Wstęp do informatyki, I rok Matematyki, 2019/2020

1. Podstawowe pojęcia. Zapis algorytmu, deklaracje i instrukcje.
2. Podprogramy. Parametry. Podprogramy rekurencyjne. Program źródłowy, kod maszynowy i kompilator.
3. Języki programowania. Składnia i semantyka.
4. Elementy analizy algorytmów. Poprawność. Złożoność czasowa i pamięciowa. Złożoność pesymistyczna, optymistyczna i średnia. Równania różnicowe.
5. Typy danych w C. Definiowanie typów.
6. Abstrakcyjne struktury danych. Stosy, kolejki i kolejki priorytetowe.
7. Algorytm HeapSort. Złożoność obliczeniowa zadania sortowania.
   Pliki.
8. Sortowanie szybkie (QuickSort).
9. Sortowanie przez scalanie. Sortowanie kubełkowe.
10. Arytmetyka zmiennopozycyjna.
11. Błędy w obliczeniach numerycznych. Algorytmy numeryczne.

---

12. Wskaźnikowe struktury danych. Zmienne dynamiczne.
    Listy jedno- i dwukierunkowe. Listowa implementacja stosu i kolejki.
13. Drzewa binarne. Drzewa binarnego wyszukiwania.
14. Drzewa AVL.
15. Kodowanie Huffmana.
16. Grafy i ich reprezentacje. Przeszukiwanie grafów wgłąb i wszerz.
17. Minimalne drzewa rozpinające. Algorytmy Prima i Kruskala.
18. Lasy rozpinające grafy skierowane. Sortowanie topologiczne.
19. Znajdowanie składowych silnie spójnych.
20. Znajdowanie najkrótszych ścieżek.
21. Sieci przepływowe.
    Klasy P i NP złożoności obliczeniowej.
22. Algorytm FFT.

Zasady zaliczania przedmiotu

Na zaliczenie przedmiotu składają się:

- Prace domowe, 40%
- Program zaliczeniowy na temat zadany przez prowadzącego ćwiczenia, 30%.
- Kolokwium (jedno w semestrze), 30%.
- Egzamin pisemny na końcu semestru — dla osób, które zaliczyły ćwiczenia, otrzymując co najmniej połowę punktów możliwych do zdobycia w semestrze.
- Osoby, u których wystąpi duża niezgodność między oceną z egzaminu pisemnego i ambicjami, mogą przystąpić do egzaminu ustnego.

W drugim semestrze zasady będą te same.

W latach 2005/2006 i 2006/2007 językiem programowania nauczanym na tych zajęciach był Pascal, w związku z czym zadania na kolokwiach i egzaminie mają elementy w tym języku. Ponadto zmieniona została kolejność przedstawianych zagadnień, co wpływa na zakres egzaminów po każdym semestrze.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10 : 15 12 grudnia 2005.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Program w Pascalu zawiera deklaracje

```
type tab = array[ 1 .. 10 ] of integer;

procedure P1 ( var a : tab; var b : tab );
  var i : integer;
begin
  for i := 1 to 10 do a[i] := b[11-i]
end {P1};

procedure P2 ( var a : tab; b : tab );
  var i : integer;
begin
  for i := 1 to 10 do a[i] := b[11-i]
end {P2};

procedure P3 ( var a : tab; var b : tab );
  var i : integer;
begin
  for i := 2 to 10 do b[i] := a[i-1]
end {P3};
```

Jaka będzie zawartość tablicy `c` po wykonaniu instrukcji `P1 ( c, c )`, `P2 ( c, c )` oraz `P3 ( c, c )`, jeśli przed wykonaniem każdej z tych instrukcji tablica ta zawiera liczby $1, \dots, 10$, tj. `c[i] = i` dla `i = 1,...,10`?

2. Tablica a na pozycjach od 1 do 10 zawiera liczby od 1 do 10 uporządkowane rosnąco. Liczby w tej tablicy zostały następnie posortowane algorytmem HeapSort (priorytet każdej liczby jest równy tej liczbie). Ile przestawień było wykonanych? Podaj zawartość tablicy po każdym przestawieniu.

3. Algorytm znajdowania liczb pierwszych został zrealizowany w następujący sposób:

```
1:  type tablica = array [ 2 .. n ] of boolean;
2:
3:  procedure Sito ( var t : tablica );
4:    var i, j : integer;
5:  begin
6:    for i := 2 to n do t[i] := true;
7:    for i := 2 to n div 2 do begin
8:      j := i+i;
9:      while j <= n do begin
10:       t[j] := false;
11:       j := j+i
12:     end
13:   end
14: end {Sito};
```

Wskaż numer linii z instrukcją, którą można uznać za operację dominującą w tym algorytmie. Oblicz jego koszt. Wiedząc, że $\sum_{i=2}^{n} \frac{1}{i} < \ln n$ dla $n \geq 2$, oszacuj rząd złożoności tego algorytmu.

4. Tablica `a`, indeksowana od 1 do n ($n > 0$) zawiera ciąg n liczb uporządkowany niemalejąco. Napisz podprogram w Pascalu, który dla ustalonego x (podanego jako parametr) znajduje w takiej tablicy liczbę `a[i]` taką, że $|x - \texttt{a[i]}| \leq |x - \texttt{a[j]}|$ dla `j=1,...,n`. Indeks `i` ma być wynikiem. Zrealizowany algorytm ma być możliwie szybki, ale przede wszystkim poprawny i dobrze objaśniony (za pomocą komentarzy w kodzie lub dodatkowego opisu). Podaj złożoność tego algorytmu.

Można użyć standardowej w Pascalu funkcji `abs`, która oblicza wartość bezwzględną parametru. Typ tej funkcji jest taki sam jak typ parametru, tj. `integer` albo `real`.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10 : 15 11 grudnia 2006.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Program w Pascalu zawiera deklaracje

```
type tab = array [ 1 .. 14 ] of integer;
var a : tab;  k : integer;

procedure P1 ( var t : tab; i, j : integer; k : integer );
  var l : integer;
begin
  t[i] := k;  k := k+1;
  if j-i > 1 then begin
    l := (i+j) div 2;
    P1 ( t, i+1, l, k );
    P1 ( t, l+1, j-1, k )
  end;
  t[j] := k
end {P1};

procedure P2 ( var t : tab; i, j : integer; var k : integer );
  var l : integer;
begin
  t[i] := k;  k := k+1;
  if j-i > 1 then begin
    l := (i+j) div 2;
    P2 ( t, i+1, l, k );
    P2 ( t, l+1, j-1, k )
  end;
  t[j] := k
end {P2};
```

Jaka będzie wartość zmiennej k i zawartość tablicy a

a) po wykonaniu instrukcji `k := 1;  P1 ( a, 1, 14, k )`,

b) po wykonaniu instrukcji `k := 1;  P2 ( a, 1, 14, k )` ?

2. Tablica a na miejscach $1, \dots, 12$ zawiera liczby $1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12$. Zakładamy, że każda liczba jest swoim priorytetem. Ile przestawień liczb w tablicy nastąpiłoby podczas porządkowania kopca za pomocą instrukcji

a)
```
for i := 2 to 12 do
  UpHeap ( a, i );
```

b)
```
for i := 12 div 2 downto 1 do
  DownHeap ( a, i, 12 );
```

Napisz końcową zawartość tablicy uporządkowanej każdym z tych sposobów.

3. Tablica o długości n zawiera liczby całkowite $a_0, \dots, a_{n-1}$. Napisz procedurę, która otrzymuje taką tablicę jako parametr i znajduje liczby i_0, i_1, takie że $0 \leq i_0 \leq i_1 \leq n$, oraz suma

$$\sum_{i:\ i_0 \leq i < i_1} a_i$$

jest największa (<u>uwaga</u>: jeśli $i_0 = i_1$, to suma jest równa 0).

Algorytm zrealizowany w procedurze powinien być możliwie szybki (tj. mieć małą złożoność obliczeniową), ale ważniejsza jest jego poprawność (można założyć, że podczas sumowania liczb a_i nie wystąpi nadmiar).

Wskaż instrukcję z operacją dominującą w tej procedurze i oblicz złożoność pesymistyczną algorytmu w zależności od n.

<u>Wskazówka:</u> W obliczeniu złożoności mogą (nie muszą) przydać się wzory

$$\sum_{k=0}^{n} k = \frac{1}{2}n(n+1),$$
$$\sum_{k=0}^{n} k^2 = \frac{1}{6}n(n+1)(2n+1),$$
$$\sum_{k=0}^{n} k^3 = \frac{1}{4}n^2(n+1)^2.$$

Kolokwium ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 12 : 15 10 grudnia 2007.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Podany niżej podprogram w C:

```
void SelectionSort ( int n, float tab[] )
{
  int   i, j, k;
  float a;

  for ( i = n-1; i > 0; i-- ) {
    for ( k = i, j = k-1;  j >= 0;  j-- )
      if ( tab[j] > tab[k] )
        k = j;
    a = tab[k];  tab[k] = tab[i];  tab[i] = a;
  }
} /*SelectionSort*/
```

realizuje tzw. algorytm sortowania przez wybieranie.

Wskaż operację dominującą w tym algorytmie i oblicz jego złożoność pesymistyczną i optymistyczną w zależności od długości sortowanego ciągu.

Co można powiedzieć o złożoności średniej tego algorytmu?

2. W tablicy `tab` znajduje się początkowo następujący ciąg liczb:

$$12, 1, 11, 2, 10, 3, 9, 4, 8, 5, 7, 6$$

Przyjmujemy, że priorytetem liczby jest ona sama. Rozważamy dwa sposoby uporządkowania kopca zbudowanego z tych liczb:

a) `for ( i = 1; i < 12; i++ ) UpHeap ( tab, i );`

b) `for ( i = 5; i >= 0; i-- ) DownHeap ( tab, i, 11 );`

Dla każdego z tych sposobów podaj wynik, czyli narysuj uporządkowany kopiec lub napisz wynikowy ciąg elementów w tablicy, oraz podaj liczbę wykonanych operacji porównywania priorytetów.

3. Napisz w języku C (ale czytelnie!) podprogram, którego pierwsze dwa parametry opisują ciąg $a_0, \ldots, a_{n-1}$ liczb typu float (pierwszy parametr typu int określa długość ciągu, drugi jest tablicą), a następne dwa parametry to wskaźnik `m` zmiennej typu int i tablica `b` zmiennych typu int.

Zadaniem podprogramu jest znalezienie wszystkich nie dających się wydłużyć podciągów $a_k, \ldots, a_l$, złożonych z kolejnych elementów ciągu $a_0, \ldots, a_{n-1}$, i takich że ciąg iloczynów $a_k, a_k \cdot a_{k+1}, \ldots, a_k \cdot \ldots \cdot a_l$ jest ściśle rosnący. Dla każdego takiego podciągu należy do tablicy `b` wpisać (na kolejnych pozycjach) liczby k i l. Zmiennej wskazywanej przez parametr `m` należy przypisać liczbę znalezionych podciągów.

Należy założyć, że tablica `b` jest dostatecznie długa.

Uwaga: liczby a_i nie muszą być dodatnie.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. $10:15$ 15 grudnia 2008.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Podana niżej procedura w C służy do wyszukiwania w uporządkowanym niemalejąco ciągu liczb o długości $n > 0$ położenia liczby x lub (jeśli jej tam nie ma) największej liczby mniejszej od x:

```
int ZnajdzPozycje ( int n, float tab[], float x )
{
  int i, j, k;

  if ( x < tab[0] )
    return -1;    /* sygnalizacja braku rozwiązania */
  i = 0, j = n;
  do {
    k = (i+j)/2;  /* jeśli i+j jest nieparzyste, */
                  /* to zaokrąglanie jest w dół */
    if ( tab[k] > x ) j = k;
                 else i = k;
  } while ( j-i > 1 )
  return i;
} /*ZnajdzPozycje*/
```

Wskaż dowolną instrukcję, która w tym algorytmie realizuje operację dominującą.

Oblicz pesymistyczną i optymistyczną złożoność obliczeniową algorytmu w zależności od długości ciągu n.

Wskazówka: Zacznij od napisania i rozwiązania równania różnicowego, przy założeniu, że $n = 2^k$ dla pewnego $k \in \mathbb{N}$, a następnie uogólnij otrzymany wynik na przypadek dowolnego $n \in \mathbb{N}$.

2. Jeśli wiadomo, że tablica zawiera n liczb uporządkowanych rosnąco, napisz, ile trzeba wykonać porównań i ile przestawień elementów, aby utworzyć w tej tablicy uporządkowany kopiec (realizujący kolejkę priorytetową, w której większa liczba ma większy priorytet). Odpowiedź uzasadnij.

Napisz w C procedurę, która dokonuje takiego uporządkowania; tablica i jej długość mają być przekazane jako parametry.

3. Funkcja f przyporządkowuje macierzy A o wymiarach $m \times n$, o współczynnikach a_{ij} (dla $0 \le i < m$, $0 \le j < n$, przy czym i jest numerem wiersza, a j — kolumny), liczbę

$$f(A) = \max_{j \in \{0,\ldots,n-1\}} \sum_{i=0}^{m-1} |a_{ij}|.$$

Napisz podprogram w C obliczający wartość funkcji f dla macierzy reprezentowanej przez odpowiednie parametry: liczby m i n oraz tablicę współczynników typu `double`, przy założeniu, że tablica ta jest jednowymiarowa, ma długość mn i współczynniki macierzy są w niej przechowywane „wierszami”, tj. pierwsze n miejsc zajmują kolejne współczynniki wiersza o numerze 0, następne n miejsc współczynniki wiersza o numerze 1 itd.

W podprogramie użyj funkcji `fabs`, która ma prototyp (zadeklarowany w pliku `math.h`)

```
double fabs ( double x );
```

i której wartością jest wartość bezwzględna parametru.

Ocenie będzie podlegać poprawność *i czytelność* kodu.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10 : 15 14 grudnia 2009.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Napisz procedurę w języku C, która ma pięć parametrów: długość pierwszej tablicy wejściowej, pierwszą tablicę wejściową, długość drugiej tablicy wejściowej, drugą tablicę wejściową i tablicę wyjściową. Tablice wejściowe zawierają ciągi liczb typu `float` posortowane niemalejąco. Zadaniem procedury jest *scalenie* tych ciągów, tj. przepisanie liczb z tablic wejściowych do wyjściowej tak, aby powstał w niej ciąg niemalejący.

2. Jeśli liczba naturalna n jest parzysta, to algorytm sortowania przez scalanie do posortowania ciągu o długości n musi posortować dwa podciągi o długości $n/2$, a następnie (metodą opisaną w pierwszym zadaniu) scalić te dwa podciągi. Ich scalenie wymaga wykonania co najmniej $n/2$ i co najwyżej $n-1$ porównań elementów.

   Zakładając, że porównanie elementów ciągu jest operacją dominującą, znajdź optymistyczną złożoność obliczeniową sortowania tą metodą ciągu o długości $n = 2^k$, gdzie $k \in \mathbb{N}$; w tym celu napisz i rozwiąż odpowiednie równanie różnicowe.

   Wiedząc, że jest to niemalejąca funkcja długości sortowanego ciągu, podaj rząd złożoności optymistycznej algorytmu sortowania przez scalanie ciągu o długości $n \in \mathbb{N}$.

3. Początkową zawartością tablicy jest następujący ciąg liczb:

   $$3,\ 5,\ 7,\ 9,\ 11,\ 6,\ 10,\ 12,\ 8,\ 4,\ 2,\ 1.$$

   Ciąg ten został posortowany za pomocą algorytmu HeapSort. Wypisz ciąg po każdym przestawieniu elementów w tablicy i podaj liczby wykonanych porównań elementów w fazie porządkowania kopca i w fazie wyjmowania z niego elementów.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 8 : 30 13 grudnia 2010.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Elementy tablicy a o długości $n > 0$ oraz zmienne `m` i `M` są typu `float`. Zmienne `n` oraz `i` są typu `int`. Napisz, czym są końcowe wartości zmiennych `m` i `M` po wykonaniu instrukcji

```
if ( n % 2 )
  { m = M = a[0];  i = 1; }
else {
  if ( a[0] < a[1] ) { m = a[0];  M = a[1]; }
              else { m = a[1];  M = a[0]; }
  i = 2;
}
for ( ; i < n; i += 2 )
  if ( a[i] < a[i+1] ) {
    if ( a[i] < m ) m = a[i];
    if ( a[i+1] > M ) M = a[i+1];
  }
  else {
    if ( a[i+1] < m ) m = a[i+1];
    if ( a[i] > M ) M = a[i];
  }
```

   Przyjmując, że porównywanie liczb zmiennopozycyjnych jest operacją dominującą, podaj złożoność zapisanego wyżej algorytmu w zależności od n. Odpowiedzi uzasadnij.

2. Uzupełnij kod podany w zadaniu 1 o nagłówek procedury i odpowiednie deklaracje zmiennych tak, aby powstał poprawny podprogram w C, otrzymujący dane i przekazujący wyniki obliczeń za pomocą parametrów (można nie przepisywać kodu, tylko napisać w odpowiednim miejscu komentarz np. `/*w tym miejscu ma być umieszczony podany kod*/`).

3. Początkową zawartością tablicy jest następujący ciąg liczb:

   $$1,\ 2,\ 4,\ 8,\ 12,\ 10,\ 6,\ 11,\ 9,\ 7,\ 5,\ 3.$$

   Ciąg ten został posortowany za pomocą algorytmu HeapSort. Wypisz ciąg po każdym przestawieniu elementów w tablicy i podaj liczby wykonanych porównań elementów w fazie porządkowania kopca i w fazie wyjmowania z niego elementów.

4. Drzewo czwórkowe o wysokości 0 jest puste, natomiast drzewo czwórkowe o wysokości $h > 0$ składa się z jednego wierzchołka zwanego korzeniem, oraz z czterech poddrzew, będących drzewami czwórkowymi o wysokości mniejszej niż h, z których co najmniej jedno ma wysokość $h - 1$.

   Rozwiązując odpowiednie równania różnicowe, znajdź wzory opisujące dla każdego $h \in \mathbb{N}$ minimalną i maksymalną liczbę wierzchołków w drzewie czwórkowym o wysokości h. Razem z równaniami różnicowymi podaj uzasadnienie, skąd one się wzięły.

Kolokwium poprawkowe ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 12 : 15 12 stycznia 2011.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Odległość euklidesowa punktów jest to pierwiastek kwadratowy z sumy kwadratów różnic poszczególnych współrzędnych tych punktów. Średnicę zbioru definiujemy jako największą odległość między jego punktami.

   Napisz podprogram w języku C obliczający średnicę zbioru n punktów w przestrzeni trójwymiarowej. Punkty powinny być opisane za pomocą odpowiedniej struktury wprowadzonej przy użyciu konstrukcji językowej z `typedef`.

   Wszystkie dane wejściowe i wynik powinny być przekazywane za pomocą parametrów i innych środków opisanych w nagłówku podprogramu.

2. Wskaż w podprogramie napisanym w zadaniu 1 dowolną operację dominującą i oblicz optymistyczną i pesymistyczną złożoność obliczeniową realizowanego przez ten podprogram algorytmu.

3. Początkową zawartością tablicy jest następujący ciąg liczb:

   $$3,\ 5,\ 7,\ 9,\ 11,\ 6,\ 10,\ 12,\ 8,\ 4,\ 2,\ 1.$$

   Ciąg ten został posortowany za pomocą algorytmu HeapSort. Wypisz ciąg po każdym przestawieniu elementów w tablicy i podaj liczby wykonanych porównań elementów w fazie porządkowania kopca i w fazie wyjmowania z niego elementów.

4. Rozwiąż równanie różnicowe

   $$a_k = 2a_{k-1} - a_{k-2} + 1$$

   z warunkiem początkowym $a_0 = 1,\ a_1 = 2$.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 8 : 30 7 grudnia 2015.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. a) Uzasadnij, że po wykonaniu procedury

```
void ZnajdzNCD ( int n, int ncd[] )
{
  int i, j;

  for ( i = 0; i < n; i++ )
    ncd[i] = 1;
  for ( i = 2; i+i < n; i++ )
    for ( j = i+i; j < n; j += i )
      ncd[j] = i;
} /*ZnajdzNCD*/
```

w tablicy przekazanej jako parametr `ncd`, dla $i = 2, \ldots, n-1$, liczba `ncd[i]` jest największym całkowitym dzielnikiem liczby i mniejszym niż i (w szczególności, jeśli i jest liczbą pierwszą, to `ncd[i]` jest równe 1).

b) Napisz podprogram w języku C, który korzystając z tablicy wypełnionej przez procedurę `ZnajdzNCD`, dla $k \in \{2, \ldots, n-1\}$, znajdzie rozkład liczby k na czynniki pierwsze. Tablicę `ncd` i liczbę k należy przekazać jako parametry. Znalezione czynniki należy wpisać do dodatkowej tablicy przekazanej jako parametr, lub „wyprowadzić", wywołując dla każdego czynnika procedurę o nagłówku `void Output ( int czynnik );` .

2. Napisz podprogram w języku C, który ma 3 parametry; pierwszy to tablica liczb typu `int`; długość tej tablicy, n, jest podana jako wartość drugiego parametru. Zadaniem procedury jest znalezienie najdłuższego spójnego fragmentu tablicy, zawierającego wyłącznie zera.

   Wynik zwracany jako wartość podprogramu jest długością tego fragmentu tablicy (i jest liczbą całkowitą od 0 do n). Trzeci parametr podprogramu ma być użyty do przekazania pozycji (indeksu) początku znalezionego fragmentu tablicy (jeśli taki fragment jest niepusty, tj. jeśli w tablicy jest przynajmniej jedno zero).

3. Wskaż w podprogramie, napisanym w poprzednim zadaniu, przynajmniej jedną instrukcję dominującą i znajdź (podając odpowiednie uzasadnienie) czasowe złożoności obliczeniowe algorytmu realizowanego przez ten podprogram: optymistyczną i pesymistyczną.

4. Rozwiąż równanie różnicowe

$$a_k = 2a_{k-1} - a_{k-2} + 2^k + 4$$

z warunkiem początkowym $a_0 = a_1 = 0$.

Kolokwium poprawkowe ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 16 : 00 11 stycznia 2016.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Program w języku C zawiera taki fragment:

```
#define N 1000    /* lub inna liczba */

typedef struct {
    ...           /* tu zamiast kropek jest coś konkretnego */
  } element;

element stosy[N];
```

W tablicy `stosy` należy zaimplementować dwa stosy o łącznej pojemności `N` — pierwszy przechowuje elementy „na początku", a drugi „na końcu" tablicy. Napisz procedury `InitStacks`, `Push1`, `Pop1`, `Stack1Empty` oraz `Push2`, `Pop2`, `Stack2Empty`, które inicjalizują te stosy oraz umieszczają element na odpowiednim stosie, zdejmują element ze stosu i sprawdzają, czy stos jest pusty. Procedury umieszczania elementu na stosie powinny sygnalizować przepełnienie w sytuacji, gdy w chwili wywołania *w obu stosach łącznie* jest już `N` elementów; należy wtedy wywołać procedurę o nagłówku

```
void StacksOverflow ( void )
```

2. Sprawdzenie, czy dana liczba naturalna n jest pierwsza i znalezienie jej najmniejszego dzielnika większego niż 1 może być wykonane przy użyciu takiego algorytmu: sprawdzamy, czy liczba n jest parzysta i jeśli nie jest, to sprawdzamy, czy jest podzielna przez kolejne liczby nieparzyste (zaczynając od 3) nie większe niż $\sqrt{n}$.

   Napisz podprogram, który dla liczby n typu `unsigned int` realizuje ten algorytm; liczba n ma być podana jako parametr, zaś jej najmniejszy dzielnik większy niż 1 ma być przekazywany jako wartość podprogramu.

3. Algorytm opisany w poprzednim zadaniu może być użyty także dla wielkich liczb naturalnych, reprezentowanych przy użyciu tablic cyfr (np. dziesiętnych). Zakładając, że koszt sprawdzenia podzielności liczby k-cyfrowej przez liczbę l-cyfrową jest równy ckl operacji dominujących, gdzie c jest pewną stałą, znajdź złożoność optymistyczną i rząd złożoności pesymistycznej tego algorytmu dla liczby k-cyfrowej. Czy jest to algorytm odpowiedni do praktycznego stosowania dla wielkich liczb? Odpowiedź uzasadnij.

4. Tak zwany rozkład QR macierzy o wymiarach $n \times n$ może być znaleziony metodą odbić Householdera: dla $n > 1$ trzeba wykonać w tym celu $2n^2 - 1$ mnożeń liczb rzeczywistych (które są operacjami dominującymi), a następnie znaleźć rozkład QR pewnej macierzy o wymiarach $(n-1) \times (n-1)$. Dla $n = 1$ nie trzeba wykonywać żadnych działań (bo wtedy czynnik Q jest macierzą jednostkową, a czynnik R oryginalną macierzą, która ma jeden współczynnik).

   Ułóż i rozwiąż odpowiednie równanie różnicowe ze stosownym warunkiem początkowym, aby obliczyć koszt znajdowania rozkładu QR macierzy $n \times n$.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 8 : 30 12 grudnia 2016.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań (w tym czytelność napisanego kodu w C) i poprawność uzasadnienia każdej odpowiedzi.

1. Każdą permutacje zbioru n-elementowego możemy reprezentować za pomocą tablicy o długości n, w której są przechowywane wszystkie liczby całkowite od 0 do $n-1$; permutacja σ jest reprezentowana w ten sposób, że $\sigma(i) = \mathtt{a[i]}$ dla $i = 0, \ldots, n-1$. Napisz w języku C podprogramy

   a) `SprawdzPermutacje`, który sprawdza, czy podany w tablicy ciąg liczb jest poprawną reprezentacją permutacji (tj. czy występują w nim *wszystkie* liczby od 0 do $n-1$). Podprogram ma zwracać wartość `1`, jeśli ciąg jest poprawny, albo `0` w przeciwnym razie. Podprogram może mieć dodatkowy parametr — tablicę roboczą zmiennych typu <u>char</u> o długości n.

   b) `ZlozPermutacje`, króry znajduje permutację σ będącą złożeniem dwóch permutacji danych, $\sigma(i) = \sigma_2(\sigma_1(i))$ dla $i = 0, \ldots, n-1$, reprezentowanch przez zawartość odpowiednich tablic. Wynik ma być wpisany do trzeciej tablicy przekazanej jako parametr.

   Oba proprogramy nie mogą mieć żadnych efektów ubocznych i nie mogą zmieniać zawartości tablic z danymi ciągami liczb.

2. Dwie tablice o długości n zawierają pewne liczby całkowite (ustawione w dowolnej kolejności). Zaproponuj (podając opis słowami, *nie* kod w C) algorytm znajdowania w tych tablicach największego wspólnego elementu, tj. największej liczby x, która występuje w obu tablicach. Algorytm może, w razie potrzeby, przestawiać elementy w każdej z tablic. Przyjmując jako operację dominującą porównywanie liczb, podaj optymistyczną i pesymistyczną złożoność obliczeniową tego algorytmu.

3. Znajdź ciąg $(a_k)_{k\in\mathbb{N}}$, taki że $a_0 = 0$, $a_1 = 1$ oraz

$$a_k = 4a_{k-2} + k \cdot 2^k, \quad \text{dla } k > 1.$$

4. Wytęż wzrok i znajdź błędy w podanym niżej podprogramie w języku C (podprogram ma znaleźć w przekazanej jako parametr tablicy o długości n liczbę najmniejszą i największą). Dla każdego błędu napisz, na czym on polega.

```
void MinMax ( int n, int a[], int *min, *max );
{
  if ( n % 2 == 1 ) {
    min = max = a[0];
    i = 1;
  }
  else {
    if ( a[0] < a[1] ) { min = a[0], max = a[1] };
    else { min = a[1], max = a[0] };
    i = 2;
  }
  for ( ; i < n; i += 2 ) {
    if ( a[i] < a[i+1] ) {
      if a[i] < min  min = a[i],
      if a[i+1] > max  max = a[i+1];
    }
    else {
      if a[i+1] < min  min = a[i+1],
      if a[i] > max  max = a[i];
    }
  }
  return max-min;
} /*MinMax*/
```

Kolokwium poprawkowe ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 16 : 00 16 stycznia 2017.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań (w tym czytelność napisanego kodu w C) i poprawność uzasadnienia każdej odpowiedzi.

1. Każdą permutację zbioru n-elementowego możemy reprezentować za pomocą tablicy o długości n, w której są przechowywane wszystkie liczby całkowite od 0 do $n-1$; permutacja σ jest reprezentowana w ten sposób, że $\sigma(i) = \mathtt{a[i]}$ dla $i = 0, \dots, n-1$.

   Niech k oznacza liczbę par (i, j), takich że $i < j$ oraz $\sigma(i) > \sigma(j)$.
   Znak permutacji σ jest równy $+1$ jeśli k jest parzyste, albo -1 w przeciwnym przypadku.

   Napisz w języku C podprogram `ZnakPermutacji`, który oblicza znak permutacji reprezentowanej przez ciąg liczb podany w tablicy (przy założeniu, że ciąg ten ma długość n i są w nim wszystkie liczby całkowite od 0 do $n-1$). Czasowa złożoność obliczeniowa algorytmu zaimplementowanego w tym podprogramie, w miarę możliwości, powinna być rzędu mniejszego niż n^2. Podprogram może mieć dodatkowy parametr — tablicę roboczą zmiennych typu `int` o długości n — ale nie może zmieniać zawartości tablicy z danym ciągiem liczb.

   Wskazówka. Przestawienie dowolnych dwóch liczb w ciągu powoduje powstanie ciągu reprezentującego permutację o przeciwnym znaku.

2. Tablica a o wymiarach $n \times n$ (gdzie $n > 1$) zawiera liczby całkowite a_{ij}. Napisz podprogram w języku C, który znajduje liczby $k, l \in \{0, \dots, n\}$, takie że suma

   $$s_{kl} = \sum_{i=0}^{k-1} \sum_{j=0}^{l-1} a_{ij}$$

   jest największa (uwaga: $s_{k0} = s_{0l} = 0$ dla każdego k, l).
   Złożoność obliczeniowa zaimplementowanego algorytmu powinna być rzędu n^2, przy czym dla osiągnięcia tego celu można korzystać z dodatkowej tablicy $n \times n$ liczb typu `int`.

3. Znajdź ciąg $(a_k)_{k \in \mathbb{N}}$, taki że $a_0 = 1$, $a_1 = 1$ oraz

   $$a_k = 2a_{k-1} - a_{k-2} + k, \quad \text{dla } k > 1.$$

4. Wytęż wzrok i znajdź błędy w podanym niżej podprogramie w języku C (podprogram ma znaleźć punkt w przedziale $[a, b]$, w którym dana ciągła funkcja f osiąga minimalną wartość — zastosowana jest metoda tzw. złotego podziału). Dla każdego błędu napisz, na czym on polega.

```
float GoldenRat ( float a, b; float (*f)(x) )
#define TAU 0.618;
{
  float c, d, fc, fd;

  c = TAU*a+(1-TAU)*b,  fc = f(c);
  d = (1-TAU)*a+TAU*b,  fd = f(b);
  while |c-d| > eps {
    if fc < fd {
      b = d,  d = c;
      c = TAU*a+(1-TAU)*b,  fc = f(c)
    }
    else {
      a = c,  c = d;
      d = (1-TAU)*a+TAU*b,  fd = f(d)
    }
  }
  return if fc < fd ? c else d;
} /*GoldenRat*/
```

Kolokwium ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 8 : 30 11 grudnia 2017.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań (w tym czytelność napisanego kodu w C) i poprawność uzasadnienia każdej odpowiedzi.

1. Algorytm FFT na podstawie danego ciągu liczb zespolonych o długości n oblicza ciąg liczb zespolonych o długości n. Jeśli $n = 1$, to koszt tego obliczenia jest równy 0. Jeśli liczba n jest podzielna przez 3, to algorytm rozwiązuje 3 zadania dla ciągów o długości $n/3$, po czym „scala” wyniki w wynik końcowy, wykonując przy tym $3n$ mnożeń zespolonych, które są operacjami dominującymi.

   Znajdź wzór opisujący złożoność obliczeniową algorytmu FFT dla zadań o rozmiarze $n = 3^k$, gdzie k jest liczbą naturalną, układając i rozwiązując odpowiednie równanie różnicowe.

2. Dane są dwa ciągi rosnące liczb całkowitych, o długościach n i m. Napisz podprogram w języku C, którego parametry reprezentują te ciągi (podane w tablicach), i który oblicza i podaje jako wynik (wartość funkcji) liczbę wspólnych elementów obu ciągów (tj. liczbę liczb występujących na pewnych miejscach w obu ciągach). Algorytm realizowany przez ten podprogram ma mieć złożoność obliczeniową rzędu $n + m$.

3. W tablicy jest podany ciąg liczb

   $$1,\ 12,\ 4,\ 11,\ 7,\ 9,\ 10,\ 6,\ 2,\ 3,\ 5,\ 8.$$

   Liczby te zostały posortowane przy użyciu algorytmu HeapSort. Napisz zawartość tablicy po każdym przestawieniu jej elementów i znajdź liczbę porównań wykonanych podczas sortowania.

Kolokwium poprawkowe ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14 : 00 11 stycznia 2018.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań (w tym czytelność napisanego kodu w C) i poprawność uzasadnienia każdej odpowiedzi.

1. Napisz procedurę w C, która wykonuje mnożenie wielomianów, tj. mając dane współczynniki $a_0, \ldots, a_n$ i $b_0, \ldots, b_m$ wielomianów $a(x) = \sum_{i=0}^{n} a_i x^i$ oraz $b(x) = \sum_{j=0}^{m} b_j x^j$, oblicza współczynniki $c_0, \ldots, c_{n+m}$ wielomianu $c(x) = \sum_{k=0}^{n+m} c_k x^k$. Stopnie wielomianów a i b oraz tablice z ich współczynnikami, a także tablica, w której ma być umieszczony wynik, mają być przekazane jako parametry.

   Wskazówka. $c(x) = \sum_{i=0}^{n}\sum_{j=0}^{m} a_i b_j x^{i+j}$.

2. Algorytm odbić Householdera służy do znajdowania tzw. rozkładu ortogonalno-trójkątnego macierzy o wymiarach $m \times n$ (gdzie $m > n$). Wymaga on wykonania $m(2n-1)$ mnożeń, które są operacjami dominującymi, a następnie dla $n = 1$ to koniec, a jeśli $n > 1$, to trzeba rozwiązać podobne zadanie dla macierzy o wymiarach $(m-1) \times (n-1)$. Ułóż i rozwiąż odpowiednie równanie różnicowe, aby otrzymać wzór opisujący koszt obliczenia w zależności od m i n.

   Wskazówka. Wprowadź parametr pomocniczy $k = m - n$.

3. Liczby z ciągu $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12$ należy ustawić w kopiec reprezentujący kolejkę priorytetową (w której większa liczba ma większy priorytet). Narysuj drzewo tego kopca i podaj liczbę porównań priorytetów elementów tego ciągu, jeśli

   (a) do początkowo pustego kopca wstawiane są kolejne elementy ciągu,

   (b) cały ciąg jest dany w tablicy i porządkowanie kopca odbywa się „od dołu” w tej tablicy.

Kolokwium ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. $8:30$ 10 grudnia 2018.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań (w tym czytelność napisanego kodu w C) i poprawność uzasadnienia każdej odpowiedzi.

1. Napisz podprogram w języku C, który dla danej jako parametr liczby naturalnej x znajduje liczby naturalne a i n, takie że $x = a^n$, a ponadto liczba a jest najmniejsza, dla której ta równość zachodzi z pewnym n. Jeśli x nie jest całkowitą potęgą żadnej mniejszej liczby naturalnej, to podprogram ma znaleźć $a = x$ i $n = 1$. Znalezione liczby a i n należy przypisać zmiennym wskazywanym przez odpowiednie parametry podprogramu.

2. W tablicy `a` o długości $2n$ znajdują się liczby całkowite od 0 do $n-1$, przy czym każda z nich występuje dwukrotnie, a poza tym kolejność liczb w tablicy jest przypadkowa. Napisz podprogram w języku C, którego zadaniem jest obliczenie dla każdej z tych liczb odległości między miejscami, na których występuje w tablicy `a`. Wyniki mają być wpisane do tablicy `b` podanej jako trzeci parametr podprogramu (ma być `b[i] == k-j`, gdzie liczby `j`, `k` są indeksami, takimi że `a[k] == a[j] == i` oraz `k > j`).

   Wskaż instrukcje dominujące w algorytmie realizowanym przez ten podprogram. Algorytm ten ma mieć złożoność obliczeniową rzędu n.

3. Algorytm Choleskiego służy do znajdowania tzw. rozkładu trójkątno-trójkątnego macierzy symetrycznej i dodatnio określonej o wymiarach $n \times n$. Dla $n = 1$ wymaga to jednego obliczenia pierwiastka kwadratowego, zaś dla $n > 1$ trzeba obliczyć pierwiastek kwadratowy i dodatkowo wykonać $\frac{1}{2}(n^2+n)-1$ innych działań arytmetycznych (mnożeń lub dzieleń), a następnie rozwiązać podobne zadanie o wymiarze $n-1$. Ułóż i rozwiąż odpowiednie równanie różnicowe, aby otrzymać jawny (nie-rekurencyjny) wzór opisujący złożoność tego algorytmu.

Kolokwium poprawkowe ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. $14:15$ 17 stycznia 2019.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań (w tym czytelność napisanego kodu w C) i poprawność uzasadnienia każdej odpowiedzi.

1. Napisz podprogram w języku C, który otrzymuje jako parametry dwie liczby, n i N, oraz trzy tablice: pierwsza, o długości n, zawiera dowolne liczby całkowite dodatnie, a druga tablica liczba zawiera N liczb całkowitych. Trzecia tablica, do której należy wpisać wynik, ma długość n i elementy typu <u>`unsigned`</u> <u>`int`</u>. Dla każdego i elementowi i-temu trzeciej tablicy należy przypisać liczbę tych elementów drugiej tablicy, które są podzielne przez i-ty element pierwszej tablicy.

2. W tablicach dane są dwa ciągi rosnące liczb całkowitych, o długościach m i n. Napisz podprogram w języku C, którego parametry reprezentują te ciągi i który znajduje i podaje jako wynik liczbę takich elementów pierwszego ciągu, których kwadraty występują w drugim ciągu. Skorzystaj z warunku, że ciągi są rosnące, aby otrzymać algorytm o jak najmniejszej pesymistycznej złożoności obliczeniowej.

   Wskaż instrukcje dominujące w algorytmie realizowanym przez ten podprogram i podaj, z uzasadnieniem, jego pesymistyczną złożoność obliczeniową.

3. Algorytm sortowania przez scalanie polega na podzieleniu ciągu o długości n na dwa fragmenty (których długości są takie same lub różnią się o 1), rekurencyjnym posortowaniu tych fragmentów, a następnie „scaleniu” posortowanych fragmentów, co wymaga co najmniej $\lfloor n/2 \rfloor$ i co najwyżej $n-1$ operacji porównywania elementów ciągu. Zakładając, że $n = 2^k$, gdzie k jest liczbą naturalną, napisz równania różnicowe z odpowiednimi warunkami początkowymi, opisujace złożoności optymistyczną i pesymistyczną tego algorytmu, a następnie rozwiąż jedno z tych równań, aby otrzymać jawny wzór na złożoność.

Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 13 : 00 30 stycznia 2006.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Program w Pascalu zawiera następujący fragment:

```
type tab = array [ 1 .. n ] of real;

function Suma ( var t : tab; i, j : integer ) : real;
  var k : integer;
begin
  if i = j then Suma := t[i]
  else begin
    k := (i+j) div 2;
    Suma := Suma ( t, i, k ) + Suma ( t, k+1, j )
  end
end {Suma};
```

Funkcja wywołana z parametrami t oraz $i \leq j$ oblicza z dokładnością do błędów zaokrągleń sumę $\sum_{k=i}^{j} t[k]$ (funkcja realizuje tzw. algorytm sumowania parami, interesujący właśnie z uwagi na te błędy).

a) Przypuśćmy, że $n \geq 10$ i funkcja została wywołana z parametrami i=1, j=10. Narysuj drzewo binarne, którego liśćmi są elementy tablicy t, i którego pozostałe wierzchołki odpowiadają wykonanym dodawaniom, a poddrzewa każdego z tych wierzchołków są argumentami dodawania.

Wskazówka: Zacznij rysowanie od korzenia.

b) Jak wysokość drzewa zależy od parametrów wywołania procedury? Odpowiedź uzasadnij.

2. Tablica a, indeksowana od 1 do n, zawiera liczby typu integer.
Program zawiera instrukcję

```
for i := 2 to n do
  for j := 1 to i-1 do a[j+1] := a[j] + a[j+1];
```

Napisz instrukcję, której wykonanie bezpośrednio po instrukcji podanej wyżej spowodowałoby przywrócenie początkowej zawartości tablicy a.

3. Przy użyciu typów zdefiniowanych niżej:

```
type pdlista = ^dlista;
     dlista = record
       x : integer;
       poprz, nast : pdlista
     end;
```

program tworzy dwukierunkowe listy zamknięte (tj. cykliczne). Napisz podprogram — funkcję, która ma dwa parametry, będące wskaźnikami do pewnych niepustych i rozłącznych list. Zadaniem podprogramu jest połączenie tych dwóch list w jedną listę zamkniętą. Wartość funkcji ma być wskaźnikiem do pewnego elementu połączonej listy.

4. Tablica a, indeksowana od 1 do n, zawiera pewien ciąg liczb rzeczywistych. Napisz podprogram, który znajduje niepusty podciąg tego ciągu, złożony z jego kolejnych elementów, którego suma ma najmniejszą wartość bezwzględną. Wynikiem podprogramu mają być indeksy pierwszego i ostatniego elementu podciągu.

Podaj, z uzasadnieniem, złożoność obliczeniową algorytmu realizowanego przez ten podprogram.

5. Dyskretna transformata Fouriera ciągu liczb (zespolonych) $a_0, \ldots, a_{n-1}$ jest pewnym ciągiem liczb $f_0, \ldots, f_{n-1}$. Jeśli $n = 1$, to $f_0 = a_0$ i koszt obliczenia transformaty jest równy 0. Jeśli liczba n jest podzielna przez m, to zadanie obliczenia transformaty można sprowadzić do obliczenia m transformat podciągów o długości $\frac{n}{m}$, a następnie wykonania $c(m-1)n$ działań arytmetycznych (które są operacjami dominującymi; liczba c jest stałą).

Przy założeniu, że $n = 3^k$, gdzie k jest liczbą naturalną, znajdź złożoność algorytmu obliczania transformaty, polegającego na rekurencyjnym stosowaniu podanego wyżej pomysłu — ciąg dany dłuższy niż 1 jest dzielony na 3 podciągi o jednakowej długości. Transformaty podciągów, po ich obliczeniu, są „scalane” w celu uzyskania transformaty całego ciągu.

Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10:00 27 stycznia 2007.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Rozważamy algorytm QuickSort, korzystający z takiej funkcji `FindPivot`, która kosztem wykonania najwyżej trzech porównań kluczy elementów sortowanej tablicy gwarantuje, że jeśli wszystkie klucze w tablicy są różne, to klucz elementu dzielącego nigdy nie będzie najmniejszy ani największy w przetwarzanym fragmencie tablicy o długości większej niż 2 (funkcja nie wykonuje porównań, jeśli fragment ma tylko 2 elementy).

   a) Kiedy koszt sortowania (mierzony liczbą porównań kluczy) jest największy? Odpowiedź uzasadnij.

   b) Oblicz złożoność pesymistyczną sortowania tym algorytmem tablicy o długości n.

2. Typy zdefiniowane w następujący sposób:

```
type pdlista = ^dlista;
     dlista = record
       k : integer;
       poprz, nast : pdlista
     end;
```

   zostały użyte w programie do utworzenia list dwukierunkowych; pola `poprz` i `nast` wskazują odpowiednio poprzedni i następny element listy, przy czym pole `poprz` pierwszego elementu i pole `nast` ostatniego elementu listy mają wartość `nil`.

   a) Napisz procedurę, która otrzymuje parametr (przekazywany przez wartość), będący wskaźnikiem dowolnego elementu takiej niepustej listy. Zadaniem tej procedury jest przestawienie tego elementu w liście z jego elementem następnym (jeśli parametr wskazuje element ostatni, to procedura ma niczego w liście nie zmieniać).

   b) Zaproponuj algorytm i napisz procedurę sortowania listy w kolejności niemalejących wartości pól k, korzystający z procedury działającej zgodnie z opisem w punkcie a).

3. Do początkowo pustego drzewa binarnego wyszukiwania zostało wstawione n wierzchołków o różnych kluczach, przy czym każdy wierzchołek bezpośrednio po dołączeniu do drzewa był liściem i drzewo nigdy nie było przebudowywane.

   a) Jaka może być największa, a jaka najmniejsza liczba porównań kluczy wykonana podczas tego wstawiania? Odpowiedź uzasadnij.

   b) Narysuj drzewo otrzymane po wstawieniu wierzchołków kolejno z kluczami $f, k, c, d, e, a, b, h, g, j, i$ (przy założeniu, że porządek w zbiorze kluczy jest alfabetyczny).
   Ile porównań kluczy zostało wykonane podczas budowania tego drzewa?

4. Tablica a, indeksowana od 1 do $n > 1$ zawiera liczby całkowite $a_1, \ldots, a_n$. Dla $j \in \{0, \ldots, n\}$ oznaczymy

$$b_j = \sum_{i=1}^{j} a_i - \sum_{i=j+1}^{n} a_i,$$

   przy czym jeśli $i = 0$ to pierwsza, a jeśli $i = n$ to druga suma w powyższym wzorze jest równa 0.

   a) Napisz funkcję w Pascalu, której wartością jest liczba $k \in \{0, \ldots, n\}$, taka że liczba b_k ma najmniejszą wartość bezwzględną.

   b) Wskaż w napisanym podprogramie instrukcję dominującą i oblicz pesymistyczną złożoność zastosowanego algorytmu.

5. Jeśli liczba $n > 1$ jest podzielna przez $m > 1$, to zadanie obliczenia dyskretnej transformaty Fouriera ciagu liczb (zespolonych) $a_0, \ldots, a_{n-1}$ można sprowadzić do obliczenia m transformat podciągów o długości $\frac{n}{m}$, a następnie do obliczenia wyniku kosztem $c(m-1)n$ działań arytmetycznych ($c = \text{const}$). Dla $n = 1$ koszt obliczenia transformaty jest zerowy.

   Przypuśćmy, że liczba n jest podzielna przez 4. Mając dane 4 transformaty odpowiednich podciągów o długości $\frac{n}{4}$, można obliczyć transformatę całego ciągu danego, obliczając najpierw dwie transformaty podciągów o długości $\frac{n}{2}$, a następnie końcowy wynik, albo można obliczyć końcowy wynik bezpośrednio na podstawie tych czterech transformat podciągów.
   Która metoda wymaga wykonania mniejszej liczby działań?
   Odpowiedź uzasadnij.

Egzamin poprawkowy ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 16:00 9 marca 2007.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dla każdego wierzchołka w drzewie binarnym określamy jego poziom w następujący sposób: Poziom korzenia jest równy 0. Jeśli poziom pewnego wierzchołka jest równy k, to poziomy korzeni obu poddrzew tego wierzchołka są równe $k+1$.

   Program w Pascalu tworzy drzewa BST ze zmiennych dynamicznych typu `drzewo`:

```
type pdrzewo = ^drzewo;
     drzewo = record
                 klucz : integer;
                 lewe, prawe : pdrzewo
               end;
```

   Napisz procedurę do tego programu, która dla każdego wierzchołka drzewa (którego korzeń jest wskazywany przez parametr) wypisze na plik `output` jedną linię tekstu, zawierającą dwie liczby: klucz i poziom wierzchołka. Kolejność, w jakiej mają być wypisane dane dla wierzchołków, ma być zgodna z porządkiem kluczy przyjętym podczas tworzenia drzewa.

2. Tablica zawiera liczby (rzeczywiste) zmiennopozycyjne $a_1, \ldots, a_n$. Dla dowolnego zbioru $S \subset \{1, \ldots, n\}$ określamy liczbę b_S, równą iloczynowi tych elementów ciągu $a_1, \ldots, a_n$, których indeksy należą do S (uwaga: $b_\emptyset = 1$). Dowolny taki zbiór S możemy reprezentować przy użyciu tablicy elementów typu `boolean` (`S[i]` ma wartość `true` wtedy i tylko wtedy, gdy $i \in S$).

   Napisz podprogram — funkcję Pascalową, który otrzymuje jako parametry tablicę `a` z pewnym ciągiem liczb i tablicę `S`. Zadaniem podprogramu jest wyznaczenie takiego zbioru S, że liczba b_S ma największą wartość bezwzględną, i umieszczenie reprezentacji tego zbioru w tablicy `S`. Wartością funkcji ma być liczba b_S.

   Wskaż instrukcję dominującą i podaj rząd złożoności zrealizowanego algorytmu.

3. Jeśli liczba $n > 1$ jest podzielna przez $m > 1$, to zadanie obliczenia dyskretnej transformaty Fouriera ciagu liczb (zespolonych) $a_0, \ldots, a_{n-1}$ można sprowadzić do obliczenia m transformat podciągów o długości $\frac{n}{m}$, a następnie do obliczenia wyniku kosztem $c(m-1)n$ działań arytmetycznych ($c = \text{const}$).

   Przypuśćmy, że liczba n jest podzielna przez 10. Mając dane 10 transformat odpowiednich podciągów o długości $\frac{n}{10}$, można obliczyć transformatę całego ciągu danego, obliczając najpierw dwie transformaty podciągów o długości $\frac{n}{2}$, a następnie końcowy wynik, albo najpierw 5 transformat podciągów o długości $\frac{n}{5}$, a potem końcowy wynik. Który sposób wymaga wykonania mniejszej liczby działań? Odpowiedź uzasadnij.

4. Niech $k \geq 2$. Podczas sortowania za pomocą algorytmu QuickSort pewnej tablicy o długości n okazało się, że po każdym wywołaniu funkcji `Partition`, z parametrami i oraz j określającymi odpowiedni fragment tablicy o długości większej niż 2, pozycja p elementu dzielącego spełniała następujący warunek:

   a) $p - i \leq k$ lub $j - p \leq k$.

   b) $3(p - i) \geq j - i$ oraz $3(j - p) \geq j - i$.

   Jaki jest w każdym z tych przypadków rząd złożoności algorytmu? Odpowiedź uzasadnij.

   Wskazówka: Zbadaj głębokość rekurencji w wywołaniach procedury `QuickSort` w obu tych przypadkach.

5. Typy zdefiniowane w następujący sposób:

```
type pdlista = ^dlista;
     dlista = record
       klucz : integer;
       poprz, nast : pdlista
     end;
```

   zostały użyte w programie do utworzenia zamkniętych list dwukierunkowych; pola `poprz` i `nast` wskazują odpowiednio poprzedni i następny element listy.

   Napisz procedurę, która otrzymuje parametr, będący wskaźnikiem dowolnego elementu takiej niepustej listy. Zadaniem tej procedury jest przestawienie tego elementu w liście z jego elementem poprzednim.

Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 12:00 28 stycznia 2008.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Palindrom jest to napis identyczny z napisem powstałym po odwróceniu kolejności znaków.

   a) Napisz w języku C procedurę `Palindrom`, która ma dwa parametry: liczbę całkowitą `n` i tablicę elementów typu `char`, zawierającą napis o długości `n`. Procedura ma zwrócić wartość różną od zera wtedy i tylko wtedy, gdy podany napis jest palindromem.

   b) Napisz procedurę `ZnajdzPalindromy`, której parametry reprezentują pewien napis jak wyżej. Procedura ta ma znaleźć w podanym napisie wszystkie palindromy o długości większej niż 2. Na przykład w napisie `BACABADACELE` należy znaleźć `BACAB`, `ACA`, `ABA`, `ADA` i `ELE`.

   Dla każdego znalezionego palindromu należy wywołać procedurę o nazwie `Wypisz`, przekazując jej jako parametry długość palindromu i wskaźnik jego pierwszego znaku.

2. Jeśli sprawdzenie, czy dany napis o długości n jest palindromem, zajmuje co najmniej jedno i co najwyżej $\lfloor \frac{n}{2} \rfloor$ operacji porównywania znaków, a procedura wyszukiwania palindromów sprawdza każdy fragment napisu o długości $k \in \{3, \dots, n\}$, to jakie są (mierzone liczbą porównań) złożoności pesymistyczna i optymistyczna tej procedury w zależności od n? Wyprowadź dokładne wzory i podaj rzędy tych złożoności.

3. Pewien algorytm rekurencyjny rozwiązuje pewne zadanie o rozmiarze $n = 2$ kosztem co najwyżej 8 operacji (uznanych za dominujące). Jeśli $k \in \mathbb{N}$, to aby rozwiązać zadanie o rozmiarze $n = 2^k + 1$ algorytm wykonuje dwie operacje, a następnie musi znaleźć rozwiązania co najwyżej siedmiu analogicznych zadań o rozmiarze $(n+1)/2$.

   Przy założeniu, że dla $n \geq 2$ pesymistyczna złożoność obliczeniowa tego algorytmu jest funkcją rosnącą, znajdź jej oszacowania (z góry i z dołu), układając i rozwiązując odpowiednie równanie różnicowe.

4. W tablicy dwuwymiarowej `a` są podane współczynniki a_{ij} wielomianu

$$w(x, y) = \sum_{i=0}^{n} \sum_{j=0}^{m} a_{ij} x^i y^j = \sum_{i=0}^{n} \Big(\sum_{j=0}^{m} a_{ij} y^j \Big) x^i.$$

   a) Napisz w C procedurę, która przy użyciu schematu Hornera oblicza wartość wielomianu w dla podanych liczb x, y. Tablica `a` (elementów typu `float`) oraz liczby x, y typu `float` mają być parametrami tej procedury.

   Przyjmij założenie, że liczby m i n są określone „na stałe” w programie, za pomocą odpowiednich dyrektyw `#define`.

   Obliczona wartość wielomianu ma być wartością procedury.

   b) Wiedząc, że wartość wielomianu rzeczywistego $w(x) = \sum_{k=0}^{n} a_k x^k$, obliczona za pomocą schematu Hornera zrealizowanego w arytmetyce zmiennopozycyjnej, jest dokładną wartością pewnego wielomianu $\tilde{w}(x) = \sum_{k=0}^{n} \tilde{a}_k x^k$, gdzie $\tilde{a}_k = a_k(1 + \gamma_k)$, $|\gamma_k| \leq (2k+1)\nu$ (symbol $\nu = 2^{-t}$ oznacza maksymalny błąd względny zaokrąglenia), wykaż, że algorytm zrealizowany w punkcie a) jest numerycznie poprawny.

5. Używając procedury `Partition`, takiej jak na wykładzie, oraz stosu liczb całkowitych, obsługiwanego przez procedury `InitStack`, `Push`, `Pop` i `StackEmpty`, można zrealizować algorytm QuickSort bez używania rekurencji. W tym celu na stos wstawiamy dwie liczby, 0 i $n-1$. Następnie w pętli, wykonywanej aż do opróżnienia stosu, wykonujemy kolejno następujące czynności:

   - Zdejmujemy ze stosu dwie liczby, określające koniec i początek fragmentu tablicy do posortowania.
   - Wywołujemy procedurę `Partition`. Wyznacza ona punkt podziału fragmentu tablicy na dwa mniejsze.
   - Dla każdego z tych fragmentów, jeśli jest dłuższy niż 1, wstawiamy na stos parę liczb, określających początek i koniec fragmentu.

   Napisz w C podprogram `QuickSort`, który działa w opisany wyżej sposób.

   Czy użycie kolejki zamiast stosu daje poprawny algorytm sortowania, i jeśli tak, to czy warto dokonać takiej zmiany? Odpowiedź uzasadnij.

Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 13:00 26 stycznia 2009.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dany jest ciąg liczb całkowitych o długości n i liczba dodatnia $m < n$. Zadanie polega na znalezieniu w tym ciągu m największych elementów.

   a) Mając do dyspozycji procedury `UpHeap` i `DownHeap`, operujące na kopcu (nie trzeba ich pisać), napisz procedurę w C, która otrzymuje jako parametry liczbę n, tablicę z ciągiem danym, liczbę m i dodatkową tablicę o długości m, w której procedura ma umieścić m największych elementów ciągu danego. Kolejność elementów w tablicy z danym ciągiem może zostać zmieniona.

   b) Znając rzędy pesymistycznej złożoności czasowej procedur obsługi kopca (trzeba je znać), podaj rząd złożoności czasowej procedury napisanej w punkcie a). Odpowiedź uzasadnij.

2. Procedura `Partition`, używana w algorytmie QuickSort, wywołana z parametrami `a`, `i`, `j`, gdzie $0 \leq i < j < n$, kosztem wykonania $j - i$ porównań kluczy przestawia elementy tablicy `a` między pozycjami `i` i `j` w taki sposób, że elementy o kluczach mniejszych niż klucz elementu dzielącego znajdują się przed elementem dzielącym, a pozostałe za nim.

   Przypuśćmy, że algorytm QuickSort ma być użyty do posortowania ciągu n elementów o jednakowych kluczach.

   a) Oblicz koszt sortowania takich danych.

      Wskazówka: Zbadaj końcową pozycję elementu dzielącego w przetwarzanym przez procedurę `Partition` fragmencie tablicy.

   b) Zaproponuj modyfikację procedury `Partition` (polegającą na wykonaniu dodatkowych porównań kluczy), prowadzącą do zmniejszenia kosztu sortowania takich danych. Nie trzeba pisać kodu w C, wystarczy opisać słowami, na czym ta modyfikacja polega.

      Uzasadnij skuteczność proponowanej modyfikacji.

3. a) Wartość wyrażenia $a^2 - ab + b^2$ została obliczona za pomocą instrukcji

      ```
      c = a - b;
      x = 0.5*((a*a + b*b) + c*c);
      ```

      przy czym zmienne `a`, `b`, `c` i `x` są zmiennopozycyjne. Zakładając, że w obliczeniach nie wystąpił nadmiar ani niedomiar, napisz wyrażenie (zawierające liczby a i b oraz popełnione błędy zaokrągleń), którego wartość została nadana zmiennej `x`.

   b) Reprezentacja zmiennopozycyjna liczby ma t bitów mantysy. Jaka jest największa liczba całkowita n, taka że wszystkie liczby całkowite ze zbioru $\{-n, \ldots, n\}$ można reprezentować dokładnie? Odpowiedź uzasadnij.

4. Przypuśćmy, że zadanie obliczenia sumy n liczb danych w tablicy należy rozwiązać przy użyciu algorytmu sumowania parami.

   a) Napisz w języku C procedurę rekurencyjną, realizującą ten algorytm (określony za pomocą parametrów fragment tablicy dłuższy niż 1 należy podzielić na dwie części i obliczyć i dodać sumy liczb w tych częściach).

   b) Uzasadnij, że algorytm korzystający z kolejki:

      ```
      InitQueue ();
      for ( i = 0; i < n; i++ ) Enqueue ( a[i] );
      for ( i = 0; i < n-1; i++ ) {
        Dequeue ( &b1 );  Dequeue ( &b2 );
        Enqueue ( b1 + b2 );
      }
      Dequeue ( &s );
      ```

      realizuje algorytm sumowania parami, z dokładnością do kolejności składników.

5. Dane są dwa napisy, reprezentowane jako ciągi znaków ASCIIZ w tablicach, przy czym pierwszy napis (`slowo`) jest krótszy niż napis drugi (`tekst`).

   a) Napisz w C procedurę która oblicza liczbę wystąpień słowa w tekście.

   b) Oblicz złożoność pesymistyczną i optymistyczną algorytmu zrealizowanego przez tę procedurę w zależności od długości słowa, m i długości tekstu, n.

Egzamin poprawkowy ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 16:00 5 marca 2009.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dana jest tablica o długości n, której elementy $a_0, \ldots, a_{n-1}$ są liczbami całkowitymi.

   a) Napisz podprogram, który otrzymuje parametry opisujące taką tablicę oraz dwie liczby całkowite, $l > 0$ i s. Zadaniem podprogramu jest obliczenie liczby takich indeksów k, że
   $$\sum_{i=k}^{k+l-1} a_i = s.$$

   b) Oblicz złożoność pesymistyczną algorytmu realizowanego przez procedurę napisaną w punkcie a). Opisz słowami tańszy algorytm, który można zastosować do rozwiązania tego zadania, jeśli wiadomo, że ciąg liczb w tablicy jest niemalejący.

2. Dane są liczby zmiennopozycyjne $x_1, \ldots, x_n$. Zadanie polega na obliczeniu liczby
   $$a = \Big(\Big(\big((x_1 + x_2)^2 + x_3\big)^2 + x_4\Big)^2 + \cdots + x_n\Big)^2.$$

   a) Napisz wyrażenie, którego wartość jest obliczonym w arytmetyce zmiennopozycyjnej na podstawie tego wzoru wynikiem, przy założeniu, że w obliczeniu nie wystąpił nadmiar ani niedomiar.

   b) Znajdź wyrażenia opisujące liczby $\gamma_1, \ldots, \gamma_n$, takie że $\tilde{x}_i = x_i(1 + \gamma_i)$ dla każdego $i \in \{1, \ldots, n\}$ i otrzymany wynik $\tilde{a}$ jest dokładnym rozwiązaniem zadania z liczbami x_i zastąpionymi przez $\tilde{x}_i$.

   Czy na podstawie tych wyrażeń można stwierdzić, że rozpatrywany algorytm obliczania liczby a jest numerycznie poprawny? Odpowiedź uzasadnij.

   Wskazówka: Dla liczby ε o małej wartości bezwzględnej można przyjąć, że $\sqrt{1+\varepsilon} \approx 1 + \frac{\varepsilon}{2}$.

3. Napisz podprogram, który otrzymuje parametr — tablicę elementów typu `char`. W tablicy jest napis złożony z cyfr dziesiętnych (w kodzie ASCII), zakończony bajtem o wartości 0. Zadaniem podprogramu jest obliczenie przy użyciu schematu Hornera (i przekazanie jako wyniku) liczby całkowitej reprezentowanej przez te cyfry.

   Wskazówka: Kody ASCII cyfr `0`, ..., `9` są kolejnymi liczbami całkowitymi. Liczba a reprezentowana przez cyfry $c_n, \ldots, c_0$ jest wartością wielomianu $w(x) = c_n x^n + \cdots + c_1 x + c_0$ dla $x = 10$.

4. a) Napisz procedurę realizującą zmodyfikowany algorytm QuickSort. Modyfikacja polega na tym, że jeśli długość fragmentu tablicy określonego przez parametry jest mniejsza niż 10, to zamiast rekurencyjnego wywołania procedury `QuickSort` jest wywoływana procedura `InsertionSort`, która realizuje algorytm sortowania przez wstawianie. Zamiast procedur `Partition` i `InsertionSort` napisz tylko ich nagłówki.

   b) Algorytm sortowania przez wybieranie polega na znalezieniu w sortowanym ciągu elementu najmniejszego i przestawienie go na początek tablicy, następnie znalezieniu najmniejszego wśród pozostałych i przestawienie go na drugie miejsce itd.

   Oblicz złożoność optymistyczną i pesymistyczną tego algorytmu, przyjmując porównanie elementów za operację dominującą.

   Ustaw algorytmy sortowania przez wstawianie, sortowania szybkiego (QuickSort) i sortowania przez wybieranie w kolejności rosnących złożoności optymistycznych, a następnie pesymistycznych.

5. Tablica zawiera ciąg liczb $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12$. Należy w tej tablicy zbudować kopiec realizujący kolejkę priorytetową, przy czym każda liczba jest równa swojemu priorytetowi.

   a) Podaj zawartość tablicy po każdym przestawieniu oraz całkowitą liczbę wykonanych przestawień, jeśli do porządkowania kopca została użyta procedura `UpHeap`.

   b) Podaj zawartość tablicy po każdym przestawieniu oraz całkowitą liczbę wykonanych przestawień, jeśli do porządkowania kopca została użyta procedura `DownHeap`.

Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 2 lutego 2010.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Tablica a o długości $n > 0$ zawiera liczby całkowite (typu int).

   a) Mając do dyspozycji podprogram `Partition`, za pomocą którego można zaimplementować algorytm QuickSort, napisz w języku C podprogram rekurencyjny `WybierzKNajmniejszych`, taki aby instrukcja

   ```
   WybierzKNajmniejszych ( a, 0, n-1, k );
   ```

   przy założeniu, że $0 < k \leq n$, przestawiła liczby w tablicy a, umieszczając k najmniejszych liczb na jej początku (liczby te nie muszą być uporządkowane).

   b) Zakładając, że podprogram `Partition`, wywołany z parametrami `a, i, j` wykonuje $j - i$ porównań elementów tablicy a, znajdź złożoność optymistyczną algorytmu realizowanego przez podprogram napisany w punkcie a). Odpowiedź uzasadnij.

2. Za pomocą działań zmiennopozycyjnych należy obliczyć $x = a^3 + a^2b + ab^2 + b^3$. Napisz wyrażenia opisujące obliczone wyniki (z uwzględnieniem błędów zaokrągleń) dla algorytmów realizowanych przez następujące instrukcje, w których wszystkie zmienne są typu float:

   a) `d = b*b;   x = ((a+b)*a+d)*a+d*b;`

   b) `c = a*a;   d = b*b;   x = (c*c-d*d)/(a-b);`

3. Tablica, zawierająca początkowo ciąg liczb $8, 7, 6, 5, 4, 3, 2, 1, 0$, została poddana sortowaniu algorytmem HeapSort.

   a) Opisz ogólny sposób porządkowania kopca i znajdź liczbę wykonanych porównań elementów tablicy o podanej zawartości podczas porządkowania w niej kopca.

   b) Podaj zawartość tablicy po każdym przestawieniu elementów i całkowitą liczbę wykonanych porównań elementów tablicy podczas etapu wyjmowania elementów z kopca.

4. Dane są podprogramy o prototypach

   ```
   void PrzepiszMacierz ( int n, float a[], float b[] );
   void DodajDiag ( int n, float a[], float b );
   void MnozMacierze ( int n, float a[], float b[], float c[] );
   ```

   Tablicowe parametry tych podprogramów reprezentują macierze $n \times n$. `PrzepiszMacierz` kopiuje zawartość tablicy `a` do tablicy `b`, `DodajDiag` dodaje liczbę b do wszystkich współczynników na diagonali macierzy A, reprezentowanej przez tablicę `a`, zaś `MnozMacierze` oblicza współczynniki macierzy $C = AB$, i umieszcza je w tablicy `c`.

   a) Napisz podprogram w C, którego parametrami są liczba k, tablica liczb typu `float`, zawierająca współczynniki $a_0, \ldots, a_{k-1}$ wielomianu $w(x) = x^k + a_{k-1}x^{k-1} + \cdots + a_1x + a_0$, liczba n, tablica `x` ze współczynnikami macierzy X o wymiarach $n \times n$ i tablica `w`. Zadaniem podprogramu jest obliczenie współczynników macierzy $w(X)$ i umieszczenie ich w tablicy `w`. Podprogram ma być implementacją schematu Hornera, korzystającą z opisanych wyżej podprogramów.

   b) Uznając za działania dominujące dodawanie i mnożenie liczb zmiennopozycyjnych i wiedząc, że podprogram `DodajDiag` wykonuje n, a podprogram `MnozMacierze` $2n^3 - n^2$ takich działań, oblicz koszt procedury z punktu a) w zależności od n i k.

5. Algorytm eliminacji Gaussa znajduje rozkład macierzy $n \times n$ na czynniki trójkątne. Jeśli $n = 1$, to koszt tego algorytmu jest równy 0. Dla $n > 1$ należy wykonać $n - 1$ dzieleń oraz $(n-1)^2$ mnożeń i odejmowań zmiennopozycyjnych, a następnie zastosować ten algorytm do macierzy $(n-1) \times (n-1)$, otrzymanej w wyniku tych obliczeń.

   a) Przyjmując, że operacje dominujące to mnożenia i dzielenia zmiennopozycyjne, oblicz złożoność obliczeniową algorytmu eliminacji Gaussa, układając i rozwiązując odpowiednie równanie różnicowe.

   b) Dodatkowym elementem algorytmu eliminacji Gaussa jest tzw. wybór elementu głównego: w kolumnie macierzy $n \times n$ należy znaleźć współczynnik o największej wartości bezwzględnej i przestawić wiersz z tym współczynnikiem z pierwszym wierszem macierzy. Rozwiązując odpowiednie równanie różnicowe, znajdź liczbę wykonanych w trakcie *całej* eliminacji porównań wartości bezwzględnych współczynników.

Egzamin poprawkowy ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 16:00 4 marca 2010.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Tablica `a` o długości $n > 0$ zawiera różne liczby całkowite (typu <u>int</u>).

   Mając do dyspozycji podprogram `Partition`, za pomocą którego można zaimplementować algorytm QuickSort, napisz w języku C podprogram `WybierzZeSrodka`, taki aby instrukcja

   ```
   WybierzZeSrodka ( a, 0, n-1, k, l );
   ```

   przy założeniu, że $0 \leq k \leq l < n$, przestawiła liczby w tablicy `a`, w taki sposób, że wszystkie liczby na miejscach $0, \ldots, k-1$ są mniejsze, a wszystkie liczby na miejscach $l, \ldots, n-1$ są większe od każdej z liczb na pozycjach $k, \ldots, l-1$. Należy, o ile to możliwe, uniknąć sortowania całej tablicy.

2. Napisz procedurę typu <u>int</u>, która ma jeden parametr — tablicę elementów typu <u>char</u>, zawierającą napis ASCIIZ. Napis ten składa się ze znaków `'0'` i `'1'` (tj. kodów ASCII cyfr 0 i 1) i reprezentuje liczbę całkowitą w układzie dwójkowym. Procedura ma obliczyć tę liczbę i zwrócić ją jako wynik.

   Należy zastosować schemat Hornera. <u>Wskazówka:</u> jeśli kolejne cyfry rozwinięcia liczby w układzie pozycyjnym o dowolnej podstawie x, to $a_n, a_{n-1}, \ldots, a_1, a_0$, to liczba ta jest równa $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$.

3. Napisz procedurę w C obliczającą współczynnik dwumianowy Newtona $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ dla podanych jako parametry liczb całkowitych n i k. Do obliczeń wykorzystaj wzór

   $$\binom{n}{0} = 1,$$
   $$\binom{n}{k} = \binom{n}{k-1}\frac{n-k+1}{k} \quad \text{dla } k = 1, \ldots, n.$$

   Podaj koszt algorytmu realizowanego przez napisaną procedurę.

4. Tablica, zawierająca początkowo ciąg liczb $8, 7, 5, 6, 10, 4, 3, 1, 2, 9$, została poddana sortowaniu algorytmem HeapSort.

   Podaj zawartość tablicy po każdym przestawieniu jej elementów i podaj całkowitą liczbę wykonanych podczas sortowania przestawień.

5. Algorytm odbić Householdera znajduje rozkład macierzy $n \times n$ na czynniki, z których pierwszy jest macierzą ortogonalną, a drugi jest macierzą trójkątną (tzw. rozkład QR). Operacją dominującą w tym algorytmie jest mnożenie liczb zmiennopozycyjnych.

   Jeśli $n = 1$, to koszt jest zerowy. Dla $n > 1$ należy skonstruować odbicie, kosztem $n+1$ mnożeń, i zastosować to odbicie do $n-1$ kolumn macierzy, przy czym odbicie każdej kolumny wymaga wykonania $2n+1$ mnożeń. Następnie należy wykonać cały algorytm na podmacierzy $n-1 \times n-1$ macierzy otrzymanej za pomocą tych odbić.

   Napisz i rozwiąż odpowiednie równanie różnicowe, aby znaleźć koszt znajdowania rozkładu QR algorytmem odbić Householdera.

Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 31 stycznia 2011.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. a) Dany jest następujący ciąg liczb:

$$2,\ 11,\ 4,\ 9,\ 6,\ 7,\ 8,\ 5,\ 10,\ 3,\ 12,\ 1.$$

Ciąg ten został posortowany za pomocą algorytmu HeapSort.

Podaj zawartość tablicy po każdym wykonanym przestawieniu elementów oraz liczby wykonanych porównań elementów ciągu w fazie porządkowania kopca i w fazie wyjmowania z niego elementów.

b) Rozwiąż równanie różnicowe

$$a_k = a_{k-1} + a_{k-2} + 1$$

z warunkiem początkowym $a_0 = 0$, $a_1 = 1$.

c) Algorytm QuickSort został użyty do posortowania tablicy o długości a_k, takiej że liczba a_k (dla pewnego $k \in \mathbb{N}$) jest elementem ciągu będącego rozwiązaniem zadania z poprzedniego punktu. Okazało się, że na każdym etapie sortowany fragment tablicy miał długość a_l (dla pewnego $l \in \{1, \ldots, k\}$), i jeśli ta długość była większa od 1, to element dzielący znalazł się na pozycji odległej o a_{l-1} od początku fragmentu.

Napisz równanie różnicowe (i warunek początkowy), którego rozwiązanie opisuje koszt sortowania (mierzony liczbą porównań elementów tablicy).

2. a) Korzystając z definicji struktur danych

```
typedef struct { float x, y, z; } punkt;
typedef struct {
    float xmin, xmax, ymin, ymax, zmin, zmax;
  } kostka;
```

napisz podprogram, który znajduje najmniejszą kostkę (w przestrzeni $\mathbb{R}^3$) o krawędziach równoległych do osi układu współrzędnych, zawierającą n punktów danych w tablicy przekazanej jako parametr.

b) Wskaż operację dominującą w algorytmie realizowanym przez podprogram napisany w poprzednim punkcie i podaj złożoność tego algorytmu.

3. Dane są macierze $\boldsymbol{u}$, $\boldsymbol{v}$ i $\boldsymbol{w}$ o wymiarach $n \times 1$ (gdzie $n > 1$). Należy obliczyć macierz $\boldsymbol{x} = \boldsymbol{u}\boldsymbol{v}^T\boldsymbol{w}$.

a) Przyjmując za operację dominującą mnożenie liczb zmiennopozycyjnych, oblicz koszt algorytmów opartych na wzorach

$$\boldsymbol{x} = (\boldsymbol{u}\boldsymbol{v}^T)\boldsymbol{w} \quad \text{oraz} \quad \boldsymbol{x} = \boldsymbol{u}(\boldsymbol{v}^T\boldsymbol{w}).$$

b) Napisz podprogram w C realizujący tańszy z tych algorytmów. Do obliczania potrzebnych sum liczb rzeczywistych użyj zwykłego algorytmu sumowania „po kolei”.

Dane (w tym liczba n, nieokreślona „z góry”) i wyniki należy przekazywać do i z podprogramu za pomocą parametrów.

c) Dla $n = 3$ napisz wyrażenia opisujące wynik (tj. współczynniki macierzy $\tilde{\boldsymbol{x}}$, będącej przybliżeniem $\boldsymbol{x}$) otrzymany wskutek zastosowania arytmetyki zmiennopozycyjnej, przy założeniu, że w obliczeniach nie wystąpił nadmiar ani niedomiar.

4. Napis ASCIIZ jest ciągiem znaków (elementów typu <u>char</u>) przechowywanym w tablicy, którego koniec jest zaznaczony znakiem o kodzie 0 (znak ten do napisu nie należy).

a) Napisz podprogram w C, którego parametrem jest napis ASCIIZ i który zwraca wartość `-1`, jeśli napis jest pusty lub zawiera nie tylko cyfry, wartość 0, jeśli napis składa się z samych cyfr i jest dziesiętnym zapisem nieujemnej liczby całkowitej niepodzielnej przez 3, wartość 1, jeśli liczba reprezentowana przez ten napis jest podzielna przez 3 i niepodzielna przez 9, albo wartość 2, jeśli liczba ta jest podzielna przez 9.

b) Napisz podprogram w C, którego parametrami są napis ASCIIZ oraz tablica liczb całkowitych o długości 10. Zadaniem podprogramu jest znalezienie liczb wystąpień poszczególnych cyfr dziesiętnych w napisie i umieszczenie tych liczb w tablicy podanej jako drugi parametr (jeśli więc tablica ma nazwę `t`, to `t[0]` ma być liczbą wystąpień znaku `'0'` itd.).

<u>Wskazówka.</u> Do zbadania, czy dana liczba jest kodem ASCII cyfry, można użyć funkcji o prototypie `int isdigit ( int c );`.
Jeśli wartość zmiennej `c` jest kodem ASCII cyfry, to wartość tej cyfry jest równa `c-'0'`.

Egzamin poprawkowy ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 3 marca 2011.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Do początkowo pustej kolejki priorytetowej zrealizowanej za pomocą kopca zostały wstawione (kolejno, po jednej) liczby $2, 4, 6, 8, 10, 12$ (priorytet liczby jest tą liczbą).

   Następnie sześciokrotnie została wyjęta jedna liczba, a po jej wyjęciu wstawiona kolejna liczba nieparzysta (najpierw 1, potem $3, 5, 7, 9, 11$). Podaj zawartość tablicy, w której są przechowywane elementy kopca, po każdym wstawieniu i usunięciu elementu, oraz liczbę wykonanych w całym obliczeniu porównań elementów.

2. Korzystając z definicji struktury

   ```
   typedef struct { float x, y, z, m; } punkt;
   ```

   napisz podprogram, który oblicza środek ciężkości układu n punktów (w przestrzeni $\mathbb{R}^3$), danych w tablicy przekazanej jako parametr. Pola x, y, z struktury zawierają współrzędne kartezjańskie punktu, wartością pola m jest masa punktu. Wynik obliczenia powinien być przekazany za pomocą parametru lub instrukcji return.

3. Korzystając z definicji struktury

   ```
   typedef struct { float re, im; } complex;
   ```

   napisz podprogram, który oblicza wartość wielomianu zespolonego $w(z) = a_n z^n + \cdots + a_1 z + a_0$ za pomocą schematu Hornera. Stopień n, współczynniki $a_0, \ldots, a_n$ i argument z mają być przekazywane za pomocą parametrów, wynik obliczenia ma być przekazany za pomocą instrukcji return.

   Wzory realizujące mnożenie i dodawanie liczb zespolonych można zakodować bezpośrednio w podprogramie, lub użyć osobnych procedur realizujących te działania — w drugim przypadku można tych procedur nie pisać, wystarczy podać ich nagłówki.

4. Opisz działanie algorytmu QuickSort w przypadku, gdy jest on użyty do posortowania ciągu stałego o długości n. W szczególności podaj pozycję, na której procedura `Partition` umieści element dzielący dany fragment tablicy i oblicz koszt algorytmu dla takich danych.

   W jaki sposób można zmodyfikować algorytm QuickSort, aby zmniejszyć rząd złożoności, jeśli ma on być często stosowany do sortowania ciągów, których elementy mogą występować wielokrotnie?

5. Napis ASCIIZ jest ciągiem znaków (elementów typu char) przechowywanym w tablicy, którego koniec jest zaznaczony znakiem o kodzie 0 (znak ten do napisu nie należy).

   Napisz procedurę, która bada, czy w napisie przekazanym jako parametr pewne fragmenty występują wielokrotnie. Jeśli tak, to należy znaleźć pozycję pierwszego znaku pierwszego wystąpienia takiego fragmentu i jego długość, przy czym ma to być najdłuższy taki fragment.

   Na przykład w napisie

   `"Odo do dodo dodaj to do dossier ..."`

   napis siedmioznakowy `"o do do"` występuje dwa razy, zaczynając od pozycji 2 i 19 (litera `'O'` jest na pozycji 0). Dwa wystąpienia mogą mieć niepustą część wspólną (np. fragmenty pięcioznakowe `"do do"` zaczynające się od pozycji 1 i 4).

   Podaj pesymistyczną złożoność obliczeniową algorytmu realizowanego przez napisany podprogram, w zależności od liczby n, która jest długością napisu.

Egzamin ze Wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 15:00 28 stycznia 2016.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Indukowana norma maksimum macierzy rzeczywistej A o wymiarach $m \times n$ jest to funkcja określona wzorem

$$\|A\|_\infty = \max_i \sum_j |a_{ij}|,$$

gdzie a_{ij} oznacza współczynnik macierzy A w i-tym wierszu i j-tej kolumnie. Napisz podprogram w C, który dla macierzy o współczynnikach podanych w tablicy przekazanej jako parametr obliczy wartość tej funkcji (tj. normę macierzy A) i przekaże ją jako wartość zwracaną.

Przyjmij założenia, że wymiary macierzy m i n są podane jako parametry typu `int`; wiersze i kolumny są numerowane odpowiednio od 0 do $m-1$ i od 0 do $n-1$. Współczynniki macierzy są liczbami typu `double` w tablicy *jednowymiarowej*, w której są przechowywane wierszami, tj. pierwsze n elementów to współczynniki pierwszego wiersza, następne n — drugiego itd. Do obliczania wartości bezwzględnej użyj standardowego podprogramu o nagłówku `double fabs ( double x )`.

2. Tablica początkowo zawiera ciąg liczb $5, 1, 3, 2, 7, 6, 4, 9, 8, 0$. Ciąg ten został posortowany rosnąco przy użyciu algorytmu HeapSort. Napisz zawartość tablicy po każdym przestawieniu w niej liczb i podaj całkowitą liczbę porównań wykonanych przez ten algorytm.

3. Tablica początkowo zawiera taki sam ciąg liczb, jak w poprzednim zadaniu. Do jej posortowania rosnąco został użyty algorytm sortowania przez scalanie (MergeSort). Napisz ciągi powstające w wyniku scalania i podaj liczbę wykonanych przez ten algorytm porównań elementów ciągu.

4. Napisz podprogram w C, który dla podanej liczby n generuje wszystkie $n!$ permutacji zbioru n-elementowego.

Algorytm generowania permutacji ma być zrealizowany przez podprogram rekurencyjny, używający tablicy o długości n, w której są elementy zbioru (różne liczby całkowite, np. od 1 do n). Oprócz liczby n i tablicy, przekazanych jako parametry, podprogram otrzymuje trzeci parametr, liczbę $k \in \{1, \dots, n\}$. Jeśli $k = 1$, to należy wywołać procedurę o nagłówku

```
void wyprowadz ( int n, int a[] )
```

(jej zadaniem jest wypisanie permutacji w tablicy `a`). Dla $k > 1$ podprogram ma w pętli, odpowiednio przestawiając elementy, umieścić na miejscu $k-1$ w tablicy kolejno wszystkie elementy tablicy obecne na pozycjach od 0 do $k-1$ i wykonać wywołanie rekurencyjne, przekazując jako parametry liczbę n, tablicę, oraz liczbę $k-1$ (program główny ma wpisać odpowiednie liczby do tablicy i wywołać procedurę z trzecim parametrem równym n — instrukcji, które to robią, ani procedury `wyprowadz` nie piszemy).

5. a) Podaj definicje czasowej złożoności obliczeniowej algorytmu: optymistycznej, pesymistycznej i średniej.

b) Podaj (wymieniając nazwy) trzy algorytmy sortowania przedstawione na wykładzie i podaj (z krótkim uzasadnieniem) rzędy ich złożoności obliczeniowych optymistycznych, pesymistycznych i średnich.

Egzamin poprawkowy ze Wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 15:00 18 lutego 2016.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dana jest tablica o długości n elementów typu int, z których co najmniej jeden jest dodatni. Należy w niej znaleźć fragment (złożony z kolejnych elementów) o największej sumie elementów. Napisz podprogram w C, który to robi; liczba n i tablica mają być przekazane jako parametry. Dodatkowe dwa parametry są wskaźnikami do zmiennych typu int. Podprogram ma tym zmiennym przypisać odpowiednio indeks pierwszego i ostatniego elementu znalezionego fragmentu.

   Podprogram może dokonać tymczasowej zmiany zawartości tablicy, obliczając i zapisując w niej tzw. *sumy prefiksowe*, tj. sumy elementów od początku tablicy do danego miejsca. Mając do dyspozycji ciąg sum prefiksowych można opracować algorytm o mniejszej złożoności czasowej, ale po rozwiązaniu zadania (tj. znalezienia poszukiwanego fragmentu) podprogram ma przywrócić początkową zawartość tablicy.

   Znajdź lub oszacuj pesymistyczną czasową złożoność obliczeniową algorytmu realizowanego przez napisany podprogram.

   Pisząc podprogram przyjmij założenie, że w obliczeniach nie wystąpi nadmiar stałopozycyjny (bo np. suma wartości bezwzględnych wszystkich elementów tablicy nie przekracza `INT_MAX`).

2. Tablica początkowo zawiera ciąg liczb $0, 8, 9, 4, 6, 7, 2, 3, 1, 5$. Ciąg ten został posortowany rosnąco przy użyciu algorytmu HeapSort. Napisz zawartość tablicy po każdym przestawieniu w niej liczb i podaj całkowitą liczbę porównań wykonanych przez ten algorytm.

3. Tablica początkowo zawiera taki sam ciąg liczb, jak w poprzednim zadaniu. Do jej posortowania rosnąco został użyty algorytm QuickSort, w którym procedura `FindPivot` wywołana dla fragmentu sortowanej tablicy od miejsca `i` do `j` zwracała wartość wyrażenia `(i+j)/2`. Wskaż położenia kolejnych elementów dzielących na każdym poziomie podziału tablicy i podaj liczbę wykonanych przez ten algorytm porównań elementów ciągu.

4. Napisz podprogram w C, który dla podanych liczb n i k spełniających warunek $0 < k < n$ generuje wszystkie podciągi (których jest $\binom{n}{k}$) o długości k ciągu $(0, \ldots, n-1)$. Na przykład dla $n = 4$, $k = 2$ to są podciągi $(0, 1)$, $(0, 2)$, $(0, 3)$, $(1, 2)$, $(1, 3)$, $(2, 3)$.

   Algorytm ma być zrealizowany przez podprogram rekurencyjny, wpisujący kolejne podciągi do tablicy o długości k. Liczby n i k oraz poziom wywołania rekurencyjnego (początkowo 1) i tablica mają być przekazywane jako parametry. Na poziomie p podprogram ma w pętli wpisywać p-ty element podciągu do tablicy i wykonywać wywołanie rekurencyjne, albo, jeśli $p = k$, wywoływać procedurę o nagłówku

   ```
   void wyprowadz ( int k, int a[] )
   ```

   (jej zadaniem jest wypisanie podciągu w tablicy `a`).

5. Opisz powody (na wykładzie podanych było pięć), dla których programy zapisuje się w postaci wielu podprogramów, zamiast jednego.

Egzamin ze Wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 15:00 30 stycznia 2017.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Norma indukowana maksimum macierzy rzeczywistej A o wymiarach $m \times n$ jest to funkcja określona wzorem

$$\|A\|_\infty = \max_i \sum_j |a_{ij}|.$$

Macierz rzadka jest to taka macierz, która ma niewiele (np. rzędu $m + n$) współczynników różnych od zera. Taką macierz można reprezentować za pomocą tablicy o długości N struktur typu

```
typedef struct {
          int i, j;
          double a;
        } wspolczynnik;
```

gdzie N jest liczbą niezerowych współczynników, a każdy element tablicy reprezentuje jeden z nich. Pola `i` oraz `j` elementu określają pozycję (numery wiersza i kolumny) współczynnika macierzy, zaś w polu `a` jest przechowywana wartość współczynnika.

Zakładając, że wiersze i kolumny macierzy są numerowane odpowiednio od 0 do $m-1$ i od 0 do $n-1$ i tablica nie jest uporządkowana, napisz podprogram w C, który obliczy normę indukowaną macierzy reprezentowanej w tej postaci. Parametry procedury opisują wymiary macierzy, liczbę niezerowych współczynników i tablicę struktur opisanych wyżej (oraz dodatkową tablicę roboczą zmienych typu `double` o długości m); obliczona norma ma być zwracana jako wartość funkcji realizowanej przez napisany podprogram.

2. Tablica początkowo zawiera ciąg liczb $5, 1, 3, 2, 7, 6, 4, 9, 8, 0, 4, 7$. Ciąg ten został posortowany niemalejąco przy użyciu algorytmu HeapSort. Napisz zawartość tablicy po każdym przestawieniu w niej liczb i podaj całkowitą liczbę porównań wykonanych przez ten algorytm.

3. Tablica początkowo zawiera taki sam ciąg liczb, jak w poprzednim zadaniu. Do jej posortowania rosnąco został użyty algorytm sortowania przez scalanie (MergeSort). Wypisz kolejne ciągi powstałe w wyniku scalania i podaj liczbę wykonanych przez ten algorytm porównań elementów ciągu.

4. W tablicy o długości N znajdują się liczby całkowite. Napisz podprogram w C, który znajdzie w ciągu liczb danym w tej tablicy najdłuższe podciągi malejące, składające się z kolejnych elementów ciągu. Dla każdego takiego podciągu należy wypisać (przy użyciu procedury `printf`) dwie liczby — indeks pierwszego i ostatniego elementu podciągu.

5. Podaj definicję operacji dominującej w algorytmie, a także definicje czasowych złożoności obliczeniowych algorytmu: optymistycznej, pesymistycznej i średniej. Dla dowolnego algorytmu omawianego na wykładzie podaj, z krótkim uzasadnieniem, rząd każdej z tych złożoności.

Egzamin poprawkowy ze Wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 15:00 24 lutego 2017.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Macierz rzadka jest to taka macierz, która ma niewiele (np. rzędu $m + n$) współczynników różnych od zera. Taką macierz można reprezentować za pomocą tablicy o długości N struktur typu

   ```
   typedef struct {
             int i, j;
             double a;
           } wspolczynnik;
   ```

   gdzie N jest liczbą niezerowych współczynników, a każdy element tablicy reprezentuje jeden z nich. Pola `i` oraz `j` elementu określają pozycję (numery wiersza i kolumny) współczynnika macierzy, zaś w polu a jest przechowywana wartość współczynnika.

   Zakładając, że tablica a opisanych wyżej struktur jest uporządkowana w ten sposób, że jeśli `i < j`, to

   `a[i].i < a[j].i` albo `a[i].i == a[j].i` oraz `a[i].j < a[j].j`

   i podobnie uporządkowana jest tablica b reprezentująca drugą macierz, napisz podprogram w C, który obliczy sumę macierzy reprezentowanej w tej postaci. Parametry procedury opisują liczby niezerowych współczynników i tablice struktur opisanych wyżej. Wynik ma być umieszczony w trzeciej tablicy takich struktur (zakładamy, że tablica ta ma wystarczającą długość), a wartość zwracana przez podprogram ma być liczbą niezerowych współczynników sumy macierzy.

2. Tablica początkowo zawiera ciąg liczb $7, 4, 0, 8, 9, 4, 6, 7, 2, 3, 1, 5$. Ciąg ten został posortowany niemalejąco przy użyciu algorytmu HeapSort. Napisz zawartość tablicy po każdym przestawieniu w niej liczb i podaj całkowitą liczbę porównań wykonanych przez ten algorytm.

3. Napisz podprogram w C, który sprawdza czy dany ciąg liczb całkowitych o długości $n + 1$ jest ciągiem współczynników wielomianu $(2x + a)^n$ dla pewnej liczby całkowitej a i jeśli tak, to przekazuje tę liczbę przy użyciu odpowiedniego parametru.

4. W tablicy a o długości N znajdują się liczby rzeczywiste (typu `float`), różne od zera. Napisz podprogram w C, który znajdzie liczbę zmian znaku liczb w tym ciągu, tj. liczbę indeksów `i`, takich że

   `a[i] < 0 < a[i+1]` albo `a[i] > 0 > a[i+1]`.

5. Podaj definicję operacji dominującej w algorytmie, a także definicje czasowych złożoności obliczeniowych algorytmu: optymistycznej, pesymistycznej i średniej. Dla dowolnego algorytmu omawianego na wykładzie podaj, z krótkim uzasadnieniem, rząd każdej z tych złożoności.

Egzamin ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 2 lutego 2018.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Prostokątna tablica elementów typu `int` ma M wierszy i N kolumn. Należy znaleźć w tej tablicy prostokąt (tj. przecięcie sąsiednich kolumn i wierszy), w którym suma elementów jest największa. Napisz podprogram w C, który to robi; wymiary tablicy są określone z góry (za pomocą `#define`). Tablica ma być przekazana jako parametr, pozostałe parametry służą do przekazania wyników, którymi są indeksy pierwszego wiersza i kolumny, a także wymiary prostokąta i suma elementów w nim.

   Pisząc podprogram, przyjmij założenie, że w obliczeniach nie wystąpi nadmiar stałopozycyjny (bo np. suma wartości bezwzględnych wszystkich elementów tablicy nie przekracza `INT_MAX`).

2. Dany ciąg liczb $7, 4, 0, 8, 9, 4, 6, 7, 2, 3, 1, 5$ ma być posortowany przy użyciu algorytmu sortowania przez scalanie (MergeSort). Wypisz podciągi otrzymane w poszczególnych operacjach scalania i podaj liczbę operacji porównywania elementów sortowanego ciągu wykonanych przez ten algorytm.

3. Napisz podprogramy (funkcje) w C, które dla danej liczby całkowitej bez znaku znajdują liczby jedynek w jej rozwinięciu

   a) dwójkowym,

   b) trójkowym.

   Znaleziona liczba ma być przekazana jako wartość funkcji.

4. Napisz podprogram, którego parametrami są: liczba $n > 1$ i trzy tablice liczb całkowitych o długości n. W pierwszych dwóch tablicach są dane ciągi liczb otrzymane przez zastosowanie pewnych permutacji σ_1 i σ_2 do ciągu $0, 1, \ldots, n-1$. Podprogram ma wpisać do trzeciej tablicy liczby $0, 1, \ldots, n-1$ w kolejności reprezentującej złożenie $\sigma_1 \circ \sigma_2$.

5. Podaj definicję operacji dominującej w algorytmie, a także definicje czasowych złożoności obliczeniowych algorytmu: optymistycznej, pesymistycznej i średniej.

   Napisz równania różnicowe z odpowiednimi warunkami początkowymi opisujące te trzy złożoności algorytmu sortowania przez wstawianie (InsertionSort) i rozwiąż jedno z tych równań, aby znaleźć jawny wzór opisujący złożoność.

   Dla złożoności średniej przyjmij założenie, że każda z $n!$ możliwych permutacji elementów sortowanego zbioru jest jednakowo prawdopodobna.

Egzamin ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10:00 23 lutego 2018.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dane są dwie tablice liczb całkowitych o długościach m i n. W żadnej z tablic liczby w niej obecne nie powtarzają się. Napisz podprogram w C, który wyszukuje i zlicza liczby obecne w obu tablicach oraz podaje (jako wartość funkcji) liczbę takich wspólnych elementów, przy założeniu, że oba ciągi w tablicach są posortowane rosnąco.

   Algorytm zapisany w tym podprogramie powinien mieć złożoność rzędu $m + 1$.

2. Napisz proprogram w C, który ma dwa parametry: liczbę typu `float`, będącą miarą kąta, oraz (dostatecznie długą) tablicę elementów typu `char`. Zadaniem podprogramu jest utworzenie w tej tablicy napisu, który wyraża miarę kąta, podaną w stopniach, minutach i sekundach, np. `87.27'15"`.

   Wskazówka. Możesz użyć procedury `sprintf`, której pierwszy parametr jest tablicą elementów typu `char`, a kolejne parametry są takie jak parametry procedury `fprintf`.

3. Ciąg $10, 5, 8, 9, 2, 3, 7, 4, 1, 6$ należy posortować za pomocą algorytmu QuickSort. Zakładając, że element dzielący jest brany z początku fragmentu tablicy przetwarzanego przez każde wywołanie procedury `Partition`, wypisz zawartości tablicy po każdym przestawieniu i podaj całkowitą liczbę porównań elementów sortowanego ciągu.

4. Podaj definicję operacji dominującej w algorytmie, a także definicje czasowych złożoności obliczeniowych algorytmu: optymistycznej, pesymistycznej i średniej.

   Napisz równania różnicowe z odpowiednimi warunkami początkowymi opisujące te złożoności pesymistyczna i optymistyczną algorytmu QuickSort i rozwiąż jedno z tych równań, aby znaleźć jawny wzór opisujący złożoność.

5. Napisz podprogram w C, który dla danej liczby całkowitej n znajduje liczbę rzeczywistą x, która jest ułamkiem otrzymanym przez odwrócenie cyfr zapisu dziesiętnego liczby n. Na przykład dla $n = 2018$ ma być $x = 0.8102$.

Egzamin ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10:00 28 stycznia 2019.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Wskaż błędy w podanym niżej podprogramie w C, którego zadaniem jest znalezienie liczby najmniejszej i największej w tablicy o długości N zawierającej liczby całkowite i wyjaśnij, na czym te błędy polegają.

   Przepisz ten podprogram, poprawiając błędy.

```
void FindMinMax ( int a[], N, min, max );
{
  float i;

  min = max == a[0];
  for ( i = 1; i <= N; i++ )
    if ( a[i] < min ) min = a[i]
    else if ( a[i] > max ) max = a[i];
  }
  return min, max;
} /*FindMinMax*/
```

2. Podaj definicję operacji dominującej w algorytmie. Wskaż operacje dominujące w poprawionym podprogramie będącym rozwiązaniem poprzedniego zadania, podając uzasadnienie dla każdej ze wskazanych operacji.

   Uwaga: Ze względu na to, że rozwiązania każdego zadania ma sprawdzać inna osoba, przepisz ten podprogram na kartce z rozwiązaniem tego zadania.

3. Napisz w języku C podprogram `FindPivot`, którego parametrami są tablica z ciągiem struktur typu `element`, zawierających liczbowe pola `klucz`, oraz liczby `i`, `j` wyznaczające podciąg (tj. fragment tablicy) do przetworzenia przez procedurę `Partition` algorytmu QuickSort. Podprogram `FindPivot` ma spośród trzech elementów tablicy na pozycjach `i`, `j` oraz `k = (i+j)/2` wybrać element, którego klucz jest medianą (tj. leży pomiędzy pozostałymi dwoma kluczami) i przekazać pozycję (indeks) tego elementu.

4. a) Rozwiąż równanie różnicowe $N_k = N_{k-1} + N_{k-2} + 1$ z warunkiem początkowym $N_0 = 0$, $N_1 = 1$.

   b) Ciąg o długości N_k (dla pewnego $k \in \mathbb{N}$) został posortowany algorytmem

QuickSort, przy czym okazało się, że za każdym razem procedura `Partition` była wywołana dla podciągu o długości N_l (gdzie $l \in \{2, \ldots, k\}$) i wskutek działania tej procedury element dzielący znalazł się N_{l-2} pozycje za początkiem podciągu (a więc dalej trzeba było sortować rekurencyjnie podciągi o długościach N_{l-2} i N_{l-1}). Napisz równanie różnicowe z odpowiednim warunkiem początkowym, którego rozwiązanie opisuje koszt (liczbę porównań kluczy wykonanych przez algorytm podczas) sortowania takiego ciągu dla każdego k.

5. Do początkowo pustej kolejki priorytetowej zaimplementowanej w postaci kopca (obsługiwanego przez opisane na wykładzie procedury `UpHeap` i `DownHeap`) zostały wstawione kolejne liczby naturalne od 1 do 16, przy czym po wstawieniu każdej nowej pary liczb jedna liczba (o największym priorytecie, czyli największa) została usunięta z kolejki. Podaj zawartość kopca przed i po każdym usunięciu liczby.

Egzamin poprawkowy ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 21 lutego 2019.)

Proszę bardzo uważnie przeczytać treść zadań. Należy rozwiązać cztery zadania z podanych pięciu. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Palindrom jest to napis identyczny z napisem powstałym po odwróceniu kolejności znaków.

   a) Napisz w języku C procedurę `Palindrom`, która ma dwa parametry: liczbę całkowitą n i tablicę elementów typu `char`, zawierającą napis o długości n. Procedura ma zwrócić wartość różną od zera wtedy i tylko wtedy, gdy podany napis jest palindromem.

   b) Napisz procedurę `ZnajdzPalindromy`, której parametry reprezentują pewien napis jak wyżej. Procedura ta ma znaleźć w podanym napisie wszystkie palindromy o długości większej niż 2. Na przykład w napisie `BACABADACELE` należy znaleźć `BACAB`, `ACA`, `ABA`, `ADA` i `ELE`.

   Dla każdego znalezionego palindromu należy wywołać procedurę o nazwie `Wypisz`, przekazując jej jako parametry długość palindromu i wskaźnik jego pierwszego znaku.

2. Jeśli sprawdzenie, czy dany napis o długości n jest palindromem, zajmuje co najmniej jedno i co najwyżej $\lfloor \frac{n}{2} \rfloor$ operacji porównywania znaków, a procedura wyszukiwania palindromów sprawdza każdy fragment napisu o długości $k \in \{3, \ldots, n\}$, to jakie są (mierzone liczbą porównań) złożoności pesymistyczna i optymistyczna tej procedury w zależności od n? Wyprowadź dokładne wzory i podaj rzędy tych złożoności.

3. Do początkowo pustej kolejki priorytetowej zaimplementowanej w postaci kopca (obsługiwanego przez opisane na wykładzie procedury `UpHeap` i `DownHeap`) zostały wstawione kolejne liczby naturalne od 1 do 15, przy czym po wstawieniu każdej nowej trójki liczb jedna liczba (o największym priorytecie, czyli największa) została usunięta z kolejki. Podaj zawartość kopca przed i po każdym usunięciu liczby.

4. Przypuśćmy, że zadanie obliczenia sumy n liczb danych w tablicy należy rozwiązać przy użyciu algorytmu sumowania parami.

a) Napisz w języku C procedurę rekurencyjną, realizującą ten algorytm (określony za pomocą parametrów fragment tablicy dłuższy niż 1 należy podzielić na dwie części i obliczyć i dodać sumy liczb w tych częściach).

b) Uzasadnij, że algorytm korzystający z kolejki:

```
InitQueue ();
for ( i = 0; i < n; i++ ) Enqueue ( a[i] );
for ( i = 0; i < n-1; i++ ) {
  Dequeue ( &b1 );  Dequeue ( &b2 );
  Enqueue ( b1 + b2 );
}
Dequeue ( &s );
```

realizuje algorytm sumowania parami, z dokładnością do kolejności składników.

5. a) Rozwiąż równanie różnicowe $N_k = N_{k-1} + N_{k-2} + 1$ z warunkiem początkowym $N_0 = 0$, $N_1 = 1$.

b) Ciąg o długości N_k (dla pewnego $k \in \mathbb{N}$) został posortowany algorytmem QuickSort, przy czym okazało się, że za każdym razem procedura `Partition` była wywołana dla podciągu o długości N_l (gdzie $l \in \{2, \dots, k\}$) i wskutek działania tej procedury element dzielący znalazł się N_{l-2} pozycje za początkiem podciągu (a więc dalej trzeba było sortować rekurencyjnie podciągi o długościach N_{l-2} i N_{l-1}). Napisz równanie różnicowe z odpowiednim warunkiem początkowym, którego rozwiązanie opisuje koszt (liczbę porównań kluczy wykonanych przez algorytm podczas) sortowania takiego ciągu dla każdego k.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14 : 15 24 kwietnia 2006.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

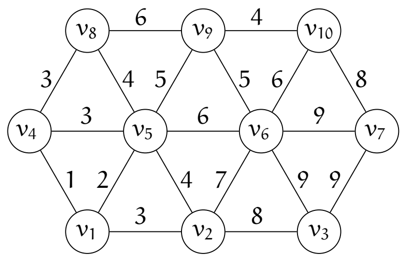
1. Do początkowo pustego drzewa AVL zostały wstawione kolejno wierzchołki z kluczami $1, 9, 2, 8, 3, 7, 4, 6, 5$. Narysuj obrazy drzewa przed wykonaniem każdej rotacji oraz zaznacz wyraźnie na każdym obrazie, lub napisz obok, w którym wierzchołku i w którą stronę ta rotacja ma być wykonana.

2. Długi tekst składa się z liter „a”, …, „h”, przy czym prawdopodobieństwa ich wystąpienia są podane w tabelce:

a	b	c	d	e	f	g	h
0,45	0,2	0,1	0,09	0,08	0,05	0,02	0,01

Znajdź kod binarny optymalny do zakodowania tego tekstu.
Uzasadnij, dlaczego jest on optymalny.

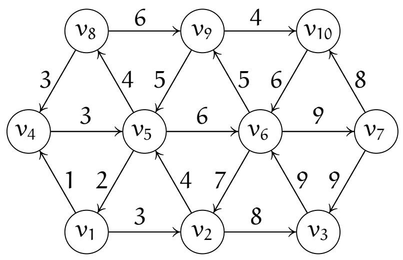
3. Dany jest graf, którego krawędzie mają przyporządkowane wagi, przedstawione na rysunku.



Narysuj minimalne drzewo rozpinające ten graf i podaj kolejność, w jakiej algorytm Kruskala znajdzie krawędzie tego drzewa.

Czy istnieje tylko jedno minimalne drzewo rozpinające ten graf? Odpowiedź uzasadnij.

4. Dany jest graf skierowany, którego krawędzie mają przyporządkowane wagi, przedstawione na rysunku.



Narysuj drzewo składające się z najkrótszych ścieżek prowadzących z wierzchołka v_5 do pozostałych wierzchołków tego grafu skierowanego. Podaj kolejność, w jakiej algorytm Dijkstry znajdzie krawędzie tego drzewa.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14 : 15 23 kwietnia 2007.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

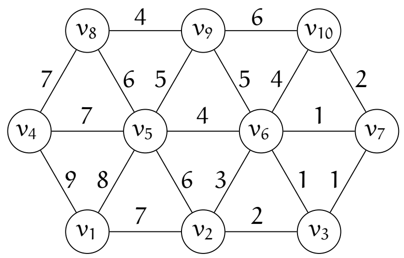
1. Do początkowo pustego drzewa AVL zostały wstawione kolejno wierzchołki z kluczami $5, 6, 4, 7, 3, 8, 2, 9, 1, 10$. Narysuj obrazy drzewa przed wykonaniem każdej rotacji oraz zaznacz wyraźnie na każdym obrazie, lub napisz obok, w którym wierzchołku i w którą stronę ta rotacja ma być wykonana.

2. Długi tekst składa się z liter „a", ..., „h", przy czym prawdopodobieństwa ich wystąpienia są podane w tabelce:

a	b	c	d	e	f	g	h
0,48	0,15	0,12	0,1	0,09	0,03	0,02	0,01

Znajdź kod binarny optymalny do zakodowania tego tekstu.
Uzasadnij, dlaczego jest on optymalny.

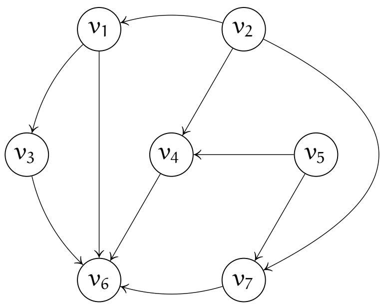
3. Dany jest graf, którego krawędzie mają przyporządkowane wagi, przedstawione na rysunku.



Narysuj minimalne drzewo rozpinające ten graf i podaj kolejność, w jakiej algorytm Prima znajdzie krawędzie tego drzewa.

Czy istnieje tylko jedno minimalne drzewo rozpinające ten graf? Odpowiedź uzasadnij.

4. Dany jest graf skierowany G, przedstawiony na rysunku.



a) Znajdź kolejność, w jakiej algorytm DFS nada wierzchołkom tego grafu kolory szary i czarny (tj. wartości atrybutów b i f wierzchołków).

b) Dla każdej krawędzi grafu G określ, jak ją algorytm DFS zaklasyfikuje (drzewowa, powrotna, skierowana w przód, boczna).

c) Znajdź składowe silnie spójne i narysuj graf zredukowany grafu G.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 8:30 21 kwietnia 2008.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Napisz procedurę, której parametrem jest wskaźnik początku listy złożonej z elementów typu opisanego w taki sposób:

```
typedef struct lista {
          int klucz, tresc;
          struct lista *nast;
        } lista;
```

Zadaniem procedury jest utworzenie drugiej listy, zawierającej kopie tych elementów, których klucze są liczbami parzystymi. Elementy tej listy należy tworzyć przy użyciu procedury `malloc`. Wskaźnik początku utworzonej listy ma być zwrócony jako wartość procedury.

Kolejność elementów w utworzonej liście może być zgodna z ich uporządkowaniem w liście danej, lub odwrotna; napisz, jaka jest ta kolejność w napisanej przez siebie procedurze.

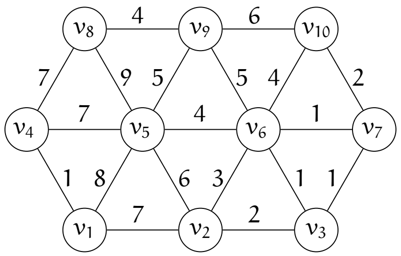
2. Do początkowo pustego drzewa AVL zostały wstawione kolejno wierzchołki z kluczami $10, 1, 9, 2, 8, 3, 7, 4, 5, 6$. Narysuj obrazy drzewa przed wykonaniem każdej rotacji oraz zaznacz wyraźnie na każdym obrazie, lub napisz obok, w którym wierzchołku i w którą stronę ta rotacja ma być wykonana.

3. Długi tekst składa się z liter „a”, ..., „i”, przy czym prawdopodobieństwa ich wystąpienia są podane w tabelce:

a	b	c	d	e	f	g	h	i
0,42	0,2	0,12	0,1	0,06	0,04	0,03	0,02	0,01

Znajdź kod binarny optymalny do zakodowania tego tekstu.
Zakoduj przy jego użyciu tekst „`cichababababadaecha`”.

4. Dany jest graf, którego krawędzie mają przyporządkowane wagi, przedstawione na rysunku.



Narysuj minimalne drzewo rozpinające ten graf i podaj kolejność, w jakiej algorytm Kruskala znajdzie krawędzie tego drzewa. Jeśli kolejność nie jest jednoznacznie określona przez dany graf, to należy podać jedną z poprawnych możliwości.

Czy istnieje tylko jedno minimalne drzewo rozpinające ten graf? Odpowiedź uzasadnij.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10:15 20 kwietnia 2009.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Napisz procedurę, której parametrem jest wskaźnik pierwszego elementu listy złożonej ze zmiennych typu opisanego w taki sposób:

```
typedef struct lista {
          int klucz;
          struct lista *nast;
        } lista;
```

   Zadaniem procedury jest znalezienie i zwrócenie jako wyniku liczby par kolejnych elementów tej listy, takich że klucze elementów należących do pary różnią się o liczbę parzystą (uwaga: każdy element listy oprócz pierwszego i ostatniego należy do dwóch par, które trzeba zbadać).

2. Do początkowo pustego drzewa AVL zostały wstawione kolejno wierzchołki z kluczami $1, 2, 3, 4, 5, 10, 9, 8, 7, 6$. Narysuj obrazy drzewa przed wykonaniem każdej rotacji oraz zaznacz wyraźnie na każdym obrazie, lub napisz obok, w którym wierzchołku i w którą stronę ta rotacja ma być wykonana. Narysuj także drzewo otrzymane na końcu.

3. Pewien tekst składa się z liter „a”, ..., „j”, przy czym litery „a” i „b” występują po jednym razie, a liczba wystąpień każdej następnej litery alfabetu jest sumą liczb wystąpień dwóch poprzednich liter (np. „c” występuje dwa razy, a „d” trzy razy).

   Znajdź kod binarny optymalny do zakodowania tego tekstu.

   Co można powiedzieć o optymalnym kodzie utworzonym dla alfabetu składającego się z dowolnej liczby n znaków, jeśli liczby wystąpień poszczególnych liter w tekście spełniają analogiczną regułę?
   Odpowiedź uzasadnij.

4. Drzewa binarne w programie są zbudowane ze zmiennych typu podanego niżej:

```
typedef struct drzewo {
          int klucz;
          struct drzewo *lewe, *prawe;
        } drzewo, *pdrzewo;
```

Mając do dyspozycji procedury obsługi stosu i kolejki, mogących przechowywać wskaźniki wierzchołków drzewa, o nagłówkach

```
void InitStack ( void );
void Push ( pdrzewo el );
void Pop ( pdrzewo *el );
char StackEmpty ( void );
void InitQueue ( void );
void Enqueue ( pdrzewo el );
void Dequeue ( pdrzewo *el );
char QueueEmpty ( void );
```

oraz procedurę o nagłówku

```
void Wyprowadz ( pdrzewo el );
```

napisz procedurę, która otrzymuje parametr będący wskaźnikiem do korzenia drzewa i której zadaniem jest wywołanie procedury `Wyprowadz` kolejno dla wszystkich wierzchołków drzewa (wskaźnik wierzchołka należy przekazać jako parametr), przy czym pierwszy ma być wskaźnik korzenia, następnie wskaźniki korzeni jego poddrzew itd. — wierzchołki drzewa położone bliżej korzenia mają być wyprowadzone wcześniej od bardziej oddalonych.

Napisz, jaki ogólny algorytm został użyty do wyprowadzenia wierzchołków drzewa w tej kolejności.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10:15 26 kwietnia 2010.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Program w C zawiera deklarację

```
typedef struct lista {
          int klucz;
          struct lista *poprz, *nast;
        } lista;
```

Ze zmiennych typu `lista` została utworzona zamknięta lista jednokierunkowa, w której pole `nast` każdego elementu wskazuje element następny w liście. Pole `poprz` każdego elementu w liście ma nieokreśloną wartość.

Napisz w języku C podprogram z jednym parametrem wskaźnikowym, którego wartością jest adres dowolnego elementu takiej listy lub `NULL`. Zadaniem podprogramu jest nadanie takich wartości polom `poprz` wszystkich elementów listy, aby pole `poprz` każdego elementu wskazywało element poprzedni (zatem ma powstać zamknięta lista dwukierunkowa). Podprogram ma ponadto zwrócić wartość będącą liczbą elementów listy.

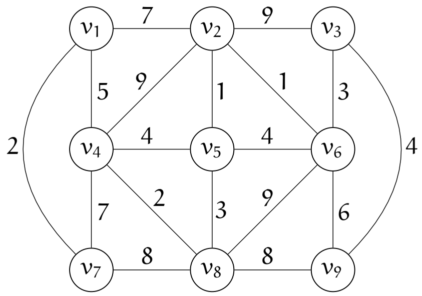
2. Do początkowo pustego drzewa AVL zostały wstawione kolejno wierzchołki z kluczami $1, 2, 3, 4, 10, 9, 8, 5, 6, 7$. Narysuj obrazy drzewa przed wykonaniem każdej rotacji oraz zaznacz wyraźnie na każdym obrazie, lub napisz obok, w którym wierzchołku i w którą stronę ta rotacja ma być wykonana. Narysuj także drzewo otrzymane na końcu.

3. Znajdź kod binarny optymalny do zakodowania tekstu

   `ABRAKADABRAALIBABIESIĘBABRA`

   i użyj tego kodu do zakodowania tego tekstu.

4. Dany jest graf, którego krawędzie mają przyporządkowane wagi przedstawione na rysunku.



Narysuj minimalne drzewo rozpinające ten graf i podaj kolejność, w jakiej algorytm Prima znajdzie krawędzie tego drzewa. Jeśli kolejność nie jest jednoznacznie określona przez dany graf, to należy podać jedną z poprawnych możliwości.

Czy istnieje tylko jedno minimalne drzewo rozpinające ten graf? Odpowiedź uzasadnij.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 8:30 18 kwietnia 2011.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Napisz podprogram w C, którego parametrem jest wskaźnik do korzenia drzewa binarnych wyszukiwań. Procedura ma obliczyć (i przekazać jako wartość) liczbę wierzchołków drzewa, których odległość od korzenia jest podana jako drugi parametr.

   Tekst procedury poprzedź odpowiednią definicją typu strukturalnego opisującego wierzchołek drzewa. Typy klucza i danych przechowywanych w wierzchołkach mogą być dowolne.

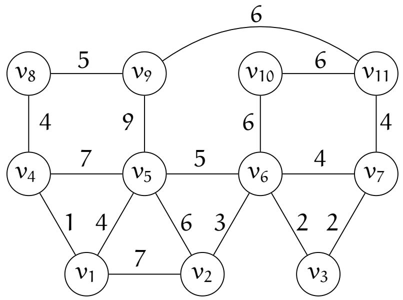
2. Do początkowo pustego drzewa AVL zostały wstawione kolejno wierzchołki z kluczami $7, 6, 8, 5, 9, 4, 10, 3, 2, 1$. Narysuj obrazy drzewa przed wykonaniem każdej rotacji oraz zaznacz wyraźnie na każdym obrazie, lub napisz obok, w którym wierzchołku i w którą stronę ta rotacja ma być wykonana.

3. Długi tekst składa się z liter „a”, ..., „j”, przy czym prawdopodobieństwa ich wystąpienia są podane w tabelce:

a	b	c	d	e	f	g	h	i	j
0,37	0,17	0,12	0,1	0,09	0,08	0,03	0,02	0,01	0.01

   Znajdź kod binarny optymalny do zakodowania tego tekstu.
   Oblicz współczynnik kompresji, tj. iloraz liczby bitów tekstu zakodowanego do liczby bitów reprezentujących tekst oryginalny, jeśli znaki są dane w tablicy elementów typu <u>char</u>, w kodzie ASCII.

4. Dany jest graf, którego krawędzie mają przyporządkowane wagi, przedstawione na rysunku.



Narysuj minimalne drzewo rozpinające ten graf i podaj kolejność, w jakiej algorytm Prima znajdzie krawędzie tego drzewa. Jeśli kolejność nie jest jednoznacznie określona przez dany graf, to należy podać jedną (dowolną) z poprawnych możliwości.

Czy istnieje tylko jedno minimalne drzewo rozpinające ten graf? Odpowiedź uzasadnij.

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 8:30 8 maja 2017.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dana jest następująca deklaracja typu:

```
typedef struct ElemListy {
    int wartosc;
    struct ElemListy *nast; /* następny */
  } ElemListy;
```

Napisz funkcję o nagłówku

```
ElemListy* ListaRoznic ( ElemListy* pocz )
```

która tworzy nową listę krótszą o 1 element, zawierającą różnice między kolejnymi elementami listy podanej jako parametr i zwraca jako wartość wskaźnik początku tej nowej listy. Na przykład, jeśli lista wejściowa zawiera liczby 1, 3, 2, 6, 11, to lista wyściowa ma zawierać elementy −2, 1, −4, −5.

2. Opisz, jak wygląda drzewo AVL o wysokości $h > 0$, w którym lewe poddrzewo ma największą możliwą liczbę wierzchołków, a prawe poddrzewo najmniejszą. Znajdź maksymalną różnicę między liczbami wierzchołków lewego i prawego poddrzewa korzenia takiego drzewa w funkcji wysokości h.

3. Do początkowo pustego drzewa AVL zostały wstawione wierzchołki z kluczami — liczbami całkowitymi — w następującej kolejności: $1, 10, 3, 8, 5, 6, 7, 4, 9, 2$. Narysuj obraz drzewa przed każdą rotacją wykonaną podczas tych wstawień i zaznacz, w którym wierzchołku i w którą stronę rotacja została wykonana.

4. Znajdź kod optymalny do zakodowania fragmentu wiersza St. Barańczaka:

```
Reputację Papuy
Psuły póły bibuły
Puchła pod chałup pułap
Pochłaniana co tchu, łap-
Czywie przez Papuasa
Półnielegalna prasa:
```

Pominń spacje i znaki interpunkcyjne i zaniedbaj różnicę między wielkimi i małymi literami. Podaj długość (liczbę bitów) zakodowanego tekstu.

Wskazówka. W tekście występują tylko następujące litery:

`A B C D E Ę G H I J L Ł N O Ó P R S T U W Y Z`

Kolokwium poprawkowe ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 16:15 31 maja 2017.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dana jest następująca deklaracja typu:

```
typedef struct vertex {
    int klucz;
    struct vertex *lewe, *prawe;
  } vertex;
```

Ze struktur tego typu jest zbudowane drzewo binarne. Napisz podprogram w C, który otrzymuje jako parametr wskaźnik korzenia takiego drzewa i który sprawdza, czy jest to poprawne drzewo wyszukiwań binarnych (a zatem, czy dla każdego wierzchołka jego poddrzewo lewe nie zawiera klucza większego, a poddrzewo prawe klucza mniejszego od klucza w tym wierzchołku).

Czasowa złożoność pesymistyczna algorytmu realizowanego przez ten podprogram ma być rzędu liczby wierzchołków w drzewie.

2. Ze struktur typu

```
typedef struct elem {
    int x;
    struct elem *nast;
  } elem;
```

jest zbudowana lista cykliczna. Napisz procedurę, której parametrem jest wskaźnik jednego (dowolnego) elementu takiej listy i której zadaniem jest przebudowanie listy w taki sposób, aby odwrócić kolejność elementów.

3. Do początkowo pustego drzewa AVL zostały wstawione wierzchołki z kluczami — liczbami całkowitymi — w następującej kolejności: $10, 1, 4, 3, 8, 5, 6, 7, 2, 9$. Narysuj obraz drzewa przed każdą rotacją wykonaną podczas tych wstawień i zaznacz, w którym wierzchołku i w którą stronę rotacja została wykonana.

4. Znajdź kod optymalny do zakodowania fragmentu wiersza St. Barańczaka:

```
Tu kaczka i tam kaczka.
Czy ta kaczka, czy ta czka?
Nie: acz klacz czka i pstra czka
Paczka perliczek — kaczka
Baczy raczej na kacze
Obyczaje i kwacze.
```

Pomiń spacje i znaki interpunkcyjne i zaniedbaj różnicę między wielkimi i małymi literami. Podaj długość (liczbę bitów) zakodowanego tekstu.

Wskazówka. W tekście występują tylko następujące litery:

```
A B C E I J K L M N O P R S T U W Y Z
```

Kolokwium ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 8:30 23 kwietnia 2018.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dana jest następująca deklaracja typu:

```
typedef struct ElemListy {
    int wartosc;
    struct ElemListy *nast; /* następny */
  } ElemListy;
```

Napisz podprogram o nagłówku

```
char TaSama ( ElemListy *l1, ElemListy *l2 )
```

której parametry mają wartość `NULL` albo wskazują pewne elementy list cyklicznych (tj. zamkniętych, pole `nast` ostatniego elementu wskazuje pierwszy element). Wartość funkcji `TaSama` ma być równa `1`, jeśli oba parametry są wskaźnikami pustymi lub gdy oba wskazywane przez nie elementy należą do tej samej listy, oraz `0` w przeciwnym razie.

2. Drzewo binarnych wyszukiwań jest zbudowane ze zmiennych typu

```
typedef struct vertex {
    int klucz;
    struct vertex *lewe, *prawe;
  } vertex;
```

Napisz podprogram, którego pierwszy parametr jest wskaźnikiem korzenia drzewa, a dalsze dwa parametry wskazują dwa wierzchołki tego drzewa (nie trzeba sprawdzać, czy wskazywane wierzchołki są w drzewie obecne). Zadaniem podprogramu jest znalezienie (i przekazanie jako wartość funkcji) wskaźnika wierzchołka, który jest „ostatnim wspólnym przodkiem” wierzchołków wskazywanych przez parametry, tzn. korzeniem najmniejszego poddrzewa, do którego należą oba te wierzchołki.

3. Do początkowo pustego drzewa AVL zostały wstawione wierzchołki z kluczami — liczbami całkowitymi — w następującej kolejności: $6, 3, 5, 1, 10, 4, 9, 8, 2, 0, 7$. Narysuj obraz drzewa przed każdą rotacją wykonaną podczas tych wstawień i zaznacz, w którym wierzchołku i w którą stronę rotacja została wykonana.

4. Zdanie (J. Tuwima)

   `Ci mali hulali po polu i pili kakao.`

   zostało zakodowane (z pominięciem spacji i znaków interpunkcyjnych) przy użyciu optymalnego kodu znalezionego dla tego tekstu. Podaj najmniejszą i największą liczbę bitów reprezentujących znaki (litery) w tym kodzie i oblicz całkowitą długość (liczbę bitów) zakodowanego tekstu.

Kolokwium poprawkowe ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:15 28 maja 2018.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Klucze w wierzchołkach drzewa binarnych wyszukiwań są liczbami zmiennopozycyjnymi (typu `float`); zakładamy, że drzewo ma co najmniej dwa wierzchołki. Napisz podprogram w C, który znajdzie w uporządkowanym niemalejąco ciągu kluczy wierzchołków takiego drzewa najmniejszą i największą różnicę kolejnych kluczy.

2. Ze struktur typu

```
typedef struct elem {
    int x;
    struct elem *poprz, *nast;
  } elem;
```

   jest zbudowana cykliczna lista dwukierunkowa. Napisz podprogram w C, którego parametr jest wskaźnikiem pewnego elementu takiej listy. Zadaniem podprogramu jest zamiana kolejności w liście: wskazywany element ma się stać następnym elementem swojego początkowego następnika.

3. Do początkowo pustego drzewa AVL zostały wstawione wierzchołki z kluczami — liczbami całkowitymi — w następującej kolejności: $10, 5, 9, 4, 8, 3, 7, 2, 6, 1$. Narysuj obraz drzewa przed każdą rotacją i zaznacz, w którym wierzchołku i w którą stronę rotacja została wykonana.

4. Znajdź kod optymalny do zakodowania fragmentu wiersza L.J. Kerna:

```
A w ogóle rodziców trzeba trzymać ostro.
Jak masz brata - to z bratem.  Jak siostrę - to z siostrą.
Bo inaczej rozpuszczą się do takich granic,
Że gotowi są dzieciom nie pozwolić na nic.
```

Pomiń spacje i znaki interpunkcyjne i zaniedbaj różnicę między wielkimi i małymi literami. Podaj długość (liczbę bitów) zakodowanego tekstu.

Wskazówka. W tekście występują tylko następujące litery:

`A Ą B C Ć D E Ę G H I J K L M N O Ó P R S T U W Y Z Ż`

Kolokwium ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 8:30 29 kwietnia 2019.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Klucze w wierzchołkach drzewa binarnych wyszukiwań są liczbami zmiennopozycyjnymi (typu `float`); zakładamy, że drzewo ma co najmniej dwa wierzchołki. Napisz podprogram w C, który znajdzie w uporządkowanym niemalejąco ciągu kluczy wierzchołków takiego drzewa najmniejszą i największą różnicę kolejnych kluczy. Nie należy przepisywać całego ciągu kluczy do dodatkowej tablicy, aby w niej znaleźć różnice.

2. Ze struktur typu

```
typedef struct elem {
    int x;
    struct elem *nast;
  } elem;
```

są zbudowane listy (jednokierunkowe). Napisz podprogram w C, którego parametrami są wskaźniki początkowych elementów dwóch takich list, uporządkowanych w kolejności niemalejących pól x. Z elementów tych list podprogram ma zbudować jedną listę uporządkowaną w taki sam sposób.

3. Do początkowo pustego drzewa AVL zostały wstawione wierzchołki z kluczami — liczbami całkowitymi — w następującej kolejności: $10, 1, 9, 7, 8, 3, 2, 4, 6, 5$. Narysuj obraz drzewa przed każdą rotacją i zaznacz, w którym wierzchołku i w którą stronę rotacja została wykonana.

4. Znajdź kod binarny optymalny do zakodowania fragmentu wiersza St. Barańczaka:

```
O, oazo, o, opoko
Oka, ucha, ust i uda!
O, omijaj ją, epoko,
Bo kto ufa, że się uda

Naukowy opis - sto sond
Dowód jemu jeno da,
Że o sobie mylny osąd -
A i o Aidzie - ma.
```

Liczby wystąpień samogłosek w tym tekście to a — 12, ą — 2, e — 7, ę — 1, i — 7, o — 24, ó — 1, u — 6, y — 3. Aby znaleźć kod, policz poszczególne spółgłoski, pomiń spacje i znaki interpunkcyjne i zaniedbaj różnicę między wielkimi i małymi literami. Podaj długość (liczbę bitów) zakodowanego tekstu.

Kolokwium poprawkowe ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:15 3 czerwca 2019.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Napisz podprogram w C, który ma 4 parametry: pierwszy jest wskaźnikiem korzenia drzewa binarnych wyszukiwań, a pozostałe 3 są wskaźnikami zmiennych całkowitych, którym podprogram ma przypisać wyniki. Wynikami tymi są maksymalny poziom liścia w drzewie (tj. liczba o 1 mniejsza niż wysokość drzewa), minimalny poziom liścia oraz maksymalna różnica wysokości lewego i prawego poddrzewa dla *wszystkich* wierzchołków drzewa.

   Wskazówka: Zadeklaruj odpowiednie zmienne w podprogramie i przekaż ich adresy jako parametry wywołań rekurencyjnych.

2. Ze struktur typu

```
typedef struct elem {
    int x;
    struct elem *nast;
  } elem;
```

   są zbudowane listy cykliczne. Napisz podprogram w C, którego parametrami są wskaźniki (dowolnych) elementów dwóch takich list. Zadaniem podprogramu jest połączenie tych dwóch list w jedną listę cykliczną (tj. „rozerwanie” list i dołączenie początku pierwszej listy na końcu drugiej i dołączenie początku drugiej listy na końcu pierwszej). Wynik — wskaźnik do dowolnego elementu połączonej listy ma być przekazany jako wartość podprogramu.

   Uwaga: Każda z list może być pusta, może też mieć tylko jeden albo więcej elementów. Podprogram ma poprawnie obsługiwać wszystkie te przypadki.

3. Do początkowo pustego drzewa AVL zostały wstawione wierzchołki z kluczami — liczbami całkowitymi — w następującej kolejności: $5, 6, 4, 2, 3, 8, 7, 9, 1, 10$. Narysuj obraz drzewa przed każdą rotacją i zaznacz, w którym wierzchołku i w którą stronę rotacja została wykonana.

4. Znajdź kod binarny optymalny do zakodowania fragmentu wiersza Ludwika Jerzego Kerna:

   ```
   Zamiast nocą nad książkami pochylać swe czółka,
   Młodzi będą dany przedmiot
   Przyjmować w pigułkach.

   Można by zastrzyki także zaprząc do tej roli,
   Zastrzyk jednak ma tę wadę, że troszeczkę boli.

   A pigułki można łykać bez trudu
   Jak kluski:
   Ta pigułka na algebrę,
   A ta na francuski.
   ```

   Pomiń spacje i znaki interpunkcyjne i zaniedbaj różnicę między wielkimi i małymi literami. Liczby wystąpień wystąpień poszczególnych liter: `A` 29, `Ą` 4, `B` 5, `C` 7, `Ć` 3, `D` 9, `E` 10, `Ę` 4, `F` 1, `G` 4, `H` 2, `I` 14, `J` 4, `K` 16, `L` 5, `Ł` 6, `M` 8, `N` 9, `O` 11, `Ó` 1, `P` 7, `R` 10, `S` 8, `T` 11, `U` 7, `W` 4, `Y` 7, `Z` 15, `Ż` 4.

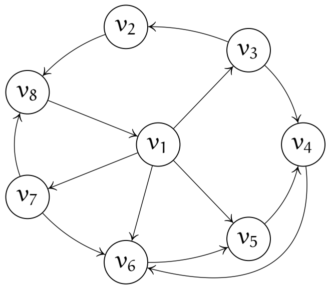
   Podaj długość (liczbę bitów) zakodowanego tekstu.

Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 27 czerwca 2006.)

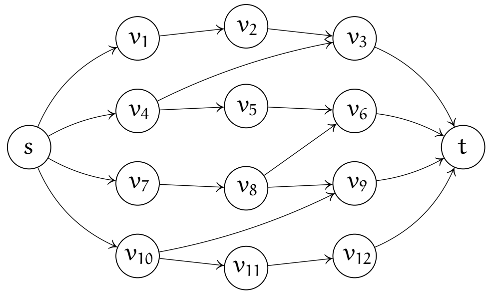
Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Niech G oznacza graf skierowany przedstawiony na rysunku obok:



   a) Podaj kolejność, w której wierzchołki tego grafu zostaną „pomalowane” na szaro i czarno podczas przeszukiwania algorytmem DFS.
   b) Podaj wierzchołki i krawędzie drzew w lesie rozpinającym ten graf.
   c) Metodą podaną na wykładzie znajdź składowe silnie spójne grafu G.

2. Sieć przepływowa przedstawiona na rysunku obok ma wszystkie krawędzie o przepustowości 1.



   a) Stosując algorytm Forda-Fulkersona, znajdź maksymalny przepływ w tej sieci (między wierzchołkami s i t).
   b) Wypisz wszystkie kolejne ścieżki powiększające znalezione przez ten algorytm.
   c) Narysuj sieć residualną dla znalezionego maksymalnego przepływu.

3. a) Napisz w Pascalu podprogram (z parametrem, który służy do przekazania wartości argumentu x), obliczający wartość funkcji

$$f(x) = \frac{x^3 - 2x^2 + 3x - 4}{x^3 + 2x^2 + 3x + 4}$$

przy użyciu arytmetyki zmiennopozycyjnej i schematu Hornera.

b) Napisz wyrażenie opisujące wynik obliczenia wykonywanego przez podprogram napisany jako rozwiązanie punktu a). Czy błąd względny tego wyniku jest ograniczony? Odpowiedź uzasadnij.

c) Pragnąc otrzymać wartości pewnej funkcji f w punktach $\frac{1}{30}, \frac{2}{30}, \dots, \frac{29}{30}, 1$, ktoś napisał następujący fragment programu:

```
...
var x, h : real;
begin
  x := 0.0;  h := 1.0/30.0;
  repeat
    x := x + h;
    writeln ( 'f(', x, ') = ', f(x) )
  until x = 1.0
end;
```

Czy (przy założeniu, że podprogram `f` obliczający wartość funkcji f działa poprawnie dla każdego możliwego argumentu x) można oczekiwać, że powyższy program zadziała zgodnie z intencją autora? Odpowiedź uzasadnij i jeśli jest ona przecząca, to zaproponuj ulepszenie (tj. napisz odpowiedni kod, z uzasadnieniem, dlaczego jest lepszy).

Egzamin poprawkowy ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 7 września 2006.)

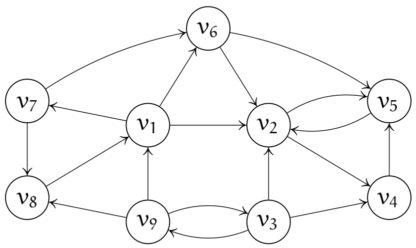
Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. a) Narysuj drzewo AVL otrzymane po wstawieniu do początkowo pustego drzewa ciągu liczb $12, 6, 16, 2, 9, 14, 18, 1, 5, 7, 10, 13$. Podaj liczbę rotacji wykonanych podczas budowania tego drzewa.

b) Czy jest możliwe, że wstawiając elementy pewnego (dowolnie długiego) ciągu jednocześnie do „zwykłego” drzewa BST i do drzewa AVL, oba drzewa będą miały identyczną budowę po wstawieniu każdego elementu? Odpowiedź uzasadnij.

c) Czy algorytm sortowania, który polega na zbudowaniu z elementów sortowanego ciągu drzewa AVL, a następnie wypisaniu elementów z drzewa we właściwej kolejności, ma optymalny rząd złożoności obliczeniowej? Odpowiedź uzasadnij.

2. Niech G oznacza graf skierowany przedstawiony na rysunku obok:

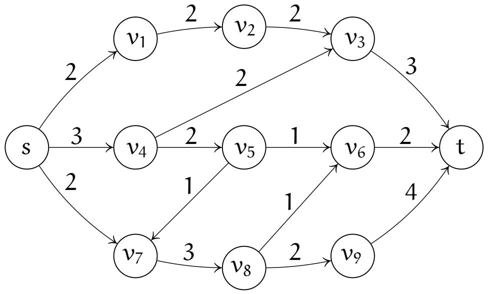


a) Podaj kolejność, w której wierzchołki tego grafu zostaną „pomalowane” na szaro i czarno podczas przeszukiwania algorytmem DFS.

b) Podaj wierzchołki i krawędzie drzew w lesie rozpinającym ten graf.

c) Metodą podaną na wykładzie znajdź składowe silnie spójne grafu G i narysuj graf zredukowany.

3. Rysunek obok przedstawia pewną sieć przepływową, z zaznaczonymi przepustowościami krawędzi.



a) Stosując algorytm Forda-Fulkersona, znajdź maksymalny przepływ w tej sieci (między wierzchołkami s i t).

b) Wypisz wszystkie kolejne ścieżki powiększające znalezione przez ten algorytm.

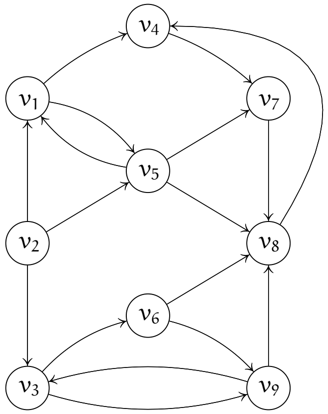
c) Narysuj sieć residualną dla znalezionego maksymalnego przepływu.

Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 9:00 14 czerwca 2007.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Niech G oznacza graf skierowany przedstawiony na rysunku obok.

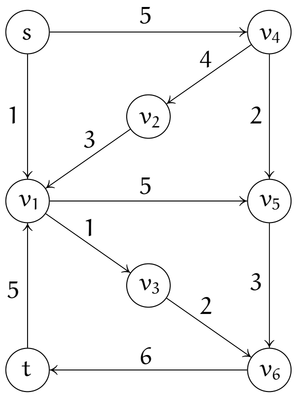


a) Podaj kolejność, w jakiej wierzchołki grafu G zostaną „pomalowane" na szaro i na czarno podczas przeszukiwania tego grafu algorytmem DFS.

b) Metodą podaną na wykładzie znajdź składowe silnie spójne grafu G i narysuj graf zredukowany.

c) Ile krawędzi wystarczy dodać do grafu G, aby otrzymać graf silnie spójny? Czy można zawsze podać odpowiedź na to pytanie, znając tylko graf zredukowany? Odpowiedź uzasadnij.

2. Rysunek obok przedstawia sieć przepływową z zaznaczonymi przepustowościami krawędzi.



a) Stosując algorytm Forda-Fulkersona, znajdź maksymalny przepływ między wierzchołkami s i t w tej sieci. Czy istnieje więcej niż jeden maksymalny przepływ? Odpowiedź uzasadnij.

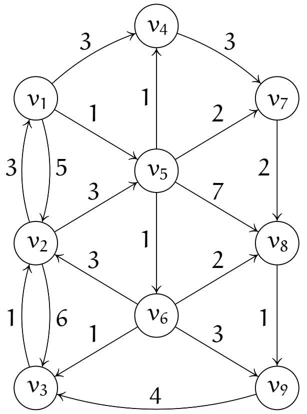
b) Wypisz kolejne ścieżki powiększające znalezione przez ten algorytm.

c) Narysuj sieć residualną dla znalezionego maksymalnego przepływu.

3. Dany jest graf skierowany G, przedstawiony na rysunku obok, którego krawędzie mają określone wagi.

   a) Narysuj drzewo składające się z najkrótszych ścieżek z wierzchołka v_1 do pozostałych.

   b) Podaj kolejność, w jakiej algorytm Dijkstry znajdzie krawędzie tego drzewa.



4. a) Wartość wyrażenia $w = a^2 + ab + b^2$ dla ustalonych liczb a, b, została obliczona na podstawie wzoru $w = \frac{1}{2}\big((a^2 + b^2) + (a + b)^2\big)$. Napisz wyrażenie opisujące otrzymany wynik przy założeniu, że w obliczeniu przy użyciu arytmetyki zmiennopozycyjnej nie wystąpił nadmiar ani niedomiar.

   b) Dane są 3 wielomiany jednej zmiennej: $a(x) = \sum_{i=0}^{n} a_i x^i$, $b(x) = \sum_{i=0}^{m} b_i x^i$ oraz $c(x) = \sum_{i=0}^{l} c_i x^i$, za pomocą współczynników w bazie potęgowej, odpowiednio $a_0, \ldots, a_n$, $b_0, \ldots, b_m$ i $c_0, \ldots, c_l$. Podaj możliwie szybki algorytm obliczania iloczynu tych trzech wielomianów, tj. współczynników w bazie potęgowej wielomianu $d(x) = a(x)b(x)c(x)$.

Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 12 czerwca 2008.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Zmienne typu zdefiniowanego następująco:

```
typedef struct BSTree {
          int klucz;
          struct BSTree *lewe, *prawe;
          int eqi;
        } BSTree;
```

   zostały użyte do zbudowania drzewa binarnych wyszukiwań.

   a) Napisz procedurę, która dla każdego wierzchołka drzewa obliczy jego wskaźnik zrównoważenia i przypisze go polu `eqi`, o nieokreślonej wartości początkowej. <u>Wskazówka:</u> Użyj rekurencji.

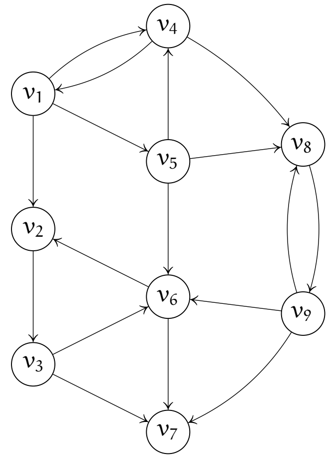
   b) Napisz procedurę, która oblicza (i zwraca jako swoją wartość) wysokość drzewa, którego korzeń jest wskazywany przez parametr. Procedura ma działać w czasie proporcjonalnym do wysokości drzewa, przy założeniu, że wartość pola `eqi` każdego wierzchołka jest poprawnym wskaźnikiem jego zrównoważenia. <u>Wskazówka:</u> Nie używaj rekurencji.

2. Niech G oznacza graf skierowany przedstawiony na rysunku obok.

   a) Przy użyciu algorytmu DFS znajdź las rozpinający ten graf. Dla każdej krawędzi napisz, czy jest ona drzewowa, skierowana w przód, powrotna, czy boczna.

   b) Przy użyciu algorytmu DFS znajdź składowe silnie spójne grafu G.

   c) Narysuj graf zredukowany grafu G; dla każdego wierzchołka grafu zredukowanego wypisz odpowiadające mu wierzchołki grafu G.



Egzamin ze wstępu do informatyki, I rok Mat.

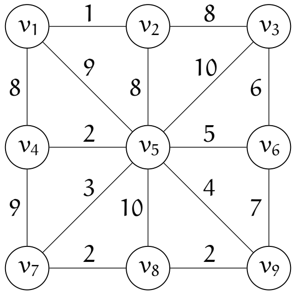
(*Ściśle tajne* przed godz. 14:00 8 czerwca 2009.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dla grafu G przedstawionego na rysunku obok należy znaleźć minimalne drzewo rozpinające. Podaj kolejność znajdowania krawędzi drzewa

   a) przez algorytm Prima,

   b) przez algorytm Kruskala.

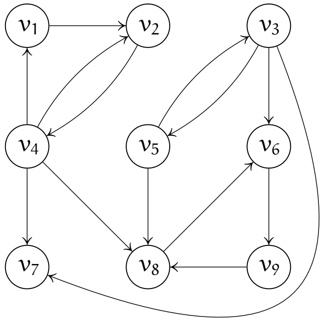


2. Dla grafu skierowanego G przedstawionego na rysunku obok

   a) znajdź las rozpinający za pomocą algorytmu DFS i podaj zbiory krawędzi drzewowych, skierowanych w przód, powrotnych i bocznych,

   b) znajdź za pomocą algorytmu DFS składowe silnie spójne,

   c) narysuj graf zredukowany i podaj minimalną liczbę krawędzi, których dołączenie przekształci graf G w graf silnie spójny. Odpowiedź uzasadnij.

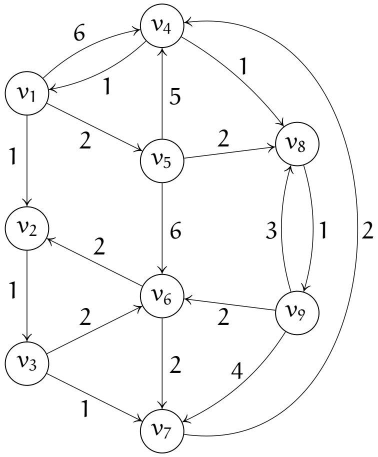


3. Niech G oznacza przedstawiony na rysunku obok graf skierowany, którego krawędzie mają podane długości.

   a) Przy użyciu algorytmu Dijkstry znajdź w grafie G najkrótsze ścieżki z wierzchołka v_1 do pozostałych. Podaj kolejność, w jakiej zostały znalezione te krawędzie.

   b) Narysuj drzewo, którego krawędzie należą do tych ścieżek. Czy jest tylko jedno takie drzewo?

   Czy w ogólności, jeśli drzewo najkrótszych ścieżek jest tylko jedno, kolejność, w jakiej jego krawędzie mogą być wyznaczone przez algorytm Dijkstry, jest jednoznacznie określona? Odpowiedzi uzasadnij.

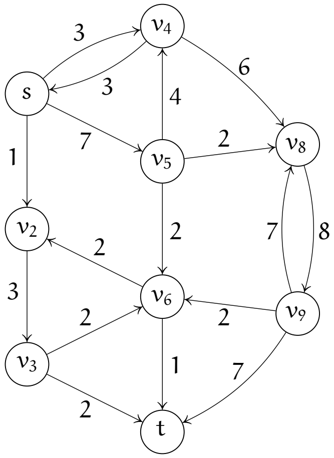


4. Niech G oznacza przedstawioną na rysunku obok sieć przepływową, której krawędzie mają podane przepustowości.

   a) Przy użyciu algorytmu Forda-Fulkersona znajdź przepływ maksymalny w sieci G od zaznaczonego źródła s do ujścia t. Wykonaj rysunek sieci, z zaznaczoną dla każdej krawędzi wartością tego przepływu. Podaj jego sumę.

   b) Wypisz wybrane w kolejnych krokach ścieżki powiększające.

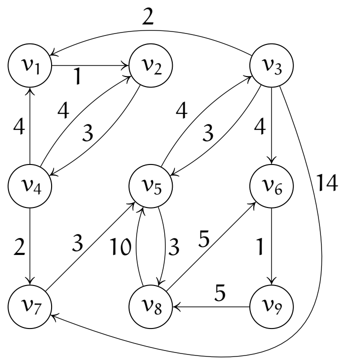
   c) Narysuj sieć residualną znalezionego przepływu maksymalnego i zaznacz na rysunku przepustowości jej krawędzi. Podaj zbiory wierzchołków S i T, stanowiące przekrój odpowiadający znalezionemu przepływowi.



3. Przy krawędziach grafu skierowanego narysowanego obok są podane ich długości.

   a) Podaj kolejność, w jakiej algorytm Dijkstry znajdzie krawędzie najkrótszych ścieżek z wierzchołka v_5 do pozostałych.

   b) Jeśli pewne krawędzie (dowolnego) grafu skierowanego G mają ujemne długości, to czy algorytm Dijkstry użyty do znalezienia najkrótszych ścieżek w G może dać wynik poprawny i czy może dać wynik niepoprawny?
   Odpowiedź uzasadnij.

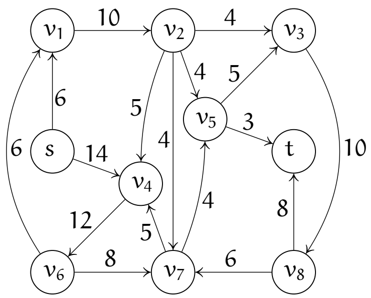


4. Na rysunku obok jest przedstawiona sieć przepływowa z zaznaczonymi przepustowościami krawędzi.

   a) Znajdź przepływ maksymalny ze źródła s do ujścia t w tej sieci za pomocą algorytmu Forda-Fulkersona. Podaj kolejne ścieżki powiększające wybrane przez ten algorytm.

   b) Narysuj sieć residualną dla znalezionego przepływu.

   c) Przypuśćmy, że w pewnej sieci przepływowej dla każdej krawędzi istnieje krawędź skierowana przeciwnie, o tej samej przepustowości. Udowodnij, że jeśli funkcja f jest przepływem maksymalnym ze źródła s do ujścia t, to funkcja $-f$ jest przepływem maksymalnym ze źródła t do ujścia s.



Egzamin ze wstępu do informatyki, I rok Mat.

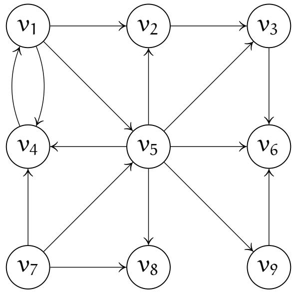
(*Ściśle tajne* przed godz. 10:00 7 czerwca 2010.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. a) Za pomocą algorytmu DFS znajdź las rozpinający graf przedstawiony na rysunku obok. Dla każdej krawędzi grafu napisz, czy jest to krawędź drzewowa, skierowana w przód, powrotna, czy boczna.

   b) Za pomocą algorytmu DFS znajdź składowe silnie spójne tego grafu.

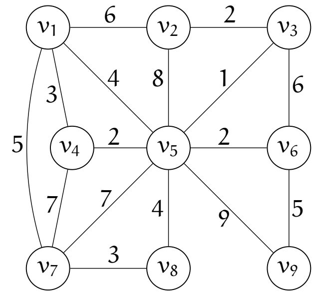
   c) Narysuj graf zredukowany i podaj uporządkowanie topologiczne jego wierzchołków.



2. Krawędzie grafu G na rysunku obok mają przypisane podane wagi. Znajdź minimalne drzewo rozpinające i podaj kolejność wykrywania jego krawędzi

   a) przez algorytm Prima,

   b) przez algorytm Kruskala.

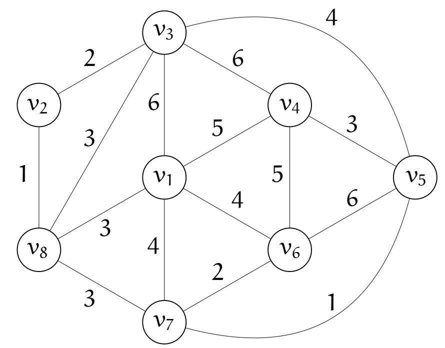


Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10:00 6 września 2010.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

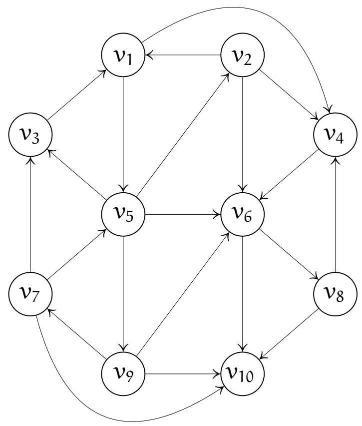
1. Krawędzie grafu G na rysunku obok mają przypisane podane wagi.



   a) Znajdź minimalne drzewo rozpinające i podaj kolejność wykrywania jego krawędzi przez algorytm Prima

   b) Czy istnieje tylko jedno minimalne drzewo rozpinające graf G? Odpowiedź uzasadnij.

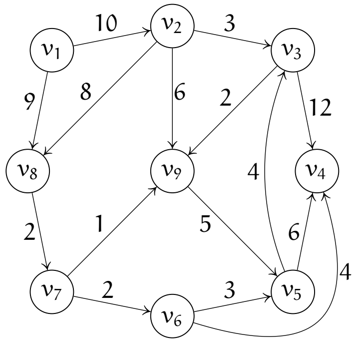
2. a) Za pomocą algorytmu DFS znajdź las rozpinający graf przedstawiony na rysunku obok. Dla każdej krawędzi grafu napisz, czy jest to krawędź drzewowa, skierowana w przód, powrotna, czy boczna.



   b) Za pomocą algorytmu DFS znajdź składowe silnie spójne tego grafu.

   c) Narysuj graf zredukowany. Jaką najmniejszą liczbę krawędzi należy dodać do grafu rozpatrywanego w tym zadaniu, aby powstał graf silnie spójny? Odpowiedź uzasadnij.

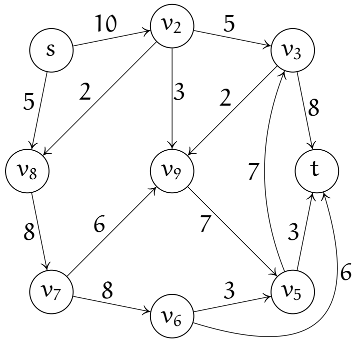
3. Przy krawędziach grafu skierowanego narysowanego obok są podane ich długości.



   a) Za pomocą algorytmu Dijkstry znajdź drzewo najkrótszych ścieżek od wierzchołka v_2 do pozostałych wierzchołków. Czy istnieje tylko jedno takie drzewo? Odpowiedź uzasadnij.

   b) Algorytm Bellmana-Forda w każdej iteracji dokonuje relaksacji wszystkich krawędzi w pewnej kolejności; dla grafu o n wierzchołkach algorytm wykonuje $n-1$ takich iteracji. Czy można uporządkować krawędzie grafu tak, aby wszystkie krawędzie drzewa najkrótszych ścieżek były znalezione w pierwszej iteracji? Czy dla dowolnego grafu można uporządkować krawędzie grafu tak, aby pewne krawędzie drzewa najkrótszych ścieżek były znalezione dopiero w ostatniej iteracji? Odpowiedź uzasadnij.

4. Na rysunku obok jest przedstawiona sieć przepływowa z zaznaczonymi przepustowościami krawędzi.



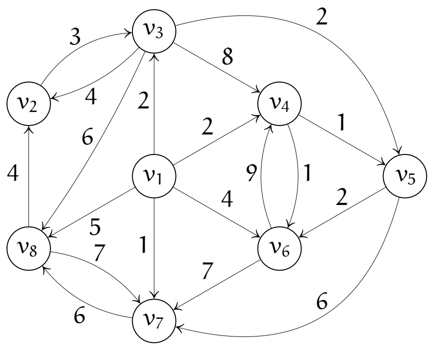
   a) Znajdź przepływ maksymalny ze źródła s do ujścia t w tej sieci za pomocą algorytmu Forda-Fulkersona. Podaj kolejne ścieżki powiększające wybrane przez ten algorytm.

   b) Narysuj sieć residualną dla znalezionego przepływu.

   c) Dla znalezionego przepływu maksymalnego f podaj przekrój (S,T), taki że $f(S,T)=c(S,T)$, gdzie c oznacza funkcję przepustowości sieci.

3. Przy krawędziach grafu skierowanego narysowanego obok są podane ich długości.

   a) Za pomocą algorytmu Dijkstry znajdź drzewo najkrótszych ścieżek od wierzchołka v_5 do pozostałych wierzchołków. Czy istnieje tylko jedno takie drzewo? Odpowiedź uzasadnij.



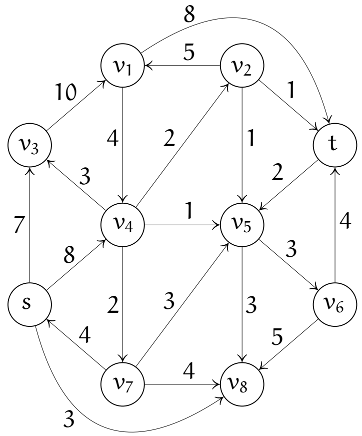
   b) Przypuśćmy, że graf skierowany G o n wierzchołkach jest silnie spójny. Algorytmy DFS i BFS można wykorzystać do uporządkowania krawędzi w kolejności ich przetwarzania przy przeszukiwaniu grafu, zaczynając od pewnego wierzchołka v, w ten sposób, że kolejny numer nadajemy krawędzi w chwili badania jej (sprawdzania, czy wierzchołek końcowy był odwiedzony) po raz pierwszy.
   Czy któreś z tych uporządkowań może spowodować, że algorytm Bellmana-Forda znajdzie wszystkie najkrótsze ścieżki z wierzchołka v po wykonaniu mniej niż $n-1$ iteracji (jedna iteracja polega na wykonaniu relaksacji wszystkich krawędzi)?
   Czy można określić, ile iteracji wystarczy do znalezienia najkrótszych ścieżek przy którymś z tych uporządkowań krawędzi?
   Odpowiedzi uzasadnij.

4. Na rysunku obok jest przedstawiona sieć przepływowa z zaznaczonymi przepustowościami krawędzi.

   a) Znajdź przepływ maksymalny ze źródła s do ujścia t w tej sieci za pomocą algorytmu Forda-Fulkersona. Podaj kolejne ścieżki powiększające wybrane przez ten algorytm.

   b) Narysuj sieć residualną dla znalezionego przepływu.

   c) Dla znalezionego przepływu maksymalnego f podaj przekrój (S, T), taki że $f(S, T) = c(S, T)$, gdzie c oznacza funkcję przepustowości sieci.



Egzamin ze wstępu do informatyki, I rok Mat.

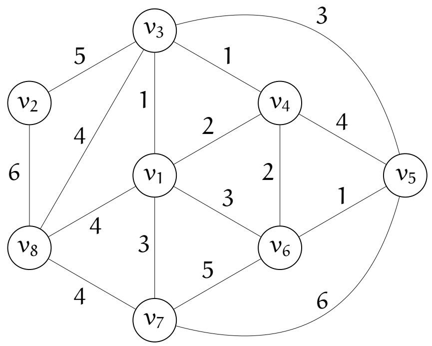
(*Ściśle tajne* przed godz. 10:00 7 czerwca 2011.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Krawędzie grafu G na rysunku obok mają przypisane podane wagi.

   a) Znajdź minimalne drzewo rozpinające i podaj kolejność wykrywania jego krawędzi przez algorytm Kruskala.

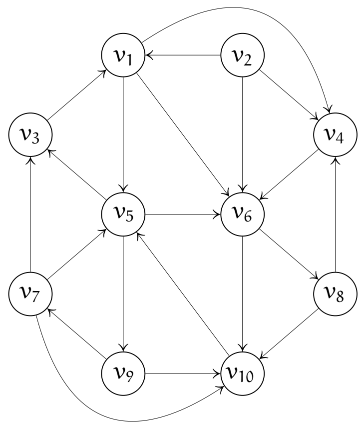
   b) Czy istnieje więcej niż jedno minimalne drzewo rozpinające graf G? Odpowiedź uzasadnij.



2. a) Za pomocą algorytmu DFS znajdź las rozpinający graf G przedstawiony na rysunku obok. Dla każdej krawędzi grafu napisz, czy jest to krawędź drzewowa, skierowana w przód, powrotna, czy boczna.

   b) Za pomocą algorytmu DFS znajdź składowe silnie spójne tego grafu.

   c) Narysuj graf bezcyklowy G', powstały z G przez usunięcie wszystkich krawędzi powrotnych znalezionych podczas rozwiązywania zadania w punkcie a). Następnie znajdź uporządkowanie topologiczne zbioru wierzchołków grafu G'. Opisz krótko metodę użytą do znalezienia tego porządku.



Egzamin ze wstępu do informatyki, I rok Mat.

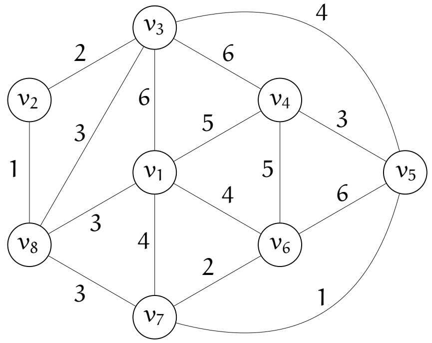
(*Ściśle tajne* przed godz. 10:00 6 września 2011.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Krawędzie grafu G na rysunku obok mają przypisane podane wagi.

   a) Znajdź minimalne drzewo rozpinające i podaj kolejność wykrywania jego krawędzi przez algorytm Prima

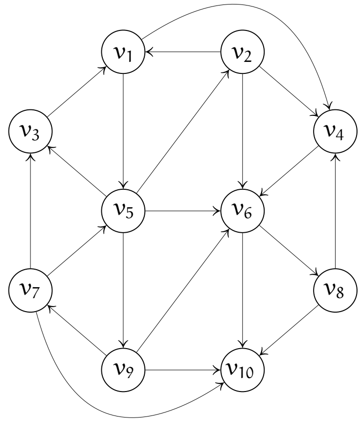
   b) Czy istnieje tylko jedno minimalne drzewo rozpinające graf G? Odpowiedź uzasadnij.



2. a) Za pomocą algorytmu DFS znajdź las rozpinający graf przedstawiony na rysunku obok. Dla każdej krawędzi grafu napisz, czy jest to krawędź drzewowa, skierowana w przód, powrotna, czy boczna.

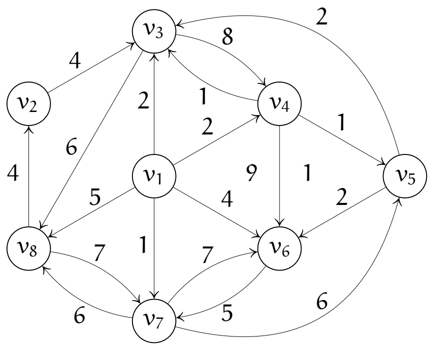
   b) Za pomocą algorytmu DFS znajdź składowe silnie spójne tego grafu.

   c) Narysuj graf zredukowany. Jaką najmniejszą liczbę krawędzi należy dodać do grafu rozpatrywanego w tym zadaniu, aby powstał graf silnie spójny? Odpowiedź uzasadnij.



3. Przy krawędziach grafu skierowanego narysowanego obok są podane ich długości.

   a) Za pomocą algorytmu Dijkstry znajdź drzewo najkrótszych ścieżek od wierzchołka v_5 do pozostałych wierzchołków. Czy istnieje tylko jedno takie drzewo? Odpowiedź uzasadnij.



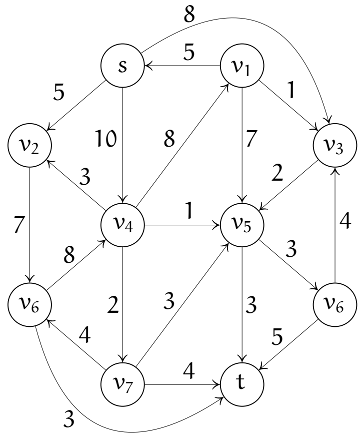
   b) Rozważamy graf skierowany G o n wierzchołkach, którego krawędzie mogą mieć długości o dowolnych znakach, ale w którym nie ma ujemnych cykli. Dla takiego grafu chcemy znaleźć najkrótsze ścieżki między wszystkimi parami wierzchołków. Które ze znanych z wykładu algorytmów mogą być użyte do rozwiązania tego zadania? Jaki jest rząd złożoności dla tych algorytmów? Rozważ przypadki, gdy liczba krawędzi jest rzędu n oraz n^2. Odpowiedzi uzasadnij.

4. Na rysunku obok jest przedstawiona sieć przepływowa z zaznaczonymi przepustowościami krawędzi.

   a) Znajdź przepływ maksymalny ze źródła s do ujścia t w tej sieci za pomocą algorytmu Forda-Fulkersona. Podaj kolejne ścieżki powiększające wybrane przez ten algorytm.

   b) Narysuj sieć residualną dla znalezionego przepływu.

   c) Dla znalezionego przepływu maksymalnego f podaj przekrój (S, T), taki że $f(S, T) = c(S, T)$, gdzie c oznacza funkcję przepustowości sieci.



Egzamin ze wstępu do informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10:00 6 czerwca 2016.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dana jest definicja typu

```
typedef struct wierzch {
      int klucz;
      struct wierzch *lewe, *prawe;
    } wierzch;
```

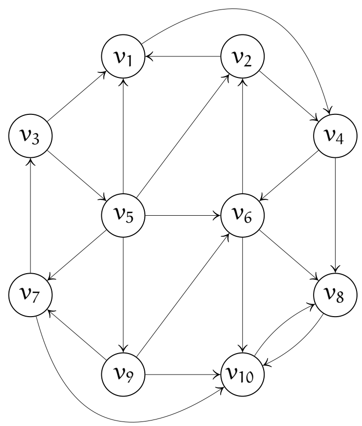
   a) Napisz podprogram, który dla drzewa binarnego o wierzchołkach tego typu znajduje minimalny i maksymalny poziom, na którym znajdują się liście tego drzewa. Wskaźnik korzenia drzewa ma być podany jako parametr, dalsze parametry mają być użyte do przekazania wyniku.

   b) Czy jeśli minimalny poziom znaleziony przez podprogram napisany w punkcie a) jest taki sam jak maksymalny, to to oznacza, że drzewo ma minimalną możliwą wysokość dla ustalonej liczby wierzchołków? Odpowiedź uzasadnij.

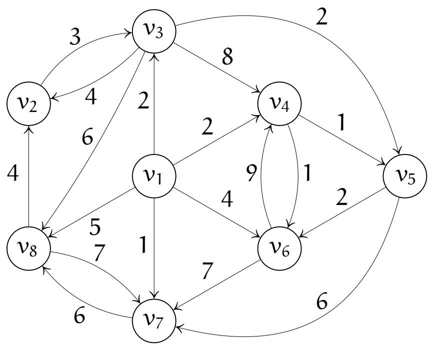
2. a) Za pomocą algorytmu DFS znajdź las rozpinający graf przedstawiony na rysunku obok. Dla każdej krawędzi grafu napisz, czy jest to krawędź drzewowa, skierowana w przód, powrotna, czy boczna.

   b) Za pomocą algorytmu DFS znajdź składowe silnie spójne tego grafu i narysuj graf zredukowany.

   c) Czy istnieją takie krawędzie w grafie G, że zmiana orientacji którejś z nich (tylko jednej) spowoduje powstanie grafu silnie spójnego? Jeśli tak, to wskaż wszystkie takie krawędzie.



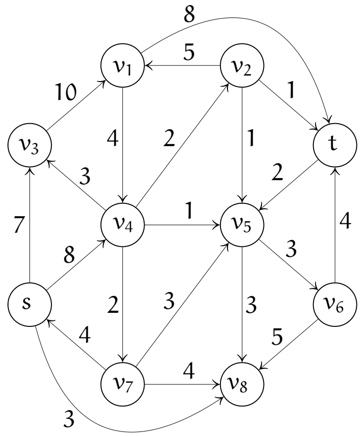
3. Przy krawędziach grafu skierowanego narysowanego obok są podane ich długości.



   a) Za pomocą algorytmu Dijkstry znajdź drzewo najkrótszych ścieżek od wierzchołka v_5 do pozostałych wierzchołków. Czy istnieje tylko jedno takie drzewo? Odpowiedź uzasadnij.

   b) Przypuśćmy, że graf skierowany G o n wierzchołkach jest silnie spójny. Algorytmy DFS i BFS można wykorzystać do uporządkowania krawędzi w kolejności ich przetwarzania przy przeszukiwaniu grafu, zaczynając od pewnego wierzchołka v, w ten sposób, że kolejny numer nadajemy krawędzi w chwili badania jej (sprawdzania, czy wierzchołek końcowy był odwiedzony) po raz pierwszy.
   Czy któreś z tych uporządkowań może spowodować, że algorytm Bellmana-Forda znajdzie wszystkie najkrótsze ścieżki z wierzchołka v po wykonaniu mniej niż $n-1$ iteracji (jedna iteracja polega na wykonaniu relaksacji wszystkich krawędzi)?
   Czy można określić, ile iteracji wystarczy do znalezienia najkrótszych ścieżek przy którymś z tych uporządkowań krawędzi?
   Odpowiedzi uzasadnij.

4. Na rysunku obok jest przedstawiona sieć przepływowa z zaznaczonymi przepustowościami krawędzi.



   a) Znajdź przepływ maksymalny ze źródła s do ujścia t w tej sieci za pomocą algorytmu Forda-Fulkersona. Podaj kolejne ścieżki powiększające wybrane przez ten algorytm.

   b) Narysuj sieć residualną dla znalezionego przepływu.

   c) Dla znalezionego przepływu maksymalnego f podaj przekrój (S,T), taki że $f(S,T) = c(S,T)$, gdzie c oznacza funkcję przepustowości sieci.

3. Dla pewnych pięciu miast koszt budowy bezpośredniego połączenia kolejowego pomiędzy miastem i-tym i j-tym jest równy a_{ij}, przy czym liczby a_{ij} są współczynnikami macierzy

$$A = \begin{bmatrix} 0 & 5 & 4 & 3 & 1 \\ 5 & 0 & 2 & 8 & 7 \\ 4 & 2 & 0 & 6 & 3 \\ 3 & 8 & 6 & 0 & 4 \\ 1 & 7 & 3 & 4 & 0 \end{bmatrix}.$$

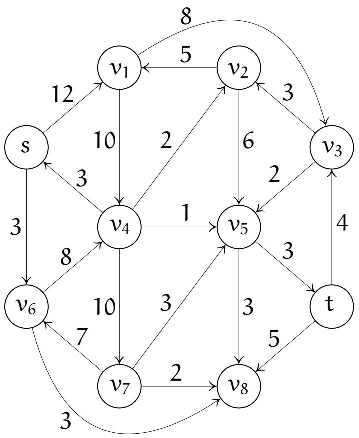
Zaprojektuj najtańszą w budowie sieć kolejową łączącą wszystkie miasta. Wykonaj rysunek — schemat tej sieci i uzasadnij, że jest ona najtańsza. Jaki jest koszt budowy?

4. Na rysunku obok jest przedstawiona sieć przepływowa z zaznaczonymi przepustowościami krawędzi.

   a) Znajdź przepływ maksymalny ze źródła s do ujścia t w tej sieci za pomocą algorytmu Forda-Fulkersona. Podaj kolejne ścieżki powiększające wybrane przez ten algorytm.

   b) Narysuj sieć residualną dla znalezionego przepływu.

   c) Dla znalezionego przepływu maksymalnego f podaj przekrój (S, T), taki że $f(S, T) = c(S, T)$, gdzie c oznacza funkcję przepustowości sieci.



Egzamin poprawkowy ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 31 sierpnia 2016.)

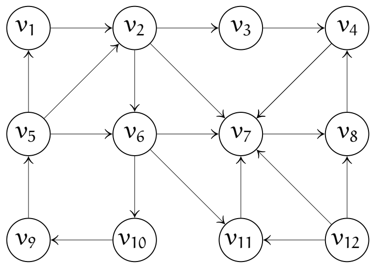
Proszę bardzo uważnie przeczytać treść zadań. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Rozważamy sortowanie ciągu liczb o długości n. W tym celu tworzymy puste drzewo binarnych wyszukiwań, a następnie wstawiamy do niego kolejno elementy ciągu. Następnie obchodzimy drzewo w głąb, wyprowadzając dla każdego wierzchołka elementy z lewego poddrzewa, z danego wierzchołka i z prawego poddrzewa.

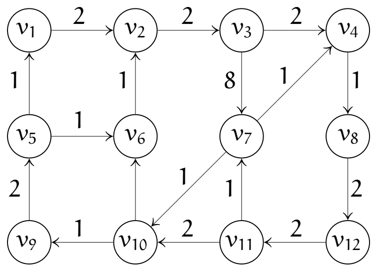
   a) Napisz w języku C podprogram, który wyprowadza elementy zgodnie z powyższym opisem.

   b) Jaki jest rząd złożoności optymistycznej i pesymistycznej takiego algorytmu sortowania w przypadku, gdy drzewo jest zwykłym drzewem binarnych wyszukiwań oraz gdy jest to drzewo AVL? Odpowiedź uzasadnij.

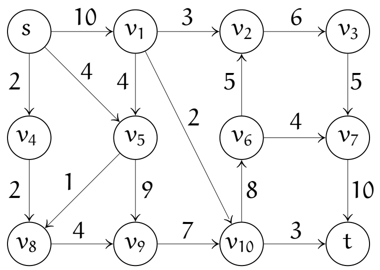
2. Dla grafu skierowanego przedstawionego na rysunku poniżej, metodą przeszukiwania w głąb, znajdź las rozpinający, a następnie znajdź składowe silnie spójne i narysuj graf zredukowany. Jaką najmniejszą liczbę krawędzi wystarczy dołączyć do grafu, aby powstał graf silnie spójny? Odpowiedź uzasadnij.



3. Korzystając z algorytmu Dijkstry, znajdź najkrótsze ścieżki z wierzchołka v_6 narysowanego niżej grafu do wszystkich pozostałych wierzchołków. Obok każdej krawędzi na rysunku jest podana jej długość. Czy jest tylko jedno rozwiązanie tego zadania? Odpowiedź uzasadnij.



4. Znajdź maksymalny przepływ od źródła s do ujścia t w sieci przepływowej pokazanej na rysunku niżej. Obok każdej krawędzi jest podana jej przepustowość. Narysuj sieć residualną, zaznacz przekrój tej sieci i podaj sumę znalezionego przepływu.



Egzamin ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10:00 19 czerwca 2017.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Struktury typu

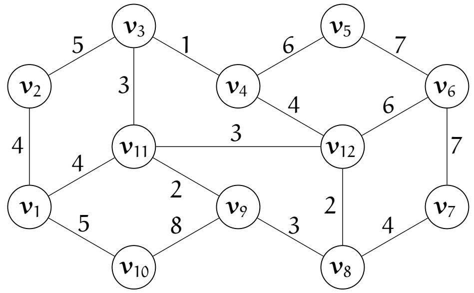
```
typedef struct wdi { /* wdi - wierzchołek drzewa iglastego */
    float     klucz;
    int       licznik;
    struct wdi *lewe, *prawe;
  } wdi;
```

mają być użyte do zbudowania drzewa binarnych wyszukiwań (BST).

a) Napisz podprogram wstawiania wierzchołka do takiego drzewa; parametrem procedury ma być wskaźnik wskaźnika korzenia drzewa (mającego wartość `NULL` przed pierwszym wywołaniem procedury) i klucz, który ma być zapamiętany w nowym wierzchołku. Pole `licznik` każdego wierzchołka drzewa ma mieć wartość równą liczbie wierzchołków poddrzewa, którego korzeniem jest ten wierzchołek (jeśli np. wierzchołek jest liściem, to jego `licznik` ma mieć wartość 1, jeśli jest korzeniem całego drzewa, to jego `licznik` ma przechowywać liczbę wszyskich wierzchołków drzewa itd.).

b) Napisz podprogram, który dla danego k wyszuka w drzewie, którego wierzchołki mają liczniki o wartościach jak wyżej, klucz, który jest k-tym elementem uporządkowanego niemalejąco ciągu kluczy przechowywanych w drzewie (przy założeniu, że w drzewie jest co najmniej k wierzchołków).

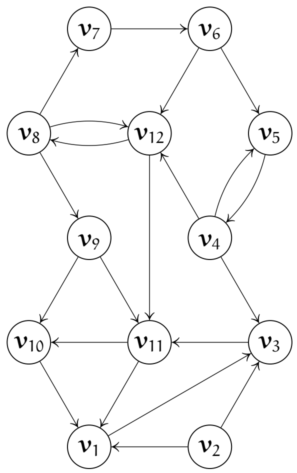
2. Znajdź minimalne drzewo rozpinające graf przedstawiony na rysunku obok, przy użyciu algorytmu Kruskala. Podaj kolejność, w jakiej ten algorytm znajduje krawędzie tego drzewa. Czy jest tylko jedna możliwa kolejność? Czy jest tylko jedno minimalne drzewo rozpinające? Odpowiedzi uzasadnij.



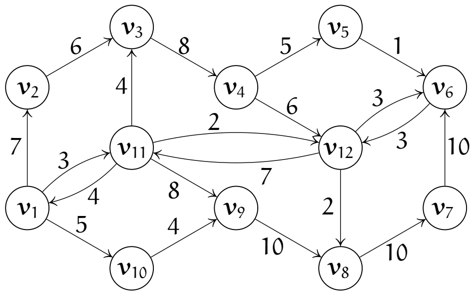
3. a) Metodą DFS znajdź las rozpinający graf skierowany G przedstawiony na rysunku obok.

   b) Korzystając z rozwiązania poprzedniego punktu, znajdź składowe silnie spójne tego grafu i narysuj graf zredukowany.

   c) Podaj minimalny zbiór krawędzi, których dołączenie do grafu G spowoduje powstanie grafu silnie spójnego.



4. Znajdź drzewo najkrótszych ścieżek z wierzchołka v_1 do pozostałych wierzchołków grafu skierowanego pokazanego na rysunku poniżej, za pomocą algorytmu Dijkstry. Obok każdej krawędzi jest podana jej długość. Podaj kolejność, w jakiej algorytm Dijkstry znajduje krawędzie tego drzewa. Czy drzewo najkrótszych ścieżek w tym grafie jest jednoznaczne? Odpowiedź uzasadnij.



Egzamin poprawkowy ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 30 sierpnia 2017.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

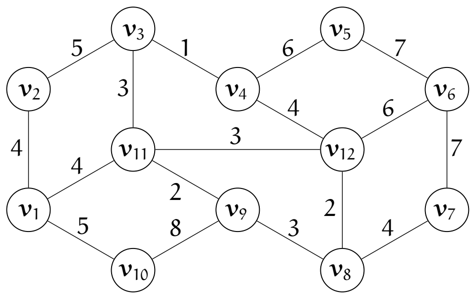
1. Struktury typu

```
typedef struct el {
    float     klucz;
    int       dana;
    struct el *nast;
  } el;
```

   są używane do zbudowania list otwartych (tzn. nie-cyklicznych).

   Napisz podprogram, który przegląda daną listę (której wskaźnik pierwszego elementu jest podany jako parametr) i wszystkie elementy, których klucze mają wartość bezwzględną większą niż podana jako parametr liczba a przenosi do nowej listy (utworzonej przez ten podprogram), zachowując ich oryginalną kolejność. Procedura ma przekazać jako wynik dwa wskaźniki: pierwszego elementu pozostałości danej listy, z której odpowiednie elementy zostały usunięte i pierwszego elementu nowej listy.

2. Znajdź minimalne drzewo rozpinające graf przedstawiony na rysunku obok, przy użyciu algorytmu Prima. Podaj kolejność, w jakiej ten algorytm znajduje krawędzie tego drzewa. Czy jest tylko jedna możliwa kolejność? Czy jest tylko jedno minimalne drzewo rozpinające? Odpowiedzi uzasadnij.

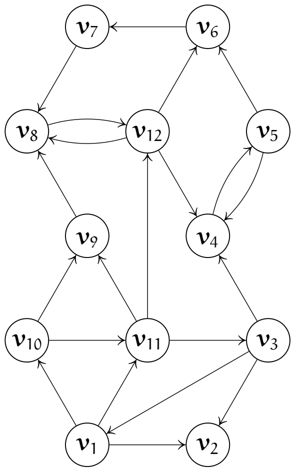


Podaj definicję algorytmu zachłannego i wymień trzy przykłady zachłannych algorytmów przetwarzania grafów, omawianych na wykładach.

3. a) Metodą DFS znajdź las rozpinający graf skierowany G przedstawiony na rysunku obok.

   b) Korzystając z rozwiązania poprzedniego punktu, znajdź składowe silnie spójne tego grafu i narysuj graf zredukowany.

   c) Podaj minimalny zbiór krawędzi, których dołączenie do grafu G spowoduje powstanie grafu silnie spójnego.



4. Podaj definicje sieci przepływowej, przepływu, sumy przepływu, ścieżki powiększającej, sieci residualnej i przekroju w sieci.

   Narysuj dowolną sieć przepływową mającą od 4 do 6 wierzchołków i wskaż w niej ścieżką powiększającą, a następnie narysuj sieć residualną dla przepływu wzdłuż tej ścieżki.

Egzamin ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10:00 18 czerwca 2018.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę bardzo duży wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

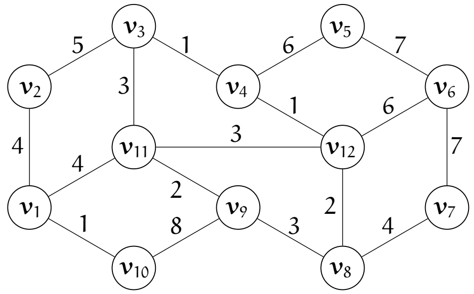
1. Struktury typu

```
typedef struct el {
    float    klucz;
    int      dana;
    struct el *nast;
  } el;
```

są używane do zbudowania list otwartych (tzn. nie-cyklicznych).

Napisz podprogram o nazwie `Zamien`, który ma parametr `p` typu `el**` (wskaźnik wskaźnika struktury `el`). Zadaniem podprogramu jest zamiana kolejnych dwóch elementów listy: elementu wskazywanego przez `*p` i elementu następnego w liście, jeśli taki istnieje. Wyjaśnij, dlaczego użycie wskaźnika do wskaźnika jest potrzebne w takim podprogramie.

2. Znajdź minimalne drzewo rozpinające graf przedstawiony na rysunku obok, przy użyciu algorytmu Kruskala. Podaj kolejność, w jakiej ten algorytm znajduje krawędzie tego drzewa. Czy jest tylko jedna możliwa kolejność? Czy jest tylko jedno minimalne drzewo rozpinające? Odpowiedzi uzasadnij.



   Podaj definicję algorytmu zachłannego i wymień trzy przykłady zachłannych algorytmów przetwarzania grafów, omawianych na wykładach.

Egzamin poprawkowy ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 10:00 18 września 2018.)

Proszę bardzo uważnie przeczytać treść zadań. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Dana jest definicja typu

```
typedef struct wierzch {
     int klucz;
     struct wierzch *lewe, *prawe;
    } wierzch;
```

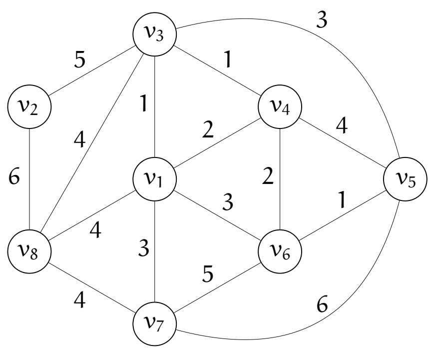
a) Napisz podprogram, który dla drzewa binarnego o wierzchołkach tego typu znajduje minimalny i maksymalny poziom, na którym znajdują się liście tego drzewa. Wskaźnik korzenia drzewa ma być podany jako parametr, dalsze parametry mają być użyte do przekazania wyniku.

b) Czy jeśli minimalny poziom znaleziony przez podprogram napisany w punkcie a) jest taki sam jak maksymalny, to to oznacza, że drzewo ma minimalną możliwą wysokość dla ustalonej liczby wierzchołków? Odpowiedź uzasadnij.

2. Krawędzie grafu G na rysunku obok mają przypisane podane wagi.

a) Znajdź minimalne drzewo rozpinające i podaj kolejność wykrywania jego krawędzi przez algorytm Kruskala.

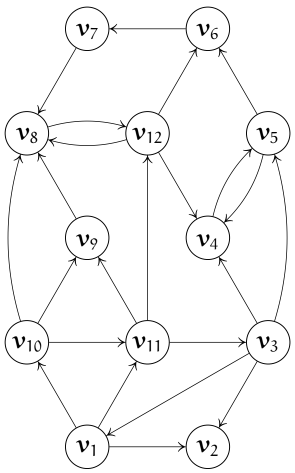
b) Czy istnieje więcej niż jedno minimalne drzewo rozpinające graf G? Odpowiedź uzasadnij.



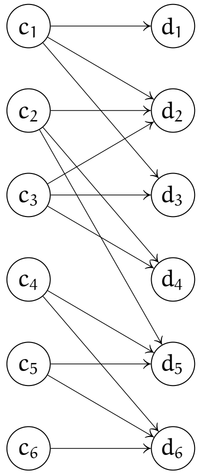
3. a) Metodą DFS znajdź las rozpinający graf skierowany G przedstawiony na rysunku obok.

b) Korzystając z rozwiązania poprzedniego punktu, znajdź składowe silnie spójne tego grafu i narysuj graf zredukowany.

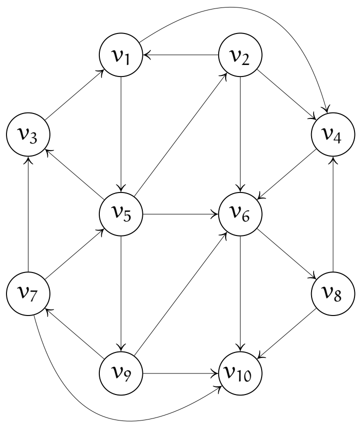
c) Podaj minimalny zbiór krawędzi, których dołączenie do grafu G spowoduje powstanie grafu silnie spójnego.



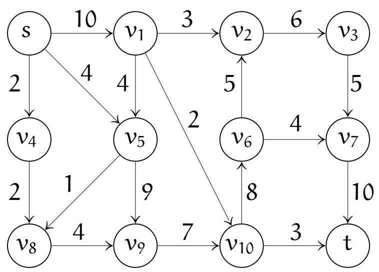
4. Do grafu pokazanego na rysunku obok dodaj źródło s, z którego wychodzą krawędzie do wszystkich wierzchołków c_i oraz ujście t, do którego prowadzą krawędzie od wszystkich wierzchołków d_i. Zakładając, że wszystkie krawędzie powstałej w ten sposób sieci przepływowej mają przepustowość 1, znajdź maksymalny przepływ i odpowiedz na pytanie, czy w tym grafie dwudzielnym istnieje skojarzenie. Jeśli tak, to je wypisz. Jeśli nie, to uzasadnij, dlaczego.



3. a) Za pomocą algorytmu DFS znajdź las rozpinający graf przedstawiony na rysunku obok. Dla każdej krawędzi grafu napisz, czy jest to krawędź drzewowa, skierowana w przód, powrotna, czy boczna.
   b) Za pomocą algorytmu DFS znajdź składowe silnie spójne tego grafu.
   c) Narysuj graf zredukowany. Jaką najmniejszą liczbę krawędzi należy dodać do grafu rozpatrywanego w tym zadaniu, aby powstał graf silnie spójny? Odpowiedź uzasadnij.



4. Znajdź maksymalny przepływ od źródła s do ujścia t w sieci przepływowej pokazanej na rysunku niżej. Obok każdej krawędzi jest podana jej przepustowość. Narysuj sieć residualną, zaznacz przekrój tej sieci i podaj sumę znalezionego przepływu.



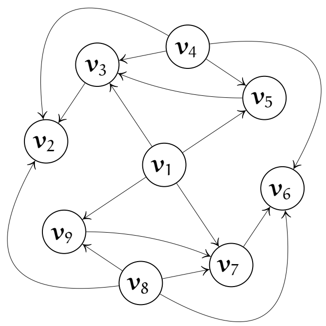
Egzamin ze Wstępu do Informatyki, I rok Mat.

(*Ściśle tajne* przed godz. 14:00 24 czerwca 2019.)

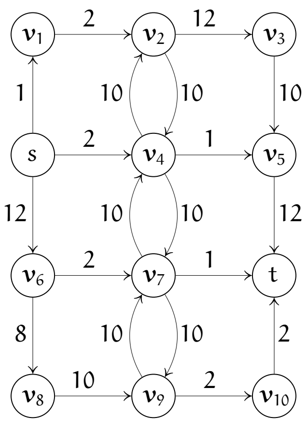
Proszę bardzo uważnie przeczytać treść zadań. Na ocenę **bardzo duży** wpływ będzie miała czytelność rozwiązań i poprawność uzasadnienia każdej odpowiedzi.

1. Napisz procedury obsługi kolejki (zwykłej): `QueueInit`, `Enqueue`, `Dequeue` i `QueueEmpty`, takie że kolejka jest implementowana w postaci listy jednokierunkowej. Typ elementu przechowywanego w kolejce jest strukturalny i ma nazwę `element`. Kolejka ma być reprezentowana przez strukturę zawierającą wskaźniki początku i końca listy elementów, zaś wskaźnik takiej struktury ma być pierwszym parametrem wymienionych wyżej procedur.
2. Podaj definicję pojęcia algorytmu zachłannego. Wymień przykłady zadań grafowych, przedstawionych na wykładzie, które można rozwiązać za pomocą algorytmów zachłannych. Dla jednego, dowolnie wybranego przykładu zadania przedstaw rozwiązujący je algorytm zachłanny i uzasadnij, dlaczego ten algorytm daje poprawny wynik.

3. a) Za pomocą algorytmu DFS znajdź las rozpinający graf przedstawiony na rysunku obok. Dla każdej krawędzi grafu napisz, czy jest to krawędź drzewowa, skierowana w przód, powrotna, czy boczna.
   b) Za pomocą algorytmu DFS znajdź składowe silnie spójne tego grafu.
   c) Narysuj graf zredukowany. Jaką najmniejszą liczbę krawędzi należy dodać do grafu rozpatrywanego w tym zadaniu, aby powstał graf silnie spójny? Odpowiedź uzasadnij.



4. Na rysunku niżej jest pokazana sieć przepływowa z zaznaczonym źródłem s, ujściem t i przepustowością każdej krawędzi. Znajdź maksymalny przepływ od źródła do ujścia w tej sieci. Podaj ścieżki powiększające kolejno znalezione podczas poszukiwania maksymalnego przepływu. Narysuj sieć residualną, zaznacz przekrój (S, T) tej sieci, taki że $f(S, T) = c(S, T)$ i podaj sumę znalezionego przepływu.



Zadania do zaprogramowania (I semestr)

1. Napisz program, który układa słownik dla początkujących angielskich poetów: ma on przeczytać z pliku wejściowego pewną liczbę słów (rzędu 1000), a następnie posortować te słowa w kolejności alfabetycznej, przy odwróconej kolejności liter, i wyprowadzić do pliku wyjściowego.
2. Napisz program, który czyta z pliku wejściowego tekst i zlicza wystąpienia znalezionych w nim słów. Wynikiem jest plik, w którym n słów, które pojawiły się najwięcej razy, zostało wypisanych razem z liczbami ich wystąpień.
3. Napisz program, który czyta tekst z pliku wejściowego i sprawdza, czy w tym tekście występuje pewne ustalone słowo; jeśli tak, to należy podać pierwszą pozycję, na której słowo to wystąpiło.
4. Napisz program, który rysuje wykres funkcji ciągłej jednej zmiennej. Program powinien umożliwiać narysowanie kilku wykresów na jednym rysunku, a ponadto osie powinny być opisane liczbami dobranymi do długości jednostki osi.
5. Napisz program, który rysuje wykres funkcji dwóch zmiennych w postaci powierzchni złożonej z trójkątów. Funkcja może być nieokreślona w pewnym obszarze prostokąta $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ i w takim razie trójkąty, których dowolne wierzchołki są przez to nieokreślone, powinny być pominięte.
6. Napisz program, który rysuje warstwice funkcji ciągłej w pewnym prostokącie $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$.
7. Napisz program, który dla wszystkich nieparzystych większych niż 1 i nie większych niż ustalone n znajduje najmniejsze całkowite dzielniki właściwe.
8. Napisz program, który dla kolejnych liczb naturalnych znajduje ułamki powstałe przez odwrócenie cyfr w ustalonym układzie. Na przykład, jeśli to jest układ dziesiętny, to liczbie 62984 odpowiada 0.48926, ale podstawa układu może być wybierana dowolnie.
9. Napisz program, który rozstawia n hetmanów na szachownicy o wymiarach $n \times n$ tak, aby żaden nie bił drugiego, i wyprowadza wynik w postaci pięknego rysunku.
10. Napisz program, który znajduje drogę konika szachowego przez wszystkie pola szachownicy o wymiarach $n \times n$ i jeśli znajdzie, to ją rysuje.
11. Napisz program, który dla danego n znajduje i wypisuje wszystkie permutacje zbioru $\{1, \ldots, n\}$.
12. Napisz program, który dla danego czworokąta, którego wierzchołki mają współrzędne całkowite, wypełnia ekran parkietażem zbudowanym z czworokątów do niego przystających.

13. Napisz program realizujący grę „życie” Conwaya w tablicy $n \times n$ traktowanej jako torus. Gra jest taka: komórka (element tablicy) może być „żywa” albo „martwa”. Jeśli z daną komórką sąsiadują dwie „żywe”, to nie zmienia ona stanu. Jeśli trzy „żywe”, to w następnym kroku jest ona „żywa”. Jeśli dowolna inna liczba (od 0 do 8), to komórka będzie „martwa”. Gra rozpoczyna się od pewnego (np. losowego) stanu początkowego komórek i w kolejnych krokach należy rysować obrazy tablicy z komórkami.
14. Napisz program, który czyta z pliku dwie macierze zespolone, mnoży je i wyprowadza macierz będącą iloczynem.
15. Napisz program, który dla wielokąta, którego brzeg jest łamaną o wierzchołkach przeczytanych z pliku, sprawdza, czy ten wielokąt jest poprawny (tj. łamana nie ma samoprzecięć) i wypukły.
16. Napisz program, który dla wielokąta, którego brzeg jest łamaną o wierzchołkach przeczytanych z pliku, oblicza pole, obwód i współrzędne środka ciężkości tego wielokąta.
17. Napisz program, który dla wielokąta, którego brzeg jest łamaną o wierzchołkach przeczytanych z pliku i pewnego ciągu punktów (też przeczytanego z pliku) sprawdza, czy każdy z tych punktów leży wewnątrz wielokąta.
18. Napisz program, który dane przeczytane z pliku (kilka do kilkunastu liczb dodatnich) przedstawia na wykresie „kołowym” lub „słupkowym”.
19. Napisz program, który rozwiązuje uogólniony problem wież Hanoi: należy przenieść n krążków z pręta nr. 1 na pręt nr. 2, przy użyciu dwóch prętów dodatkowych. Obecność dodatkowego pręta ma na celu zmniejszenie liczby wykonanych przeniesień.
20. Napisz program, który dokonuje konwersji liczby naturalnej podanej w układzie dziesiętnym na zapis rzymski i w drugą stronę.
21. Napisz program, który oblicza liczbę dni między dwiema datami podanymi przez użytkownika.

Podstawowe pojęcia

Def. Informatyka jest to nauka o przetwarzaniu informacji przy użyciu automatycznych środków pomocniczych.

Sedno powyższej definicji jest zawarte w słowie „automatycznych”; nie obejmuje ona przetwarzania informacji takiego jak podczas stawiania hipotez lub (z pewnymi wyjątkami) dowodzenia twierdzeń w matematyce. Aby można było zastosować jakiś „automat”, potrzebny jest algorytm.

Def. Algorytm jest to jednoznaczny i zrozumiały dla wykonawcy opis postępowania, który prowadzi do otrzymania wyniku (informacji końcowej) na podstawie danych (informacji początkowej).

Najbardziej interesujące (ale nie jedyne) są algorytmy rozwiązywania różnych zadań. Zadanie definiuje się przez określenie zbioru D wszystkich dopuszczalnych danych i zbioru W wszystkich możliwych wyników, oraz określenie sposobu (pewnej funkcji $f\colon D \to W$), w jaki wyniki zależą od danych.

Przykłady zadań

- Dla danych liczb naturalnych n, k znajdź ich największy wspólny dzielnik. Mamy zatem $D = \mathbb{N} \times \mathbb{N}$, $W = \mathbb{N}$ oraz $f(n,k) = \mathrm{NWD}(n,k)$.
- Znajdź wszystkie pierwiastki rzeczywiste trójmianu kwadratowego $w(x) = ax^2 + bx + c$, $a \neq 0$. W tym przypadku $D = \mathbb{R} \setminus \{0\} \times \mathbb{R} \times \mathbb{R}$, $W = \mathbb{R}^2 \cup \mathbb{R} \cup \{\emptyset\}$. Funkcja f ma postać
$$f(a,b,c) = \begin{cases} \{\frac{-b-\sqrt{\Delta}}{2a}, \frac{-b+\sqrt{\Delta}}{2a}\} & \text{dla } \Delta > 0, \\ \{-\frac{b}{2a}\} & \text{dla } \Delta = 0, \\ \emptyset & \text{dla } \Delta < 0, \end{cases}$$
gdzie $\Delta = b^2 - 4ac$.
- Posortuj dany ciąg liczb rzeczywistych $(a_0, a_1, \ldots, a_{n-1})$, tj. ustaw te liczby w kolejności niemalejącej. W tym zadaniu $D = \mathbb{R}^n$, $W = \{\,(x_0, \ldots, x_{n-1})\colon x_0 \leq \cdots \leq x_{n-1}\,\} \subset \mathbb{R}^n$, zaś $f(a_0, a_1, \ldots, a_{n-1}) = (a_{\sigma(0)}, a_{\sigma(1)}, \ldots, a_{\sigma(n-1)})$, gdzie σ jest dowolną permutacją zbioru n-elementowego taką że $a_{\sigma(0)} \leq a_{\sigma(1)} \leq \cdots \leq a_{\sigma(n-1)}$.

Zwróćmy uwagę na słowo „dowolną” w ostatnim przykładowym zadaniu; jeśli dany ciąg nie jest różnowartościowy, to istnieje więcej niż jeden poprawny sposób rozwiązania zadania.

Postawienie zadania nie jest równoznaczne z określeniem algorytmu jego rozwiązania. Zobaczmy przykład. Niech zadanie polega na obliczeniu liczby $c = a^2 - b^2$, gdzie a, b są liczbami rzeczywistymi. Algorytm „oczywisty” jest taki:

1. Oblicz $x = a^2$ (wykonując jedno mnożenie),
2. Oblicz $y = b^2$,
3. Oblicz $c = x - y$.

To samo zadanie można rozwiązać przy użyciu algorytmu opartego na algebraicznie równoważnym wzorze $c = (a+b)(a-b)$, w którym trzeba wykonać tylko jedno mnożenie. Jeśli czas potrzebny do wykonania mnożenia jest inny niż czas dodawania, to obliczenia przy użyciu obu algorytmów będą zajmować różne ilości czasu. Bardziej istotny jest w tym przypadku fakt, że liczby *rzeczywiste* są reprezentowane *w przybliżeniu*, najczęściej jako liczby *zmiennopozycyjne*, i działania arytmetyczne mogą wprowadzać błędy zaokrągleń. Ich skutkiem może być otrzymanie różnych wyników po zastosowaniu tych algorytmów i żaden z tych wyników może nie być prawdziwy. Co więcej, wyniki pośrednie (czyli liczby a^2 i b^2 albo $a+b$ i $a-b$) mogą nie mieć reprezentacji wskutek tzw. nadmiaru. Wtedy algorytm, który oblicza taki wynik pośredni może w ogóle zawieść, co oznacza, że dla różnych algorytmów rozwiązywania tego samego zadania zbiory wszystkich dopuszczalnych danych mogą być różne. Jak widać z tego przykładu, wybór algorytmu do rozwiązania zadania jest sprawą istotną i nietrywialną.

Zapis algorytmu

Opis algorytmu zawiera dwa elementy: deklaracje, czyli opis obiektów używanych przez algorytm (trzeba opisać dane wejściowe, wyniki i wszystkie dane pośrednie) oraz instrukcje, czyli opis czynności wykonywanych na tych obiektach.

Przed przystąpieniem do kodowania (czyli pisania programu w postaci, która będzie dalej przetwarzana przez komputer), jest bardzo pożądane zapisanie algorytmu w postaci bardziej zrozumiałej dla człowieka; w takiej postaci algorytm jest łatwiejszy do zbadania i do zrozumienia przez kogoś innego niż autor (a także dla autora, szczególnie po dłuższej przerwie w pracy nad programem). W tym celu można użyć języka polskiego, albo *pseudojęzyka*, będącego czymś pośrednim między językiem polskim i językiem programowania.

Deklaracje

Dane przetwarzane przez algorytmy to zmienne, czyli obiekty, które mogą przechowywać wartości określonego typu. W najprostszym przypadku zmienna przechowuje jedną wartość np. liczbową lub logiczną i wtedy deklaracja w pseudojęzyku może wyglądać tak:

całkowita `i`, `liczba_boków`; rzeczywista `obwód`, `pole`; logiczna `wypukły`.

W języku programowania C wygląda to tak:

```
int   i, liczba_bokow;
float obwod, pole, suma;
char  wypukly;
```

Każda z powyższych deklaracji wprowadza nazwę, czyli identyfikator zmiennej, oraz typ, który określa zbiór wartości, które zmienne mogą przyjmować. W języku C jest rozróżnienie wielkich i małych liter, zatem np. nazwy `nazwa`, `Nazwa` i `NAZWA` są różne. Warto ponosić trochę trudu i nadawać zmiennym nazwy znaczące (takie jak w powyższym przykładzie, z wyjątkiem `i`), bo to ułatwia ludziom (także autorowi) niesłychanie ważną czynność, jaką jest czytanie programów.

Pewne nazwy są tzw. słowami kluczowymi; w piśmie ręcznym lub drukowanym bywają podkreślone. Ich znaczenie jest ustalone i nie mogą one być w programie w C używane w żaden inny sposób.

W bardziej skomplikowanym przypadku zmienne mogą być strukturalne; zmienna taka składa się z części, które mogą być przetwarzane osobno. Na przykład zmienna tablicowa w języku C może być zadeklarowana tak:

```
float tab[11];
```

Powyższa deklaracja opisuje zmienną, która składa się z 11 zmiennych przyjmujących wartości tego samego typu: zmiennopozycyjnego (który jest używany do przybliżonego reprezentowania liczb rzeczywistych). Poszczególne elementy tablicy identyfikujemy przez podanie indeksu; jest on wyrażeniem, które w powyższym przykładzie może przyjmować wartości od 0 do 10, przy czym `tab[2]` oznacza trzeci element tablicy (w języku C zawsze pierwszy element tablicy ma indeks 0).

Jeśli w programie napisanym w języku C chcemy mieć tablicę indeksowaną od 1, to najprościej jest zadeklarować tablicę indeksowaną od zera i nie przejmować się tym, że jeden jej element jest nieużywany. Z drugiej strony, jeśli mamy tablicę indeksowaną od 1980 do 2079, to trzeba zadeklarować tablicę o długości 100 i od indeksu (przyjmującego wartości między 1980 i 2079) odejmować *w każdym odwołaniu do tablicy* dolną granicę zbioru indeksów (czyli 1980).

Instrukcje

Instrukcje mogą być proste i strukturalne. Przykładem instrukcji prostej jest instrukcja przypisania o postaci

zmienna = *wyrażenie* ;

Powyższy napis jest zdaniem metajęzyka, czyli pewnego sposobu opisywania składni języka programowania — będzie o nim mowa później. Instrukcja przypisania jest szczególnym przypadkiem tzw. instrukcji wyrażeniowej. Jej wykonanie polega na obliczeniu wartości wyrażenia z prawej strony operatora przypisania „=", a następnie nadaniu tej wartości zmiennej występującej po lewej stronie. Symbol „=" tego operatora oznacza słowa „staje się". Przykłady:

```
delta = b*b-4*a*c;
tab[i] = tab[k];
wypukly = kat <= PI;
```

Inne instrukcje proste to wywołanie podprogramu (które też jest szczególnym przypadkiem instrukcji wyrażeniowej), instrukcje skoku i instrukcja pusta; będzie o nich mowa później. Zwróćmy uwagę, że każda instrukcja prosta (w tym wszystkie powyższe przykłady) kończy się średnikiem.

Instrukcje strukturalne są takie:

Instrukcja złożona składa się z ciągu instrukcji (prostych lub strukturalnych), które mają być wykonane kolejno — każda po zakończeniu wykonywania poprzedniej. Składnia w języku C jest następująca:

{ *instrukcja instrukcja ... instrukcja* }

Instrukcję złożoną tworzy się przy użyciu tzw. nawiasów instrukcyjnych, którymi są nawiasy klamrowe. Między nimi jest ciąg instrukcji, który może być pusty.

Instrukcja warunkowa składa się z instrukcji poprzedzonej warunkiem, mającym postać wyrażenia. Instrukcja jest wykonywana wtedy, gdy warunek jest spełniony, co ma miejsce wtedy, gdy wyrażenie ma wartość niezerową. Wartość wyrażenia jest liczbą; w szczególności operatory porównywania „==" (równe), „!=" (różne), „<" (mniejsze), „>" (większe), „<=" (nie większe) i „>=" (nie mniejsze) dostarczają wartości liczbowych (całkowitych): różnej od zera jeśli wartości porównywanych wyrażeń spełniają odpowiednią relację, albo 0, jeśli nie spełniają. Składnia instrukcji warunkowej jest taka:

`if` (*wyrażenie*) *instrukcja*

Druga postać instrukcji warunkowej składa się z dwóch instrukcji poprzedzonych warunkiem. Jeśli jest on spełniony (wartość wyrażenia jest różna od zera), to wykonywana jest pierwsza instrukcja, a w przeciwnym razie druga. Składnia tej wersji instrukcji warunkowej jest taka:

`if` (*wyrażenie*) *instrukcja* `else` *instrukcja*

Opisane dalej instrukcje strukturalne służą do opisania pętli, czyli fragmentu programu wykonywanego wielokrotnie. Cechą każdego poprawnego algorytmu jest to, że powinien on zatrzymać się po wykonaniu skończenie wielu czynności. Różne instrukcje pętli mają zastosowanie w różnych sytuacjach, przy czym wybór rodzaju pętli jest kwestią wygody.

Instrukcja pętli „tak długo, jak ... wykonuj" ma następującą składnię:

`while` (*wyrażenie*) *instrukcja*

Wykonanie tej instrukcji zaczyna się od obliczenia wartości wyrażenia opisującego warunek. Jeśli warunek jest spełniony, to wykonywana jest instrukcja podana po warunku i wykonywanie instrukcji pętli zaczyna się od nowa. Niespełnienie warunku powoduje zakończenie wykonywania instrukcji pętli.

Instrukcja pętli „wykonuj ... tak długo jak" różni się od poprzedniej tym, że warunek jest podany na końcu. Składnia jest taka:

`do` *instrukcja* `while` (*wyrażenie*);

Najpierw jest wykonywana instrukcja, a potem jest sprawdzany warunek, tj. jest obliczana wartość wyrażenia. Jeśli jest niezerowa, to instrukcja jest wykonywana

ponownie. Instrukcja takiej pętli jest zatem wykonywana co najmniej raz.

Instrukcja pętli „dla" jest najbardziej użyteczna wtedy, gdy liczba iteracji (tj. wykonań instrukcji w pętli) może być określona „z góry". Najczęściej ma to miejsce podczas przetwarzania tablic (w każdym przebiegu pętli przetwarzamy jeden element tablicy), ale nie tylko. Składnia instrukcji pętli „dla" w języku C jest taka:

`for` (*wyrażenie*$_1$; *wyrażenie*$_2$; *wyrażenie*$_3$) *instrukcja*

Wykonanie pętli zaczyna się od obliczenia wartości pierwszego wyrażenia w nawiasach. Drugie wyrażenie opisuje warunek; jego wartość jest obliczana po pierwszym wyrażeniu. Jeśli jest różna od 0, to program wykonuje instrukcję, a następnie oblicza wyrażenie trzecie i ponownie drugie, w celu podjęcia decyzji o ponownym wykonaniu instrukcji, itd.

Na przykład, aby obliczyć sumę elementów tablicy, możemy napisać

```
for ( i = 0, suma = 0.0;  i < 11;  i++ ) suma += tab[i];
```

albo

```
for ( i = 10, suma = 0.0;  i >= 0;  i-- ) suma += tab[i];
```

W powyższych instrukcjach występują operatory zwiększania wartości zmiennej, „++" oraz zmniejszania wartości zmiennej, „--". Wartość zmiennej i jest za ich pomocą zwiększana lub zmniejszana o 1. Ponadto mamy tu operator przypisania „+=". Instrukcja `suma += tab[i];` jest poleceniem obliczenia sumy wartości zmiennych `suma` i `tab[i]`, a następnie przypisania tej sumy zmiennej `suma`.

Ponieważ możliwość czytania tekstu programu przez ludzi jest co najmniej tak samo ważna jak możliwość sterowania przez program pracą komputera, więc niesłychanie ważne są w programie komentarze, będące w zasadzie fragmentami ignorowanymi. Niezależnie od tego, w jakim języku programujemy, pisanie sensownych komentarzy jest ważne, ale w C jest wyjątkowo ważne, ponieważ język ten jest dla zwykłego człowieka dosyć „słabo czytelny". Komentarzem w C jest dowolny fragment tekstu, którego pierwsze dwa znaki to „/*", a ostatnie dwa znaki to „*/" i który nie zawiera wewnątrz znaków „*/" napisanych bezpośrednio po sobie.

Przykłady:

Algorytm Euklidesa znajdowania największego wspólnego dzielnika liczb naturalnych n i k oparty jest na spostrzeżeniu, że jeśli reszta z dzielenia większej z nich przez mniejszą jest dodatnia, to $\mathrm{NWD}(n, k)$ też jest jej dzielnikiem. Algorytm ten można w pseudojęzyku zapisać tak:

1. Przypisz zmiennej a nie mniejszą, a zmiennej b nie większą z liczb n i k;
2. Tak długo jak b > 0 wykonuj
   (a) Przypisz zmiennej r resztę z dzielenia a przez b;
   (b) Przypisz zmiennej a wartość zmiennej b, a zmiennej b wartość zmiennej r
3. Wartość zmiennej a jest równa $\mathrm{NWD}(n, k)$.

W języku C (zakładamy, że liczby n i k są wartościami zmiennych n i k „danymi z góry" — kompletny kod musi zawierać odpowiednie uzupełnienie):

```
int a, b, r;

if ( n > k ) { a = n;  b = k; }
        else { a = k;  b = n; }
while ( b > 0 ) {
  r = a % b;
  a = b;  b = r;
}
/* a = NWD(n,k) */
```


Rozwiązywanie równania kwadratowego. Posłużymy się wzorem z „deltą" do obliczenia pierwiastka, który ma większą wartość bezwzględną i wzoru Viète'a do obliczenia drugiego pierwiastka. W rachunkach z błędami zaokrągleń taka metoda daje dokładniejsze wyniki niż użycie wzoru z „deltą" do obliczenia obu pierwiastków. Przypomnijmy, że dane są liczby $a \neq 0$, b i c. W pseudojęzyku:

1. Oblicz $\Delta = b^2 - 4ac$.
2. Jeśli $\Delta < 0$, to nie ma pierwiastków rzeczywistych.
3. Jeśli $\Delta = 0$, to jest jeden pierwiastek, $x = -\frac{b}{2a}$.
4. Jeśli $\Delta > 0$, to
   (a) Jeśli $b > 0$, to oblicz $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$ oraz $x_2 = \frac{c}{ax_1}$,
   (b) W przeciwnym razie oblicz $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$ oraz $x_1 = \frac{c}{ax_2}$.

W języku C (tym razem też zakładamy, że zmienne a, b i c są opisane w innym miejscu i mają nadane właściwe wartości początkowe):

```
float x1, x2, delta;
int   lp;

delta = b*b-4*a*c;
if ( delta < 0 ) lp = 0;
else if ( delta == 0 ) {
  x1 = -b/(2*a);  lp = 1;
}
else {  /* tu musi być delta > 0 */
  if ( b > 0 ) {
    x1 = (-b-sqrt(delta))/(2*a);  x2 = c/(a*x1);
  }
  else {
    x2 = (-b+sqrt(delta))/(2*a);  x1 = c/(a*x2);
  }
  lp = 2;
}
/* wartość zmiennej lp jest liczbą pierwiastków rzeczywistych */
```

Sortowanie ciągu przez wstawianie. Algorytm sortowania przez wstawianie polega na następującej indukcji: przypuśćmy, że dla pewnego $i < n$ mamy początkowe i elementów ciągu ustawione w kolejności niemalejącej. Aby ustawić początkowe $i + 1$ elementów w takiej kolejności, wystarczy znaleźć właściwe miejsce między pewnymi elementami w części posortowanej i wstawić w to miejsce element i+pierwszy. Indukcję zaczynamy od $i = 1$ (elementy numerujemy od 0 do $n - 1$; ciąg jednoelementowy jest posortowany).

Algorytm w pseudojęzyku jest w przykładzie zapisany w komentarzach, umieszczonych w kodzie w języku C:

```
float a[n];
float b;
int   i, j;
   /* zakładamy, że w tablicy a są umieszczone dane do posortowania */

     /* Zaczynając od posortowanego podciągu jednoelementowego */
     /* (a[0]), kolejno dla elementów a[1], ...  a[n-1] */
for ( i = 1; i < n; i++ ) {
     /* znajdź właściwe miejsce w posortowanym podciągu; w tym celu */
     /* odłóż element a[i] na bok (skopiuj do zmiennej b) */
  b = a[i];
     /* i zaczynając od elementu a[i-1] przestaw w tablicy */
     /* elementy większe od a[i] o jedno miejsce dalej */
  for ( j = i-1; j >= 0 && a[j] > b; j-- )
    a[j+1] = a[j];
     /* wstaw element i-ty początkowego ciągu w miejsce */
     /* z którego został przestawiony ostatni wykryty element */
     /* większy od niego */
  a[j+1] = b;
}
```


Zadania i problemy na ćwiczenia i do domu

1. Jaka będzie wartość zmiennej a typu int po wykonaniu instrukcji

```
if ( 2009 )
  a = 1;
else
  a = 2;
```

2. Napisz kod w C równoważny instrukcji pętli

```
for ( i = 99; i >= 0; i-- )
  a[i] = 99-i;
```

   przy użyciu instrukcji pętli „tak długo jak ... wykonuj” oraz „wykonuj ... tak długo jak”.

3. Napisz w C instrukcję pętli, która w danej tablicy a przestawia elementy tak, aby odwrócić ich kolejność.

4. *Tablice dwuwymiarowe* w C deklaruje się na przykład tak:

```
float tab[m][n];
```

   przy czym nazwy `m` i `n` powinny oznaczać liczby (dokładniej: wyrażenia o wartościach całkowitych dodatnich, określonych w tekście programu)

   Odwołanie do elementu w i-tym wierszu i j-tej kolumnie ma postać `tab[i][j]`. Napis `tab[i]` oznacza cały wiersz macierzy. Analogicznie można deklarować i używać macierze o więcej niż dwóch indeksach.

   Przypuśćmy, że tablica `tab` reprezentuje macierz o wymiarach $m \times n$. Napisz instrukcję, która

   a) Produkuje tabliczkę mnożenia.
   b) Przestawia wiersze tak, aby odwrócić ich kolejność.
   c) Przestawia kolumny tak, aby odwrócić ich kolejność.
   d) Przy założeniu, że $m = n$ (czyli że macierz jest kwadratowa) dokonuje transpozycji macierzy (macierz transponowana ma być przechowywana w tablicy `tab` zamiast macierzy danej).

5. Napisz w C algorytm sortowania przez wybieranie; algorytm ten polega na znalezieniu w tablicy elementu najmniejszego i przestawieniu go z pierwszym. Następnie należy znaleźć element najmniejszy wśród pozostałych i przestawić go z drugim itd.

Podprogramy

Pewne fragmenty programu rozwiązują zadania, które mogą wystąpić wielokrotnie w programie. Na przykład pod nazwą sqrt, której użyliśmy do rozwiązywania równania kwadratowego, kryje się polecenie obliczenia pierwiastka kwadratowego. Polecenie to jest wykonywane przez tzw. podprogram znajdujący się w standardowej tzw. bibliotece matematycznej. Podprogram ten może być użyty także w każdym innym celu, do którego mógłby się przydać.

Sposób (czyli algorytm) obliczania pierwiastka jest z punktu widzenia zadania nieistotny, pod warunkiem, że podprogram działa dostatecznie szybko i wynik jest dostatecznie dokładny. Warto tu ponownie spojrzeć na definicję algorytmu: są w niej słowa „zrozumiały dla wykonawcy". Polecenie obliczenia pierwiastka (poprzez wywołanie podprogramu sqrt) jest zrozumiałe dla wykonawcy tak samo jak polecenia wykonywania działań arytmetycznych, oznaczone symbolami „+", „-", „*", „/".

Zwróćmy uwagę, że w przypadku podprogramu sqrt dla programu, który go wywołuje, nieistotny jest sposób obliczania pierwiastka, ale również dla podprogramu jest nieistotny cel, dla którego on został wywołany. W programach dłuższych niż 10 linii kodu prawie zawsze można wyodrębnić zadania, których rozwiązanie nie zależy od pozostałych elementów programu. Algorytmy rozwiązywania takich podzadań dobrze jest zaimplementować w postaci podprogramów. Korzyści z tego są następujące:

- Możemy się skupić na podzadaniu i napisać kod, którego działanie zależy tylko od danych istotnych w tym podzadaniu.
- Możemy raz napisać procedurę rozwiązywania podzadania, a w pozostałym kodzie umieścić wywołania procedury. Nazwa procedury staje się w ten sposób kolejnym poleceniem zrozumiałym dla wykonawcy, a jeśli ta nazwa jest znacząca (np. ObliczPierwiastki albo Posortuj), to program jest znacznie bardziej czytelny (bo nie trzeba za każdym razem zgadywać, co robi kod realizujący dany algorytm).
- Możemy tworzyć biblioteki podprogramów do zastosowania w wielu programach, a nie tylko w jednym. Możemy też korzystać z podprogramów i bibliotek napisanych przez kogoś innego.
- Możemy zmodyfikować podprogram lub zamienić go na inny, bez zmieniania reszty programu (może być ewentualnie potrzebne zmienienie tylko nazwy podprogramu w wywołaniach).
- Możemy wreszcie realizować algorytmy rekurencyjne, o których mowa dalej.

Przykład: Napiszemy program rozwiązujący równanie nieliniowe $\frac{1}{2}x = \sin x$, tj. obliczający miejsce zerowe funkcji $f(x) = \frac{1}{2}x - \sin x$ w przedziale $[1, 3]$.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

float f ( float x )
{
  return 0.5*x - sin(x);
} /*f*/

void Bisekcja ( float a, float b, float eps, float *x, char *error )
{
  float c, fa, fb, fc;

  fa = f(a);
  fb = f(b);
  if ( fa*fb <= 0.0 && eps > 0.0 ) {
    while ( fabs(b-a) > eps ) {
      c = 0.5*(a+b);
      fc = f(c);
      if ( fa*fc <= 0.0 ) b = c;
                     else a = c;
    }
    *x = a;
    *error = 0; /* 0 oznacza fałsz, w tym przypadku brak błędu */
  }
  else
    *error = 1; /* wartość różna od 0 oznacza prawdę, czyli tu błąd */
} /*Bisekcja*/

int main ( void )
{
  float x;  char err;

  Bisekcja ( 1.0, 3.0, 1.0e-6, &x, &err );
  if ( err ) printf ( "Blad\n" );
        else printf ( "rozwiazanie = %f\n", x );
  exit ( 0 );
} /*main*/
```

W powyższym programie są trzy podprogramy, a oprócz tego wykorzystaliśmy kilka podprogramów standardowych — `sin` (oblicza wartość sinusa), `fabs` (oblicza wartość bezwzględną), `printf` (wypisuje tekst) i `exit` (kończy wykonanie programu).

Pierwszy z napisanych podprogramów oblicza wartość funkcji f, której miejsce zerowe chcemy obliczyć (co jest równoważne rozwiązaniu równania). Wartość funkcji jest wartością wyrażenia występującego po słowie kluczowym `return`. Metoda numeryczna podczas działania będzie wywoływać podprogram `f`, aby otrzymać wartość funkcji f dla różnych argumentów, przy czym nie można „z góry” określić dla jakich, z wyjątkiem tego, że będą one w odpowiednim przedziale. Zauważmy, że gdybyśmy chcieli rozwiązać inne równanie nieliniowe z jedną niewiadomą, to wystarczyłoby zmienić tylko ten podprogram.

Drugi podprogram realizuje metodę bisekcji. Metoda ta jest oparta o twierdzenie (Bolzano-Weierstrassa), że jeśli funkcja rzeczywista jest ciągła w przedziale $[a, b]$ i przyjmuje na jego końcach wartości o przeciwnych znakach, to ma w tym przedziale miejsce zerowe. Procedura rekurencyjnie dzieli przedział na połowy i wybiera tę z nich, na końcach której funkcja f ma różne znaki; koniec działania następuje z chwilą otrzymania dostatecznie krótkiego przedziału, co jest równoznaczne ze znalezieniem rozwiązania z określoną dokładnością. Zauważmy, że gdybyśmy chcieli ulepszyć lub zmienić metodę numeryczną, to wystarczyłoby zmienić tylko ten podprogram (ale przy zmianie metody wypadałoby zmienić nazwę podprogramu, w nagłówku procedury i w miejscu wywołania).

Trzeci podprogram, o nazwie `main`, wywołuje podprogram `Bisekcja`, a następnie wypisuje wynik obliczeń. W każdym programie w C musi być jeden podprogram o takiej nazwie (napisanej małymi literami); pełni on rolę `programu głównego`. Wykonanie programu zaczyna się od wywołania (przez system operacyjny) podprogramu `main` i kończy się po wykonaniu jego ostatniej instrukcji, lub wskutek innego zdarzenia wymuszającego zakończenie, na przykład wywołania (przez dowolny podprogram) podprogramu `exit`.

Takim innym zdarzeniem może też być tzw. błąd wykonania, powodujący przerwanie działania przez system operacyjny, który pewnych sytuacji spowodowanych przez program nie toleruje. Ale nie każdy błąd wykonania powoduje taki skutek.

Budowa podprogramu

Każdy podprogram (procedura)[1] w C składa się z nagłówka i bloku. Na nagłówek składa się nazwa procedury, poprzedzona określeniem typu wyniku i lista parametrów. Informacje w nagłówku są konieczne i w zasadzie wystarczające do tego, by można było procedurę wywoływać.

Nazwa procedury (i nie tylko) jest ciągiem liter i cyfr (pierwsza musi być litera, przy czym litery małe i wielkie są rozróżniane, a poza tym znak podkreślenia „_” jest uznawany za literę).

Typ procedury identyfikuje zbiór, do którego należy jej wartość (a także sposób reprezentowania tej wartości). Na przykład procedura `f` w przykładzie ma typ `float`, który oznacza pewien podzbiór zbioru liczb rzeczywistych. Procedura `Bisekcja` ma typ `void`, który oznacza zbiór pusty. Informacja o wynikach obliczeń procedury typu `void` jest przekazywana tylko za pomocą parametrów lub tzw. efektów ubocznych, o których będzie mowa później.

Lista parametrów opisuje sposób przekazywania informacji między podprogramem, którego to jest lista i podprogramem wywołującym. W nawiasach okrągłych po nazwie procedury podaje się typy i nazwy kolejnych parametrów formalnych; parametry te są zmiennymi, które w chwili rozpoczęcia wykonywania procedury mają nadane (w miejscu wywołania podprogramu) wartości początkowe. Jeśli procedura nie ma parametrów, to jej lista parametrów składa się tylko z pary nawiasów `()`; można też (a nawet jest to wskazane) napisać `(void)`.

Blok procedury jest otoczony nawiasami klamrowymi; składa się on z lokalnych deklaracji oraz z ciągu instrukcji, które realizują odpowiedni algorytm. Deklaracje lokalne opisują obiekty (typy i zmienne) widoczne tylko dla instrukcji w tym bloku.

Zmienne lokalne i parametry

Zmienne lokalne (w tym parametry) przed wywołaniem procedury, a także po zakończeniu jej wykonywania, nie istnieją, a dokładniej, żadne miejsce w pamięci operacyjnej komputera nie jest zarezerwowane na te zmienne. W chwili wywołania

[1]Będziemy używać słowa „procedura”, choć oficjalna terminologia języka C określa wszystkie podprogramy słowem „funkcja”. Powodem tej decyzji jest częste używanie dalej słowa „funkcja” w znaczeniu przyjętym w matematyce. Słowa „funkcja” będziemy używać na określenie podprogramów obliczających wartości funkcji, np. takich jak `sqrt`.

procedury jest rezerwowany odpowiedni obszar pamięci, który staje się jej tzw. rekordem aktywacji. Wszelkie zmienne lokalne są przechowywane w tym obszarze. Rekord aktywacji jest likwidowany po zakończeniu działania podprogramu, dzięki czemu ten sam obszar pamięci może być użyty do przechowania rekordu aktywacji podprogramu wywołanego później. Wskutek tego, jeśli procedura jest wywoływana wielokrotnie, to żadna informacja w zmiennych lokalnych *nie może* być zapamiętana do czasu kolejnego wywołania[2]. Oczywiście, rekord aktywacji podprogramu nie znika podczas wykonywania innych podprogramów wywołanych przez ten podprogram.

Parametry są potrzebne do przekazywania informacji od procedury wywołującej do wywoływanej, a także w drugą stronę. Na przykład, parametry `a`, `b` i `eps` zawierają dane dla procedury `Bisekcja`, opisujące przedział, w którym ma być poszukiwane rozwiązanie równania, oraz maksymalny błąd, z jakim należy to rozwiązanie wyznaczyć. W miejscu wywołania procedury `Bisekcja` w procedurze `main` (w tzw. liście parametrów aktualnych) jako wartości tych parametrów są podane liczby 1, 3 i 10^{-6}.

Znalezione metodą bisekcji rozwiązanie jest przekazywane procedurze `main` za pomocą parametru `x`, podobnie informacja o tym, czy został wykryty błąd, jest przekazywana za pomocą parametru `error`. Aby przekazanie informacji w tę stronę było możliwe, parametry te są tzw. wskaźnikami: ich wartości są adresami (w pamięci operacyjnej) zmiennych, którym należy przypisać odpowiednie wartości. Zmienna wskaźnikowa ma w deklaracji nazwę poprzedzoną gwiazdką (np. `float *x`). Instrukcja przypisania `*x = a;` jest poleceniem „wartość zmiennej `a` przypisz zmiennej typu `float` wskazywanej przez parametr `x`". W liście parametrów aktualnych mamy wyrażenia „`&x`" i „`&err`". Symbol „`&`" jest operatorem adresu — jego wartością jest adres występującej po nim zmiennej. W ten sposób procedurze `Bisekcja` została podana informacja o miejscach, w których mają zostać umieszczone wyniki obliczeń.

Jeśli pewien wynik obliczeń jest wyróżniony, to zamiast parametru wskaźnikowego możemy użyć wartości procedury (funkcji). W tym celu przed nazwą procedury w nagłówku należy napisać wybrany typ (inny niż `void`). Wartością procedury jest obliczona wartość wyrażenia w instrukcji powrotu (po słowie kluczowym `return`). Wywołanie takiej procedury może być częścią większego wyrażenia (wartość procedury stanie się argumentem dalszych działań występujących w tym wyrażeniu — zobacz wywołanie funkcji `sin` w przykładzie).

[2] Można w tym celu użyć tzw. lokalnych zmiennych statycznych, o których nie będzie tu mowy dla uniknięcia komplikacji.

Jeśli procedura nie ma parametrów, to poleceniem jej wywołania jest jej nazwa, po której występuje pusta lista parametrów aktualnych `()`.

Pliki nagłówkowe

W języku C przed *użyciem* nazwy (w dowolnej deklaracji lub instrukcji), należy określić jej znaczenie; jedyny wyjątek dotyczy procedur — jeśli po nazwie, której znaczenie nie zostało wcześniej podane, występuje lista parametrów, to przyjmowane jest założenie, że nazwa ta oznacza procedurę typu `int` (którą należy wywołać), dzięki czemu program daje się skompilować, czasem nawet poprawnie.

Nie zawsze jest możliwe uporządkowanie procedur w takiej kolejności, aby każda procedura poprzedziła wszystkie instrukcje, w których jest wywoływana. Ponadto teksty źródłowe dużych programów są dzielone na wiele małych plików (w przypadku skrajnym, wcale nie bezsensownym, każda procedura może być zapisana w osobnym pliku), przy czym procedury w bibliotekach są „gotowe" i ich kod w C może być w ogóle niedostępny. Potrzebne do wywołania informacje na temat procedury są zawarte w jej nagłówku. Aby je podać, możemy napisać sam nagłówek procedury, zastępując jej blok znakiem średnika. Taki nagłówek nazywa się prototypem procedury.

Nagłówki standardowych procedur bibliotecznych, takich jak `sqrt`, `sin`, `printf` itd. są podane w tzw. plikach nagłówkowych. Oprogramowanie umożliwiające kompilację programów w C zawiera wiele takich plików, z których każdy dotyczy pewnej „grupy tematycznej" procedur. Na przykład funkcje matematyczne (`sqrt`, `sin`, `fabs` itd.) są opisane w pliku o nazwie `math.h`. Standardowe procedury wejścia/wyjścia (umożliwiające m.in. wypisywanie tekstów na ekran) są opisane w pliku `stdio.h`. Plik `stdlib.h` zawiera nagłówki procedur `exit`, `abs` i innych. Tradycyjnie używane rozszerzenie „`.h`" nazwy plików nagłówkowych pochodzi oczywiście od angielskiego słowa *header*.

Pierwszym etapem kompilacji programu w C jest użycie tzw. preprocesora, który rozpoznaje w tekście pewne napisy i je odpowiednio przetwarza. Napis `#include`, po którym występuje nazwa pliku (w nawiasach kątowych, np. `<stdio.h>`, albo w podwójnych apostrofach, np. `"stdio.h"`) jest dla preprocesora poleceniem *wstawienia* w to miejsce pliku o podanej nazwie. W ten sposób nie musimy własnoręcznie przepisywać nagłówków wykorzystywanych w programie procedur bibliotecznych.

Widoczność i zasłanianie

Zmienne globalne są to zmienne zadeklarowane poza procedurą. Zmienna globalna (nie-statyczna) jest dostępna dla wszystkich procedur programu, przy czym aby była dla nich widoczna (zwłaszcza dla procedur znajdujących się w innych plikach), powinna być odpowiednio opisana w pliku nagłówkowym (szczegóły tu pominiemy). W ten sposób dowolna procedura ma dostęp do wszystkich *swoich* zmiennych lokalnych, wszystkich zmiennych globalnych i wszystkich procedur. Jeśli jednak w procedurze zadeklarujemy zmienną (lokalną), która ma tę samą nazwę, co pewna zmienna globalna lub inna procedura, to ta zmienna lokalna zasłoni obiekt zadeklarowany na zewnątrz.

Efekty uboczne

Efekt uboczny podprogramu jest to każdy zauważalny skutek jego działania inny niż przekazanie wyniku przez wartość procedury lub za pomocą parametrów. Na przykład efektem ubocznym jest przypisanie wartości zmiennej globalnej przez procedurę.

Przekazywanie informacji w inny sposób niż dwa sposoby wymienione wyżej jest trudno (czasem bardzo trudno) zauważalne podczas czytania programu, a ponadto utrudnia wykorzystywanie podprogramów do celów innych niż początkowo przyjęte. Dlatego należy go unikać. Na przykład, można by zadeklarować zmienne `x` i `err` jako globalne (zamiast lokalne w procedurze `main`) i w procedurze `Bisekcja` napisać instrukcję przypisania wartości wyników bezpośrednio do tych zmiennych (zamiast do zmiennych wskazywanych przez parametry). Gdyby ktoś chciał przenieść podprogram `Bisekcja` do innego programu, musiałby również w nim zadeklarować takie zmienne. Niemożliwe przy tym byłoby wywołanie procedury w kilku miejscach programu, ze wskazaniem za każdym razem innych zmiennych, którym należałoby przypisać wyniki.

Efekty uboczne bywają jednak niezastąpione. Na przykład, efektem ubocznym procedury `printf` jest ukazanie się odpowiedniego tekstu na ekranie.

Rekurencja

Często rozwiązanie pewnego problemu dla parametru $n > 0$ daje się sprowadzić do rozwiązania tego samego problemu dla mniejszych wartości tego parametru. Na przykład, przypuśćmy, że mamy obliczyć współczynnik dwumianowy $\binom{n}{k}$ dla ustalonych liczb dodatnich n, k, gdzie $0 \le k \le n$. Jeden z wielu sposobów

(nienajlepszy, ale w tej chwili to nieistotne) polega na bezpośrednim użyciu wzoru rekurencyjnego (na jego podstawie opiera się schemat znany jako trójkąt Pascala):

$$\binom{n}{k} = \begin{cases} 1 & \text{dla } k = 0 \text{ lub } k = n, \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{w przeciwnym razie.} \end{cases}$$

Na podstawie tego wzoru możemy napisać procedurę obliczania współczynnika dwumianowego:

```
int Dwumian ( int n, int k )
{
  if ( k == 0 || k == n ) return 1;
    else return Dwumian ( n-1, k-1 ) + Dwumian ( n-1, k );
} /*Dwumian*/
```

Zauważmy, że rekurencyjne wywołanie podprogramu musi zależeć od pewnego warunku, który w pewnym momencie musi być niespełniony — w przeciwnym razie obliczenie nigdy się nie zakończy (a ściślej biorąc, zakończy się bardzo szybko — po utworzeniu bardzo dużej liczby rekordów aktywacji dla poszczególnych wywołań rekurencyjnych podprogramu zabraknie pamięci, w której mogą być tworzone takie rekordy, a system operacyjny w takich sytuacjach spełnia rolę ochroniarza).

Program źródłowy, kod maszynowy i kompilator

Zapis algorytmu w pseudojęzyku jest zrozumiały dla autora i dla niektórych czytelników, ale nie dla komputera; niemniej bywa niezastąpiony na etapie projektowania programu. Z drugiej strony kod maszynowy jest dla ludzi bardzo nieczytelny, ale program w postaci kodu istnieje po to, aby procesor z jak największą szybkością wykonywał zawarte w nim polecenia, bez zastanawiania się nad ich sensem. Język programowania jest w połowie drogi między pseudojęzykiem i kodem maszynowym. Jest on zaprojektowany tak, aby ludzie mogli czytać i możliwie łatwo rozumieć napisany w nim kod źródłowy programu, a jednocześnie aby dał się przetłumaczyć na kod maszynowy.

Zadanie tłumaczenia wykonuje program zwany kompilatorem. Oprócz produkowania kodu maszynowego, jego zadaniem jest wykrywanie błędów w programie, a także fragmentów niejasnych, podejrzanych o to, że są błędne (kompilator wypisuje wtedy ostrzeżenia, ale kod maszynowy produkuje dalej, chyba, że ma polecenie przerwać kompilację po pierwszym błędzie lub ostrzeżeniu).

Następne zadanie, wykonywane przez tzw. linker (zwany też konsolidatorem), który jest procedurą kompilatora albo osobnym programem, polega na połączeniu w całość procedur (które mogą być skompilowane osobno) i dołączeniu do programu procedur bibliotecznych. Procedury takie mogą stanowić większą część programu gotowego do wykonania.

Jedną z najważniejszych zalet pisania programów w językach programowania takich jak Pascal lub C jest unikanie pisania kodu maszynowego. Program napisany bezpośrednio w takim kodzie lub (co na jedno wychodzi w tzw. języku asemblera) może być wykonany tylko przez procesor, z którego rozkazów się składa. Tymczasem kod źródłowy może być skompilowany przez różne kompilatory, produkujące kod maszynowy dla różnych procesorów. Przy tym, wprawdzie można „optymalizować kod”, tj. pisać go tak, aby składał się z minimalnej liczby rozkazów, odpowiednio wykorzystywał rejestry procesora itd. Jest to jednak działalność trudna, pracochłonna i podatna na błędy, a ponadto kompilatory też nieźle optymalizują kod, nie robiąc przy tym błędów. Dlatego programowanie w kodzie maszynowym ma sens jedynie tam, gdzie nie można go uniknąć: w bezpośredniej obsłudze sprzętu (np. urządzeń graficznych, dźwiękowych, sieciowych i innych). Jeśli ktoś nie jest producentem takiego sprzętu, to raczej nie ma powodu, aby to robić.

Zadania i problemy

1. Napisz w C podprogramy znajdowania największego wspólnego dzielnika, rozwiązywania równania kwadratowego i sortowania tablicy, oparte na algorytmach podanych na poprzednim wykładzie. Istotne jest napisanie poprawnych nagłówków, a w szczególności wybranie dla każdego podprogramu typu wyniku i sposobu przekazywania poszczególnych parametrów.
2. Napisz procedury interakcyjnego czytania danych dla powyższych trzech zadań i procedury wypisywania wyników. Następnie użyj ich w programach rozwiązujących te zadania. Zadbaj o to, aby wszystkie informacje były przekazywane za pomocą parametrów i aby procedury wykonywały tylko swoje zadania (tj. na przykład procedura sortowania nie może czytać danych ani pisać wyników; podobnie procedura czytania danych nie może „wiedzieć” nic o sortowaniu itp.).
3. Napisz procedurę obliczającą miejsce zerowe funkcji f, określonej za pomocą odpowiedniego podprogramu, przy użyciu metody siecznych; na podstawie dwóch przybliżeń rozwiązania, x_{k-1} i x_k oraz wartości funkcji f dla tych argumentów, następne przybliżenie w tej metodzie otrzymuje się ze wzoru
$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} f(x_k).$$
Procedura ma mieć taką samą listę parametrów jak procedura `Bisekcja` podana na wykładzie. Zidentyfikuj sytuacje, które prowadzą do błędu obliczeń. Procedura ma wykrywać te sytuacje i sygnalizować je przy użyciu parametru `error`.
4. Napisz procedurę, która metodą bisekcji znajduje miejsce w tablicy zawierającej niemalejący ciąg liczb całkowitych (tj. odpowiedni indeks), na którym w tablicy znajduje się liczba y (dana jako parametr) lub ostatnia liczba mniejsza niż y.
5. Napisz procedurę rekurencyjną rozwiązującą problem wież Hanoi; ma ona wypisać odpowiedni ciąg instrukcji przenoszenia krążków.

Języki programowania

Przykłady programów i ich fragmentów przedstawione wcześniej miały na celu pokazanie „jak to wygląda”, natomiast aby pisać własne programy należy przestrzegać pewnych reguł formalnych. Chodzi o to, aby algorytm zrozumiały (miejmy nadzieję) dla autora i innych osób był także zrozumiały dla maszyny, która go będzie wykonywać.

Na opis języka programowania składają się dwa elementy:

- Składnia, czyli opis budowy konstrukcji językowych,
- Semantyka, czyli opis znaczenia poszczególnych konstrukcji.

Opis składni jest łatwiejszy do sformalizowania; wprowadzamy pewien zbiór pojęć gramatycznych, reprezentowanych przez symboliczne nazwy; w gramatykach języków używanych przez ludzi takimi pojęciami są *zdanie*, *podmiot*, *orzeczenie*, *przydawka*, *dopełnienie*, *okolicznik* itd. Zaczynając od symbolu *zdanie*, który reprezentuje cel analizy danego tekstu, możemy ten tekst *wyprowadzić*, stosując tzw. produkcje gramatyki, czyli reguły zastępowania jednych symboli przez inne. Na przykład symbol *zdanie* możemy zastąpić ciągiem symboli *podmiot orzeczenie*. Następnie *podmiot* możemy zastąpić przez ciąg *rzeczownik przydawka*, itd., na końcu dochodząc do słów danego tekstu.

Choć takie „mechaniczne” postępowanie wobec języków „mówionych” jest nie całkiem skuteczne, ale tzw. języki formalne, w tym języki, w których pisze się programy komputerowe, są projektowane w taki sposób, aby powyżej opisany sposób ich analizowania działał dobrze. Podobnie, jak w języku mówionym, analiza przebiega na dwóch poziomach: na pierwszym wyodrębniamy słowa, reprezentowane przez pewne ciągi liter tekstu. Każde takie słowo możemy zakwalifikować do jednej z grup: *rzeczownik*, *czasownik*, *przymiotnik*, *przysłówek* itp. Rozbiór gramatyczny zdania następuje na poziomie drugim, gdzie nie zajmujemy się już poszczególnymi literami słów. Podczas kompilacji programu znaki jego tekstu są najpierw grupowane w tzw. symbole leksykalne (albo leksemy, ang. *tokens*, czyli dosłownie żetony). Analiza składni zdania w języku programowania (czyli na przykład programu) jest dokonywana na podstawie ciągu symboli leksykalnych.

Symbole leksykalne w języku C

Symbole leksykalne w języku C dzielą się na sześć grup: identyfikatory, słowa kluczowe, stałe, napisy, operatory i separatory. Rozpoznając symbole leksykalne kompilator stara się utworzyć *jak najdłuższe* ciągi znaków; jeśli w programie występuje na przykład ciąg cyfr, to reprezentuje on jedną stałą — liczbę wielocyfrową, a nie wiele liczb jednocyfrowych.

Identyfikator jest niepustym ciągiem znaków, z których pierwszy jest literą (alfabetu angielskiego), a każdy następny jest literą lub cyfrą, przy czym ciąg ten nie jest żadnym słowem kluczowym. Litery małe i wielkie, np. „a” i „`A`” w identyfikatorach są uważane za różne. Ponadto za literę w C uważa się znak podkreślenia „_”.

Słowo kluczowe jest jednym z następujących ciągów liter:

`auto` `break` `case` `char` `continue` `const` `default` `do` `double` `else` `enum` `extern` `float` `for` `goto` `if` `int` `long` `register` `return` `short` `signed` `sizeof` `static` `struct` `switch` `typedef` `union` `unsigned` `void` `volatile` `while`

Pisząc tekst programu na papierze, słowa kluczowe zwykle wyróżnia się za pomocą podkreślenia. Edytory tekstowe używane do pisania progamów w C mogą słowa kluczowe wyświetlać na ekranie w wyróżnionym kolorze. Pewne identyfikatory nie znajdujące się na powyższej liście mogą być słowami kluczowymi dla pewnych kompilatorów[3].

Stałe są liczbowe, znakowe i wyliczeniowe; wśród stałych liczbowych mamy liczby całkowite i zmiennopozycyjne, zapisywane najczęściej w układzie dziesiętnym[4]. Ciąg cyfr dziesiętnych, np. `137`, reprezentuje stałą całkowitą. Stała zmiennopozycyjna składa się z ciągu cyfr reprezentującego część całkowitą, po której jest część ułamkowa i/lub wykładnik, np: `3.14`, `1e-6` lub `1.05e6`. Część ułamkowa jest od części całkowitej oddzielona kropką. Wykładnik (zaczynający się od litery `e` lub `E`) określa potęgę liczby 10, przez którą należy pomnożyć liczbę występującą przed wykładnikiem.

[3] Poprawny program w C jest w zasadzie również poprawnym programem w języku C++, pod warunkiem, że nie występują w nim identyfikatory, które są słowami kluczowymi C++ (dodatkowe ograniczenie to tzw. ścisła typizacja w C++, która wymaga stosowania jawnej konwersji typu wskaźników — w tej chwili to nieistotne). Słów kluczowych w C++ jest dużo więcej niż w C. Jeśli edytor wyświetla słowa kluczowe w specjalnym kolorze, to zwykle wyświetlane są w ten sposób wszystkie słowa kluczowe C++.

[4] Możliwość stosowania układów ósemkowego i szesnastkowego odnotujmy, bez wchodzenia w szczegóły.

Stałe znakowe są znakami w apostrofach, np. `'A'`, `'!'`. Stałe znakowe są zamieniane na liczby, prawie zawsze jest w tym celu stosowany tzw. kod ASCII (ang. *American Standard Code for Information Interchange*), w którym np. literze `A` odpowiada liczba 65 itd. Zbiór znaków zawiera też różne znaki specjalne, które można reprezentować za pomocą odwrotnego ukośnika. Na przykład `'\n'` oznacza znak przejścia do nowej linii, `'\t'` — tabulator, `'\b'` — cofnięcie, `'\a'` — dzwonek itp.

Stałe wyliczeniowe są identyfikatorami. Podczas kompilacji identyfikatory te są zamieniane na liczby całkowite, ale dla czytelności programu ważna jest treść identyfikatora, a nie reprezentująca go liczba. Na przykład możemy napisać w programie

```
enum kolory {czerwony, zolty, zielony, turkus, niebieski, fioletowy};
```

a następnie napisać instrukcję

```
if ( kolor == czerwony ) ...
```

w której odpowiednie działanie jest podejmowane, jeśli wartością zmiennej `kolor` jest `czerwony`, co dla osoby czytającej program jest bardziej znaczące niż fakt, że dla komputera `czerwony` to `0`.

Napisy to ciągi znaków w podwójnych apostrofach, np. `"napis"`. Napisy są reprezentowane jako tablice, przy czym długość tablicy jest o 1 większa od liczby jego znaków (nie licząc apostrofów). Kolejne znaki są liczbami (w kodzie ASCII), a na dodatkowym miejscu w tablicy jest umieszczane zero, które oznacza koniec napisu. Taka reprezentacja tekstów jest nazywana ASCIIZ.

Operatory są zapisywane przy użyciu znaków specjalnych innych niż litery i cyfry, i w większości służą do tworzenia wyrażeń. Pełna lista operatorów jest podana dalej.

Separatory to ciągi znaków istotne dla czytelności programu, w pewnych miejscach konieczne dla rozdzielenia innych symboli, a poza tym ignorowane przez kompilator. Separatorami są spacje, znaki tabulacji, znaki końca linii i komentarze, czyli dowolne teksty zawarte między parami znaków `/*` i `*/`. Na przykład między symbolami `do` i `x` w instrukcji `do x = f(x); while ( x > 0 );` spacja lub inny separator jest konieczny, aby kompilator nie odczytał identyfikatora `dox`.

Składnia języka C

Nie będzie tu pełnej składni języka C; osoby zainteresowane znajdą ją w książce Kerninghana i Ritchiego, natomiast na wykładzie przyjrzymy się sposobowi jej zapisywania. W zapisie występują symbole leksykalne, które pojawiają się w tekście programu, oraz symbole odpowiadające abstrakcyjnym pojęciom (analogicznym do pojęć *zdanie*, *podmiot* itd. w języku polskim), które „wyprowadzając” tekst programu w C należy zastąpić przez inne symbole (w końcowym efekcie tekst będzie się składał tylko z symboli leksykalnych). Punktem wyjścia jest *jednostka-tłumaczenia*. Mamy dla niej produkcje

jednostka-tłumaczenia:
  deklaracja-zewnętrzna
  jednostka-tłumaczenia deklaracja-zewnętrzna

Odczytujemy to tak: symbol *jednostka-tłumaczenia* może być zastąpiony przez ciąg symboli podany w jednej z występujących poniżej linii. Może to więc być jedna *deklaracja-zewnętrzna* lub też *jednostka-tłumaczenia*, po której występuje *deklaracja-zewnętrzna*. Zauważmy, że druga z podanych produkcji jest formułą rekurencyjną. Z obu produkcji wynika, że poprawne zdanie w języku C składa się z jednej lub większej liczby fragmentów, z których każdy jest jedną *deklaracją-zewnętrzną*.

Symboli takich jak *jednostka-tłumaczenia*, czy *deklaracja-zewnętrzna* jest w sumie 65, dla każdego z nich jest od jednej do kilkunastu produkcji, tj. alternatywnych sposobów zastępowania tego symbolu. Przedstawione niżej są tylko niektóre produkcje, przy czym część z nich jest uproszczona, bo nie chodzi tu o szczegóły, tylko o zasadę definiowania składni. Każdy symbol ma określone znaczenie, które opisuje się słowami.

deklaracja-zewnętrzna:
  definicja-funkcji
  deklaracja

Deklaracja-zewnętrzna jest to podprogram lub deklaracja np. zmiennych nie umieszczona wewnątrz podprogramu.

Symbol *definicja-funkcji* oznacza podprogram, którego budowę określają dalsze reguły gramatyczne. Symbol *deklaracja* może oznaczać m.in. deklarację zmiennych. Zatem, program w C może składać się z podprogramów i deklaracji zmiennych, występujących w kolejności, która z punktu widzenia składni może być dowolna, ale ponieważ znaczenie każdego identyfikatora powinno być określone w miejscu jego użycia, więc np. deklaracje zmiennych napiszemy przed wyrażeniami, w których zmienne te występują. W ten sposób kolejność *deklaracji-zewnętrznych* jest podporządkowana znaczeniu (semantyce) poszczególnych części programu.

definicja-funkcji:
  specyfikatory-deklaracji$_{opc}$ *deklarator* *instrukcja-złożona*
  ...

Indeks $_{opc}$ oznacza, że dany symbol można pominąć (co skraca zapis gramatyki), natomiast ... oznacza inne możliwości (tj. produkcje), które pominąłem. Dalej mamy produkcje

specyfikatory-deklaracji:
  specyfikator-typu
  ...

specyfikator-typu: jeden z
  `void` `char` `short` `int` `long` `float` `double` `signed` `unsigned` ...

Specyfikator-typu określa typ wyniku zwracanego jako wartość podprogramu. Ma on zasadnicze znaczenie podczas obliczania wartości wyrażenia, w którym występuje wywołanie podprogramu. *Specyfikator-typu* może w definicji funkcji się nie pojawić — kompilator przyjmuje wtedy, że jest to `int`.

deklarator:
  bezpośredni-deklarator
  ...

bezpośredni-deklarator:
  identyfikator (*lista-typów-parametrów*)
  ...

Z podanych wyżej produkcji wynika możliwość zapisania nagłówka podprogramu złożonego ze *specyfikatora-typu* i *deklaratora*, który składa się z *identyfikatora* i listy parametrów (w nawiasach) — w programie omawianym na poprzednim wykładzie mamy tego przykłady. Aby nie obciążać zanadto pamięci, zobaczmy jeszcze tylko składnię instrukcji.

instrukcja:
  instrukcja-etykietowana
  instrukcja-wyrażeniowa
  instrukcja-złożona
  instrukcja-wyboru
  instrukcja-powtarzania
  instrukcja-skoku

instrukcja-wyrażeniowa:
  wyrażenie$_{opc}$;

Wykonanie instrukcji wyrażeniowej polega na obliczeniu wartości wyrażenia (i odrzuceniu jej). Jego skutkiem są wszelkie efekty uboczne tego wyrażenia, które na przykład może być wywołaniem podprogramu. Wyrażenia i ich efekty uboczne omówimy za chwilę. Jeśli wyrażenie jest nieobecne (jest tylko średnik), to instrukcja wyrażeniowa jest instrukcją pustą — nic nie robi, ale jest w języku programowania tak potrzebna jak 0 w zbiorze liczb całkowitych.

instrukcja-skoku:
  `goto` *identyfikator* ;
  `continue` ;
  `break` ;
  `return` *wyrażenie*$_{opc}$;

Instrukcje skoku służą do wskazania następnej instrukcji, która ma być wykonana, zwykle innej niż instrukcja następna w tekście programu.

Instrukcja powrotu (`return`) powoduje zakończenie wykonywania podprogramu i przekazanie sterowania do instrukcji, która ten podprogram wywołała. Wyrażenie w instrukcji powrotu jest obliczane i obliczona wartość jest wartością podprogramu — wyrażenie musi być odpowiedniego typu. Instrukcja powrotu bez wyrażenia jest właściwa dla podprogramu typu `void`. Instrukcja powrotu bez wyrażenia jest też umieszczana automatyczne przez kompilator na końcu każdego podprogramu (więc nie musimy jej pisać), ale wartość podprogramu typu innego niż `void` dla instrukcji powrotu bez wyrażenia jest nieokreślona (kompilator w takich przypadkach zwykle wypisuje ostrzeżenia).

Instrukcja przerwania (`break`) może wystąpić w dowolnej pętli (`while`, `do` ... `while` lub `for`) lub w instrukcji przełącznika (`switch`). Skutkiem jej wykonania jest zakończenie wykonywania tej pętli lub przełącznika i rozpoczęcie wykonywania następnej instrukcji. Istotne jest, że instrukcja przerwania kończy działanie tylko jednej pętli lub przełącznika, tej, która ją bezpośrednio otacza. W razie konieczności zakończenia działania pętli zagnieżdżonych trzeba użyć instrukcji goto albo (najlepiej) return.

Instrukcja kontynuacji (continue) może wystąpić w pętli; jej skutkiem jest pominięcie dalszych instrukcji wewnątrz pętli i, jeśli warunek wykonywania następnego przebiegu pętli jest spełniony, rozpoczęcie jego wykonywania. Jeśli warunek ten nie jest spełniony, wykonywana jest pierwsza instrukcja za pętlą.

Instrukcja skoku (goto) umożliwia przekazanie sterowania do dowolnej wskazanej instrukcji (w tym samym podprogramie), którą należy opatrzyć identyfikatorem (tzw. etykietą); w instrukcji skoku podaje się ten identyfikator. Za pomocą instrukcji skoku łatwo jest uczynić program całkowicie nieczytelnym i dlatego najlepiej jest unikać używania tej instrukcji w ogóle. Instrukcja ta przydaje się w razie konieczności zakończenia wykonywania zagnieżdżonej pętli, lub obsługi sytuacji wyjątkowych. *Zawsze* skoki powinny być „w przód”, tj. do instrukcji występujących dalej w tekście programu. Ponadto *nigdy* nie należy wskakiwać do środka pętli lub innej instrukcji strukturalnej.

instrukcja-etykietowana:
  identyfikator : *instrukcja*
  case *wyrażenie-stałe* : *instrukcja*
  default : *instrukcja*

Instrukcja może być opatrzona etykietą, i jeśli jest, to jej wykonanie nie musi następować po instrukcji poprzedzającej w tekście. Identyfikator i dwukropek przed instrukcją oznaczają, że do tej instrukcji może nastąpić skok spowodowany przez instrukcję goto. Pozostałe postaci etykiet występują w instrukcji przełącznika i omówimy je dalej.

Instrukcje wyrażeniowe i skoku są tak zwanymi instrukcjami prostymi, w odróżnieniu od instrukcji strukturalnych, których częściami składowymi mogą być inne instrukcje.

instrukcja-złożona:
  { *lista-deklaracji*$_{opc}$ *lista-instrukcji*$_{opc}$ }

lista-instrukcji:
  instrukcja
  lista-instrukcji instrukcja

Instrukcja-złożona jest ciągiem instrukcji ujętym w klamry; jeśli nie występują skoki, to instrukcje te są wykonywane kolejno. Na początku instrukcji złożonej można zadeklarować zmienne, które zostaną utworzone na początku wykonywania tej instrukcji i zlikwidowane na zakończenie.

instrukcja-wyboru:
  if (*wyrażenie*) *instrukcja*
  if (*wyrażenie*) *instrukcja* else *instrukcja*
  switch (*wyrażenie*) *instrukcja*

Instrukcje warunkowe if (...) ... oraz if (...) ... else ... zawierają warunek (*wyrażenie*), na podstawie którego instrukcja opatrzona warunkiem jest wykonywana lub pomijana, albo następuje wykonanie jednej z dwóch instrukcji. Warunek jest uważany za spełniony wtedy i tylko wtedy, gdy wartość wyrażenia jest różna od 0.

Instrukcja przełącznika (switch) służy do zareagowania na jedną z wielu możliwości. Na przykład, mając zmienną kolor, przyjmującą jedną z wartości czerwony, zielony, niebieski lub inną, możemy napisać

```
  switch ( kolor ) {
case czerwony:  printf ( "czerwony\n" );   break;
case zielony:   printf ( "zielony\n" );    break;
case niebieski: printf ( "niebieski\n" );  break;
default:        printf ( "inny\n" );
  }
```

Ten sam efekt moglibyśmy uzyskać, pisząc

```
if ( kolor == czerwony ) printf ( "czerwony\n" );
else if ( kolor == zielony ) printf ( "zielony\n" );
else if ( kolor == niebieski ) printf ( "niebieski\n" );
else printf ( "inny\n" );
```

ale w ten sposób wymuszamy określoną kolejność sprawdzania warunków.

Zwróćmy uwagę na instrukcje przerwania (break;) w instrukcji przełącznika. Gdyby ich nie było, to dla zmiennej kolor o wartości czerwony program wypisałby cztery linie tekstu informujące o wszystkich kolorach. Instrukcja z etykietą default jest wykonywana wtedy, gdy żadna możliwość opisana przez inne etykiety nie ma miejsca. Etykiety `default` nie musi być w instrukcji przełącznika i wtedy jeśli wartość zmiennej nie odpowiada żadnej innej etykiecie, to instrukcja przełącznika kończy działanie. Etykieta `default` *nie musi być* ostatnia w przełączniku. Oczywiście każda etykieta w przełączniku może wystąpić tylko raz.

instrukcja-powtarzania:
  `while` (*wyrażenie*) *instrukcja*
  `do` *instrukcja* `while` (*wyrażenie*)
  `for` (*wyrażenie*$_{opc}$; *wyrażenie*$_{opc}$; *wyrażenie*$_{opc}$) *instrukcja*

Instrukcje powtarzania, czyli pętle, służą do wielokrotnego wykonywania tych samych czynności. Ich przykłady mieliśmy na pierwszym wykładzie. W tym miejscu wspomnimy o sposobach przerywania pętli: może ono nastąpić wskutek niespełnienia warunku (odpowiednie wyrażenie ma wartość 0), lub wskutek wykonania skoku (`break`, `goto` lub `return`).

Zagnieżdżanie instrukcji warunkowej wiąże się z pewną łatwą do uniknięcia pułapką, wynikającą z pewnej niejednoznaczności składni. Rozważmy dwa fragmenty programu:

`if` (*warunek*$_1$)
  `if` (*warunek*$_2$) *instrukcja*$_1$
`else` *instrukcja*$_2$

`if` (*warunek*$_1$)
  `while` (*warunek*$_2$)
    `if` (*warunek*$_3$) *instrukcja*$_1$
`else` *instrukcja*$_2$

W pierwszym przypadku zależy nam na tym, aby jeśli *warunek*$_1$ jest niespełniony, wykonać *instrukcję*$_2$, a w przeciwnym razie zbadać *warunek*$_2$ i ewentualnie wykonać *instrukcję*$_1$. W drugim przypadku, jeśli *warunek*$_1$ jest spełniony, wykonujemy pętlę, w której *instrukcja*$_1$ jest wykonywana lub nie, zależnie od *warunku*$_3$. Jeśli *warunek*$_1$ jest niespełniony, to chcemy jednorazowo wykonać *instrukcję*$_2$. Tymczasem w obu przypadkach kompilator uzna tekst `else` *instrukcja*$_2$ za część drugiej (wewnętrznej) instrukcji warunkowej. Aby program działał zgodnie z opisaną wyżej intencją, trzeba go inaczej zapisać. Mamy dwa wyjścia. W pierwszym możemy do wewnętrznej instrukcji warunkowej dołączyć `else`; (czyli użyć instrukcji pustej). Drugi sposób, chyba lepszy, to umieścić

wewnętrzne instrukcje w instrukcji złożonej:

`if` (*warunek*$_1$) {
  `if` (*warunek*$_2$) *instrukcja*$_1$
}
`else` *instrukcja*$_2$

`if` (*warunek*$_1$) {
  `while` (*warunek*$_2$)
    `if` (*warunek*$_3$) *instrukcja*$_1$
}
`else` *instrukcja*$_2$

Składnia wyrażeń w C

Składnia wyrażeń jest opisana za pomocą około 20 symboli pomocniczych i ich wypisywania na tablicy nikt by nie przetrzymał. Jest ona zamieszczona w zadaniach. Wyrażenia składają się z identyfikatorów, stałych i operatorów. Zadaniem składni jest określenie dwóch rzeczy: priorytetów poszczególnych operatorów, czyli rozstrzygnięcia, jak należy zinterpretować na przykład wyrażenie `a+b*c` oraz łączności operatorów, czyli rozstrzygnięcia kolejności działań na przykład w wyrażeniu `a-b-c`, dla operatorów o takm samym priorytecie. Semantyka operatorów dotyczy skutków wykonania działań przez nie reprezentowanych.

Operator	l.a.	łączność
`( ) [ ] -> .`	1,2	lewostronna
`! ~ ++ -- + - * &` (*nazwa-typu*) `sizeof`	1	prawostronna
`* / %`	2	lewostronna
`- +`	2	lewostronna
`<< >>`	2	lewostronna
`< <= > >=`	2	lewostronna
`== !=`	2	lewostronna
`&`	2	lewostronna
`^`	2	lewostronna
`|`	2	lewostronna
`&&`	2	lewostronna
`||`	2	lewostronna
`? :`	3	prawostronna
`= += -= *= /= %= |= ^= <<= >>=`	2	prawostronna
`,`	2	lewostronna

W tabelce mamy wszystkie operatory języka C występujące w wyrażeniach; w drugiej kolumnie jest podana liczba argumentów każdego z nich. Pewne operatory są oznaczane takim samym symbolem, np. są jednoargumentowe operatory `&` i `*` oraz dwuargumentowe operatory `&` i `*` — kompilator umie je rozróżnić i my też powinniśmy się nauczyć.

Jeśli w wyrażeniu mamy dwa operatory, które występują w różnych wierszach, to pierwszeństwo ma operator podany wyżej. Zatem, podwyrażenie zamknięte w nawiasach okrągłych lub kwadratowych będzie obliczone najpierw. Ponieważ łączność operatora „`-`” jest lewostronna, więc obliczenie wartości wyrażenia `a-b-c` polega na obliczeniu różnicy `a-b` i odjęciu od niej wartości zmiennej `c`.

Operatory przypisania „`=`”, „`+=`”, „`-=`”, „`*=`”, „`/=`” itd. nadają zmiennej po lewej stronie wartość wyrażenia po prawej stronie, albo wartość obliczoną na podstawie wartości obu stron. Na przykład instrukcja `a += 5;` jest równoważna instrukcji `a = a + 5;`.

Operatory przypisania mają określony efekt uboczny; jest nim przypisanie nowej wartości jednemu z dwóch argumentów. Formalnie rzecz biorąc, wartością wyrażenia obliczoną przez operator przypisania jest przypisana wartość. Zatem w instrukcji wyrażeniowej `a=b;` mamy obliczaną wartość wyrażenia (jeśli zmienne `a` i `b` mają ten sam typ, to jest nią wartość zmiennej `b`). Po obliczeniu wartość wyrażenia jest odrzucana, ale pozostaje efekt uboczny operatora „`=`” — przypisanie nowej wartości zmiennej `a`. Dlaczego jest to istotne? Ponieważ wartość *wyrażenia-przypisania* może być argumentem kolejnego operatora. Na przykład w instrukcji `a=b=c;` wartość zmiennej `c` jest przypisywana zmiennej `b`, a następnie zmiennej `a`. Taka kolejność wynika stąd, że operatory przypisania mają łączność prawostronną.

Operatory nawiasowe „`(`” i „`)`” obejmują podwyrażenie, które ma być obliczone w pierwszej kolejności. Jeśli nie pamiętamy wszystkich poziomów priorytetów, to zawsze możemy dowolne podwyrażenie zamknąć w nawiasach, aby mieć pewność, co zostanie obliczone.

Operatory „`[`” i „`]`” służą do indeksowania tablic (i też mają najwyższy priorytet).

Operator „`.`” służy do dostępu do pola struktury lub unii. Operator „`->`” daje dostęp do pola struktury lub unii wskazywanej przez wartość wyrażenia z lewej strony (o strukturach, uniach i wskaźnikach będzie mowa później).

Operator „`!`” jest operatorem negacji logicznej.
Operator „`~`” jest operatorem negacji bitowej.

Operatory „`++`” i „`--`” są operatorami zwiększania i zmniejszania zmiennej o 1 — są to więc operatory, które mają skutki uboczne. Na przykład instrukcja

```
C++;
```

jest równoważna instrukcji

```
C = C+1;
```

Można pisać `C++;` i `++C;`, co oznacza to samo, ale instrukcje

`a = C++;` i `a = ++C;`

są równoważne odpowiednio instrukcjom

`a = C, C = C+1;` oraz `C = C+1, a = C;`.

Jednoargumentowe operatory „`+`” i „`-`” są arytmetyczne.

Operator „`(`*nazwa-typu*`)`” służy do konwersji typów. Na przykład jeśli zmienna `x` jest typu `float`, to wyrażenie `(int) x` oznacza wynik obcięcia (w kierunku zera) wartości zmiennej `x`, który jest reprezentowany jako liczba całkowita (typu `int`).

Wartością operatora „`sizeof`” jest wielkość (w bajtach) obszaru pamięci zajętego przez zmienną daną jako argument lub przez dowolną zmienną typu danego jako argument.

Operatory dwuargumentowe „`+`”, „`-`”, „`*`”, „`/`” i „`%`” realizują zwykłe działania arytmetyczne („`%`” powoduje obliczenie reszty z dzielenia pierwszego argumentu przez drugi).

Operatory „`<<`” i „`>>`” realizują „przesunięcia bitowe”. Wyniki obliczenia `a << b` i `a >> b`, gdzie `a` i `b` są liczbami całkowitymi, są równe odpowiednio $\lfloor \mathtt{a}\cdot 2^{\mathtt{b}}\rfloor$ i $\lfloor \mathtt{a}\cdot 2^{-\mathtt{b}}\rfloor$, pod warunkiem, że nie wystąpi nadmiar.

Bitowe operatory dwuargumentowe „`&`”, „`^`” i „`|`” realizują operacje koniunkcji, alternatywy wyłącznej i alternatywy na odpowiadających sobie bitach argumentów (bit o wartości 1 jest traktowany jako „prawda", a 0 jako „fałsz").

Operatory „&&" oraz „||" są logiczne. W odróżnieniu od operatorów bitowych traktują każdy argument jako jedną wielkość logiczną (0 jako „fałsz", każdą inną liczbę jako prawdę).

Operatory relacyjne „==", „!=", „<", „>", „<=", „>=" służą do porównywania wyrażeń.

Operator trójargumentowy „? :" może zastąpić w prostych przypadkach instrukcję warunkową. Na przykład instrukcja

```
a = b > c ? b : c;
```

ma identyczny skutek jak instrukcja

```
if ( b > c ) a = b; else a = c;
```

Operatory jednoargumentowe „&" i „*" służą do otrzymania adresu zmiennej i do sięgnięcia pod ten adres. Przykład użycia był w programie, który przekazywał adres jako parametr podprogramu.

Operator przecinkowy „," umożliwia połączenie kilku wyrażeń w jedno. Rozważmy przykład algorytmu, który z tablicy o długości N wybiera elementy na pozycjach nieparzystych w tablicy a i wpisuje je na kolejne pozycje w tablicy b:

```
for ( i = 0, j = 1;  j < N;  i++, j += 2 )
  b[i] = a[j];
```

Preprocesor i makrodefinicje

Do sparametryzowania programów w C często używa się dyrektywy preprocesora `#define`. Można napisać np.

```
#define N 10
```

i od tej pory każde wystąpienie identyfikatora `N` w programie będzie zamieniane

na ciąg dwóch cyfr „10", który następnie zostanie rozpoznany jako literał[5] (liczba całkowita 10). Zastosowanie dyrektywy `#define` jest znacznie szersze. Przede wszystkim można napisać dowolny tekst w C. Dzięki temu na przykład dyrektywy

```
#define PI 3.14159265358979
#define deg *(PI/180.0)
```

umożliwiają poprawne działanie instrukcji

```
  s = sin ( 10 deg );
```

której zadaniem jest obliczenie sinusa 10 stopni przy użyciu funkcji `sin`, która oczekuje parametru, którego wartość jest miarą kąta w radianach. Ponadto można definiować makra z parametrami. Na przykład definicja

```
#define Exchange(a,b) { c = a; a = b; b = c; }
```

działa tak, że każde wystąpienie identyfikatora `Exchange`, po którym są w nawiasach podane dwie zmienne, np.

```
  Exchange ( x[i], x[j] )
```

zostanie zamienione na odpowiednią instrukcję złożoną, w tym przypadku

```
  { c = x[i]; x[i] = x[j]; x[j] = c; }
```

która będzie przestawiać elementy tablicy `x`, pod warunkiem, że taka tablica istnieje (tj. została wcześniej zadeklarowana) i istnieje również zmienna `c` odpowiedniego typu. W definicji makra z parametrami otwierający nawias okrągły musi być podany natychmiast po identyfikatorze makra, tj. nie może między nimi być spacji.

Tekst makra zaczyna się po jego nazwie lub po liście parametrów, a kończy się z końcem linii. Jeśli makro ma być długie, to możemy je przenieść do następnej linii, pisząc znak \, np.

[5]Literał jest to symbol leksykalny reprezentujący stałą, której wartość jest możliwa do odczytania z ciągu znaków składających się na ten symbol. Na przykład ciąg cyfr reprezentujących liczbę całkowitą, a także napis, jest literałem. Nie jest literałem identyfikator `N`, o znaczeniu nadanym przez dyrektywę `#define`

```
#define Exchange(a,b) \
  { c = a; a = b; b = c; }
```

Jeśli chcemy zlikwidować makro (bo np. ma ono znaczenie tylko w pewnym podprogramie, a w następujących po nim mogłoby szkodzić), to piszemy dyrektywę `#undef`, np.

```
#undef Exchange
```

Jeszcze jedno: Odpowiednio zastosowane makra umożliwiają znaczne skrócenie tekstu programu w C, przy jednoczesnym zwiększeniu jego czytelności. Nieodpowiednio zastosowane makra mogą doprowadzić do otrzymania programu koszmarnie nieczytelnego, co może być zamierzonym efektem tylko wtedy, gdy autor programu chce zrobić komuś (a przy okazji sobie) na złość. Wobec tego należy ponowić apel o jak najczytelniejsze i klarowne zapisywanie algorytmów i starania o ułatwianie zrozumienia treści programu osobom, które będą ten program czytać. Służą temu: dzielenie problemu na niezależne od siebie części, realizowane przez osobne procedury, pisanie komentarzy, stosowanie wcięć i używanie z wyczuciem makrodefinicji.

Zadania i problemy

1. Przejrzyj składnię wyrażeń w C i skonfrontuj ją z podaną na wykładzie tabelką z operatorami. Zwróć uwagę na sposób, w jaki priorytety i łączność poszczególnych operatorów są odzwierciedlone w gramatyce.
   <u>Nie ucz się tego na pamięć</u>.

wyrażenie:
  wyrażenie-przypisania
  wyrażenie `,` *wyrażenie-przypisania*

wyrażenie-przypisania:
  wyrażenie-warunkowe
  wyrażenie-jednoargumentowe operator-przypisania wyrażenie-przypisania

operator-przypisania: jeden z
  `=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=`

wyrażenie-warunkowe:
  logiczne-wyrażenie-OR
  logiczne-wyrażenie-OR `?` *wyrażenie* `:` *wyrażenie-warunkowe*

logiczne-wyrażenie-OR:
  logiczne-wyrażenie-AND
  logiczne-wyrażenie-AND `||` *logiczne-wyrażenie-OR*

logiczne-wyrażenie-AND:
  wyrażenie-OR
  logiczne-wyrażenie-AND `&&` *wyrażenie-OR*

wyrażenie-OR:
  wyrażenie-XOR
  wyrażenie-OR `|` *wyrażenie-XOR*

wyrażenie-XOR:
  wyrażenie-AND
  wyrażenie-XOR `^` *wyrażenie-AND*

wyrażenie-AND:
  wyrażenie-przyrównania
  wyrażenie-przyrównania `&` *wyrażenie-AND*

wyrażenie-przyrównania:
  wyrażenie-relacyjne
  wyrażenie-przyrównania `==` *wyrażenie-relacyjne*
  wyrażenie-przyrównania `!=` *wyrażenie-relacyjne*

wyrażenie-relacyjne:
  wyrażenie-przesunięcia
  wyrażenie-relacyjne `<` *wyrażenie-przesunięcia*

wyrażenie-relacyjne > *wyrażenie-przesunięcia*
wyrażenie-relacyjne <= *wyrażenie-przesunięcia*
wyrażenie-relacyjne >= *wyrażenie-przesunięcia*

wyrażenie-przesunięcia:
wyrażenie-addytywne
wyrażenie-przesunięcia << *wyrażenie-addytywne*
wyrażenie-przesunięcia >> *wyrażenie-addytywne*

wyrażenie-addytywne:
wyrażenie-multiplikatywne
wyrażenie-addytywne + *wyrażenie-multiplikatywne*
wyrażenie-addytywne - *wyrażenie-multiplikatywne*

wyrażenie-multiplikatywne:
wyrażenie-rzutowania
wyrażenie-multiplikatywne * *wyrażenie-rzutowania*
wyrażenie-multiplikatywne / *wyrażenie-rzutowania*
wyrażenie-multiplikatywne % *wyrażenie-rzutowania*

wyrażenie-rzutowania:
wyrażenie-jednoargumentowe
(*nazwa-typu*) *wyrażenie-rzutowania*

wyrażenie-jednoargumentowe:
wyrażenie-przyrostkowe
++ *wyrażenie-jednoargumentowe*
-- *wyrażenie-jednoargumentowe*
operator-jednoargumentowy *wyrażenie-rzutowania*
...

wyrażenie-przyrostkowe:
wyrażenie-proste
wyrażenie-przyrostkowe [*wyrażenie*]
wyrażenie-przyrostkowe (*lista-argumentów*$_{opc}$)
wyrażenie-przyrostkowe ++
wyrażenie-przyrostkowe --
...

wyrażenie-proste:
identyfikator
stała
napis
(*wyrażenie*)

lista-argumentów:
wyrażenie-przypisania
lista-argumentów , *wyrażenie-przypisania*

2. Wskaż kolejność wykonywania poszczególnych operacji w poniższym wyrażeniu, oraz jego skutki uboczne:

```
!( a = b*c+d ) && Nicea || Muerte,
sin(2.0*alfa+pi/3) + 3.14*(beta+gamma/(a-b))
```

3. Przypuśćmy, że zmienne `a`, `b`, `c`, `d` typu <u>`int`</u> mają początkowe wartości odpowiednio 1, 2, 3, 4. Jakie będą końcowe wartości tych zmiennych po wykonaniu instrukcji

```
a += b += c += d += 5;
```

Elementy analizy algorytmów

Dla każdego algorytmu trzeba brać pod uwagę jego dwie istotne cechy: poprawność i złożoność obliczeniową.

Poprawność

Poprawność algorytmu jest to własność, która zapewnia, że algorytm rozwiązuje zadanie, do którego jest przeznaczony. Poprawność rozpatruje się na dwóch poziomach: abstrakcyjnym, gdzie przyjmujemy, że działania wykonywane przez algorytm są wyidealizowane (np. wykonujemy cztery działania arytmetyczne na liczbach rzeczywistych — wtedy jedyny przypadek, gdy obliczenia mogą skończyć się niepowodzeniem, to dzielenie przez zero, oczywiście algorytm nie może do tego dopuścić), zaś to co obliczymy jest dokładnie tym, co mieliśmy na myśli programując użyte wzory, oraz programistycznym, gdzie ilość dostępnej pamięci jest ograniczona, nie każdy wynik poprawnego działania można reprezentować (z powodu nadmiaru), a do tego wykonując obliczenia zmiennopozycyjne, komputer wprowadza błędy zaokrągleń. Tak więc abstrakcyjna poprawność algorytmu *nie jest* warunkiem dostatecznym niezawodności programu, który ten algorytm realizuje. Mimo wszystko, od niej trzeba zaczynać pracę nad programem.

Rozważmy *zadanie*, określone w postaci funkcji $f\colon D \to W$; jego rozwiązanie polega na obliczeniu wyniku $w = f(d)$ dla konkretnych danych $d \in D$. *Algorytm* A jest też funkcją, która dowolnym danym przyporządkowuje pewien wynik. Algorytm jest poprawny (dla zadania f), jeśli funkcja A w zbiorze D jest identyczna z f (czyli $\forall_{d\in D} A(d) = f(d)$). Stwierdzenie poprawności algorytmu często nie jest łatwe.

Dowód poprawności algorytmu najczęściej polega na wykazaniu dwóch rzeczy. Pierwszą z nich jest tzw. własność stopu algorytmu, czyli fakt, że dla dowolnych danych $d \in D$ algorytm ten zakończy działanie po wykonaniu skończenie wielu obliczeń. Algorytm jest poprawny, jeśli ma własność stopu i zakończenie obliczeń może nastąpić tylko wskutek znalezienia rozwiązania zadania, które ten algorytm ma rozwiązywać.

Własność stopu mają oczywiście algorytmy, w których nie ma żadnych instrukcji wykonywanych wielokrotnie, czyli nie ma pętli ani rekurencji. Dowody własności stopu algorytmów iteracyjnych i rekurencyjnych najczęściej korzystają z indukcji. Przypuśćmy, że algorytm jest zrealizowany za pomocą pojedynczej pętli (`while` albo `for`) i warunek zakończenia wykonywania pętli można przedstawić w postaci $g(n) \le 0$, z funkcją g, która może zależeć od danych i której argument $n \in \mathbb{N}$ jest

numerem kolejnego przebiegu pętli. Aby algorytm miał własność stopu, wystarczy, że funkcja g przyjmuje wartości całkowite i jest (ściśle) malejąca.

Dowód, że zakończenie działania algorytmu iteracyjnego następuje wskutek znalezienia rozwiązania, często polega na wskazaniu i zbadaniu tzw. niezmienników pętli, czyli własności, które mają wartości zmiennych przetwarzanych przez algorytm w pewnym miejscu (np. przed wykonaniem pierwszej instrukcji w pętli). Instrukcje wykonywane w pętli mają za zadanie tak zmienić wartości zmiennych, aby niezmiennik nadal był zachowany, przy czym otrzymanie rozwiązania wynika z niezmiennika i z faktu, że funkcja g (ta od własności stopu) ma wartość niedodatnią.

Przykład. Zadanie polega na obliczeniu dla danej liczby $m \in \mathbb{N}$ liczby $x \in \mathbb{N}$, takiej że $x = \lfloor\sqrt{m}\rfloor$. Udowodnimy, że algorytm zapisany w instrukcji

```
for ( x = 1, y = x+1;  y*y <= m;  x = y++ )
  ;
```

jest poprawnym algorytmem rozwiązywania tego zadania. Po pierwsze, dla dowolnego $m \in \mathbb{N}$ wynik x jest większy lub równy 1. Fakt, że liczba n (numer przebiegu pętli, określający, który raz jest wykonywana instrukcja pusta) jest wartością zmiennej x *przed* obliczeniem wyrażenia `x = y++` (które przebiega tak: wartość zmiennej y jest przypisywana zmiennej x, *a następnie* zwiększana o 1) jest niezmiennikiem pętli. Drugim niezmiennikiem jest nierówność $x^2 \le m$. Algorytm ma własność stopu, ponieważ funkcja g, określona wzorem $g(n) = m - (n+1)^2 + 1$, przyjmuje wartości całkowite i jest malejąca. Wreszcie, z faktu, że $g(n) \le 0$ wynika, że $(n+1)^2 = (x+1)^2 > m$, zatem w chwili zatrzymania algorytmu mamy $x^2 \le m < y^2 = (x+1)^2$, co kończy dowód.

Zwróćmy uwagę, że ten dowód jest oparty na założeniu, że dysponujemy możliwością wykonywania rachunków na liczbach naturalnych. Jeśli w implementacji użyjemy liczb typu `int` i przyjmiemy za m największą reprezentowalną w tym typie liczbę całkowitą, to liczba $(x+1)^2$ nie ma reprezentacji. Zatem, algorytm na poziomie abstrakcyjnym jest poprawny (to udowodniliśmy), ale jego implementacja może być błędna.

W praktyce (niestety) bardzo mała część programów powstaje razem z dowodem poprawności. Pisząc program, warto jednak zastanawiać się nad ewentualnym udowodnieniem poprawności algorytmu. W szczególności dla każdej pętli warto sformułować jej niezmienniki i napisać je w komentarzu.
To się niesamowicie opłaca.

Koszt i złożoność

Druga istotna cecha algorytmu to jego złożoność obliczeniowa. Rozróżniamy złożoność czasową, związaną z liczbą elementarnych czynności wykonywanych przez algorytm i złożoność pamięciową, która dotyczy objętości danych przetwarzanych przez ten algorytm.

Bardziej elementarnym pojęciem, które prowadzi do złożoności, jest koszt (czasowy i pamięciowy) algorytmu. Dotyczy on wykonania algorytmu na konkretnych danych; jest to liczba operacji (albo taktów zegara, które można przeliczyć na dokładny czas obliczeń), albo liczba komórek pamięci (bajtów lub słów) potrzebnych w obliczeniach. Koszt zależy od rozmiaru zadania, czyli liczby danych (np. długości danego ciągu liczb) lub liczby wyników (np. pikseli na obrazie, który jest wynikiem, liczby liczb pierwszych, które należy znaleźć itd.).

Rozmiar zadania może być scharakteryzowany jedną liczbą, ale często naturalne jest użycie kilku liczb. Na przykład rozmiar zadania sortowania ciągu lub rozkładania liczby naturalnej na czynniki jest jedną liczbą n, która określa w pierwszym przypadku długość ciągu, a w drugim liczbę cyfr dziesiętnych. Natomiast zadanie mnożenia macierzy prostokątnych lepiej jest „zmierzyć” trzema liczbami: dla $A \in \mathbb{R}^{m,n}$ i $B \in \mathbb{R}^{n,k}$ mamy obliczyć macierz $C = AB \in \mathbb{R}^{m,k}$; w tym przypadku informacja niesiona przez trójkę liczb (m, n, k) jest znacznie pełniejsza niż łączna liczba $(m + k)n$ współczynników danych macierzy A i B.

Wyznaczanie kosztu można sprowadzić do liczenia tzw. operacji dominujących. Operacje dominujące to takie działania w algorytmie, że każdemu z nich można przyporządkować co najwyżej pewną ustaloną liczbę pozostałych operacji. Na przykład, dodanie dwóch macierzy liczbowych polega na obliczeniu sum odpowiednich współczynników. Odpowiednie współczynniki trzeba pobrać z pamięci (co wiąże się z obliczeniem ich adresów), wykonać dodawanie, obliczyć adres, pod którym należy umieścić wynik i przesłać ten wynik na miejsce. Operacją dominującą może tu być dodawanie. Podobnie, za operację dominującą w algorytmach sortowania przyjmuje się porównywanie — po dokonaniu porównania może, ale nie musi nastąpić przestawienie elementów, tak więc przestawień nie będzie więcej niż porównań.

Algorytm może dla różnych danych o ustalonym rozmiarze wykonywać różne liczby operacji. Na przykład posortowanie ciągu uporządkowanego może sprowadzić się do stwierdzenia, że ciąg jest uporządkowany, co może trwać krócej niż przestawianie wszystkich elementów innego ciągu o tej samej długości.

W związku z tym są następujące rodzaje złożoności (niżej zakładamy, że rozmiar danych jest jedną liczbą n, a D_n oznacza zbiór wszystkich danych o tym rozmiarze; $t(d)$ oznacza koszt działania algorytmu na danych d):

Złożoność pesymistyczna:

$$T^{\mathrm{wor}}(n) \stackrel{\mathrm{def}}{=} \sup\{\, t(d)\colon d \in D_n \,\}.$$

Mamy gwarancję, że algorytm dla danych o rozmiarze n nigdy nie wykona więcej niż $T^{\mathrm{wor}}(n)$ operacji dominujących.

Złożoność optymistyczna:

$$T^{\mathrm{bst}}(n) \stackrel{\mathrm{def}}{=} \inf\{\, t(d)\colon d \in D_n \,\}.$$

Mamy gwarancję, że dla danych o rozmiarze n algorytm musi wykonać przynajmniej $T^{\mathrm{bst}}(n)$ operacji.

Złożoność średnia:

$$T^{\mathrm{ave}}(n) \stackrel{\mathrm{def}}{=} \sum_{k=0}^{\infty} k p_n(k),$$

gdzie $p_n(k)$ jest prawdopodobieństwem wystąpienia danych $d \in D_n$, dla których koszt algorytmu jest równy k

$$p_n(k) = \Pr\{\, d \in D_n\colon t(d) = k \,\}.$$

To oczywiście wymaga znajomości rozkładu prawdopodobieństwa wystąpienia określonych danych. Zawsze są spełnione nierówności

$$T^{\mathrm{bst}}(n) \leq T^{\mathrm{ave}}(n) \leq T^{\mathrm{wor}}(n).$$

Równości zachodzą wtedy, gdy koszt zależy tylko od rozmiaru danych.

Przykład. Rozważmy zadanie, które polega na wyszukaniu danego elementu (liczby całkowitej) x w ciągu o długości n umieszczonym w tablicy. Jeśli x jest obecny w ciągu, to należy podać jego położenie (indeks). W przeciwnym razie należy zasygnalizować brak elementu, podając indeks 0. Algorytm polega na przeszukaniu tablicy, tj. sprawdzeniu kolejno wszystkich elementów do znalezienia x lub do końca.

Uwaga: Aby przekazać tablicę jako parametr procedury w C, w liście parametrów piszemy `int a[]`. Tak opisany parametr nie określa długości tablicy, choć można napisać także np. int a[10] i wtedy długość jest określona. Należy pamiętać, że tablica przekazywana jako parametr nie jest kopiowana. Zmienna a — parametr procedury — jest adresem w pamięci operacyjnej pierwszego elementu tablicy (o indeksie 0). Równoważnie możemy dać taki opis parametru: int *a, jeszcze do tego tematu wrócimy.

```
int Pozycja ( int n, int a[], int x )
{
  int i;

  for ( a[0] = x, i = n;  a[i] != x;  i-- )
    ;
  return i;
} /*Pozycja*/
```

W powyższym podprogramie zakładamy, że algorytmie tablica a ma długość $n+1$ i elementy ciągu są w niej umieszczone na pozycjach od 1 do n. Miejsce 0 tablicy jest wykorzystane jako tzw. strażnik; daną, którą mamy wyszukać, wpisujemy na to miejsce, dzięki czemu mamy pewność, że znajdziemy w tablicy element o wartości x i nie musimy sprawdzać, czy indeks i nie jest ujemny.

Za operację dominującą tego algorytmu możemy przyjąć porównywanie x z elementem tablicy. Złożoność optymistyczna tego algorytmu jest oczywiście równa 1 — wykonamy co najmniej jedno porównanie, po którym może się okazać, że od razu znaleźliśmy x. Zatem $T^{\mathrm{bst}}(n) = 1$.

Koszt będzie równy złożoności pesymistycznej w przypadku, gdy ciąg w tablicy nie zawiera elementu x — wtedy wykonamy $n+1$ porównań (i procedura będzie miała wartość 0), jeśli zaś ciąg zawiera x, to koszt będzie mniejszy.
Tak więc $T^{\mathrm{wor}}(n) = n+1$.

Aby obliczyć złożoność średnią, trzeba przyjąć jakieś założenia na temat prawdopodobieństwa wystąpienia określonych danych. Przyjmiemy najpierw, że prawdopodobieństwo, że x jest elementem ciągu, jest równe p, przy czym założymy, że x może wystąpić tylko raz, na dowolnej pozycji z jednakowym prawdopodobieństwem. Zatem z prawdopodobieństwem $1-p$ algorytm wykona $n+1$ porównań. Prawdopodobieństwo wykonania $k \in \{1,\dots,n\}$ porównań jest

równe $\frac{p}{n}$. Przy takich założeniach mamy

$$T^{\mathrm{ave}}(n) = (n+1)(1-p) + \sum_{k=1}^{n} k\frac{p}{n} = (n+1)(1-p) + \frac{p}{n}\sum_{k=1}^{n} k =$$
$$(n+1)(1-p) + \frac{p}{n}\frac{n(n+1)}{2} = (n+1)(1-p/2).$$

Gdybyśmy przyjęli inne założenie, np. że każdy element ciągu, niezależnie od pozostałych, może mieć wartość x z prawdopodobieństwem $q > 0$, to złożoność średnia tego samego algorytmu byłaby równa

$$T^{\mathrm{ave}}(n) = (n+1)r^n + \sum_{k=1}^{n} kqr^{k-1},$$

gdzie $r = 1-q$. Przeliczmy to do końca. Aby obliczyć $\sum_{k=1}^{n} kr^{k-1}$ zauważamy, że to jest pochodna funkcji

$$g_n(r) = \sum_{k=0}^{n} r^k = \frac{r^{n+1}-1}{r-1}, \quad \text{czyli}$$
$$g_n'(r) = \frac{nr^{n+1}-(n+1)r^n+1}{(r-1)^2} = \frac{nr^{n+1}-(n+1)r^n+1}{q^2}.$$

Mamy zatem

$$T^{\mathrm{ave}}(n) = (n+1)r^n + \frac{nr^{n+1}-(n+1)r^n+1}{q} = \frac{1-(1-q)^{n+1}}{q}.$$

Dla ustalonego $q \in (0,1)$ złożoność średnia algorytmu rośnie ze wzrostem n, przy czym wartością graniczną jest $1/q$ — mimo, że złożoność pesymistyczna dla $n \to \infty$ rośnie nieograniczenie, to możemy się spodziewać, że na ogół znajdziemy wynik po wykonaniu stałej liczby (ok. $1/q$) porównań.

Rząd złożoności obliczeniowej algorytmu jest pojęciem, które zaniedbuje czynnik stały w koszcie. Na przykład, algorytm eliminacji Gaussa rozwiązywania układu n równań liniowych z n niewiadomymi wykonuje około $\frac{1}{3}n^3$ operacji dominujących, którymi są mnożenia z dodawaniem. Inny algorytm (odbić Householdera) rozwiązuje to samo zadanie wykonując ok. $\frac{2}{3}n^3$ takich operacji (czyli jego złożoność jest w przybliżeniu dwukrotnie większa). Cechą wspólną obu tych algorytmów jest to, że złożoność ich obu jest w przybliżeniu równa cn^3, dla pewnej (innej dla każdego algorytmu) stałej dodatniej c.

Dokładniej, mówimy, że funkcja $f(n)$ jest rzędu co najwyżej $g(n)$, jeśli istnieją liczby c i n_0, takie że

$$\forall_{n>n_0} f(n) \le cg(n).$$

Zapisujemy to symbolicznie grecką literą „duże omikron”:

$$f(n) = O(g(n)).$$

Def. Mówimy, że funkcja $f(n)$ jest rzędu co najmniej $g(n)$, jeśli istnieją liczby $c > 0$ i n_0, takie że

$$\forall_{n>n_0} f(n) \geq cg(n).$$

Zapisujemy to symbolicznie przy użyciu litery „duże omega”:

$$f(n) = \Omega(g(n)).$$

Def. Mówimy, że funkcja $f(n)$ jest rzędu $g(n)$, jeśli jest rzędu co najwyżej $g(n)$ i rzędu co najmniej $g(n)$. To zapisujemy symbolicznie w postaci

$$f(n) = \Theta(g(n)).$$

Rząd złożoności obliczeniowej obu wspomnianych algorytmów rozwiązywania układu n równań z n niewiadomymi jest zatem taki sam: $\Theta(n^3)$. Na podstawie tej informacji wiemy, że jeśli dowolnego z tych algorytmów użyjemy do rozwiązania układu $2n$ równań, to czas obliczeń będzie dłuższy w przybliżeniu ośmiokrotnie w porównaniu z czasem rozwiązywania układu $n \times n$.

Używając zdefiniowanych tu notacji stwierdzamy, że zbadany wcześniej algorytm wyszukiwania elementu x w ciągu ma rząd złożoności pesymistycznej $\Theta(n)$, a optymistycznej $\Theta(1)$. Zależnie od rozkładu prawdopodobieństwa pojawienia się szczególnych danych, złożoność średnia algorytmu jest rzędu n w pierwszym zbadanym przypadku, albo rzędu 1 w drugim.

Wiele algorytmów rozwiązywania różnych zadań praktycznych ma rząd złożoności taki jak w poniższej liście:

$O(1)$	—	stała,
$O(\log n)$	—	logarytmiczna,
$O(n)$	—	liniowa,
$O(n \log n)$	—	liniowo-logarytmiczna,
$O(n^2)$	—	kwadratowa,
$O(n^3)$	—	sześcienna,
$O(n^k)$	—	wielomianowa (z wykładnikiem k),
$O(a^n)$	—	wykładnicza (dla pewnego $a > 1$),
$O(n!)$, $O(n^n)$	—	jeszcze większe.

W zasadzie algorytmy o złożoności logarytmicznej są bardzo szybkie, algorytmy o złożoności wielomianowej są praktycznie użyteczne (choć im większy wykładnik, tym gorzej to wygląda), natomiast algorytmy o złożoności wykładniczej i większej są praktycznie bezużyteczne — jeśli zwiększenie rozmiaru danych o 1 powoduje dwu- lub dziesięciokrotny wzrost czasu obliczeń, to zadania, które możemy rozwiązać, stosując taki algorytm, są nieduże.

Zalety notacji „O” to uproszczenie wzorów i rachunków oraz skupienie się na najważniejszej cesze złożoności. Czynnik stały często daje się trochę poprawić, w ostateczności można wziąć szybszy komputer. Obniżenie rzędu złożoności, na przykład zmniejszenie wykładnika, tj. znalezienie algorytmu wielomianowego o mniejszym wykładniku, otwiera drogę do rozwiązywania znacznie większych zadań.

Wada notacji „O” to zaniedbanie stałych, które może zafałszować obraz sytuacji. Przypuśćmy, że rząd złożoności algorytmu A_1 jest $O(n^2)$, zaś rząd złożoności algorytmu A_2, jest $O(n^3)$, czyli jest większy. Jeśli jednak złożoności tych algorytmów są równe odpowiednio $10^5 \cdot n^2$ i $5 \cdot n^3$, to dla niedużych n drugi algorytm może jednak rozwiązywać zadanie w znacznie krótszym czasie.

Rząd złożoności zadania jest to najmniejszy rząd złożoności poprawnego algorytmu, który rozwiązuje to zadanie. Wiele zadań można rozwiązać przy użyciu algorytmów o różnych złożonościach obliczeniowych. Jednym z podstawowych działów informatyki jest obliczanie (lub szacowanie) złożoności zadań. Dla wielu zadań nie jest znany rząd złożoności — pole do działania jest tu ogromne. Na przykład nie wiadomo, jaki jest rząd złożoności zadania mnożenia macierzy kwadratowych $n \times n$ — wiadomo, że nie mniejszy niż n^2, bo z tylu liczb składa się wynik, ale algorytm „zwykły” ma koszt $\Theta(n^3)$. Znane są algorytmy o mniejszym rzędzie złożoności (to nie są jednak bardzo praktyczne algorytmy, m.in. czynnik stały jest w nich bardzo duży).

Równania różnicowe

Chyba najważniejszym (a w każdym razie skutecznym w wielu przypadkach) środkiem do obliczania złożoności obliczeniowej różnych algorytmów iteracyjnych i rekurencyjnych jest rozwiązywanie równań różnicowych. Możemy użyć tego środka, jeśli dany algorytm rozwiązywania zadania o rozmiarze $k > 1$ wymaga rozwiązania ustalonej liczby zadań mniejszych (np. o rozmiarze $k - 1$ lub $k/2$), oraz wykonania pewnej liczby $f(k)$ operacji w celu uzyskania końcowego rozwiązania.

Niech liczby $a_0, \ldots, a_{n-1}$ będą dane i niech dla każdego całkowitego $k \geq n$

$$a_k = c_{n-1}a_{k-1} + \cdots + c_0 a_{k-n} + f(k), \tag{*}$$

gdzie liczby $c_0, \ldots, c_{n-1}$ i funkcja f są ustalone. Równanie o podanej wyżej postaci nazywa się równaniem różnicowym liniowym rzędu n. Razem z warunkiem początkowym, tj. danymi liczbami $a_0, \ldots, a_{n-1}$, określa ono jednoznacznie nieskończony ciąg liczbowy $(a_k)_{k=0}^{\infty}$. Koszty i złożoności wielu algorytmów (a także inne wielkości występujące w różnych zastosowaniach) można przedstawić za pomocą równań różnicowych. Aby uzyskać wynik w postaci jawnej (w której jest od razu widoczny na przykład rząd złożoności algorytmu), trzeba rozwiązać takie równanie.

Równanie o postaci

$$a_k = c_{n-1}a_{k-1} + \cdots + c_0 a_{k-n}, \tag{**}$$

nazwiemy równaniem jednorodnym. Łatwo możemy zauważyć, że jeśli ciągi $(a_k)_{k=0}^{\infty}$ i $(b_k)_{k=0}^{\infty}$ są rozwiązaniami równania (*), to różnica tych ciągów jest rozwiązaniem równania (**) i na odwrót: suma rozwiązania równania (*) i dowolnego rozwiązania odpowiadającego mu równania jednorodnego (otrzymanego przez usunięcie składnika $f(k)$) też jest rozwiązaniem równania (*).

Metoda rozwiązywania równań różnicowych liniowych jest następująca. Należy znaleźć (np. przez odgadnięcie, co bywa łatwe dla pewnych funkcji f) dowolne rozwiązanie równania. Następnie należy znaleźć takie rozwiązanie równania jednorodnego, aby suma obu rozwiązań spełniała warunki początkowe. Zaczniemy od zbadania, jakie ciągi spełniają równania jednorodne.

Zbadamy hipotezę, że rozwiązanie równania jednorodnego ma postać $a_k = \lambda^k$. Wstawiając to do (**), po podzieleniu stron przez λ^{k-n} i uporządkowaniu, dostaniemy tzw. równanie charakterystyczne:

$$\lambda^n - c_{n-1}\lambda^{n-1} - \cdots - c_1\lambda - c_0 = 0,$$

czyli $w(\lambda) = 0$. Wielomian w stopnia n nazywa się wielomianem charakterystycznym równania; liczba λ musi być jego miejscem zerowym. Jeśli wielomian ten ma n różnych (jednokrotnych) miejsc zerowych, to mamy n liniowo niezależnych ciągów liczbowych spełniających równanie jednorodne.

Przykład. Jednorodne równanie różnicowe drugiego rzędu $F_k = F_{k-1} + F_{k-2}$ z warunkiem początkowym $F_0 = 0$, $F_1 = 1$ określa znany ciąg Fibonacciego.

Podstawiając $a_k = \lambda^k$, otrzymamy równanie kwadratowe

$$\lambda^2 - \lambda - 1 = 0,$$

którego rozwiązaniami są liczby $\lambda_1 = \frac{1}{2}(1-\sqrt{5})$ i $\lambda_2 = \frac{1}{2}(1+\sqrt{5})$. Dowolne rozwiązanie równania jednorodnego, w tym rozwiązanie poszukiwane, ma postać $F_k = b_1\lambda_1^k + b_2\lambda_2^k$. Na podstawie warunków początkowych

$$F_0 = b_1 + b_2 = 0,$$
$$F_1 = b_1\lambda_1 + b_2\lambda_2 = 1,$$

skąd możemy obliczyć $b_1 = -\frac{1}{\sqrt{5}}$, $b_2 = \frac{1}{\sqrt{5}}$, czyli ostatecznie

$$F_k = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^k - \left(\frac{1-\sqrt{5}}{2}\right)^k\right)$$

Jeśli pierwiastki wielomianu w są zespolone, to (dla równania o współczynnikach rzeczywistych) występują w parach sprzężonych, $(\lambda_i, \overline{\lambda}_i)$, i możemy znaleźć dla każdej takiej pary dwa liniowo niezależne ciągi liczb rzeczywistych, $(\lambda_i^k + \overline{\lambda_i}^k)_{k=0}^{\infty}$ oraz $(i(\lambda_i^k - \overline{\lambda_i}^k))_{k=0}^{\infty}$, które spełniają równanie (**). Ten przypadek ma mniejsze znaczenie w obliczaniu kosztów algorytmów, bo rozwiązania odpowiadające zespolonym pierwiastkom równania charakterystycznego oscylują (przyjmując także wartości ujemne).

Jeśli wielomian w ma miejsca zerowe o krotności większej niż 1, to liniowo niezależnych ciągów geometrycznych spełniających równanie (**) jest mniej niż n, zatem może ich nie wystarczyć do znalezienia rozwiązania spełniającego dowolne warunki początkowe. Jeśli liczba λ jest miejscem zerowym wielomianu charakterystycznego o krotności r, to ciąg $\big((d_0 + d_1k + \cdots + d_{r-1}k^{r-1})\lambda^k\big)_{k=0}^{\infty}$ jest rozwiązaniem. Aby spełnić warunek początkowy, należy odpowiednio dobrać współczynniki $d_0, \ldots, d_{r-1}$.

Przykład. Rozwiążemy równanie $a_k = 2a_{k-1} - a_{k-2}$ z warunkiem $a_0 = 1$, $a_1 = 2$. Mamy $w(\lambda) = \lambda^2 - 2\lambda + 1 = (\lambda-1)^2$, skąd wynika, że liczba $\lambda = 1$ jest pierwiastkiem wielomianu w o krotności 2. Zatem rozwiązanie ogólne ma postać $(d_0 + d_1k)\cdot 1^k$, i łatwo możemy sprawdzić, że ciąg arytmetyczny $a_k = k+1$ spełnia to równanie i warunek początkowy.

Teraz zajmiemy się znajdowaniem rozwiązań szczególnych równania niejednorodnego. Postać rozwiązania zależy od funkcji f. Jeśli $f(k) = p(k)\cdot\mu^k$, gdzie p jest wielomianem stopnia s (funkcje o tej postaci mają dla nas największe znaczenie) i μ nie jest miejscem zerowym wielomianu charakterystycznego w, to istnieje rozwiązanie szczególne o postaci $q(k) \cdot \mu^k$, gdzie q jest wielomianem stopnia s. Jeśli liczba μ jest miejscem zerowym o krotności r wielomianu charakterystycznego, to pewne rozwiązanie szczególne równania (*) ma postać $k^r q(k) \cdot \mu^k$, gdzie wielomian q ma stopień s; współczynniki tego wielomianu otrzymamy, podstawiając odpowiednie wyrażenie do równania.

<u>Przykład.</u> Niech $a_k = 2a_{k-1} + k$, oraz $a_0 = 0$. Funkcja f jest tu wielomianem stopnia 1, liczba $\mu = 1$ nie jest miejscem zerowym wielomianu $w(\lambda) = \lambda - 2$, zatem pewne rozwiązanie szczególne ma postać $a_k = ck + d$. Podstawiamy,

$$\begin{aligned} ck + d &= 2\bigl(c(k-1) + d\bigr) + k, \\ ck + d &= 2ck + 2(d - c) + k, \\ -ck + 2c - d &= k, \end{aligned}$$

skąd wynika $-c = 1$, $2c - d = 0$, czyli $c = -1$, $d = -2$. Rozwiązanie równania jednorodnego $a_k = 2a_{k-1}$ ma postać $a_k = e \cdot 2^k$, zatem rozwiązanie ogólne naszego równania niejednorodnego ma postać $a_k = -k - 2 + e \cdot 2^k$. Na podstawie warunku początkowego $-2 + e \cdot 2^0 = 0$, skąd $e = 2$. Poszukiwanym rozwiązaniem równania jest ciąg $a_k = 2^{k+1} - k - 2$.

Więcej przykładów poznamy, analizując konkretne algorytmy.

Zadania i problemy

1. Udowodnij, że algorytm

```
for ( x = a, e = b, z = 1; ; ) {
  if ( e % 2 == 1 ) z *= x;
  e /= 2;
  if ( e == 0 ) break;
  x *= x;
}
```

   jest poprawnym algorytmem obliczania liczby $z = a^b$ dla $a, b \in \mathbb{N}$ (o ile nie wystąpi nadmiar w obliczeniach); wszystkie zmienne są typu `int`.
   <u>Wskazówka:</u> Przedstaw wykładnik `b` i kolejne wartości zmiennej `e` w układzie dwójkowym.
2. Wskaż w algorytmie z poprzedniego zadania instrukcje, które można uznać za operacje dominujące. Jaka jest złożoność tego algorytmu?
3. Algorytm FFT obliczania dyskretnej transformaty Fouriera (informacja, co to jest, będzie podana później) ciągu liczb (zespolonych) o długości n wykonuje $f(n) = cn(n_1 + \ldots + n_k)$ działań arytmetycznych, gdzie c jest stałą dodatnią niezależną od n, natomiast $n_1, \ldots, n_k$ są to liczby pierwsze, których iloczynem jest n.

   Znajdź funkcje monotoniczne g i h, takie że $f(n) = O(g(n))$ oraz $f(n) = \Omega(h(n))$.

   Czy istnieje taka funkcja monotoniczna k, że $f(n) = \Theta(k(n))$?
4. Zadanie polega na obliczeniu iloczynu trzech macierzy: $D = ABC$, gdzie $A \in \mathbb{R}^{m,n}$, $B \in \mathbb{R}^{n,k}$ i $C \in \mathbb{R}^{k,l}$. Podaj koszty (czasowe i pamięciowe) dwóch algorytmów rozwiązywania tego zadania oparte na „zwykłym” wzorze na iloczyn macierzy. Pierwszy z tych algorytmów oblicza najpierw $E = AB$, a następnie $D = EC$, a drugi najpierw $F = BC$, a następnie $D = AF$.

   Napisz podprogram w C, który na podstawie liczb m, n, k, l podaje informację, który z tych dwóch algorytmów wykonuje mniej działań.
5. Dla zadania mnożenia macierzy $A \in \mathbb{R}^{m,n}$ i $B \in \mathbb{R}^{n,k}$ dana jest tylko sumaryczna liczba współczynników tych macierzy: $l = (m + k)n$. Jaki może być maksymalny koszt mnożenia (w „zwykły” sposób) tych macierzy?

   Porównaj otrzymany wynik z kosztem mnożenia macierzy, jeśli wiadomo, że $m = n = k$ (tj. $l = 2n^2$), a także w przypadku, gdy $m = k = 1$ (tj. $l = 2n$).
6. Na podstawie definicji udowodnij, że dowolny wielomian w zmiennej n stopnia k jest funkcją rzędu co najwyżej n^k (czyli $w(n) = O(n^k)$).
7. Oblicz złożoność pesymistyczną i optymistyczną algorytmu sortowania przez wstawianie (z pierwszego wykładu), przyjmując za operację dominującą porównywanie elementów sortowanego ciągu.

Zakładając, że wszystkie permutacje, które porządkują ciąg, są jednakowo prawdopodobne, oblicz złożoność średnią tego algorytmu.

Jakie są rzędy wszystkich tych złożoności?

8. Rozważmy zadanie mnożenia liczb n-cyfrowych (którego wynikiem jest liczba $2n$-cyfrowa). Dodanie dwóch takich liczb jest wykonalne kosztem $O(n)$ operacji, ponieważ trzeba wykonać odpowiednie działanie na n parach cyfr, do których dochodzą przeniesienia. Natomiast mnożenie w „zwykły” sposób zabiera $O(n^2)$ operacji; wszystkie cyfry jednego czynnika mnożymy przez każdą cyfrę drugiego czynnika, a potem trzeba dodać wyniki tych mnożeń (odpowiednio je przesuwając — to jest algorytm mnożenia „pisemnego”).

   Aby uzyskać algorytm o mniejszej złożoności, przedstawimy czynniki, które są liczbami n-cyfrowymi (dla $n > 1$), w postaci $a + bx$ i $c + dx$, za pomocą liczb $\lceil n/2 \rceil$-cyfrowych a, b, c, d i odpowiednio dobranej liczby x ($x = 2^{\lceil n/2 \rceil}$ albo $x = 10^{\lceil n/2 \rceil}$, zależnie od tego, jakiej podstawy układu używamy). Jeśli pomnożymy

$$(a + bx)(c + dx) = ac + (bc + ad)x + bdx^2,$$

   to sprowadzimy zadanie do czterech mnożeń liczb $n/2$-cyfrowych (dla uproszczenia zaniedbuję zaokrąglenie $n/2$ w górę); trzeba obliczyć ac, bc, ad i bd, a następnie wykonać dodawania (w czasie proporcjonalnym do n). W ten sposób otrzymujemy rekurencyjny algorytm mnożenia w czasie cn^2, bo najwięcej czasu zabierają w nim mnożenia; $4 \cdot c(n/2)^2 = cn^2$.

   Jeśli jednak obliczymy

$$e = (a + b)(c + d) = ac + ad + bc + bd,$$

   oraz iloczyny ac i bd, to możemy następnie obliczyć $(bc + ad) = e - ac - bd$. W ten sposób sprowadziliśmy zadanie do obliczenia tylko trzech iloczynów liczb $n/2$-cyfrowych (ściślej biorąc, liczby $a + c$ oraz $b + d$ mogą być $(\lceil n/2 \rceil + 1)$-cyfrowe, ale to zaniedbamy, tak samo jak koszt dodawań i odejmowań — w dokładnych rachunkach oczywiście nie można robić takich zaniedbań, ale tu chodzi o przedstawienie idei).

   Napisz odpowiednie równanie różnicowe i wyznacz na jego podstawie rząd złożoności opisanego wyżej algorytmu mnożenia liczb.

9. Algorytm „zwykły” mnożenia macierzy $n \times n$ wymaga wykonania $2n^3 - n^2$ działań arytmetycznych (mnożeń i dodawań zmiennopozycyjnych).
   Algorytm Strassena rozwiązuje to samo zadanie kosztem $7n^{\log_2 7} - 6n^2$ działań.
   Dla jakich n algorytm Strassena wykonuje mniej działań niż algorytm „zwykły”?

Typy danych w języku C

Typ zmiennej jest zbiorem wartości, które tej zmiennej można nadać. Ponadto typ jest związany z konkretnym sposobem reprezentowania tych wartości i w szczególności określa ilość miejsca zajmowanego przez zmienną. Na przykład typ `double` jest pewnym podzbiorem zbioru liczb rzeczywistych $\mathbb{R}$; w każdej implementacji języka C określa się, z ilu bitów składa się reprezentacja elementu tego podzbioru (obecnie chyba we wszystkich implementacjach języka C elementy typu `double` są reprezentowane za pomocą 64 bitów, zgodnie ze standardem IEEE-754, którym zajmiemy się później). Typ `int` jest podzbiorem zbioru liczb całkowitych $\mathbb{Z}$ i jest to (zwykle, ale nie zawsze) podzbiór typu `double`, ale reprezentacja elementu typu `int` jest *inna* niż reprezentacja elementu typu `double`; w szczególności może ona zajmować inną ilość miejsca w pamięci.

Typy dzielą się na typy podstawowe i typy pochodne, wśród tych ostatnich wyróżniamy typy złożone, czyli takie, których wartości składają się z elementów, które można wyodrębnić i przetwarzać osobno.

Typy podstawowe

Typy podstawowe są liczbowe; są to skończone podzbiory zbioru liczb całkowitych $\mathbb{Z}$, lub rzeczywistych $\mathbb{R}$.

Każdy typ całkowity składa się *ze wszystkich* liczb całkowitych zawartych między liczbą najmniejszą i największą w typie. Określenie typu całkowitego ma postać słowa kluczowego `char` albo `int`, które możemy poprzedzić dodatkowymi słowami kluczowymi, określającymi zakres reprezentowalnych liczb. Może tu wystąpić słowo `signed` *albo* `unsigned`; w pierwszym przypadku możemy reprezentować liczby o dowolnym znaku, w drugim tylko nieujemne. Ponadto przed słowem `int` może wystąpić słowo `short` *albo* `long`. W pierwszym przypadku zakres reprezentowalnych liczb jest mniejszy niż w drugim (ale większy niż w typie `unsigned char`). W rzeczywistości typ `int` może być tożsamy z typem `short int` albo z typem `long int`; to zależy od kompilatora (obecnie częściej zachodzi ta druga możliwość). Również od kompilatora zależy, czy typ `char` jest tożsamy z typem `unsigned char`, czy z typem `signed char`. Jeszcze jedno: jeśli napiszemy `signed`, `unsigned`, `short` lub `long`, to możemy pominąć słowo kluczowe `int`. Jeśli nie napiszemy `unsigned` przed lub zamiast `int`, to typ zawiera liczby ujemne.

Zakresy liczb w poszczególnych typach całkowitych są podane w pliku nagłówkowym `limits.h` (w systemie UNIX i podobnych plik ten jest zwykle w katalogu `/usr/include`). W tabelce niżej są podane nazwy stałych

symbolicznych (określonych za pomocą dyrektywy preprocesora `#define`) i ich przykładowe wartości.

typ	wartość minimalna	wartość maksymalna
`signed char`	`SCHAR_MIN, -128`	`SCHAR_MAX, 127`
`unsigned char`	`0`	`UCHAR_MAX, 255`
`signed short`	`SHRT_MIN, -32768`	`SHRT_MAX, 32767`
`unsigned short`	`0`	`USHRT_MAX, 65535`
`signed long`	`LONG_MIN, -2147483648`	`LONG_MAX, 2147483647`
`unsigned long`	`0`	`ULONG_MAX, 4294967295`
`int`	`INT_MIN`	`INT_MAX`
`unsigned int`	`0`	`UINT_MAX`

Oprócz typów opisanych wyżej możemy definiować typy wyliczeniowe, przez podanie identyfikatorów, które stają się stałymi całkowitymi. Przykład był podany wcześniej.

Typy zmiennopozycyjne to `float`, `double` i `long double`; będzie o nich mowa później.

Dodatkowy typ podstawowy jest identyfikowany przez słowo kluczowe `void`; zbiór jego elementów jest pusty i ich reprezentacja nie zajmuje miejsca w pamięci. Typ `void` jest często potrzebny do określania typów pochodnych.

Typy pochodne

Typy pochodne są tworzone za pomocą typów podstawowych; występują wśród nich typy tablicowe, typy strukturalne i typy wskaźnikowe. Omówimy je kolejno.

Typy tablicowe opisują tablice, tj. struktury danych składające się z elementów jednakowego typu, umieszczonych w pamięci obok siebie. Aby uzyskać dostęp do jednego z nich, należy podać w nawiasach kwadratowych jego indeks, czyli liczbę całkowitą (może ona być wartością wyrażenia), nie mniejszą niż 0 i *mniejszą* niż długość tablicy. W języku C nie możemy przypisywać tablic w całości. Jeśli więc mamy dwie zmienne tablicowe (tego samego typu), to chcąc skopiować dane z jednej tablicy do drugiej, musimy użyć odpowiedniej pętli (lub procedury standardowej `memcpy` lub `memmove`).

Tablice mogą być wielowymiarowe, tj. element tablicy może być identyfikowany za pomocą więcej niż jednego indeksu. W takim przypadku każdy indeks piszemy w osobnych nawiasach kwadratowych, np. `b[3][14]`. Kolejność elementów takiej tablicy w pamięci jest następująca: najpierw są umieszczane wszystkie elementy o *pierwszym* indeksie równym 0, bezpośrednio po nich elementy o pierwszym indeksie równym 1 itd. Zasada ta w naturalny sposób rozszerza się na tablice o więcej niż dwóch wymiarach.

Przykładowe deklaracje zmiennych tablicowych jedno- i dwuwymiarowej mają następującą postać:

```
int a[10];
float b[10][20];
```

Jeśli tablice przekazujemy jako parametry, to możemy nie podawać długości (przykład był w poprzednim wykładzie), ale dotyczy to tylko pierwszego indeksu. Możemy zatem na przykład napisać procedurę o nagłówku

```
void ZrobCos ( int a[], float b[][20] )
```

Parametr tablicowy jest w rzeczywistości zmienną wskaźnikową; jego wartość jest adresem pierwszego elementu tablicy, a *nie zmienną tablicową*, której początkowa wartość jest kopią tablicy podanej w wywołaniu procedury.

Typy strukturalne obejmują struktury i unie; zacznijmy od tych pierwszych. W programowaniu nieraz zachodzi potrzeba reprezentowania obiektu, który składa się z kilku, często różnych, części. Na przykład data składa się z trzech liczb: numeru roku, miesiąca i dnia w miesiącu — wszystko to są wprawdzie liczby całkowite, ale numer miesiąca nie bywa większy niż 12, a numer dnia nie przekracza 31. Dane osobowe kogoś mogą składać się z imienia i nazwiska (będących napisami, przechowywanymi w tablicach), oraz daty urodzenia (która sama składa się z trzech liczb). Aby określić zmienną składającą się z takich części, możemy napisać

```
struct {
  char imie[30], nazwisko[30];
  struct {
    int rok;
    unsigned char miesiac, dzien;
  } data_ur;
} osoba;
```

W ten sposób zadeklarowaliśmy zmienną o nazwie `osoba`, typu będącego

strukturą; składa się ona z trzech pól, o nazwach odpowiednio `imie`, `nazwisko` i `data_ur`. Pierwsze dwa pola są tablicami, w których można przechowywać napisy o maksymalnie 29 znakach (dlaczego?). Trzecie pole jest natomiast strukturą, która też ma trzy pola. Wyrażenia dające dostęp do poszczególnych pól zmiennej `osoba` są następujące:

```
osoba.imie
osoba.nazwisko
osoba.data_ur
osoba.data_ur.rok
osoba.data_ur.miesiac
osoba.data_ur.dzien
```

Jeśli chcemy w innym miejscu programu zadeklarować zmienną tego samego typu, to możemy powtórzyć cały opis struktury; identyczny opis umożliwia m.in. przypisywanie wartości odpowiednich zmiennych, ale powtarzanie opisu byłoby niewygodne, pracochłonne i nadzwyczaj podatne na błędy. Dlatego lepiej jest użyć deklaracji ze słowem kluczowym typedef. Przykład opisany wyżej należałoby zaprogramować tak:

```
typedef struct {
          int rok;
          unsigned char miesiac, dzien;
        } data;

typedef struct {
          char imie[30], nazwisko[30];
          data data_ur;
        } dane_os;

dane_os osoba;
```

Słowo kluczowe typedef powoduje, że identyfikatory po opisie typu, które w innym przypadku byłyby nazwami zmiennych, stają się nazwami typu, i dalej możemy ich użyć w wielu miejscach programu. Osobne zdefiniowanie typu `data` jest opłacalne, ponieważ przyczynia się do czytelności programu, a ponadto daty mogą być w programie używane nie tylko jako części danych osobowych.

Za pomocą typedef możemy wprowadzać więcej niż jedną nazwę typu; przypuśćmy, że w programie przetwarzane są punkty i wektory na płaszczyźnie, oraz liczby zespolone. Każdy z tych obiektów jest reprezentowany przez parę liczb zmiennopozycyjnych, ale poszczególne rodzaje obiektów mają inną interpretację i dlatego byłoby dobrze używać odpowiedniej nazwy typu dla każdej zmiennej. W tym celu możemy napisać

```
typedef struct { float x, y; } point2, vector2, complex;
```

albo równoważnie

```
typedef struct {
          float x, y;
        } point2;

typedef point2 vector2, complex;
```

W ten sposób jeden typ otrzymuje trzy nazwy; dalej w programie wypadałoby używać ich zgodnie z naszą interpretacją zmiennych.

Inaczej niż tablice, struktury możemy przypisywać w całości; spowoduje to skopiowanie wartości wszystkich pól. Jeśli pole jest tablicą, to cała ta tablica też zostanie skopiowana. Przypomnijmy, że przypisanie następuje podczas wykonywania operatora przypisywania „=", oraz podczas przekazywania parametrów wywoływanemu podprogramowi. Zatem, jeśli parametr procedury ma typ strukturalny, którego jedno lub więcej pól to tablice, wtedy tablice te, jako część struktury, będą kopiami tablic w zmiennej strukturalnej będącej parametrem aktualnym (tj. podanej w miejscu wywołania procedury).

Wartość procedury może być strukturalna; możemy więc napisać np. procedurę mnożenia liczb zespolonych w taki sposób:

```
complex MultComplex ( complex a, complex b )
{
  complex wynik;

  wynik.x = a.x*b.x - a.y*b.y;
  wynik.y = a.x*b.y + a.y*b.x;
  return wynik;
} /*MultComplex*/
```

Podobne do struktur są unie, które też zawierają pola. O ile pola struktury są położone w pamięci operacyjnej *obok siebie*, to pola unii są położone *w tym*

samym miejscu. Do czego to służy? Zobaczmy przykład.

```
typedef union {
          char rz;
          struct { char rz; float waga; char rodzaj; } niedzwiedz;
          struct { char rz; float waga; float DlugoscSkrzydel; } kura;
          struct { char rz; float waga; char jadowity; } waz;
        } zwierze;
```

Zmiennych typu `zwierze` możemy używać do przechowywania informacji na temat zwierząt różnych gatunków — niedźwiedzi, kur i węży. Istnieją informacje istotne dla wszystkich gatunków, na przykład waga. Inne informacje, istotne dla jednego gatunku, są zbędne dla pozostałych, na przykład informacja, czy dany zwierzak jest jadowity (tylko wąż), albo czy jest pluszowy (tylko niedźwiedź). Gdybyśmy chcieli zdefiniować strukturę z wszystkimi polami potrzebnymi dla jakichś gatunków, to struktura taka zajmowałaby dużo miejsca, z którego znaczna część byłaby zawsze niewykorzystana (czyli marnowalibyśmy pamięć operacyjną). Tymczasem unia umożliwia zajmowanie tylko takiej ilości pamięci, jaka wystarczy dla jednego („największego") gatunku.

Zauważmy, że pola unii w powyższym przykładzie są strukturami, a pola każdej struktury zajmują miejsca obok siebie. Ponadto: pierwsze pole unii, o nazwie `rz`, określa rodzaj zwierzęcia; możemy się umówić, że dla niedźwiedzi pole to ma wartość `0`, dla kur `1`, a dla węży `2` (uwaga: lepszym rozwiązaniem byłoby zdefiniowanie odpowiedniego typu wyliczeniowego i używanie identyfikatorów zamiast liczb). Pole o takiej nazwie i tego samego typu jest pierwszym polem każdej struktury i w pamięci zajmuje to samo miejsce. Dzięki temu, na przykład przypisując niedźwiedziowi wagę, nie zniszczymy informacji, która umożliwia odróżnienie niedźwiedzia od kury.

Tak samo jak dla struktur, dostęp do pól unii uzyskujemy za pomocą operatora „`.`", na przykład, mając zmienną `zw` typu `zwierze`, możemy napisać `zw.rz`, `zw.kura.waga` itd.

Wartości każdego typu wskaźnikowego są adresami w pamięci operacyjnej. Z dowolnym typem wskaźnikowym jest związana informacja, jak należy interpretować zawartość miejsca w pamięci o danym adresie, czyli na przykład jakiego typu jest zmienna wskazywana przez daną zmienną wskaźnikową.

Zmienne wskaźnikowe deklaruje się za pomocą operatora „*". Na przykład deklaracja

```
float a, *b, c, *d;
```

wprowadza cztery zmienne, z których a i c są typu float, a zmienne b i d są wskaźnikami do typu float. Należy pamiętać, że zadeklarowanie zmiennej wskaźnikowej (lub jakiejkolwiek) bez nadania jej wartości początkowej oznacza, że wartość tej zmiennej jest nieokreślona — mając zmienną wskaźnikową, jesteśmy w stanie przypisywać jej dowolne adresy i na autorze programu spoczywa odpowiedzialność za to, czy wartość zmiennej wskaźnikowej jest adresem zmiennej odpowiedniego typu. Kontynuując powyższy przykład, możemy napisać takie instrukcje:

```
b = &a;       /* zmiennej b przypisujemy adres zmiennej a */
d = &c;       /* zmiennej d przypisujemy adres zmiennej c */
a = 3.14;     /* nadajemy wartość zmiennej a */
*d = 2.0**b;  /* zmiennej c jest przypisywana podwojona wartość a */
```

Typy wskaźnikowe możemy również określać przy użyciu typedef. Na przykład deklaracja

```
typedef struct {
          float x, y;
        } point2, *point2p;
```

opisuje typ strukturalny, który otrzymuje nazwę point2 oraz typ wskaźnikowy, któremu zostaje nadana nazwa point2p. Możemy dalej w programie napisać

```
point2 a;
point2p ap;

ap = &a;
(*ap).x = 3.14;
(*ap).y = 2.78;
```

Wykonanie tych instrukcji spowoduje nadanie odpowiednich wartości polom zmiennej strukturalnej a. Zwróćmy uwagę na nawiasy wokół wyrażenia *ap — są one konieczne, ponieważ operator „." ma wyższy priorytet niż „*" i zadaniem nawiasów jest wymuszenie właściwej interpretacji całego wyrażenia. Bez nawiasów mielibyśmy błąd w programie, ponieważ kompilator uznałby, że chcemy sięgnąć

pod adres wskazywany przez pole ap.x, ale to pole nie istnieje (zmienna ap nie jest strukturą) i w szczególności nie jest wskaźnikiem. Ponieważ wskaźniki do struktur są w C wykorzystywane bardzo często (zwłaszcza jako parametry procedur), więc aby nieco skrócić i uczytelnić programy, możemy użyć operatora „->". W powyższym przykładzie ostatnie dwie instrukcje możemy zapisać tak:

```
ap->x = 3.14;
ap->y = 2.78;
```

Operator „->" ma taki sam priorytet jak „.", czyli najwyższy.

Cechą charakterystyczną języka C jest to, że nazwa zmiennej tablicowej pełni rolę wskaźnika pierwszego elementu (tj. tego, którego indeks jest równy 0). Przetwarzanie tablic (np. obliczanie adresu elementu o dowolnym indeksie) polega na dodaniu do adresu pierwszego elementu iloczynu indeksu i długości (w bajtach) miejsca zajmowanego przez element. Dowolny wskaźnik może być potraktowany jak adres początku tablicy. Z tego powodu często przekazując tablicę jako parametr, określa się odpowiedni parametr formalny jako wskaźnik. Na przykład:

```
float Suma ( int n, float *a )
{
  float suma;
  int i;

  for ( i = 0, suma = 0.0;  i < n;  i++ )
    suma += a[i];
  return suma;
} /*Suma*/

int main (void)
{
  float s, tab[10];
  ...
  s = Suma ( 10, tab );
  ...
} /*main*/
```

W powyższym programie parametr a procedury Suma jest wskaźnikiem zmiennej typu float, ale jest traktowany jak tablica. Oczywiście, w wywołaniu procedury musi być jako drugi parametr podana tablica o odpowiedniej długości — odpowiedzialność za to spoczywa na autorze programu.

Utożsamienie tablic ze wskaźnikami umożliwia m.in. przekazywanie jako parametru *części tablicy*. Przypuśćmy, że chcielibyśmy użyć przedstawionego wyżej podprogramu do obliczenia sumy elementów tablicy od miejsca trzeciego do siódmego (włącznie). W tym celu wystarczy napisać instrukcję

```
s = Suma ( 5, &tab[3] );
```

Oprócz adresów zmiennych, wskaźniki mogą przechowywać adresy podprogramów. Rozważmy przykład programu rozwiązującego metodą bisekcji równanie nieliniowe. Podprogram `f`, obliczający wartość funkcji, której miejsce zerowe należy znaleźć, możemy przekazać jako parametr procedury `Bisekcja`, której nagłówek zmienimy na taki:

```
void Bisekcja ( float (*f)(float x), float a, float b, float eps,
                float *x, char *error )
```

Opis pierwszego parametru wygląda na dość skomplikowany, ale to tylko pozory. Parametr ten ma nazwę `f` i jest wskaźnikiem do podprogramu, który ma jeden parametr typu <u>float</u> i który oblicza wartość typu <u>float</u>. W procedurze Bisekcja wywołania podprogramu przekazanego jako parametr mogą mieć postać np. `(*f)(a)`, ale możemy je również zostawić w dotychczasowej postaci `f(a)`.

Instrukcję, w której wywołujemy procedurę `Bisekcja` o powyższym nagłówku, zapiszemy w postaci

```
  Bisekcja ( &f, 1.0, 3.0, 1.0e-6, &x, &err );
```

ale pierwszy parametr zamiast `&f` może mieć też postać `f`; kompilator to uzna za poprawne. Opisana zmiana podprogramu `Bisekcja` umożliwia wykorzystanie go do rozwiązania wielu równań przez jeden program, przy czym każde z tych równań może być określone za pomocą innego podprogramu, który musi tylko mieć odpowiedni nagłówek — obliczać wartość typu <u>float</u> i mieć jeden parametr, typu <u>float</u> (i oczywiście funkcja, której wartość jest obliczana przez taki podprogram musi być ciągła, czego kompilator nie zweryfikuje).

Konwersje typów

Często w obliczeniach wartość wyrażenia pewnego typu, czyli reprezentowaną w pewien sposób, należy przekształcić do innego typu — przykładem niech będzie zamiana liczby całkowitej na zmiennopozycyjną. Nazywa się to <u>konwersją typu</u>. W języku C mamy *konwersję automatyczną* i *konwersję jawną*.

<u>Konwersja automatyczna</u> jest dokonywana na przykład wtedy, gdy wykonujemy działanie arytmetyczne, którego argumenty są liczbami reprezentowanymi na różne sposoby — jeśli jeden z argumentów jest całkowity, a drugi zmiennopozycyjny, to argument całkowity jest najpierw przekształcany do postaci zmiennopozycyjnej. Inny przykład, to działanie arytmetyczne na dwóch argumentach całkowitych, z których pierwszy jest typu <u>char</u> lub <u>short</u>, a drugi typu <u>long</u> — zawsze przekształceniu poddawany jest ten argument, którego typ ma mniejszy zakres wartości.

Konwersja automatyczna zachodzi również podczas przypisywania; obliczona wartość wyrażenia typu <u>float</u> lub <u>double</u> przed przypisaniem zmiennej typu <u>int</u> jest obcinana (tj. odrzucana jest część ułamkowa) i przekształcana do odpowiedniej postaci. Podobnie, przypisanie wartości typu <u>int</u> zmiennej typu <u>char</u> wiąże się z odrzuceniem najbardziej znaczących bitów. Zauważmy, że taka konwersja może doprowadzić do utraty części informacji. Dlatego kompilator może instrukcję wymagającą takiej konwersji uznać za poprawną (i przetłumaczyć program), ale na wszelki wypadek wypisze ostrzeżenie.

Jeśli programista wie (albo nie wie), czego chce, to ostrzeżenie zignoruje, ale aby ono się nie pojawiło, zastosuje <u>jawną konwersję</u>. W tym celu przed wyrażeniem, które ma być poddane konwersji, należy napisać <u>operator rzutowania</u> <u>typu</u>. Jest on nazwą typu ujętą w nawiasy, na przykład

```
int i;  float x;

x = 3.14;  i = (int)x;  /* zmienna i otrzymuje wartość 3 */
```

W razie potrzeby wyrażenie można zamknąć w nawiasach — zobacz tabelkę z priorytetami operatorów.

Operator rzutowania typu, jeśli służy do konwersji jednego typu liczbowego na inny, powoduje wyznaczenie odpowiedniej reprezentacji liczby (być może zaokrąglonej), tj. nowego ciągu bitów o odpowiedniej długości. Jeśli wyrażenie poddawane konwersji jest wskaźnikiem, to zastosowanie operatora konwersji umożliwia skopiowanie adresu (bez jego interpretowania). W tym przypadku reprezentacje wartości obu typów mają identyczną długość i postać — podczas działania programu (skompilowanego) wskaźnik jest identyfikatorem miejsca w pamięci operacyjnej, ale nie zawiera żadnej informacji na temat zawartości tego miejsca. Informacja taka występuje tylko w tekście źródłowym programu, dzięki czemu kompilator jest w stanie spradwzić poprawność i wygenerować kod dostosowany do typu zawartości miejsc wskazywanych przez wskaźniki. Na przykład wartość zmiennej typu int* jest adresem w pamięci operacyjnej. Może to być również wartość zmiennej typu float*, ale próba przypisania wartości jednej z tych zmiennych drugiej zmiennej zakończy się błędem kompilacji lub ostrzeżeniem. Jeśli mamy w programie zmienne

```
int *a;  float *b;
```

to kod skompilowanego wyrażenia `*a` będzie umieszczać zawartość pamięci wskazywaną przez zmienną `a` w rejestrze procesora dla liczb całkowitych, zaś kod skompilowanego wyrażenia `*b` będzie umieszczać zawartość pamięci w rejestrze zmiennopozycyjnym.

Pisząc w programie operator rzutowania typu, np. `(float*)`, programista zawiadamia kompilator, że bierze na siebie odpowiedzialność za poprawność przypisania takiego jak `b = (float*)a;`.

Zadania i problemy

1. Napisy, tj. ciągi znaków (liczb reprezentujących znaki w kodzie ASCII) są zwykle w C przechowywane w tablicach. Ostatnia z tych liczb jest równa 0, po czym poznaje się koniec napisu. Taka reprezentacja napisów znana jest pod nazwą ASCIIZ.

   Największy kłopot z napisami, i w ogóle z tablicami w C, polega na słabej kontroli nad długością tablicy — często nie dysponujemy informacją, czy dana tablica jest dostatecznie długa aby pomieścić przetwarzany napis, lub czy napis (np. wprowadzany przez użytkownika programu) ma długość nie przekraczającą pojemności tablicy, w której chcemy go umieścić.

   Napisz podprogram obliczający długość napisu przekazanego jako parametr.
2. Napisz podprogram, który kopiuje początkowy fragment napisu danego jako parametr, nie dłuższy niż pewna liczba `n`, do tablicy danej jako drugi parametr.
3. Napisz podprogram, który sprawdza, czy napis dany jako pierwszy parametr jest częścią napisu danego jako drugi parametr.
4. Napisz podprogram, który porównuje leksykograficznie dwa napisy.
5. Napisz podprogram, który usuwa z danego napisu fragment od podanego miejsca o podanej długości. Uwaga: podprogram musi działać poprawnie także wtedy, gdy fragment do usunięcia, określony przez parametry, kończy się za końcem przetwarzanego napisu.
6. Napisz podprogram, który sprawdza, czy dany napis jest palindromem.
7. Napisz podprogram, który dokonuje konwersji napisu (będącego ciągiem cyfr) na liczbę całkowitą.
8. Napisz podprogram, który dokonuje konwersji liczby całkowitej na napis — ciąg cyfr.
9. Biblioteka standardowa języka C zawiera dużo procedur przetwarzania napisów; nagłówki tych procedur znajdują się w plikach `string.h` i `stdlib.h`. Aby poznać opis procedury, a także wiele innych informacji, można wydać polecenie `man nazwa` lub `info nazwa`, gdzie `nazwa` jest nazwą interesującej nas procedury.

   Użyj polecenia `man` do otrzymania opisów procedur `strcpy`, `strncpy`, `strlen`, `strcat`, `strncat`, `atoi`, `atof` (na początek tyle wystarczy) i zapoznaj się z tymi opisami.

Abstrakcyjne struktury danych

Dane przetwarzane przez programy są przechowywane w zmiennych określonego typu. Do tej pory mieliśmy do czynienia ze zmiennymi zbudowanymi w konkretny sposób, np. tablicami lub strukturami, ale takie spojrzenie nie bardzo uwzględnia specyfikę problemu, który ma być rozwiązywany: jeśli patrzymy na tablicę tylko jak na zbiór jednakowych „komórek” umieszczonych w pamięci obok siebie, to może nam umknąć cel, dla którego te komórki zostały zarezerwowane.

Pojęcie abstrakcyjnej struktury danych ma na celu określenie sposobu wykorzystania zmiennych, w których te dane są przechowywane i w szczególności określenie dozwolonych operacji, które mogą być na tych danych wykonywane. Słowo „abstrakcyjna” oznacza tu, że nie jest istotne, jaka jest dokładna reprezentacja danych. Ważne jest tylko to, co z daną strukturą możemy robić, bo w szczególności tylko to jest istotne dla algorytmu, który tę strukturę wykorzystuje. Oczywiście w programie trzeba określić wszystkie szczegóły — to się nazywa implementacją struktury.

Najczęściej używanymi abstrakcyjnymi strukturami danych są stosy, kolejki, kolejki priorytetowe i rozmaite słowniki.

Stosy

Stos jest swego rodzaju magazynem danych jednakowego rodzaju; dane (np. struktury określonego typu) można na niego wstawiać i zdejmować. Cechą charakterystyczną stosu jest to, że jeśli wstawimy pewne dane na stos, to możemy je pozdejmować *w kolejności odwrotnej do wstawiania* (postępujemy zupełnie tak samo, jak ze stosem kartek, którego jednak nie wolno nam odwrócić i niczego nie możemy wyciągnąć ze środka). Stos jest określany skrótowcem LIFO, od angielskiego *last in, first out*.

Dla stosu mamy zatem cztery podstawowe operacje: inicjalizację, wstawienie elementu, zdjęcie elementu oraz badanie, czy stos jest pusty. Każdy magazyn, a więc także stos, ma ograniczoną pojemność, w związku z czym próba wstawienia elementu, który się nie mieści, zakończy się zerwaniem obliczeń — może się przydać dodatkowa procedura badająca, czy stos jest pełny, ale zwykle rozpatrując algorytmy w pierwszym podejściu nie zwracamy na to uwagi (bo są ważniejsze rzeczy). Oczywiście, pisząc program musimy to uwzględnić. Zerwanie obliczeń musi być również skutkiem próby zdjęcia elementu z pustego stosu (i na to zwracamy uwagę od początku).

Powiedzmy zatem, że mamy definicję typu danych, które będą przechowywane w stosie:

```
typedef struct { ... } element;
```

Operacje na stosie takich elementów mogą być wykonywane przez podprogramy o następujących prototypach:

```
void InitStack ( void );
```

Po wykonaniu tej procedury stos istnieje i jest pusty.

```
void Push ( element el );
```

Procedura `Push` umieszcza na stosie kopię wartości parametru `el`.

```
void Pop ( element *el );
```

Procedura `Pop` zdejmuje element ze stosu i przypisuje jego wartość zmiennej wskazywanej przez parametr. Zwróćmy uwagę na różnicę w sposobie przekazywania parametrów procedur `Push` i `Pop`.

```
char StackEmpty ( void );
```

Wartością funkcji `StackEmpty` jest `1`, jeśli stos jest pusty, albo `0`, jeśli na stosie jest przynajmniej jeden element.

Użyjemy stosu, w którym mogą być przechowywane znaki (elementy typu char) do rozwiązania następującego problemu: w tablicy mamy pewien tekst, w którym występują nawiasy kilku rodzajów: `'('`, `')'`, `'['`, `']'`, `'{'`, `'}'`, `'<'`, `'>'`. Nawiasy te występują w parach — w każdej parze jest nawias otwierający i nawias zamykający. Zadanie polega na zbadaniu, czy w danym tekście nawiasy są rozmieszczone w sposób *poprawny*: po każdym nawiasie otwierającym ma występować odpowiedni nawias zamykający, przy czym nawias zamykający dowolnego rodzaju nie może wystąpić, jeśli ostatni otwarty i niezamknięty nawias jest innego rodzaju.

Przykłady (znaki inne niż nawiasy są nieistotne): `(){}[<({})>]` — poprawny, `({)}` — niepoprawny, `<[]` — niepoprawny, `}(){` — niepoprawny.

Podprogram sprawdzający poprawność napisu o długości n, umieszczonego w tablicy t, może być taki:

```
char NawiasyPoprawne ( char t[], int n )
{
  int i;  char c;

  InitStack ();
  for ( i = 0; i < n; i++ ) /* tablicę t indeksujemy od 0 do n-1 */
    switch ( t[i] ) {
  case '(': Push ( ')' );  break;
  case '[': Push ( ']' );  break;
  case '<': Push ( '>' );  break;
  case '{': Push ( '}' );  break;
  case ')': case ']': case '>': case '}':
      if ( StackEmpty () )
        return 0;
      Pop ( &c );
      if ( c != t[i] )
        return 0;
      break;
  default:
      break;
    }
  return StackEmpty ();
} /*NawiasyPoprawne*/
```

Zauważmy, że są trzy możliwe rodzaje błędów rozmieszczenia nawiasów w napisie (podane przykłady błędnych napisów ilustrują wszystkie te rodzaje). Każdy z błędów jest wykrywany w innym miejscu algorytmu. Powrót z procedury z wartością 0 następuje po wykryciu błędu polegającego na osiągnięciu końca napisu z niezamkniętymi nawiasami, a także w razie odkrycia nawiasu zamykającego, który nie pasuje do nawiasu otwierającego.

„Prawdziwa" procedura będzie miała jeszcze jeden element. Jeśli utworzenie stosu powoduje trwałe efekty uboczne (np. rezerwację pamięci — nie było jeszcze o tym mowy, ale będzie), to ostatnią rzeczą, jaką musi wykonać podprogram jest „posprzątanie po sobie". Dla stosu może istnieć dodatkowa procedura, której wywołanie powoduje jego likwidację. Jeśli stos jest zaimplementowany w statycznej tablicy (jak poniżej), to niczego nie trzeba sprzątać.


```
#define MAXSTACK 1000
element tabst[MAXSTACK];
int sp;

void InitStack ( void )
{
  sp = -1;
} /*InitStack*/

void Push ( element el )
{
  if ( sp < MAXSTACK-1 )
    tabst[++sp] = el;
  else Error ();
} /*Push*/

void Pop ( element *el )
{
  if ( sp >= 0 )
    *el = tabst[sp--];
  else Error ();
} /*Pop*/

char StackEmpty ( void )
{
  return sp == -1;
} /*StackEmpty*/
```

Operatory zwiększania o 1 i zmniejszania o 1 są tu użyte w sposób, którego konsekwentne stosowanie daje bardzo krótkie i kompletnie niezrozumiałe programy w C. Wyrażenie `tabst[++sp]` oznacza, że wartość zmiennej `sp` należy zwiększyć o 1, a *następnie* użyć do indeksowania tablicy. Wyrażenie `tabst[sp--]` oznacza, że wartość zmiennej `sp` ma zostać użyta jako indeks do tablicy, a nową, mniejszą o 1 wartość zmienna `sp` ma otrzymać *potem*.

W powyższej implementacji stosu kolejno wstawiane na stos elementy są umieszczane w tablicy `tabst`. Zmienna `sp` jest tzw. <u>wskaźnikiem stosu</u>. Jej wartość jest indeksem wskazującym <u>wierzchołek stosu</u>, czyli ostatnie zajęte miejsce w tablicy `tabst`. Procedura `Error`, którą należy oczywiście dopisać do programu, ma za zadanie zerwanie obliczeń w sytuacji błędnej (można jej dorobić parametr

— napis, który wyjaśnia przyczynę; procedura przed zatrzymaniem programu wyświetliłaby ten napis na ekranie). Do zakończenia działania programu może być użyta procedura `exit`, jej prototyp znajduje się w pliku nagłówkowym `stdlib.h`.

Kolejki

Kolejka jest to struktura danych określana skrótem FIFO, od angielskiego *first in, first out* — to też jest magazyn danych jednakowego rodzaju, ale wyjmuje się z niej element, który został w kolejce umieszczony najdawniej. Operacje na kolejce są realizowane przez następujące podprogramy:

```
void InitQueue ( void );
```

Powyższa procedura tworzy pustą kolejkę.

```
void Enqueue ( element el );
```

Procedura `Enqueue` wstawia do kolejki element będący wartością parametru `el`.

```
void Dequeue ( element *el );
```

Procedura `Dequeue` usuwa z kolejki pierwszy element i przypisuje go zmiennej wskazywanej przez parametr `el`. Próba usunięcia elementu z pustej kolejki kończy się fatalnie.

```
char QueueEmpty ( void );
```

Wartość funkcji `QueueEmpty` informuje o tym czy kolejka jest pusta.

Kolejki najczęściej są używane jako bufory, tj. „przechowalnie” danych, które są „produkowane” przez pewien algorytm, a następnie dalej przetwarzane przez inny, przy czym kolejność przetwarzania ma być identyczna jak kolejność „produkowania”. Często opłaca się „hurtowe” przetwarzanie danych wtedy, gdy nazbiera się ich dostatecznie dużo, bo np. daje się to zrobić sprawniej niż przetwarzanie danych pojedynczo. Inne ważne zastosowania kolejek występują w przeszukiwaniu grafów. O tym będzie mowa później.

Implementacja kolejki w statycznej tablicy może być następująca:


```
#define MAXQUEUE 1000
element tabqu [ MAXQUEUE+1 ];
int fr, en;

void InitQueue ( void )
{
  fr = en = 0;
} /*InitQueue*/

void Enqueue ( element el )
{
  tabqu[en++] = el;
  if ( en > MAXQUEUE ) en = 0;
  if ( en == fr ) Error ();
} /*Enqueue*/

void Dequeue ( element *el )
{
  if ( fr != en ) {
    *el = tabqu[fr++];
    if ( fr > MAXQUEUE ) fr = 0;
  }
  else Error ();
} /*Dequeue*/

char QueueEmpty ( void )
{
  return fr == en;
} /*QueueEmpty*/
```

Elementy wstawione do kolejki są przechowywane w tablicy `tabqu`. Zmienna `en` wskazuje na koniec kolejki, tj. jest indeksem miejsca, w którym zostanie zapamiętany następny wstawiony element. Zmienna `fr` wskazuje początek kolejki, tj. pierwszy element, który zostanie z niej wyjęty. Jeśli obie zmienne mają tę samą wartość, to kolejka jest pusta, tj. wszystkie wstawione elementy zostały z niej wyjęte. Zauważmy, że ten sposób reprezentowania informacji o tym, czy kolejka jest pusta, wymaga, aby tablica miała o jedno miejsce więcej niż pojemność kolejki (bo w przeciwnym razie sytuacja, gdy w kolejce mamy `MAXQUEUE` elementów byłaby nieodróżnialna od sytuacji, gdy kolejka jest pusta).

Tablica używana w implementacji kolejki działa jako bufor cykliczny. Kolejne elementy są zapamiętywane na kolejnych miejscach. Po dojściu do końca tablicy wskaźniki początku i końca kolejki (zmienne `fr` i `en`) otrzymują wartość 0. W ten sposób w tablicy mamy dwie części — zajętą i wolną, przy czym w danej chwili jedna z tych części może zajmować początek i koniec tablicy.

Zauważmy, że tak stos, jak i kolejka, mogą być zrealizowane tak, że każda operacja jest wykonywana przy użyciu co najwyżej stałej liczby operacji (czyli „w czasie stałym"). Kolejki priorytetowe, opisane niżej, nie mogą być zrealizowane w ten sposób. Liczba czynności wykonywanych podczas wstawiania lub wyjmowania elementu z takiej kolejki zależy od liczby n pozostałych elementów w niej przechowywanych. Jest to istotne dla złożoności algorytmów korzystających z abstrakcyjnych struktur danych. Operację wstawienia lub usunięcia elementu można liczyć jako jedną operację tylko wtedy, gdy jej koszt jest niezależny od rozmiaru zadania (w szczególności od liczby elementów obecnych w danej chwili w stosie, kolejce lub kolejce priorytetowej).

Kolejki priorytetowe

Kolejka priorytetowa różni się od zwykłej kolejki lub stosu tym, że o kolejności wyjmowania z niej elementów nie decyduje kolejność ich wstawiania, ale tzw. priorytety przechowywanych w niej elementów. Zakładamy, że każdy element ma określony priorytet, który można porównać z priorytetem dowolnego innego elementu. Z kolejki priorytetowej wyjmujemy element o najwyższym priorytecie spośród elementów aktualnie w niej obecnych. Jeśli na przykład elementy są liczbami, to możemy chcieć wydostać z kolejki największą z nich. Jeśli różne elementy mają ten sam priorytet, to zadowala nas dowolny z nich.

Przyjrzymy się trzem różnym implementacjom kolejki priorytetowej. Pierwsza z nich jest tablicą nieuporządkowaną. Wstawienie do kolejki polega na zapamiętaniu elementu bezpośrednio za ostatnim elementem obecnym w kolejce, natomiast wyjęcie elementu wymaga wyszukania elementu o największym priorytecie. Liczba potrzebnych do tego operacji jest proporcjonalna do liczby elementów w kolejce (bo wszystkie trzeba zbadać).

```
#define NMAX 1000

typedef struct { ... } element;
typedef struct {
          int first;
          element tab[NMAX];
        } PrioQueue;
```



```
void InitPrioQueue ( PrioQueue *q )
{
  q->first = 0;
} /*InitPrioQueue*/

void InsertPrioQueue ( PrioQueue *q, element el )
{
  if ( q->first < NMAX )
    q->tab[q->first++] = el;
  else Error ();  /* kolejka pełna */
} /*InsertPrioQueue*/

void RemoveMaxPrioQueue ( PrioQueue *q, element *el )
{
  int i, j;

  if ( q->first > 0 ) {
    for ( j = 0, i = 1;  i < q->first;  i++ )
      if ( WiekszyPrio ( q->tab[i], q->tab[j] ) )
        j = i;
    *el = q->tab[j];
    q->tab[j] = q->tab[--q->first];
  }
  else Error ();  /* kolejka pusta */
} /*RemoveMaxPrioQueue*/
```

W powyższej implementacji do porównywania priorytetów używa się funkcji `WiekszyPrio` — zwróćmy uwagę, że w ten sposób nie zakładamy niczego o naturze obiektów przechowywanych w kolejce, ani o sposobie określenia priorytetów (wystarczy nam jedynie pewność, że wszystkie elementy mają określone priorytety, które można uporządkować).

Jeśli w kolejce jest $n > 0$ elementów, to są one przechowywane w tablicy na pozycjach od 0 do $n-1$, przy czym liczba n jest wartością pola `first` struktury opisującej kolejkę (jest to indeks pierwszego wolnego miejsca w tablicy, stąd taka nazwa). Zwracam uwagę na sposób, w jaki wartość pola `first` jest zmieniana. Mamy instrukcje

```
q->tab[q->first++] = el;  oraz
q->tab[j] = q->tab[--q->first];
```

Element zapisujemy w tablicy *przed* zwiększeniem, a „wyjmujemy” z końca tablicy *po* zmniejszeniu indeksu („wyjęty” element trafia do „dziury” po elemencie o dotychczas największym priorytecie).

Druga implementacja tworzy tablicę uporządkowaną — kolejne elementy mają coraz większy priorytet. Definicje typów, funkcja `WiekszyPrio` i procedura `InitPrioQueue` są takie same, zatem niżej są tylko procedury wstawiania i wyjmowania elementów:

```
void InsertPrioQueue ( PrioQueue *q, element el )
{
  int i;

  if ( q->first < NMAX ) {
    for ( i = q->first-1; i >= 0 && WiekszyPrio(q->tab[i], el); i-- )
      q->tab[i+1] = q->tab[i];
    q->tab[i+1] = el;
    q->first ++;
  }
  else Error ();  /* kolejka pełna */
} /*InsertPrioQueue*/

void RemoveMaxPrioQueue ( PrioQueue *q, element *el )
{
  if ( q->first > 0 )
    *el = q->tab[--q->first];
  else Error ();  /* kolejka pusta */
} /*RemoveMaxPrioQueue*/
```

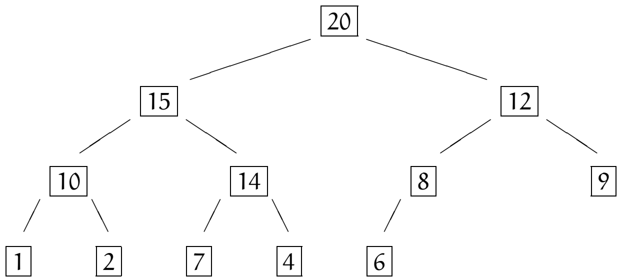
Łatwo jest sprawdzić, że w pierwszej implementacji koszt wstawiania elementu do kolejki jest stały, a koszt wyjmowania elementu jest proporcjonalny do liczby obecnych w niej elementów. W drugiej implementacji jest odwrotnie; aby wstawić element trzeba się napracować (ale niekoniecznie aż tyle, ile wyjmując element z kolejki zaimplementowanej w pierwszy sposób — złożoność optymistyczna jest nawet rzędu $O(1)$), a wyjąć element można w czasie stałym. Można zatem pytać, czy jest możliwa taka implementacja kolejki priorytetowej, w której obie operacje są wykonalne kosztem stałej liczby operacji.

Odpowiedź na to pytanie jest przecząca (co wkrótce udowodnimy). Można jednak zrealizować kolejkę priorytetową tak, aby koszt wstawiania i wyjmowania

elementu był proporcjonalny do logarytmu liczby obecnych w niej elementów. Do tego służy struktura danych zwana kopcem (ang. *heap*), którą zajmiemy się teraz.

Kopiec jest zbudowany tak: na dwóch elementach „ustawiamy” trzeci, którego priorytet jest większy. Mając dwa kopce, każdy z trzema elementami, możemy na nich „ustawić” siódmy element, którego priorytet jest większy od priorytetów elementów na wierzchołkach obu mniejszych kopców. To samo postępowanie możemy powtarzać rekurencyjnie. Otrzymany w ten sposób kopiec o wysokości h zawiera $2^h - 1$ elementów.

Jeśli chcemy ustawić kopiec z innej niż pewna potęga dwójki zmniejszona o 1 liczby elementów, to przyjmiemy, że na każdym poziomie rekurencyjnego określenia niekompletny może być „drugi” kopiec, przy czym brakujące elementy znalazłyby się na samym dole, a pierwszy może być niekompletny tylko wtedy, gdy drugi kopiec jest kompletny i o 1 niższy. Przykład (w którym elementy są liczbami, określającymi ich priorytety) jest na rysunku.



Pokazany wyżej kopiec jest szczególnym przypadkiem drzewa binarnego. Drzewo składa się z wierzchołków i krawędzi. Jeden z wierzchołków jest nazywany korzeniem i tradycyjnie na rysunkach jest umieszczany na górze (w naszym przykładzie w korzeniu jest liczba 20). Idąc od korzenia w dół wzdłuż krawędzi możemy dojść do każdego wierzchołka, przy czym taka droga do każdego wierzchołka jest tylko jedna. Jeśli z wierzchołka nie wychodzi żadna krawędź, to jest on nazywany liściem, a w przeciwnym razie wierzchołkiem wewnętrznym. Wierzchołek wewnętrzny ma dwa poddrzewa, do korzeni których prowadzą wychodzące z niego krawędzie (ale jedno z poddrzew, w naszym przypadku zawsze prawe, może być puste). Opisywane tu drzewo jest pełne, tj. nie można zbudować drzewa binarnego o mniejszej wysokości i o tej samej liczbie wierzchołków.

Opisana struktura kopca jest łatwa do reprezentowania w tablicy indeksowanej od 0. Korzeń przechowujemy na miejscu pierwszym. Korzenie jego poddrzew na miejscach 1 i 2. Ogólnie, jeśli na miejscu k-tym przechowujemy pewien wierzchołek wewnętrzny, to korzenie jego poddrzew są na miejscach $2k+1$ i $2k+2$. W ten sposób wszystkie liście mamy na końcu tablicy. Zawartość tablicy z kopcem pokazanym wyżej jest taka:

20 15 12 10 14 8 9 1 2 7 4 6

Okazuje się, że można do kopca z n elementami wstawić nowy element w czasie $O(\log n)$, a także usunąć w tym czasie element o największym priorytecie. Co więcej, okazuje się, że mając n elementów w tablicy (nieuporządkowanej) można skonstruować z nich kopiec w czasie $O(n)$, a zatem szybciej niż w przypadku wstawiania do kopca każdego elementu osobno.

Zadania i problemy

1. Napisz zestaw procedur, które implementują dwa stosy w jednej tablicy. Przepełnienie jednego lub drugiego stosu może się zdarzyć tylko wtedy, gdy *suma* liczb elementów, które mają być wstawione na oba stosy jest za duża.
2. Napisz zestaw procedur, które implementują kolejkę w tablicy. Pojemność kolejki ma być równa długości tablicy (a nie o jeden mniejsza, jak w implementacji podanej na wykładzie).
3. Napisz procedurę, która na szachownicy o wymiarach $n \times n$ umieszcza n hetmanów tak, aby żaden nie bił innego.
4. Dany jest ciąg liczb $c_1, \dots, c_n$. Zadanie polega na znalezieniu ciągu dodatnich liczb $b_1, \dots, b_n$, takich że dla każdego i jest $i - b_i \geq 0$, dla $k = i - b_i + 1, \dots, i$ jest $c_k \leq c_i$ oraz jeśli $i > b_i$, to $c_{i-b_i} > c_i$.

   Zadanie to jest łatwe do rozwiązania kosztem $O(n^2)$ operacji, za pomocą dwóch zagnieżdżonych pętli for. Aby rozwiązać je kosztem $O(n)$ operacji, można użyć stosu. Będą na nim przechowywane indeksy elementów ciągu (c_i), z których składa się najdłuższy podciąg malejący, którego ostatnim elementem jest element właśnie badany.

```
{
  for ( i = 1; i <= n; i++ ) b[i] = 1;
  c[0] = ∞;  /* strażnik */
  InitStack ();
  Push ( 0 );
  for ( i = 1; i <= n; i++ ) {
    Pop ( &k );
    while ( c[k] <= c[i] ) {
      b[i] += b[k];
      Pop ( &k );
    }
    Push ( k );  Push ( i );
  }
}
```

   Zbadaj, jak działa powyższa instrukcja (i w szczególności jak zmienia się zawartość stosu) dla ciągu

   20 15 10 12 15 15 14 13 18 22 9

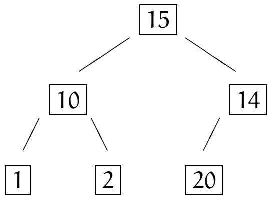
   Uzasadnij, dlaczego koszt jest proporcjonalny do n.

5. Napisz procedurę, która sprawdza poprawność rozmieszczenia nawiasów w tekście (podobnie jak procedura przedstawiona na wykładzie), przy czym dopuszcza się możliwość wystąpienia nawiasów „)”, „]”, „}” i „>”, które nie stanowią pary do odpowiedniego nawiasu otwierającego obecnego w tekście poprzedzającym, ale każdy taki nawias musi zostać „zamknięty” odpowiednio przez „(”, „[”, „{”, „<” w dalszym tekście (z uwzględnieniem pozostałych reguł poprawności takich jak na wykładzie). Na przykład napis „(>{[]][}<)” ma być uznany za poprawny.

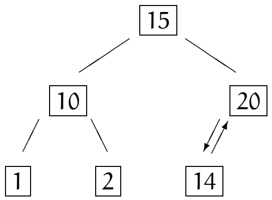
Kolejki priorytetowe, cd.

Przypuśćmy, że mamy kopiec, w którym wszystkie elementy z wyjątkiem jednego są „na swoich miejscach”. Określenie „na swoich miejscach” oznacza, że każdy węzeł drzewa zawiera element, którego priorytet jest większy niż priorytety korzeni jego poddrzew. Możemy zatem mieć najwyżej jeden element, który zaburza ten porządek. Celem jest znalezienie algorytmu, który przywraca właściwe uporządkowanie elementów w kopcu.

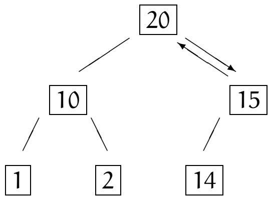
Zbadamy dwa przypadki: w pierwszym element o dowolnym priorytecie (który może zaburzać uporządkowanie) jest w ostatnim (skrajnie prawym) liściu na najniższym poziomie. Na rysunku to jest liść, który zawiera liczbę 20.



Uporządkowanie kopca jest zaburzone w wierzchołku drzewa, w którym jest element 14. Aby przywrócić ład i porządek w tym wierzchołku, należy go zamienić z elementem 20:



To przestawienie zaburzyło porządek w wierzchołku, którego poddrzewem jest wierzchołek, w którym porządek został przywrócony (czyli w korzeniu całego drzewa). Konieczne jest zatem drugie przestawienie:



Jeśli liść, w którym początkowo znajduje się element zaburzający porządek, jest na poziomie h (korzeń ma poziom 0), to może być potrzebne co najwyżej h przestawień (to wtedy, gdy element ma największy priorytet i wskutek przestawień ma znaleźć się w korzeniu). Jeśli w kopcu jest n elementów, łącznie z elementem, który trzeba przywołać do porządku, to liść z tym elementem jest na poziomie $h = \lfloor \log_2 n \rfloor$. Wykrycie zaburzenia porządku polega na porównaniu priorytetu danego elementu z priorytetem elementu bezpośrednio nad nim. Zauważmy, że jeśli trzeba wykonać przestawienie, to nie może ono zaburzyć porządku między elementem przestawionym „do góry” a elementem w korzeniu drugiego poddrzewa.

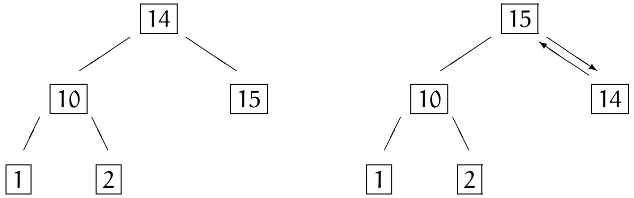
Procedura przestawiania elementów w kopcu zaimplementowanym w tablicy (korzeń jest w pozycji 0, parametr n określa położenie ostatniego elementu tablicy, jedynego, który może zaburzać porządek w kopcu) może być taka:

```
void UpHeap ( element tab[], int n )
{
  int i;

  while ( n > 0 ) {
    i = (n-1) / 2;    /* indeks wierzchołka ,,wyżej'' */
    if ( WiekszyPrio ( tab[n], tab[i] ) ) {
      Exchange ( &tab[n], &tab[i] );
      n = i;
    }
    else break;       /* dalsze przestawiania niepotrzebne */
  }
} /*UpHeap*/
```

Procedura `Exchange` zamienia wartości zmiennych wskazywanych przez parametry. Korzystając z procedury `UpHeap` możemy wstawianie do kolejki priorytetowej zaimplementować tak:

```
void InsertPrioQueue ( PrioQueue *q, element el )
{
  if ( q->first < NMAX ) {
    q->tab[q->first++] = el;
    UpHeap ( q->tab, q->first-1 );
  }
  else Error ();  /* kolejka pełna */
} /*InsertPrioQueue*/
```

Ponieważ korzeń drzewa (czyli wierzchołek kopca) zawiera element o największym priorytecie, więc dostęp do niego odbywa się w czasie stałym, ale jeśli chcemy ten element usunąć z kolejki, to trzeba miejsce po nim zapełnić (chyba, że kolejka została opróżniona) elementem o największym priorytecie spośród pozostałych. Nie można po prostu „awansować” korzenia jednego z poddrzew, bo miejsce po nim też wymaga zapełnienia itd. Można zrobić tak: do korzenia wstawić element z ostatniego liścia (który usuwamy). Oczywiście, element ten zaburza porządek, który teraz trzeba przywrócić. Trzeba go będzie zamienić z elementem w korzeniu jednego z poddrzew — tym, który ma większy priorytet. Dalej, przestawiony „w dół” element może zaburzać porządek w wierzchołku, w którym się znalazł, a zatem trzeba porządkować dalej — być może tak długo, aż trafi do liścia.



Procedurę, która przywraca porządek w drzewie, napiszemy tak, żeby element, który jest nie na swoim miejscu (tj. „za wysoko”) mógł być w dowolnym miejscu, niekoniecznie w korzeniu. Parametr `f` określa tę pozycję, zaś `l` to indeks ostatniego elementu w tablicy.

```
void DownHeap ( element tab[], int f, int l )
{
  int i, j;

  i = 2*f+1;
  while ( i <= l ) {
    j = i+1;    /* i, j - indeksy korzeni poddrzew wierzchołka f */
    if ( j <= l )
      if ( WiekszyPrio ( tab[j], tab[i] ) ) i = j;
    if ( WiekszyPrio ( tab[i], tab[f] ) ) {
      Exchange ( &tab[i], &tab[f] );
      f = i;
      i = 2*f+1;
    }
    else break;   /* dalsze przestawiania niepotrzebne */
  }
} /*DownHeap*/
```


Procedura usuwania elementu z kolejki priorytetowej może mieć postać

```
void RemoveMaxPrioQueue ( PrioQueue *q, element *el )
{
  if ( q->first > 0 ) {
    *el = q->tab[0];
    q->tab[0] = q->tab[--q->first];
    DownHeap ( q->tab, 0, q->first-1 );  /* przywracanie */
                                          /* uporządkowania kopca */
  }
  else Error ();  /* kolejka pusta */
} /*RemoveMaxPrioQueue*/
```

Procedura `DownHeap`, a zatem także procedura `RemoveMaxPrioQueue`, wykonuje co najwyżej $2\lfloor \log_2 n \rfloor$ porównań priorytetów, jeśli w kopcu (kolejce) jest n elementów. W porównaniu z poprzednimi implementacjami kolejki priorytetowej (w tablicy nieuporządkowanej i w tablicy uporządkowanej) jedna z operacji (wstawiania lub usuwania) ma większy rząd złożoności pesymistycznej, a druga mniejszy, ale suma złożoności tych operacji ma rząd mniejszy — $O(\log n)$ zamiast $O(n)$.

Algorytm HeapSort

Przypuśćmy, że do początkowo pustej kolejki priorytetowej, zrealizowanej przy użyciu kopca, mamy wstawić n elementów, a następnie wszystkie je pousuwać. Takie postępowanie jest sposobem posortowania elementów w kolejności malejących priorytetów. Policzymy wykonane przy tym (w przypadku pesymistycznym) porównania priorytetów. Maksymalny koszt wstawienia kolejno n elementów jest równy

$$T^{\mathrm{wor}}_{\mathrm{Insert}}(n) = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor.$$

Jest tak dlatego, bo wstawienie pierwszego elementu nie wymaga porównań, wstawienie następnych dwóch odbywa się z jednym porównaniem, następne cztery są wstawiane z co najwyżej dwoma porównaniami itd. Jeśli zatem $n = 2^h - 1$ dla pewnego $h \in \mathbb{N}$, to

$$T^{\mathrm{wor}}_{\mathrm{Insert}}(2^h - 1) = \sum_{k=0}^{h-1} k \cdot 2^k.$$

Wcześniej udowodniliśmy, że $\sum_{k=1}^{h} kx^{k-1} = \frac{hx^{h+1}-(h+1)x^h+1}{(1-x)^2}$, czyli

$$\sum_{k=0}^{h-1} kx^k = \frac{(h-1)x^{h+1} - hx^h + x}{(1-x)^2}. \qquad (*)$$

Na tej podstawie mamy

$$T_{\mathrm{Insert}}^{\mathrm{wor}}(2^h - 1) = 2\frac{(h-1)\cdot 2^h - h\cdot 2^{h-1} + 1}{(2-1)^2} = (h-2)\cdot 2^h + 2.$$

Podstawiając $h = \lfloor \log_2(n+1) \rfloor$ oraz $2^h = n+1$, otrzymujemy stąd

$$T_{\mathrm{Insert}}^{\mathrm{wor}}(n) = \big(\lfloor \log_2(n+1)\rfloor - 2\big)(n+1) + 2.$$

Dalej, dla dowolnego $n \in \mathbb{N}$, dla którego nie istnieje $h \in \mathbb{N}$, takie że $n = 2^h - 1$, możemy znaleźć najmniejsze $m > n$, takie że $m = 2^h - 1$ dla pewnego h. Koszt wstawienia m elementów jest oczywiście większy niż koszt wstawienia n elementów, w związku z czym

$$T_{\mathrm{Insert}}^{\mathrm{wor}}(n) < T_{\mathrm{Insert}}^{\mathrm{wor}}(m) = \big(\lfloor \log_2(m+1)\rfloor - 2\big)(m+1) + 2,$$

ale mamy $m < 2n$ oraz $\lfloor \log_2(m+1)\rfloor = \lfloor \log_2(2n+1)\rfloor$. Dzięki temu możemy oszacować (to jest zgrubne, ale chwilowo wystarczy) złożoność wstawiania dla dowolnego n:

$$T_{\mathrm{Insert}}^{\mathrm{wor}}(n) < \big(\lfloor \log_2(2n+1)\rfloor - 2\big)(2n+1) + 2,$$

czyli

$$T_{\mathrm{Insert}}^{\mathrm{wor}}(n) = O(n \log n).$$

Maksymalny koszt wyjęcia z kolejki priorytetowej wszystkich n elementów może być dwukrotnie większy, a więc złożoność pesymistyczna wyjmowania ma ten sam rząd, $O(n \log n)$.

Jeśli mamy w tablicy n elementów nieuporządkowanych (na pozycjach `0,...,n-1`), to możemy je uporządkować tak, aby dostać poprawny kopiec, kosztem $O(n)$ operacji (liczymy porównania, które są operacjami dominującymi). W tym celu kopiec będziemy budować „od dołu”, zapewniając w ten sposób, że oba poddrzewa dowolnego wierzchołka są uporządkowane, zanim zaczniemy szukać właściwego miejsca dla elementu w tym wierzchołku. Zadanie to wykonuje następujący algorytm:

```
for ( i = n/2-1; i >= 0; i-- )
  DownHeap ( tab, i, n-1 );
```


Początkowa wartość zmiennej `i` jest równa $\lfloor \frac{n}{2} \rfloor - 1$, ponieważ wszystkie pozycje w tablicy o większych indeksach reprezentują liście. Zbadajmy pesymistyczny koszt tego algorytmu. Znów przypuśćmy, że $n = 2^h - 1$ dla pewnego $h \in \mathbb{N}$. Wtedy wszystkie liście drzewa są na tym samym poziomie $(h-1)$ i jest ich $(n+1)/2$. Wszystkich poddrzew o wysokości 2 (składających się z trzech wierzchołków, w tym dwóch liści) jest $(n+1)/4$, poddrzew o wysokości 3 jest $(n+1)/8$ itd., wreszcie jest jedno drzewo o wysokości h, czyli cały kopiec do uporządkowania. Do umieszczenia na właściwej pozycji w kopcu elementu znajdującego się początkowo w korzeniu poddrzewa o wysokości k procedura `DownHeap` może potrzebować co najwyżej $2k-2$ porównań priorytetów. Zatem możemy obliczyć

$$T_{\mathrm{Order}}^{\mathrm{wor}}(2^h - 1) = \frac{n+1}{4}\cdot 2 + \frac{n+1}{8}\cdot 4 + \cdots + \frac{n+1}{n+1}\cdot 2(h-1) = \sum_{k=2}^{h} \frac{n+1}{2^k} 2(k-1) = \sum_{k=0}^{h-1} \frac{n+1}{2^k}\cdot k = 2n - 2h.$$

Aby otrzymać końcowy wynik, znów użyliśmy wzoru (*) (dla $x = \frac{1}{2}$). Jeśli $n \neq 2^h - 1$ dla każdego $h \in \mathbb{N}$, to sprawdzenie, że koszt jest rzędu n możemy wykonać tak samo, jak poprzednio. Udowodniliśmy zatem, że mając wszystkie elementy dane w tablicy, możemy uporządkować kopiec w czasie proporcjonalnym do liczby tych elementów.

Możemy teraz użyć kolejki priorytetowej zaimplementowanej w postaci kopca, do posortowania n obiektów w kolejności rosnących priorytetów, kosztem co najwyżej $O(n \log n)$ operacji. Zadanie to wykona następująca procedura:

```
void HeapSort ( element tab[], int n )
{
  int i;

  for ( i = n/2-1; i >= 0; i-- )
    DownHeap ( tab, i, n-1 );
  for ( i = n-1; i > 0; i-- ) {
    Exchange ( &tab[0], &tab[i] );
    DownHeap ( tab, 0, i-1 );
  }
} /*HeapSort*/
```

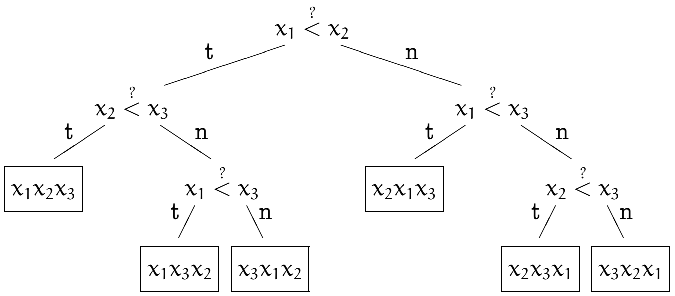
W pierwszej pętli procedura porządkuje kopiec (w czasie proporcjonalnym do n). W drugiej pętli, dla ustalonej wartości zmiennej `i` kopiec początkowo zawiera $i+1$ elementów; element o największym priorytecie (na pozycji `0` w tablicy) jest zamieniany z elementem na miejscu i-tym, co jest równoznaczne z usunięciem z kolejki elementu o największym priorytecie. Następnie kopiec jest porządkowany przez procedurę DownHeap, przy czym zawiera on teraz i elementów. Całkowity koszt tej części algorytmu jest rzędu $n \log n$.

Złożoność obliczeniowa zadania sortowania

Do tej pory koszt lub złożoność obliczeniową wiązaliśmy tylko z konkretnymi algorytmami. Mając dane zadanie i algorytm możemy zadowolić się tym algorytmem, albo poszukiwać innych, szybszych algorytmów dla tego zadania. Nie zawsze jednak znalezienie szybszego algorytmu jest możliwe, ponieważ dany algorytm może być optymalny, to znaczy może nie istnieć żaden algorytm o mniejszej złożoności (i czasem daje się to udowodnić). Złożoność zadania jest to więc złożoność optymalnego algorytmu, który to zadanie rozwiązuje.

Oczywistym oszacowaniem złożoności zadań z dołu może być rozmiar danych lub wyniku, jeśli każdy element danych trzeba wprowadzić („przeczytać”, czyli wykonać jedną operację wejścia), a każdy element wyniku „wyprowadzić” (czyli też wykonać jedną operację). Przykładem zadania, w którym to oszacowanie jest zbyt optymistyczne, jest zadanie sortowania ciągu n obiektów, przy czym obiekty te mają być porównywane parami. Zobaczymy, że żaden algorytm sortowania nie może mieć pesymistycznej złożoności (tj. maksymalnej liczby porównań dla dowolnego uporządkowania danych wejściowych) mniejszej niż $\Omega(n \log n)$.

Do udowodnienia tego twierdzenia rozważymy tzw. drzewo decyzyjne. Dowolny algorytm dokonuje kolejnych porównań sortowanych elementów, przy czym są dwa możliwe wyniki porównania: pierwszy element z porównywanej pary jest mniejszy od drugiego, albo nie. Każde porównanie dostarcza informacji, która prowadzi do otrzymania końcowego wyniku; wynikiem tym jest permutacja, której zastosowanie do danego ciągu daje ciąg uporządkowany. Na rysunku jest pokazane drzewo decyzyjne pewnego algorytmu sortującego ciąg trzyelementowy.



Sedno algorytmu sortowania polega zatem na zidentyfikowaniu na podstawie wyników porównań jednej z $n!$ permutacji, tej, która wyznacza właściwą kolejność elementów ciągu. Poszczególne permutacje są przyporządkowane liściom drzewa decyzyjnego, które jest binarne. Liczba porównań, które prowadzą do wykrycia konkretnej permutacji jest poziomem, na którym znajduje się liść związany z tą permutacją (korzeń drzewa decyzyjnego jest na poziomie 0).

Drzewo binarne, które ma k liści, i którego wszystkie wierzchołki wewnętrzne mają niepuste oba poddrzewa, ma dokładnie $k-1$ wierzchołków wewnętrznych, czego łatwy dowód indukcyjny polecam jako ćwiczenie. Różne algorytmy sortowania mają oczywiście różne drzewa decyzyjne, ale każde z tych drzew musi mieć co najmniej $2n!-1$ wierzchołków, jeśli algorytm ma poprawnie sortować ciągi o długości n. Złożoność pesymistyczna algorytmu to największy poziom, na jakim jest pewien liść. W przypadku algorytmu, którego drzewo decyzyjne jest na rysunku, złożoność jest równa 3, bo zdarzają się dane, do posortowania których algorytm ten musi wykonać aż tyle porównań.

Jeśli więc pewien algorytm sortowania jest poprawny (tj. poprawnie ustawia dane dla każdej z $n!$ możliwych permutacji danego ciągu), to jego drzewo decyzyjne ma $n!$ liści, czyli w sumie $2n!-1$ wierzchołków. Ale takie drzewo ma wysokość co najmniej $\lfloor \log_2(2n!-1) \rfloor + 1$, ponieważ drzewo binarne o wysokości h może mieć co najwyżej 2^h-1 wierzchołków. Zatem poziom przynajmniej jednego liścia nie może być mniejszy niż $\lfloor \log_2(2n!-1) \rfloor$.

Na podstawie oszacowania zgrubnego, $(n/2)^{n/2} < (2n!-1) < n^n$, prawdziwego dla każdego $n \geq 2$, możemy oszacować

$$\log_2((n/2)^{n/2}) = \frac{n}{2}(\log_2 n - 1) < \log_2(2n!-1) < \log_2(n^n) = n \log_2 n.$$

Wynika stąd istnienie stałej $c \in [\frac{1}{2}, 1]$, takiej że drzewo decyzyjne żadnego algorytmu sortowania nie może być niższe niż $cn \log_2 n$. Dokładniejsze oszacowanie stałej c można otrzymać przy użyciu wzoru Stirlinga (zobacz zadanie 5). Zauważmy, że dla każdego n istnieje drzewo binarne o $n!$ liściach, którego wszystkie liście są na poziomie co najwyżej $\lceil n \log_2 n \rceil$. Jednak przyporządkowanie par elementów do porównania poszczególnym wierzchołkom wewnętrznym takiego drzewa, dające poprawny i optymalny algorytm sortowania, jest nietrywialne. Obecnie znane są optymalne algorytmy dla liczb n nie większych niż kilkanaście, przy czym algorytmów tych raczej nie stosuje się w praktyce.

Algorytm sortowania z użyciem kopca oczywiście nie jest optymalny, ponieważ w najgorszym razie wykonuje w przybliżeniu $2n \log_2 n$ porównań. Jest on zatem gorszy od optymalnego tylko o pewien czynnik stały — oznacza to, że algorytm ten ma optymalny rząd złożoności obliczeniowej. Nawet najlepszy algorytm może być od niego szybszy tylko o pewien czynnik stały.

Pliki

Plik jest miejscem przechowywania danych w pamięci niedostępnej bezpośrednio dla programu. Najczęściej jest to miejsce w tzw. pamięci masowej (czyli na dysku, dyskietce, płycie, pendrivie itp.), ale urządzenia (klawiatura, ekran, drukarka, głośnik) mogą być dla programu dostępne jako plik. Na przykład w systemie UNIX i jego klonach absolutnie każde urządzenie jest plikiem, ale także kanały przesyłania danych między jednocześnie działającymi procesami są obsługiwane jak pliki.

Dostęp do plików jest osiągany w C za pomocą procedur biblioteki standardowej, dołączanych do programu w ostatnim etapie kompilacji. Prototypy tych procedur są umieszczone w pliku nagłówkowym `stdio.h`. Jest w nim definicja typu `FILE`, który opisuje strukturę przechowującą niezbędny zestaw informacji na temat pliku. Definicja ta zależy od systemu operacyjnego, ale w każdym z systemów używanych powszechnie, sposób używania plików w programie jest taki sam (różnice dotyczą nieistotnych dla nas tu szczegółów).

Aby czytać i pisać pliki, należy na początku programu umieścić linię

```
#include <stdio.h>
```

a następnie zadeklarować jedną lub więcej zmiennych wskaźnikowych, np.

```
FILE *mojplik;
```

W programie *nie deklarujemy* zmiennych typu `FILE`; biblioteka standardowa tworzy tablicę takich zmiennych, przy czym długość tej tablicy (zwykle co najmniej kilkanaście), będąca maksymalną liczbą plików jednocześnie otwartych przez program, zależy od systemu operacyjnego.

Związanie pliku dyskowego ze zmienną typu `FILE` — jednym z elementów tej tablicy — wykonuje procedura `fopen`, której parametry określają, co ma być z plikiem robione. Aby pisać do pliku, wykonujemy instrukcję

```
mojplik = fopen ( "nazwa.pl", "w+");
```


Jeśli plik na dysku o podanej nazwie (pierwszy parametr) nie istnieje, to zostanie wtedy utworzony (i będzie początkowo pusty). Jeśli istnieje plik o takiej nazwie, to jego zawartość zostanie skasowana. Wykonując kolejne operacje pisania, będziemy za każdym razem dopisywać dane na koniec pliku.

Jeśli chcemy dopisywać dane na koniec pliku bez kasowania jego początkowej zwartości, to drugi parametr procedury `fopen` powinien być napisem `"a+"`.

Do czytania plik zostanie przygotowany, jeśli drugi parametr procedury `fopen` jest napisem `"r+"`. Aby zamknąć plik (i zerwać jego połączenie z programem), wykonujemy instrukcję

```
fclose ( mojplik );
```

Trzy pliki są otwierane przez bibliotekę standardową automatycznie przed wykonaniem pierwszej instrukcji programu (tj. przed wywołaniem procedury `main`). Odpowiednie zmienne wskaźnikowe (*nie deklarujemy ich w programie!*, robi to dla nas plik nagłówkowy `stdio.h`) mają nazwy `stdin`, `stdout` i `stderr`. Pierwszy z tych plików służy do czytania danych z klawiatury[6], drugi i trzeci są połączone z konsolą (tj. pisanie do każdego z nich powoduje wypisywanie znaków na ekran). Do pliku `stdout` zwykle wypisujemy „normalne” wyniki, a do `stderr` komunikaty o błędach.

Pisanie danych tekstowych odbywa się za pomocą procedury `fprintf`. Pierwszy jej parametr to wskaźnik pliku (np. zmienna `mojplik`, `stdout` lub `stderr`), drugi to format, czyli napis określający sposób interpretacji kolejnych parametrów — danych do wypisania. Parametrów tych może być dowolnie wiele, ich typy mogą być różne, ale są one określone przez kolejne znaki formatu. W formacie występują „zwykłe” znaki, które będą wypisane, znaki specjalne (reprezentujące nowe linie, tabulatory itp.) oraz opisy sposobu konwersji kolejnych parametrów. Każdy taki opis zaczyna się od znaku `'%'`, po nim może być liczba (długość pola) i litera określająca typ danej.

Przykład:

```
int i, j;  float x;  char s[10];
...
fprintf ( mojplik, "%x, %3d, %8.3f, %s\n", i, j, x, s );
```

[6]To jest niezupełnie prawda, zwłaszcza w systemie UNIX, ale na początek można przyjąć, że tak jest.

Podany wyżej format opisuje cztery parametry, których wartości należy zamienić na tekst i go wypisać. Wartość zmiennej `i` będzie wypisana w postaci szesnastkowej (format `%x`), a wartość zmiennej `j` w postaci dziesiętnej (format `%3d`), przy czym jeśli to jest liczba mniej niż trzycyfrowa, to zostanie dołożone tyle spacji, aby były wypisane w sumie trzy znaki. Liczba zmiennopozycyjna `x` będzie wypisana w postaci ośmiu znaków, z czego 3 po kropce dziesiętnej. Napis `s` zostanie wypisany dosłownie (format `%s`).
Oprócz tego zostaną wypisane trzy przecinki i spacje, oraz znak końca linii (`\n`).

Do czytania z pliku służy procedura `fscanf`, na przykład dla zmiennych zadeklarowanych wyżej możemy wywołać ją tak:

```
fscanf ( mojplik, "%x %d %f %9s", &i, &j, &x, &s );
```

Zauważmy, że parametry podane po formacie są wskaźnikami do zmiennych odpowiednich typów — typy te muszą się zgadzać z kolejnymi opisami danych do przeczytania w formacie. Dokładny opis formatów jest do znalezienia w systemie pomocy kompilatora lub podręczniku systemowym. W terminalu w systemie UNIX można napisać polecenie np. `man fscanf`, które powoduje wyświetlenie szczegółowego opisu procedury `fscanf`, w tym formatu.

Wartością procedury `fscanf` jest liczba całkowita — liczba przeczytanych z pliku elementów. Może ona być mniejsza niż liczba parametrów podanych po formacie, jeśli nastąpił błąd, np. z formatu wynika, że ma być przeczytana liczba, a w pliku (po pominięciu spacji) wystąpił znak inny niż cyfra. Jeśli nie udało się niczego przeczytać (bo na przykład bieżąca pozycja w pliku jest na jego końcu), to wartość funkcji `fscanf` jest liczbą ukrytą pod nazwą symboliczną `EOF` (i tylko tej nazwy należy w programie używać).

Procedura `printf` robi to co `fprintf`, z plikiem `stdout`, którego nie podaje się (pierwszym jej parametrem jest format). Podobnie, procedura `scanf` czyta z pliku `stdin`.

Zadania i problemy

1. Tablica zawiera liczby $5, 1, 12, 3, 4, 4, 5, 6, 8, 10, 11, 7$. Narysuj drzewo z tymi liczbami, a następnie drzewa powstające po kolejnych przestawieniach, mających na celu uporządkowanie kopca (tj. ustawienie z tych liczb kolejki priorytetowej), przy czym każda liczba jest równa swojemu priorytetowi.

   Następnie narysuj drzewa powstające w wyniku umieszczania w korzeniu ostatniego liścia (który zostaje usunięty) i porządkowania kopca za pomocą procedury `DownHeap`.
2. Zbadaj koszt algorytmu znajdowania k największych liczb spośród n liczb danych w tablicy. Algorytm konstruuje kolejkę priorytetową w postaci kopca, a następnie usuwa z niej k elementów. Jaki jest rząd złożoności tego algorytmu, jeśli istnieje stała c, taka że $k \le \frac{cn}{\log_2 n}$?
3. Narysuj drzewo decyzyjne algorytmu sortowania przez wybieranie dla ciągu trzyelementowego.
4. Narysuj drzewo decyzyjne algorytmu `HeapSort` dla ciągu trzyelementowego. Ile porównań maksymalnie wykonuje ten algorytm?
5. Użyj wzoru Stirlinga

   $$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \cdot e^{\theta/12n}, \quad 0 < \theta < 1$$

   do dokładniejszego oszacowania stałej c, takiej że wysokość drzewa binarnego o $n!$ liściach i najmniejszej wysokości jest równa w przybliżeniu $cn\log_2 n$.
6. Napisz procedurę, która czyta plik tekstowy (o nazwie przekazanej jako parametr) i liczy wystąpienia w nim wszystkich znaków.
7. Napisz procedurę, która przepisuje tekst z jednego pliku do drugiego, zapisując każdą linię wspak. Procedura ma przy tym policzyć linie. Z założenia, żadna linia nie ma więcej niż 1024 znaki.

Sortowanie szybkie

Algorytm QuickSort został wynaleziony przez C.A.R. Hoare'a w 1962 r. Nadana mu wtedy nazwa odnosi się do faktu, że złożoność średnia tego algorytmu ma optymalny rząd (pesymistyczna niestety nie), co udowodnimy.

```
typedef ... typklucza;
typedef struct { typklucza k; ... } element;

int FindPivot ( element a[], int i, int j );
void Przestaw ( element *a1, element *a2 );
```

Mamy posortować n elementów tablicy w kolejności rosnących kluczy — oczywiście typ `typklucza` jest uporządkowany liniowo (dalej założymy, to są liczby, które będziemy porównywać przy użyciu operatora „<", ale gdyby to były napisy, to zamiast niego trzeba by użyć odpowiedniego podprogramu). Procedura `FindPivot` pełni w algorytmie rolę pomocniczą, choć dość istotną. Jest ona wywoływana z parametrami spełniającymi warunek $0 \leq i < j < n$ i jej wartością jest liczba ze zbioru $\{i, \dots, j\}$. Liczba ta może być dowolna, ale w jej wyborze może być uwzględniana zawartość sortowanej tablicy, dlatego tablica ta też jest przekazywana jako parametr. Procedura `FindPivot` niczego w tablicy nie zmienia, natomiast procedura `Przestaw` przestawia elementy wskazywane przez parametry. Główną część pracy w sortowaniu szybkim wykonuje podprogram `Partition`:

```
int Partition ( element a[], int i, int j )
{
  int       p, q;
  typklucza k;

  p = FindPivot ( a, i, j );        /* zawsze i ≤ p ≤ j */
  k = a[p].k;
  Przestaw ( &a[i], &a[p] );        /* element dzielący na pozycję i */
  for ( p = i, q = i+1;  q <= j;  q++ )
    if ( a[q].k < k ) {
      p++;
      Przestaw ( &a[p], &a[q] );    /* może być p == q */
    }
  Przestaw ( &a[i], &a[p] );        /* element dzielący na pozycję p */
  return p;
} /*Partition*/
```


Wartości parametrów `i` i `j` spełniają warunek $0 \leq i < j < n$ i określają fragment tablicy, w której funkcja ma poprzestawiać elementy (poza tym fragmentem zawartość tablicy ma pozostać nietknięta). Zadaniem procedury jest doprowadzenie do sytuacji, w której wszystkie elementy między pozycjami `i`-tą i `j`-tą, których klucze są *mniejsze* niż klucz elementu wskazanego przez funkcję `FindPivot` (tak zwanego elementu dzielącego) znalazły się przed tym elementem, a pozostałe za nim. Dla uproszczenia kodu element ten zostaje najpierw przestawiony na początek fragmentu (tj. na pozycję `i`-tą). Klucze wszystkich elementów są porównywane z kluczem elementu dzielącego i elementy o mniejszych kluczach są przestawiane na początek fragmentu tablicy. Ostatnie wywołanie procedury `Przestaw` przestawia element dzielący na pozycję `p` — to jest ostateczna pozycja tego elementu w posortowanej tablicy. Wszystkie elementy we fragmencie o mniejszych kluczach są umieszczone „na lewo" od elementu dzielącego, a pozostałe „na prawo". Wartością procedury `Partition` jest pozycja, na której znalazł się element dzielący.

```
void QuickSort ( element a[], int i, int j )
{
  int p;

  if ( i < j ) {
    p = Partition ( a, i, j );
    QuickSort ( a, i, p-1 );
    QuickSort ( a, p+1, j );
  }
} /*QuickSort*/
```

Aby posortować całą tablicę, należy wywołać `QuickSort ( a, 0, n-1 );`. Procedura ma posortować część tablicy od pozycji `i` do `j` włącznie. Odbywa się to w ten sposób, że wybierany jest we fragmencie tablicy element dzielący. Następnie wszystkie elementy o kluczach mniejszych od klucza elementu dzielącego trafiają przed ten element, a wszystkie elementy większe trafiają za ten element. Części fragmentu przed i za elementem dzielącym są sortowane w wywołaniach rekurencyjnych. Zauważmy, że ten algorytm sortowania (podobnie jak HeapSort, ale inaczej niż np. algorytm sortowania przez wstawianie) *nie jest* algorytmem stabilnym, tj. kolejność elementów o identycznych kluczach może ulec zmianie.

Za operację dominującą w algorytmie QuickSort można przyjąć porównania kluczy. Dla $k = j - i$, gdzie i oraz j oznaczają wartości parametrów `i` i `j`, funkcja `Partition` wykona k porównań (klucze wszystkich pozostałych elementów są porównywane z kluczem elementu dzielącego). Sortowanie fragmentu jednoelementowego ($k = 0$) nie wymaga wykonania żadnych porównań kluczy.

Największy koszt sortowania wystąpi wtedy, gdy element dzielący w każdym przetwarzanym fragmencie tablicy wypadnie na początku lub na końcu tego fragmentu. Wtedy jedno z wywołań rekurencyjnych procedury `QuickSort` otrzyma do posortowania fragment pusty, a drugie dostanie fragment o długości o 1 mniejszej. Zatem, wybierając element dzielący zawsze w najgorszy możliwy sposób, dostaniemy dla $n > 1$

$$T^{\mathrm{wor}}(n) = n - 1 + T^{\mathrm{wor}}(n-1).$$

To jest równanie różnicowe z warunkiem początkowym $T(1) = 0$, które możemy rozwiązać ogólnym sposobem podanym wcześniej (ćwiczenie). Możemy też od razu zauważyć, że

$$T^{\mathrm{wor}}(n) = \sum_{k=1}^{n} k - 1 = \frac{1}{2}n(n-1) = \Theta(n^2).$$

Rząd złożoności pesymistycznej, $\Theta(n^2)$, jest taki sam jak rząd złożoności algorytmu sortowania przez wstawianie i większy niż algorytmu HeapSort.

Najmniej porównań wykonamy wtedy, gdy element dzielący zawsze wypadnie w połowie przetwarzanego fragmentu tablicy. Dla uproszczenia załóżmy, że $n = 2^k - 1$, gdzie $k = \log_2(n+1)$ jest pewną liczbą naturalną. Wtedy mamy

$$T^{\mathrm{bst}}(n) = n - 1 + 2T^{\mathrm{bst}}((n-1)/2).$$

Jeśli oznaczymy $a_k = T^{\mathrm{bst}}(2^k - 1)$, to mamy rekurencyjną definicję ciągu

$$a_1 = 0,$$
$$a_k = 2a_{k-1} + 2^k - 2 \qquad \text{dla } k > 1.$$

Mamy tu równanie różnicowe pierwszego rzędu, którego rozwiązanie (tj. jawny wzór opisujący wyraz a_k) wyraża poszukiwany koszt. Równania takie umiemy już rozwiązywać. Rozwiązanie równania jednorodnego, $a_k = 2a_{k-1}$, ma ogólną postać $a_k = b \cdot 2^k$, a rozwiązaniem szczególnym równania niejednorodnego jest ciąg o postaci $a_k = ck \cdot 2^k + d$ (liczba 2 jest pierwiastkiem równania charakterystycznego o krotności 1). Podstawiamy:

$$\begin{aligned} ck \cdot 2^k + d &= 2\big(c(k-1) \cdot 2^{k-1} + d\big) + 2^k - 2 \\ &= 2ck \cdot 2^{k-1} - 2c \cdot 2^{k-1} + 2d + 2^k - 2, \\ \text{czyli} \quad 0 &= -c \cdot 2^k + d + 2^k - 2, \end{aligned}$$

skąd wynika $c = 1$, $d = 2$. Otrzymaliśmy rozwiązanie szczególne $a_k = k \cdot 2^k + 2$. Aby otrzymać rozwiązanie spełniające warunek początkowy, przyjmujemy $a_k = b \cdot 2^k + k \cdot 2^k + 2$ i podstawiamy $k = 1$:

$$0 = a_1 = b \cdot 2 + 1 \cdot 2 + 2,$$

skąd wynika $b = -2$. Ostatecznie, $a_k = (k-2) \cdot 2^k + 2$, czyli dla $n = 2^k - 1$ mamy

$$T^{\mathrm{bst}}(n) = (\log_2(n+1) - 2)(n+1) + 2.$$

Dla dowolnego n złożoność optymistyczną możemy oszacować z góry i z dołu, podstawiając do powyższego wzoru największe $n_1 = 2^{k_1} - 1 < n$ i najmniejsze $n_2 = 2^{k_2} - 1 > n$, gdzie $k_1, k_2 \in \mathbb{N}$, skąd otrzymujemy

$$T^{\mathrm{bst}}(n) = \Theta(n \log n).$$

Tak więc rząd złożoności optymistycznej algorytmu QuickSort jest taki sam jak rząd złożoności pesymistycznej algorytmu HeapSort i większy niż rząd złożoności optymistycznej algorytmu sortowania przez wstawianie ($\Theta(n)$).

Przypomnijmy, że złożoność średnia algorytmu sortowania przez wstawianie, *przy założeniu, że każda z $n!$ permutacji danych w tablicy jest jednakowo prawdopodobna*, jest rzędu $\Theta(n^2)$. Obliczymy rząd złożoności średniej algorytmu QuickSort, przy tym samym założeniu.

Koszt posortowania tablicy o długości n jest sumą kosztu wykonania procedury `Partition` oraz kosztów posortowania (w wywołaniach rekurencyjnych procedury `QuickSort`) dwóch fragmentów tablicy, między którymi jest element dzielący. Łatwo jest sprawdzić, że jeśli każda permutacja danych jest jednakowo prawdopodobna, oraz jeśli wartość procedury `FindPivot` nie zależy od zawartości tablicy, to prawdopodobieństwo, że pozycja p elementu dzielącego jest dowolną z liczb $0, \ldots, n-1$ jest takie samo, czyli $\frac{1}{n}$. Mniej oczywiste, ale też prawdziwe (dowód pominiemy) jest to, że po zakończeniu działania procedury `Partition` w każdym z fragmentów, które będą dalej sortowane rekurencyjnie, wszystkie permutacje danych w tych fragmentach też są jednakowo prawdopodobne. Dlatego możemy napisać

$$T^{\mathrm{ave}}(n) = n - 1 + \frac{1}{n}\sum_{p=1}^{n}\big(T^{\mathrm{ave}}(p-1) + T^{\mathrm{ave}}(n-p)\big),$$

podstawiając do prawej strony złożoności średnie algorytmu QuickSort ciągów danych o odpowiednich długościach. Oczywiście, mamy $T^{\mathrm{ave}}(0) = T^{\mathrm{ave}}(1) = 0$.

Przekształcając podany wzór, otrzymujemy

$$T^{\text{ave}}(n) = n - 1 + \frac{2}{n}\sum_{p=1}^{n} T^{\text{ave}}(p-1) = n - 1 + \frac{2}{n}\sum_{p=2}^{n-1} T^{\text{ave}}(p).$$

Ponieważ dolne oszacowanie rzędu złożoności średniej mamy (jest nim oszacowanie złożoności optymistycznej), zbadamy hipotezę, że istnieje stała c, taka że $T^{\text{ave}}(n) \le cn\ln n$. Hipoteza jest prawdziwa, jeśli dla każdego n

$$n - 1 + \frac{2c}{n}\sum_{p=2}^{n-1} p\ln p \le cn\ln n,$$

ponieważ wtedy z założenia indukcyjnego $T^{\text{ave}}(p) \le cp\ln p$ dla każdego $p < n$ wynika prawdziwość hipotezy dla $p = n$. Sumę po lewej stronie możemy oszacować za pomocą całki:

$$\sum_{p=2}^{n-1} p\ln p < \int_2^n x\ln x\,dx = \frac{1}{2}x^2\ln x - \frac{1}{4}x^2\Big|_2^n = \frac{1}{2}n^2\ln n - \frac{1}{4}n^2 - 2\ln 2 + 1$$
$$< \frac{1}{2}n^2\ln n - \frac{1}{4}n^2.$$

Przypuśćmy zatem, że

$$n - 1 + \frac{2c}{n}\left(\frac{1}{2}n^2\ln n - \frac{1}{4}n^2\right) = n - 1 + cn\ln n - \frac{c}{2}n \le cn\ln n.$$

Nierówność ta jest spełniona, jeśli przyjmiemy dowolne $c \ge 2$. W ten sposób udowodniliśmy, że

$$T^{\text{ave}}(n) \le 2n\ln n, \quad \text{czyli} \quad T^{\text{ave}}(n) \le 2n\ln n = O(n\log n),$$

a ponieważ $T^{\text{ave}}(n) \ge T^{\text{bst}}(n)$, ostatecznie mamy $T^{\text{ave}}(n) = \Theta(n\log n)$.

Pozostaje wyjaśnienie do końca roli funkcji `FindPivot`. Jak widać, ma ona zasadniczy wpływ na koszt posortowania każdych konkretnych danych; jeśli zawsze jako element dzielący wybierze ona element o największym lub najmniejszym kluczu w danym fragmencie tablicy, to koszt sortowania będzie największy. Jeśli element dzielący zawsze będzie tzw. medianą zbioru wartości kluczy w danym fragmencie tablicy, to koszt sortowania będzie optymistyczny, ale znajdowanie mediany nie ma sensu, bo wykonane przy tym porównania spowodowałyby wzrost rzędu złożoności algorytmu. Dlatego funkcja `FindPivot` albo

- stosuje prosty wzór do obliczenia indeksu elementu, który będzie elementem dzielącym, np. zwraca zawsze `i` albo `j` (co działa fatalnie, jeśli tablica jest posortowana), lub `(i+j)/2` (to akurat dla tablicy posortowanej działa najlepiej, często więc stosuje się takie rozwiązanie), albo

- losuje (z jednakowym prawdopodobieństwem) dowolną liczbę ze zbioru $\{i, \dots, j\}$ — to jest tzw. randomizacja algorytmu; zauważmy, że niezależnie od wyniku każdego losowania wynik sortowania jest poprawny, ale może być osiągnięty w różnym czasie, przy czym losowanie (jeśli jest niezależne od permutacji danych) sprawia, że rozkład prawdopodobieństwa wystąpienia określonych danych staje się nieistotny dla złożoności średniej, albo

- wykonuje niewielką liczbę porównań kluczy w odpowiednim fragmencie tablicy, w celu zapewnienia, że klucz elementu dzielącego nie będzie najmniejszy ani największy. Funkcja może wylosować trzy elementy i zwrócić wartość, która jest indeksem elementu z kluczem będącym medianą zbioru kluczy tych trzech elementów.

Dalsze ulepszenia algorytmu QuickSort mogą być takie: ponieważ dla $n < 10$ znane są proste algorytmy sortowania o mniejszej złożoności średniej (np. algorytm sortowania przez wstawianie), więc procedura `QuickSort` może, jeśli jest spełniony warunek `j-i<9`, wywołać procedurę realizującą taki szybszy algorytm. Zmniejsza to koszt sortowania o pewien czynnik stały. Rozwinięciem tego pomysłu jest zaniechanie sortowania fragmentów tablicy krótszych niż 10. Po wykonaniu takiego „niepełnego sortowania” algorytmem QuickSort należy pracę dokończyć za pomocą procedury sortowania przez wstawianie. Jeśli żaden element w tablicy nie jest dalej niż o k miejsc od miejsca, na które ma trafić, to koszt sortowania przez wstawianie jest rzędu $O(kn)$, a więc dla ustalonego k liniowy ze względu na n. Oczywiście, całkowity koszt sortowania w przypadku średnim pozostaje rzędu $\Theta(n\log n)$.

Zadania i problemy

1. Prześledź działanie algorytmu QuickSort dla tablicy, która początkowo zawiera liczby $3, 1, 4, 1, 5, 9, 2, 7, 1, 7, 2, 8$. Przyjmij, że wartość procedury `FindPivot` jest zawsze równa wartośaci parametru `i`.
2. Jak można zmodyfikować dowolny algorytm sortowania, który porównuje elementy (np. HeapSort, QuickSort), aby algorytm ten był stabilny?
3. Użyj procedury `Partition` do zrealizowania algorytmu wyszukiwania w tablicy elementu o k-tym co do wielkości kluczu. Zbadaj złożoność tego algorytmu.
4. Oblicz pesymistyczną złożoność sortowania przez wstawianie, jeśli w tablicy do posortowania każdy element jest oddalony nie dalej niż o k miejsc od miejsca, do którego trafi wskutek sortowania.
5. Napisz procedurę `QuickSort` w taki sposób, aby było w niej tylko jedno wywołanie rekurencyjne (zamiast drugiego wywołania należy zastosować pętlę).
6. Złożoność pesymistyczna algorytmu HeapSort jest rzędu $n \log n$, taki sam rząd mają złożoności optymistyczna i średnia algorytmu QuickSort. Oblicz i porównaj czynniki stałe przy wyrazach rzędu $n \log n$ dla tych złożoności (należy przyjąć wspólną podstawę logarytmu, np. 2 albo e).
7. Jaki jest rząd pesymistycznej złożoności pamięciowej algorytmu QuickSort (chodzi o pamięć dodatkową, bez uwzględnienia sortowanej tablicy — w tym przypadku należy obliczyć maksymalną głębokość wywołań rekurencyjnych, aby określić ilość miejsca potrzebnego na rekordy aktywacji).

   Podaj sposób otrzymania pesymistycznej złożoności pamięciowej rzędu $\log n$, będący modyfikacją rozwiązania zadania 5.

Sortowanie przez scalanie

Wiele algorytmów, w tym QuickSort, opiera się na pomyśle „dziel i zdobywaj”. Polega on na sprowadzeniu zadania o rozmiarze n do rozwiązania dwóch lub większej liczby takich samych zadań o rozmiarze mniejszym niż n, a następnie użycia rozwiązań tych mniejszych zadań do otrzymania rozwiązania zadania wyjściowego (w przypadku QuickSort po rozwiązaniu zadań mniejszych nic już nie trzeba robić, bo zasadnicza praca polega na dzieleniu).

Algorytmy „dziel i zdobywaj” formułuje się (i najczęściej implementuje) w sposób rekurencyjny. Algorytm taki składa się z trzech faz: dzielenia (np. zbioru danych na mniejsze podzbiory), rozwiązywania zadań mniejszych, a następnie scalania, tj. budowania rozwiązania całości.

Rozważmy algorytm sortowania przez scalanie (ang. *MergeSort*). Mając ciąg elementów o długości $n > 1$ możemy go podzielić na dwa ciągi o długościach $\lfloor \frac{n}{2} \rfloor$ oraz $\lceil \frac{n}{2} \rceil$ — to jest faza podziału. Każdy z tych ciągów sortujemy, a następnie scalamy posortowane ciągi połówkowe. Użyty niżej typ `element` jest zdefiniowany tak samo jak w poprzednio rozważanym algorytmie QuickSort.

```
void MergeSort ( element a[], element b[], int l, int p )
{
  int s, i, j, k;

  if ( p > l ) {
    s = (l+p) / 2;
    MergeSort ( a, b, l, s );
    MergeSort ( a, b, s+1, p );
    i = l;  j = s+1;  k = l;
    do {
      if ( a[i].klucz <= a[j].klucz )
        b[k++] = a[i++];
      else b[k++] = a[j++];
    } while ( i <= s && j <= p );
    if ( i <= s )
      for ( j = i; j <= s; j++ ) b[k++] = a[j];
    else
      for ( i = j; i <= p; i++ ) b[k++] = a[i];
    for ( i = l; i <= p; i++ ) a[i] = b[i];
  }
} /*MergeSort*/
```


Tablica `b` pełni tu rolę pomocniczą — podczas scalania wyników sortowania „połówek” zawartości fragmentu tablicy między pozycjami `l` i `p` (włącznie) wynik scalania procedura wpisuje do niej (a na końcu przepisuje go z powrotem do tablicy `a`). Do posortowania tablicy o długości n algorytm potrzebuje zatem pamięci dodatkowej o długości $O(n)$ (i jest to wadą tego algorytmu w porównaniu z algorytmami, które przestawiają elementy tylko w oryginalnie danej tablicy).

Możemy zauważyć, że dostęp do danych w tablicach jest realizowany w sposób sekwencyjny, a dokładniej, procedura odwołuje się do danych w każdym z dwóch fragmentów tablicy `a` po kolei i elementy te „wyjmowane” z tych fragmentów tablicy `a`, są wstawiane na kolejne miejsca odpowiedniego fragmentu tablicy `b`. Dzięki temu ta metoda, po odpowiednich modyfikacjach, może być użyta do sortowania plików (warto zaznaczyć, że dostęp do każdego elementu pliku czytanego „po kolei” w zasadzie odbywa się w stałym czasie, natomiast dostęp do elementu dowolnego odbywa się w czasie zależnym od pozycji tego elementu i elementu ostatnio czytanego i może być rzędu liczby elementów w pliku).

Jeden z wariantów modyfikacji jest taki: tworzymy dwa pliki, z których każdy zawiera połowę elementów danych. Pliki te zawierają ciągi posortowane o długości 1. Następnie, mając dwa pliki z posortowanymi ciągami o długości k, scalamy kolejne pary takich ciągów czytane z tych plików i tworzymy dwa pliki z ciągami o długości $2k$ itd (ostatni ciąg w każdym pliku może być krótszy). W ostatnim kroku, mając dwa pliki z ciągami uporządkowanymi o długościach $2^{\lceil \log_2 \frac{n}{2} \rceil}$ i $n - 2^{\lceil \log_2 \frac{n}{2} \rceil}$, scalamy te dwa ciągi w jeden.

Udowodnimy, że złożoność pesymistyczna sortowania przez scalanie jest rzędu $O(n \log n)$ (podobnie jak sortowania przy użyciu kopca), a zatem algorytm ten też jest optymalny z dokładnością do czynnika stałego. Obliczenie złożoności tego algorytmu jest jeszcze jedną okazją do przećwiczenia sposobu obliczania funkcji danych za pomocą wzorów rekurencyjnych.

Obliczymy pesymistyczną złożoność algorytmu sortowania przez scalanie. Założymy dla uproszczenia, że długość sortowanego ciągu jest całkowitą potęgą liczby 2: $n = 2^k$. Dzięki temu na każdym poziomie rekurencji fragment tablicy dzielimy na części o jednakowej długości. Obliczymy (dokładnie) maksymalną liczbę porównań kluczy. Koszt sortowania tablicy o długości n jest sumą kosztów posortowania dwóch tablic o długości $n/2$ i kosztu łączenia.

Maksymalna liczba porównań podczas łączenia dwóch posortowanych ciągów o długości $n/2$ jest równa $n - 1$. Wprowadzimy oznaczenie $a_k = T^{wor}(2^k)$, dzięki czemu równanie

$$T^{wor}(2^k) = 2T^{wor}(2^{k-1}) + 2^k - 1$$

zapiszemy w postaci

$$a_k = 2a_{k-1} + 2^k - 1,$$

w której rozpoznajemy równanie różnicowe pierwszego rzędu. Warunek początkowy jest wyznaczony przez koszt sortowania ciągu o długości 1: $a_0 = T^{wor}(1) = 0$. Rozwiązanie równania jednorodnego ma postać $e \cdot 2^k$. Ponieważ liczba 2 jest miejscem zerowym wielomianu charakterystycznego (o krotności 1), więc rozwiązanie szczególne równania niejednorodnego ma postać $bk \cdot 2^k + c$. Po podstawieniu do równania obliczamy

$$\begin{aligned} bk \cdot 2^k + c &= 2\big(b(k-1) \cdot 2^{k-1} + c\big) + 2^k - 1, \\ bk \cdot 2^k + c &= bk \cdot 2^k - b \cdot 2^k + 2c + 2^k - 1, \\ b \cdot 2^k - c &= 2^k - 1, \end{aligned}$$

skąd wynika $b = 1$, $c = 1$. Rozwiązanie ogólne ma postać $a_k = bk \cdot 2^k + c + e \cdot 2^k$. Współczynnik e obliczymy na podstawie warunku początkowego:

$$a_0 = 0 = 0 \cdot 2^0 + 1 + e \cdot 2^0,$$

czyli $e = -1$. Ostatecznie mamy $a_k = (k-1) \cdot 2^k + 1$, czyli dla $n = 2^k$

$$T^{wor}(n) = (\log_2 n - 1) \cdot n + 1 = O(n \log n).$$

Jeśli n nie jest całkowitą potęgą liczby 2, to możemy powołać się na fakt, że złożoność pesymistyczna algorytmu jest niemalejącą funkcją rozmiaru zadania, i oszacować złożoność sortowania przez złożoność dla najmniejszej potęgi dwójki większej niż n. Rząd złożoności pozostaje $O(n \log n)$.

Sortowanie licznikowe

Jak wiemy, nie może istnieć algorytm sortowania oparty na porównaniach, który sortuje każdy ciąg n-elementowy wykonując mniej niż $O(n \log n)$ porównań. Jeśli jednak zbiór wartości kluczy można odwzorować z zachowaniem uporządkowania w ograniczony zbiór liczb całkowitych, to istnieje alternatywa, która umożliwia sortowanie w czasie proporcjonalnym do n.

Pomysł polega na „rozrzucaniu” sortowanych elementów do ponumerowanych „kubełków” na podstawie wartości kluczy (nie ma tu porównań kluczy, zamiast tego na podstawie wartości klucza należy obliczyć numer kubełka). Na przykład, mając do posortowania obiekty, których klucze są słowami, możemy przygotować kubełki oznaczone literami alfabetu. Do pierwszego kubełka trafią wszystkie słowa

na literę „A”, do drugiego wszystkie na „Ą”, do trzeciego wszystkie na „B” itd. Następnie w obrębie każdego kubełka możemy posortować słowa ze względu na drugą literę, itd. Ten pomysł jeszcze nie daje algorytmu działającego w czasie liniowym, ale nieco go zmodyfikujemy.

Posortujemy tablicę elementów, których klucze są liczbami całkowitymi — typu unsigned int, o którym założymy, że każdy element tego typu zajmuje cztery bajty (liczby typu unsigned char — ośmiobitowe; bajt jest najmniejszą jednostką pamięci adresowaną przez procesory komputerów PC), $b_0, \ldots, b_3$, przy czym $x = b_0 + 256b_1 + 256^2b_2 + 256^3b_3$. Potraktujemy bajty jak cyfry rozwinięcia liczby x w układzie o podstawie 256. Aby mieć do nich dostęp, zdefiniujemy typ

```
typedef struct {
    union {
      unsigned int x;
      unsigned char b[4];
    } k;
  } typklucza;
```

Istotne założenie użyte w procedurze jest takie, że że w danym kluczu element b[0] jest najmniej znaczącym bajtem (tj. b_0), a b[3] jest bajtem najbardziej znaczącym (b_3). W związku z oparciem algorytmu o sposób przechowywania liczb w pamięci podprogram podany niżej może być *nieprzenośny* na komputery z procesorami niezgodnymi z PC, a także te, w których typ unsigned int jest zbiorem liczb 64-bitowych (to zależy od systemu operacyjnego, ewentualnie ustawień kompilatora).

Zasada działania algorytmu jest następująca: najpierw posortujemy elementy ze względu na *najmniej* znaczące bajty kluczy, tj. b_0, następnie ze względu na bajty b_1, b_2 i na końcu b_3. Stabilność algorytmu sortowania (którą musimy zapewnić) spowoduje, że jeśli dwa klucze mają identyczne bajty bardziej znaczące, a różnią się tylko bajtami mniej znaczącymi, to po posortowaniu ze względu na mniej znaczące bajty kolejność elementów z tymi kluczami w tablicy będzie właściwa i podczas sortowania ze względu na bardziej znaczące bajty (które w obu kluczach są identyczne) kolejność ta nie zostanie zmieniona.

Użyjemy 256 kubełków. Numer kubełka, do którego wrzucimy element, jest oczywiście wartością odpowiedniego bajtu. Wariant sortowania kubełkowego zrealizowany przez podaną niżej procedurę nazywa się sortowaniem licznikowym. Kubełek jest reprezentowany przez licznik elementów wrzuconych do niego. Tablicę t indeksujemy od 0 do $n-1$.

```
void CountSort ( element t[], unsigned int n )
{
  unsigned int i, j, k;
  unsigned int licznik[256];
  element u[n];

  if ( n < NMIN ) { ...  /* posortuj przez wstawianie */ }
  else {
    for ( j = 0; j < 4; j++ ) { /* od najmniej do najbardziej */
                                /* znaczących bajtów */
      for ( k = 0; k < 256; k++ ) licznik[k] = 0;
                                /* skasuj liczniki */
      for ( i = 0; i < n; i++ )
        licznik[t[i].klucz.k.b[j]] ++;
            /* teraz wiadomo, ile jest elementów w każdym kubełku */
      for ( k = 1; k < 255; k++ )
        licznik[k] += licznik[k-1];
            /* teraz wartość każdego licznika wskazuje */
            /* koniec obszaru, do którego trzeba */
            /* wyrzucić zawartość odpowiedniego kubełka */
      for ( i = n-1; i >= 0; i-- ) {
        k = t[i].klucz.k.b[j];
        u[licznik[k]] = t[i];
        licznik[k] --;
      }
      for ( i = 0; i < n; i++ ) t[i] = u[i];
    } /* for ( j ... )  */
  }
} /*CountSort*/
```

Do sortowania kubełkowego i w szczególności do podanej wyżej procedury odnoszą się następujące uwagi: czas jej działania jest proporcjonalny do sumy długości ciągu n i liczby kubełków — stąd sortowanie ciągów krótkich (krótszych niż odpowiednio dobrana stała `NMIN`, np. 30) metodą kubełkową nie opłaca się, bo lepszy wtedy jest algorytm sortowania przez wstawianie (który też jest stabilny). Ponadto procedura sortowania korzysta z pamięci dodatkowej — oprócz tablicy liczników jest potrzebna tablica `u`, do której dane są przepisywane w odpowiedniej kolejności z tablicy `t`, a na końcu przenoszone z powrotem. Można (za pomocą dalszych trików, których opis tu pominę) sortować tablice według kluczy całkowitych nie tylko dodatnich, a także kluczy zmiennopozycyjnych.

Zadania i problemy

1. Czy algorytm MergeSort jest stabilnym algorytmem sortowania (tj. czy jeśli w tablicy pewne elementy mają identyczne klucze, to kolejność tych elementów nie może zostać zmieniona)?
2. Oblicz optymistyczną złożoność sortowania przez scalanie, dla tablicy o długości $n = 2^k$.
3. Napisz procedurę sortowania pliku metodą scalania (aby otrzymać nazwy plików roboczych, wywołaj pewną funkcję typu `char*`, której dokładna treść jest tu nieistotna).
4. W tablicy a o dwóch indeksach, przyjmujących wartości od 0 do n, gdzie $n = 2^k - 1$ dla pewnego całkowitego $k > 0$, są liczby uporządkowane niemalejąco w każdym wierszu i w każdej kolumnie. Mamy następujący podprogram:

```
char Jest ( float x, int i0, int i1, int j0, int j1, int *i, int *j )
{
  int k, l;

  if ( i0+1 == i1 ) {  /* wtedy również j0+1 == j1 */
       /* zbadaj, czy któryś z czterech elementów, */
       /* a[i0][j0], a[i0][j1], a[i1][j0], a[i1][j1], jest równy x */
       /* jeśli tak, to przypisz zmiennym i, j jego indeksy, */
       /* i zwróć wartość 1, w przeciwnym razie zwróć 0 */
    .......
  }
  else {
    k = (i0+i1) / 2;  l = (j0+j1) / 2;
    if ( x == a[k][l] ) { *i = k;  *j = l;  return 1; }
    else {
      if ( Jest ( x, i0, k, l, j1, i, j ) ) return 1;
      else if ( Jest ( x, k, i1, j0, l, i, j ) ) return 1;
      else if ( x > a[k][l] ) return Jest ( x, k, i1, l, j1, i, j );
      else return Jest ( x, i0, k, j0, l, i, j );
    }
  }
} /*Jest*/
```

Możemy wywołać procedurę `Jest ( x, 0, n-1, 0, n-1, &i, &j )` w celu wyszukania w tablicy a liczby x i otrzymania jej pozycji (procedura przypisze zmiennym `i` i `j` odpowiednie indeksy), jeśli liczba ta jest obecna (procedura ma wtedy wartość 1). Wyjaśnij sposób potraktowania zasady „dziel i zdobywaj" w algorytmie realizowanym przez ten podprogram. Przyjmując za operację dominującą porównanie wartości parametru x z elementem tablicy, oblicz pesymistyczną złożoność tego algorytmu i podaj jej rząd.

5. Popraw procedurę z poprzedniego zadania tak, aby umożliwić wyszukiwanie liczb w tablicach o dowolnych wymiarach (a także w tablicach prostokątnych), oraz wyeliminować wielokrotne sprawdzania obecności liczby x w tej samej pozycji tablicy. Czy dokonana poprawka zmniejszyła rząd złożoności pesymistycznej?

Arytmetyka zmiennopozycyjna

Wszelkie dane przechowywane w pamięci (operacyjnej i masowej) komputerów są reprezentowane jako ciągi bitów, czyli cyfr dwójkowych. Różne są tylko sposoby interpretowania tych bitów — mogą to być liczby, znaki (tj. symbole alfabetu), adresy maszynowe, kody rozkazów procesora lub wzorce bitowe interpretowane jako obrazy. Oczywiście, różnie są realizowane fizyczne postaci bitów, np. jako ładunki elektryczne (w pamięci operacyjnej), namagnesowane obszary na dysku, lub zniszczone albo nietknięte miejsca w metalizowanej warstwie krążka CD. Tymi technikaliami nie warto jednak zaprzątać sobie głowy.

Zajmiemy się sposobami reprezentowania liczb przy użyciu ustalonej liczby bitów. Jest oczywiste, że ciąg bitów o długości d może mieć jedną z 2^d różnych wartości, a zatem zbiór obiektów reprezentowanych przez takie ciągi może mieć co najwyżej 2^d różnych elementów. Przy użyciu dowolnej ustalonej liczby bitów możemy reprezentować tylko elementy zbioru skończonego, choć oczywiście elementy te mogą same być zbiorami nieskończonymi.

Reprezentacje stałopozycyjne liczb

Dla porządku zacznijmy od sposobów reprezentowania liczb tzw. stałopozycyjnych. Nazwa oznacza, że w ciągu bitów reprezentujących liczbę każdy bit ma ustalone znaczenie — jest to cyfra przedstawienia liczby w układzie o podstawie 2. Liczba stałopozycyjna bez znaku, reprezentowana za pomocą bitów $b_{d-1}, \dots, b_0$ (ciągi cyfr, w tym bitów zapisujemy na papierze w kolejności *big-endian*, tj. od najbardziej do najmniej znaczącej), jest równa

$$x = \sum_{k=0}^{d-1} b_k \cdot 2^k.$$

Na przykład ciąg 8 bitów 01001101 reprezentuje liczbę

$$0 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 77.$$

Ta reprezentacja jest jednoznaczna, tj. różne ciągi bitów reprezentują różne liczby, które są całkowite, od 0 do $2^d - 1$. Działania arytmetyczne: dodawanie, odejmowanie i mnożenie, jeśli nie ma tzw. nadmiaru stałopozycyjnego, dają wynik, który daje się reprezentować. W razie nadmiaru, czyli pojawienia się wyniku, którego reprezentacja musiałaby mieć dodatkowe bity, zapamiętywane są tylko te bity, na które jest miejsce. Odrzucenie bitów pozostałych jest równoznaczne z obliczeniem reszty z dzielenia przez 2^d. Tak więc wymienione trzy działania są w rzeczywistości działaniami w pierścieniu $\mathbb{Z}_{2^d}$.

Zauważmy, że liczba 2^k, przez którą mnożymy k-ty (licząc od zera) bit, jest wybrana arbitralnie (oczywiście, taki wybór ma istotne uzasadnienie).
W ogólności można przyjąć dowolną ustaloną liczbę b i dany ciąg bitów interpretować jako liczbę

$$\sum_{k=0}^{d-1} b_k \cdot 2^{k-b},$$

co dla $b > 0$ umożliwiłoby reprezentowanie pewnych ułamków. Zauważmy, że sposób przetwarzania dwóch ciągów bitów w celu otrzymania ciągu reprezentującego sumę odpowiednich liczb, jest identyczny jak przy obliczaniu sumy liczb całkowitych. Inne musi być oczywiście mnożenie i dzielenie. Ponieważ taka reprezentacja nie jest używana zbyt często, więc nie ma rozkazów procesora realizujących takie mnożenie i dzielenie (ale można napisać w C odpowiednie podprogramy realizujące takie działania).

Aby móc reprezentować całkowite liczby ujemne, trzeba inaczej interpretować bity. Pierwszy pomysł, to interpretacja bitu b_{d-1} jako bitu znaku. Liczba reprezentowana przez ciąg $b_{d-1}, \dots, b_0$ byłaby równa

$$(-1)^{b_{d-1}} \sum_{k=0}^{d-2} b_k \cdot 2^k$$

Ta reprezentacja nie jest często używana z dwóch powodów: reprezentacja nie jest jednoznaczna, ponieważ zero ma dwie reprezentacje (a zatem, aby stwierdzić, czy wartości dwóch zmiennych są różne, nie wystarczy sprawdzić, że odpowiednie ciągi bitów są różne). Po drugie, mimo, że liczby nieujemne (od 0 do $2^{d-1} - 1$) mają identyczne reprezentacje, jak liczby stałopozycyjne bez znaku, to inne są algorytmy ich dodawania i odejmowania (i jeszcze inne musiałyby być algorytmy dodawania i odejmowania liczb o różnych reprezentacjach, a to działanie jest potrzebne często).

W związku z tym w powszechnie używanej reprezentacji stałopozycyjnej ze znakiem stosuje się tzw. kod uzupełnieniowy do dwóch. Liczba x reprezentowana przez ciąg $b_{d-1}, \dots, b_0$ jest równa

$$x = -b_{d-1} \cdot 2^{d-1} + \sum_{k=0}^{d-2} b_k \cdot 2^k.$$

Na przykład, jeśli $d = 8$, to ciąg bitów 11001101 reprezentuje liczbę

$$-1 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = -51.$$

Przy użyciu d bitów można w ten sposób reprezentować liczby całkowite od -2^{d-1} do $2^{d-1}-1$. Reprezentacja jest jednoznaczna, a dodawanie i odejmowanie liczb w takiej reprezentacji, a także dodawanie i odejmowanie liczby ze znakiem i liczby bez znaku, jest realizowane przez te same rozkazy procesora (inne muszą być tylko rozkazy mnożenia i dzielenia) — należy jednak upewnić się, że nie wystąpi nadmiar.

Reprezentacje zmiennopozycyjne

Reprezentacja zmiennopozycyjna jest najpowszechniej używanym sposobem kodowania binarnego liczb rzeczywistych. Jest ona używana wtedy, gdy trzeba reprezentować liczby o bardzo dużej lub bardzo małej (różnej od zera) wartości bezwzględnej, w którym to przypadku reprezentacja stałopozycyjna musiałaby się składać z ogromnej liczby bitów dla uniknięcia nadmiaru i niedomiaru.

Podstawowe spostrzeżenie — że ciąg d bitów może jednoznacznie określać element zbioru co najwyżej 2^d-elementowego — oznacza, że wszystkich liczb rzeczywistych, a nawet tylko liczb z dowolnego przedziału ograniczonego, których zbiór jest nieprzeliczalny, nie można reprezentować *dokładnie*. Zatem, zamiast danej liczby rzeczywistej x w ogólności możemy pamiętać element $y = \mathrm{rd}(x)$ ustalonego, skończonego podzbioru zbioru $\mathbb{R}$ (symbol „rd" jest skrótem angielskiego słowa *rounding*). Przyporządkowanie $x \to \mathrm{rd}(x)$ jest dobrane tak, aby błąd reprezentacji, równy $|x - \mathrm{rd}(x)|$, był jak najmniejszy.

Nazwa reprezentacji — zmiennopozycyjna — oznacza, że znaczenie bitów na pewnych pozycjach bywa różne, określone przez inne bity w danym ciągu. Pomysł jest podobny do tzw. półlogarytmicznego zapisu liczb dziesiętnych. Na przykład liczbę $x = 314159{,}265358979323$ możemy przedstawić w postaci iloczynu liczby z przedziału $[1, 10)$ i pewnej całkowitej potęgi podstawy układu 10:

$$x = 314159{,}265358979323 = 3{,}14159265358979323 \cdot 10^5.$$

W ten sposób (z drobnymi modyfikacjami) możemy zapisywać liczby w języku C. Najważniejsza różnica między takim zapisem i reprezentacją komputerową liczby polega na zastąpieniu podstawy układu 10 przez 2.

Liczba cyfr (bitów) reprezentacji wpływa, jak zobaczymy dalej na jej dokładność, przy czym w różnych zastosowaniach mogą być inne potrzeby. W początkach rozwoju informatyki prawie każdy producent komputerów wprowadzał inny format liczb zmiennopozycyjnych (jak wiele rzeczy, to także można robić na wiele sposobów), co prowadziło do kłopotów. Prawie wszystkie komputery

produkowane obecnie przetwarzają reprezentację liczb określoną w tzw. standardzie IEEE-754[7], dzięki czemu m.in. łatwo jest wymieniać dane w postaci binarnej między komputerami. Omówimy ten standard nieco bardziej szczegółowo. Jest rzeczą istotną, że oprócz formatów liczb zmiennopozycyjnych (tj. określenia interpretacji poszczególnych bitów) standard określa najważniejsze własności, które muszą mieć działania wykonywane na tych liczbach.

W podstawowej reprezentacji liczby w standardzie IEEE-754 najbardziej znaczący bit jest bitem znaku, a po nim występuje cecha c i mantysa m:

s	c	m

Mantysa jest liczbą rzeczywistą; jeśli reprezentuje ją ciąg bitów $b_{t-1}b_{t-2}\ldots b_1 b_0$, to $m = \sum_{k=0}^{t-1} b_k 2^{k-t}$, a zatem zawsze $0 \le m < 1$. Cecha jest liczbą całkowitą (bez znaku), która wpływa na sposób interpretacji całego ciągu bitów. Liczba reprezentowana przez taki ciąg, w zależności od cechy, jest równa

$$\begin{aligned}
x &= (-1)^s\, 2^{c-b}\,(1+m) && \text{dla } 0 < c < 2^d - 1,\\
x &= (-1)^s\, 2^{1-b}\, m && \text{dla } c = 0,\\
x &= (-1)^s\, \infty && \text{dla } c = 2^d - 1,\ m = 0,\\
x &= \mathrm{NaN}\ (\text{„nie-liczba"}) && \text{dla } c = 2^d - 1,\ m \ne 0.
\end{aligned}$$

Cechą charakterystyczną tej reprezentacji z użyciem pierwszego wzoru jest tzw. normalizacja. Mając dowolną liczbę rzeczywistą $x \ne 0$, przedstawioną w układzie dwójkowym, dobieramy cechę c (czyli równoważnie czynnik 2^{c-b}) tak, że czynnik $(1+m)$ w wyrażeniu opisującym x jest liczbą z przedziału $[1, 2)$. Jeśli otrzymana w ten sposób cecha jest za duża (większa lub równa $2^d - 1$), to mamy nadmiar zmiennopozycyjny (ang. *floating point overflow*), czyli niewykonalne zadanie reprezentowania liczby o za dużej wartości bezwzględnej.

Bardziej skomplikowana sytuacja zdarza się w przypadku, gdy cecha jest za mała (tj. gdy należałoby przyjąć $c \le 0$). Wtedy korzystamy z drugiego wzoru, w którym występuje czynnik m (przypominam, że $m \in [0, 1)$). W takim przypadku mamy do czynienia z niedomiarem zmiennopozycyjnym, czyli reprezentowaniem liczby x za pomocą mantysy o mniejszej liczbie bitów znaczących (jeśli w użyciu jest drugi wzór, to znaczące są tylko bity mantysy od pozycji najmniej znaczącej, do najbardziej znaczącej pozycji, na której jest jedynka). Najdokładniejszą reprezentacją liczb o bardzo małej wartości bezwzględnej (mniejszej niż 2^{-b-t}) jest 0. Niedomiar wiąże się z utratą dokładności reprezentacji, o czym za chwilę.

[7] *Institute of Electrical and Electronics Engineers*, 1985.

Jeśli cecha c jest równa $2^d - 1$ (gdzie d jest liczbą bitów cechy), to ciąg bitów interpretujemy jako nieskończoność (ze znakiem), albo nie-liczbę (NaN, ang. *not a number*). Sposób ich przetwarzania (w razie użycia ich jako argumentów operacji arytmetycznych) zależy od procesora, który zgodnie ze standardem powinien umożliwiać określenie sposobu działania w takim przypadku, na przykład przerwanie obliczeń lub wyprodukowanie wyniku NaN. Niestety, w językach takich jak C nie jest określona standardowa obsługa wyjątków i sterowanie mechanizmem przerwań procesora, jest to dostępne na poziomie asemblera i zawsze zależne od systemu operacyjnego.

Nie-liczby najczęściej przydają się w sytuacjach wyjątkowych. Na przykład procesory generują NaN w razie błędu, np. próby obliczenia pierwiastka z liczby ujemnej lub logarytmu liczby niedodatniej. Wartość mantysy umożliwia identyfikację powodu powstania nie-liczby. Inny sposób użycia nie-liczb (przydatny podczas uruchamiania i testowania programów) to przypisanie wartości NaN wszystkim zmiennym na początku wykonywania programu lub procedury, w której te zmienne są zadeklarowane, a następnie „normalne” obliczenia. Można w ten sposób wykryć wiele błędów typu „odwołanie do zmiennej o nienadanej wartości początkowej”.

Zastosowanie normalizacji (dla liczb reprezentowanych przy użyciu pierwszego wzoru) ma dwa skutki. Po pierwsze, każda liczba z wyjątkiem zera ma tylko jedną reprezentację (dwie reprezentacje zera mają różne bity znaku). Dzięki temu łatwo jest porównywać liczby reprezentowane w taki sposób. Niech $z(x)$ oznacza liczbę całkowitą (stałopozycyjną bez znaku), reprezentowaną przez ten sam ciąg bitów, co liczba zmiennopozycyjna x. Najbardziej znaczący bit liczby x jest bitem znaku, a po nim kolejno są bity cechy i mantysy. Załóżmy, że bit znaku w reprezentacjach pewnych liczb x i y jest równy 0. Łatwo jest przekonać się, że wynik ich porównania jest identyczny z wynikiem porównania liczb całkowitych $z(x)$ i $z(y)$. Ponadto liczba całkowita $z(+\infty)$ jest większa niż $z(x)$, a także $z(+\mathrm{NaN}) > z(+\infty)$.

Drugi skutek normalizacji to optymalne wykorzystanie bitów mantysy do uzyskania jak najmniejszego błędu względnego reprezentacji dowolnej liczby. Ponieważ każdy ciąg bitów określa inną liczbę, więc więcej różnych liczb rzeczywistych ma dokładne reprezentacje, dzięki czemu błąd reprezentacji dowolnej liczby rzeczywistej jest mniejszy. Weźmy $x \neq 0$. Istnieje liczba $f \in [0, 1)$, reprezentowana przez nieskończony ciąg cyfr dwójkowych (bitów), taka że $x = (-1)^s\, 2^{c-b}\, (1+f)$. Reprezentacja zmiennopozycyjna $\mathrm{rd}(x) = (-1)^s\, 2^{c-b}\, (1+m)$ ma cechę c i mantysę m dobrane tak, aby wartość bezwzględna różnicy $x - \mathrm{rd}(x)$ była jak najmniejsza. Ponieważ mantysa jest wielokrotnością liczby 2^{-t}, więc

wybierając odpowiednio kierunek zaokrąglania możemy zapewnić, że $|f - m| \leq 2^{-t-1}$. Błąd względny reprezentacji liczby x możemy zatem oszacować następująco:

$$\left|\frac{x-\mathrm{rd}(x)}{x}\right| = \left|\frac{(-1)^s\, 2^{c-b}\,(1+f) - (-1)^s\, 2^{c-b}\,(1+m)}{(-1)^s\, 2^{c-b}\,(1+f)}\right| \leq \frac{2^{c-b}\, 2^{-t-1}}{2^{c-b}} = 2^{-t-1}.$$

Wybierając mniej starannie kierunek zaokrąglania mamy $|x - \mathrm{rd}(x)|/|x| \leq 2^{-t}$. Natomiast w przypadku niedomiaru liczba $x = (-1)^s\, 2^{1-b}\, f$, gdzie $0 \leq f < 1$, jest przybliżana za pomocą $\mathrm{rd}(x) = (-1)^s\, 2^{1-b}\, m$ (z odpowiednio dobraną mantysą m) i mamy oszacowanie błędu za pomocą ułamka

$$\left|\frac{x-\mathrm{rd}(x)}{x}\right| = \left|\frac{f-m}{f}\right|,$$

który dla $x \to 0$ dąży do 1. Tak więc skutkiem niedomiaru jest wzrost błędu względnego reprezentacji nawet do 100%. Utrata dokładności jest stopniowa: maksymalny błąd względny reprezentacji liczb niewiele mniejszych niż 2^{-b} jest niewiele większy od 2^{-t}.

W standardzie IEEE-754 są określone reprezentacje pojedynczej i podwójnej precyzji (poza standardem istnieje też nieoficjalny i rzadko spotykany format o poczwórnej precyzji). Ponadto są określone reprezentacje rozszerzonej precyzji. W reprezentacji rozszerzonej standard określa *minimalne* liczby bitów cechy i mantysy. Mantysa jest reprezentowana za pomocą $t+1$ bitów $b_t b_{t-1} \ldots b_1 b_0$, i jest równa $m = \sum_{k=0}^{t} b_k 2^{k-t}$ (zawsze jest więc $m \in [0, 2)$); dla $c < 2^d - 1$ liczba zmiennopozycyjna rozszerzonej precyzji jest równa $x = (-1)^s 2^{c-b} m$. Jeśli $c = 2^d - 1$, to cały ciąg bitów reprezentuje $\pm\infty$ lub NaN. Reprezentacja rozszerzona nie narzuca postaci znormalizowanej, ale wyniki działań arytmetycznych, jeśli nie ma niedomiaru, są poddawane normalizacji.

Reprezentacje rozszerzonej precyzji nie zawsze są dostępne. Procesory w PC-tach obsługują reprezentację rozszerzonej precyzji podwójnej. Odbywa się to tak, że rejestry jednostki zmiennopozycyjnej przechowują liczby w formacie rozszerzonej precyzji i wszystkie obliczenia są wykonywane w tej reprezentacji. Konwersja między pojedynczą lub podwójną precyzją i rozszerzoną precyzją odbywa się podczas przesyłania danych między rejestrami i pamięcią operacyjną.

Reprezentacje liczb pojedynczej, podwójnej i podwójnej rozszerzonej precyzji na komputerach klasy PC w języku C są dostępne pod nazwami `float`, `double` i `long double`,

W tabeli niżej są przedstawione liczby bitów i inne parametry reprezentacji standardowych. Litera B oznacza całkowitą liczbę bitów, litera d — liczbę bitów cechy, t — liczbę bitów mantysy. Litera b oznacza stałą odejmowaną od cechy we wzorze opisującym reprezentowaną liczbę. Stałe $M = 2^{2^{d}-b-2}(2-2^{t})$ i $S = 2^{1-b}$ to największa i najmniejsza liczba dodatnia, które można reprezentować w postaci znormalizowanej, bez nadmiaru i niedomiaru. Błąd względny tej reprezentacji nie przewyższa $\nu = 2^{-t}$. Liczba $\mu = 2^{1-b-t}$ jest najmniejszą dodatnią liczbą zmiennopozycyjną. Liczby M, S, ν i μ są podane w przybliżeniu (tylko rząd wielkości).

	B	d	t	b	M	S	ν	μ
pojedyncza	32	8	23	127	10^{38}	10^{-38}	10^{-7}	10^{-45}
poјed. rozszerzona	44	11	31	1023	10^{308}	10^{-308}	10^{-10}	10^{-317}
podwójna	64	11	52	1023	10^{308}	10^{-308}	10^{-15}	10^{-323}
podw. rozszerzona	80	15	63	16383	10^{4932}	10^{-4932}	10^{-19}	10^{-4951}

Zmiennopozycyjne działania arytmetyczne

Działania na liczbach zmiennopozycyjnych polegają na wyznaczeniu liczby zmiennopozycyjnej możliwie mało różniącej się od wyniku „prawdziwego” działania na odpowiednich argumentach. Można to sobie wyobrażać tak, jakby po otrzymaniu „prawdziwego” wyniku procesor wyznaczał liczbę zmiennopozycyjną leżącą najbliżej tego wyniku. Techniczna realizacja wygląda tak, że rejestr zmiennopozycyjny ma trzy dodatkowe bity mantysy (za najmniej znaczącym bitem obecnym w reprezentacji). Pierwsze dwa z tych bitów są zwykłymi kolejnymi cyframi rozwinięcia binarnego mantysy wyniku działania, a trzeci jest „lepki”, tj. otrzymuje wartość 1 wtedy, gdy dowolny dalszy bit nieskończonego rozwinięcia mantysy jest różny od zera. Można udowodnić, że te trzy dodatkowe bity wystarczą do poprawnego ustalenia, w którą stronę ma nastąpić zaokrąglenie wyniku, aby otrzymać liczbę zmiennopozycyjną najbliższą liczbie danej.

Oprócz wyników czterech działań arytmetycznych, zgodnie ze standardem poprawnemu zaokrąglaniu (do najbliższej liczby zmiennopozycyjnej) musi też podlegać pierwiastek kwadratowy, a także reszta z dzielenia oraz wyniki konwersji liczb całkowitych i zmiennopozycyjnych. Wartości innych funkcji przestępnych *nie muszą* być poprawnie zaokrąglone (nie da się tego łatwo zagwarantować; dla funkcji obliczanych bezpośrednio przez procesor możemy liczyć tylko na otrzymanie małego błędu względnego). Zauważmy jednak, że poprawne zaokrąglanie dotyczy liczb w rejestrach procesora — na komputerach PC to są liczby w formacie `long double`. Jeśli liczbę taką przypiszemy zmiennej typu `float` albo `double`, to nastąpi drugie zaokrąglenie, którego wynik nie musi być najbliższy liczbie danej na początku (dokładniej: złożenie poprawnego

zaokrąglenia liczby rzeczywistej x do zmiennopozycyjnej `long double` z poprawnym zaokrągleniem tejże do formatu `float` nie musi być poprawnym zaokrągleniem x do `float`).

Wynik dowolnego działania arytmetycznego „$\diamond$” oznaczymy symbolem $\mathrm{fl}(x \diamond y)$. Jeśli nie ma nadmiaru ani niedomiaru, to

$$\mathrm{fl}(x \diamond y) = \mathrm{rd}(x \diamond y) = (x \diamond y)(1+\varepsilon),$$

gdzie $|\varepsilon| \leq \nu = 2^{-t}$. Liczba ε jest błędem <u>względnym</u> otrzymanego wyniku działania. Jeśli jest niedomiar, tj. normalizacja doprowadziłaby do powstania ujemnej cechy, to wynik działania jest obciążony błędem <u>bezwzględnym</u> α:

$$\mathrm{fl}(x \diamond y) = (x \diamond y) + \alpha,$$

gdzie $|\alpha| \leq \mu$. Jeśli nie ma niedomiaru, to to oszacowanie błędu jest oczywiście błędne. Ponieważ niedomiar (przy sensownie dobranym algorytmie numerycznym) jest raczej rzadkim wydarzeniem, więc analizę błędu często prowadzi się *tylko* w oparciu o oszacowanie błędu względnego (wzór poprzedni), które jest *błędne w razie wystąpienia niedomiaru.*

Błędy otrzymanych wyników są w większości zastosowań pomijalnie (lub dostatecznie) małe, ale czasem mogą bardzo urosnąć (jeśli wyniki działań są argumentami następnych działań). Ponadto, wskutek ich obecności działania zmiennopozycyjne <u>nie mają podstawowych własności</u> działań w ciele $\mathbb{R}$: dodawanie i mnożenie nie są łączne, mnożenie nie jest rozdzielne względem dodawania. Z tego powodu w obliczeniach numerycznych często stosuje się tzw. <u>wzory projektowane</u>, tj. wzór opisujący potrzebny wynik przekształca się do postaci algebraicznie równoważnej w ciele $\mathbb{R}$, która wytwarza mniejsze błędy (innym celem projektowania wzorów jest unikanie nadmiaru). W przedstawionych niżej rachunkach przyjmujemy (tak się robi prawie zawsze), że parametry ν i μ odpowiadają stosowanej przez nas arytmetyce (np. `float`, nawet jeśli pewne wyniki pośrednie w bardziej skomplikowanych wyrażeniach nie będą konwertowane do `float`, ponieważ będą one przechowywane tylko w rejestrach procesora, w formacie `long double`).

<u>Przykład.</u> Mając dane liczby a, b, należy obliczyć $x = a^2 - b^2$. Pierwszy sposób polega na użyciu algorytmu

$$x = a * a - b * b.$$

Przy użyciu tego wzoru otrzymamy

$$\tilde{x} = \big(a^2(1+\varepsilon_1) - b^2(1+\varepsilon_2)\big)(1+\varepsilon_3) = (a^2-b^2)(1+\gamma).$$

Błąd względny γ otrzymanego wyniku zależy od błędów wytworzonych w wykonanych działaniach; wiemy, że $|\varepsilon_1|, |\varepsilon_2|, |\varepsilon_3| < \nu$. Mamy

$$\begin{aligned}\gamma &= \frac{a^2}{a^2-b^2}(\varepsilon_1+\varepsilon_3+\varepsilon_1\varepsilon_3) - \frac{b^2}{a^2-b^2}(\varepsilon_2+\varepsilon_3+\varepsilon_2\varepsilon_3)\\ &\approx \frac{a^2}{a^2-b^2}(\varepsilon_1+\varepsilon_3) - \frac{b^2}{a^2-b^2}(\varepsilon_2+\varepsilon_3)\end{aligned}$$

(ostatnie wyrażenie powstało przez odrzucenie składników „małych wyższego rzędu — liczba mniejsza niż 10^{-14} jest zaniedbywalna w porównaniu z 10^{-7}). Możemy oszacować

$$|\gamma| \leq 2\Big(\frac{a^2}{|a^2-b^2|} + \frac{b^2}{|a^2-b^2|}\Big)\nu.$$

Ponieważ oba ułamki są zawsze większe niż 1 i mogą być nieograniczenie duże, jeśli $|a| \approx |b|$, więc błąd względny wyniku może być dowolnie duży (to nie znaczy, że zawsze jest). Jeśli użyjemy algorytmu

$$x = (a+b)*(a-b),$$

to otrzymamy wynik

$$\begin{aligned}\tilde{x} &= (a+b)(1+\varepsilon_1)(a-b)(1+\varepsilon_2)(1+\varepsilon_3)\\ &= (a^2-b^2)(1+\varepsilon_1+\varepsilon_2+\varepsilon_3+\varepsilon_1\varepsilon_2+\varepsilon_1\varepsilon_3+\varepsilon_2\varepsilon_3+\varepsilon_1\varepsilon_2\varepsilon_3)\\ &\approx (a^2-b^2)(1+\varepsilon_1+\varepsilon_2+\varepsilon_3).\end{aligned}$$

Błąd względny wyniku ma w tym przypadku oszacowanie 3ν (ale w razie wystąpienia niedomiaru w dowolnym działaniu to oszacowanie jest niepoprawne).

Przykład. Mamy obliczyć wartość bezwzględną liczby zespolonej $z = (a, b)$. Zamiast wzoru $|z| = \sqrt{a^2+b^2}$, lepiej jest użyć algorytmu

```
if ( fabs(a) > fabs(b) ) {
  p = b/a;  x = fabs(a)*sqrt(1.0+p*p);
}
else if ( b != 0.0 ) {
  p = a/b;  x = fabs(b)*sqrt(1.0+p*p);
}
else x = 0.0;
```

W algorytmie tym unikamy zarówno nadmiaru jak i niedomiaru. Postulowana przez standard poprawność zaokrąglania zapewnia, że otrzymamy $x \geq |a|$, $x \geq |b|$

— zauważmy, że to nie wynika z ograniczenia błędu względnego (spektakularnym przykładem komputerów, których arytmetyka zmiennopozycyjna nie jest zgodna ze standardem IEEE-754 są niektóre komputery CRAY — na tych maszynach, obliczając cosinus na podstawie sinusa, ze wzoru $c = \sqrt{1-s^2}$, można otrzymać $c > 1$).

Zadania i problemy

1. Wobec jednoznaczności reprezentacji liczb stałopozycyjnych ze znakiem (w kodzie uzupełnieniowym do dwóch), aby stwierdzić, że dwie liczby są różne, wystarczy sprawdzić, że różne są ich ciągi bitów. Jak to wygląda w przypadku, gdy mamy porównać liczbę w reprezentacji ze znakiem z liczbą w reprezentacji bez znaku?
2. Jaka operacja na ciągu bitów, który reprezentuje liczbę całkowitą x w kodzie uzupełnieniowym do dwóch, prowadzi do otrzymania reprezentacji liczby $-x$ (pod warunkiem, że $-x$ ma reprezentację o tej samej długości)?
3. Jakie operacje na ciągu bitów są równoważne pomnożeniu oraz podzieleniu reprezentowanej przez ten ciąg liczby całkowitej przez 2^k? Podaj odpowiedź dla reprezentacji bez znaku i dla kodu uzupełnieniowego do 2. Kiedy wystąpi nadmiar, a kiedy wystąpi zaokrąglenie?
4. Program TEX reprezentuje wymiary (w celu umieszczenia znaków drukarskich na stronie) jako całkowite liczby jednostek długości zwanych sp (ang. *scaled point*). $1\mathrm{sp} = \left(\frac{2^{-16}}{72{,}27}\right)''$ (gdzie $1'' = 25.4\mathrm{mm}$). W celu łatwego wykrywania nadmiaru w obliczaniu wymiarów, przyjęte jest założenie, że dodatnie wymiary mogą być reprezentowanie za pomocą 30 bitów (przechowuje się je w zmiennych 32-bitowych). Oblicz w milimetrach wymiary największej strony, z jaką TEX może sobie poradzić, oraz najmniejszą odległość zdaniem TEX-a różną od zera.
5. Jaki jest błąd względny reprezentacji zmiennopozycyjnej pojedynczej precyzji liczb $\frac{1}{3}$, $\frac{1}{4}$ i $\frac{1}{5}$? Napisz ciągi bitów reprezentujące te liczby.
6. Udowodnij, że jeśli nie ma nadmiaru (ale dopuszczamy niedomiar), to obliczona średnia arytmetyczna dwóch liczb zmiennopozycyjnych leży między tymi liczbami.
7. Zadanie polega na obliczeniu wartości wyrażenia $w = a^2 + ab + b^2$, dla danych liczb a, b (jest to tzw. niepełny kwadrat sumy). Napisz wyrażenia opisujące błędy względne wyników otrzymanych w arytmetyce zmiennopozycyjnej, przy użyciu algorytmów opartych na wzorach $w = (a^2 + b^2) + ab$ oraz $w = \frac{1}{2}((a+b)^2 + (a^2 + b^2))$. Który algorytm jest lepszy?

   Zbadaj, jak to wygląda w przypadku obliczania niepełnego kwadratu różnicy, $w = a^2 - ab + b^2$.

Błędy w obliczeniach numerycznych

Jakie są rodzaje błędów i jak się je otrzymuje[8]

Niech $f_1, \ldots, f_m$ będą ustalonymi funkcjami zmiennych $x_1, \ldots, x_n$. Zadanie polega na obliczeniu ich wartości (liczb rzeczywistych) $y_1, \ldots, y_m$ dla konkretnych (danych) liczb rzeczywistych $x_1, \ldots, x_n$. Błędy w obliczeniach spowodują, że zamiast dokładnego wyniku otrzymamy na ogół inne liczby, $\tilde{y}_1, \ldots, \tilde{y}_k$ (uwaga: błędy mogą w szczególności spowodować zmianę liczby wyników). Jest pięć głównych przyczyn powstawania błędów w obliczeniach numerycznych.

Błędy modelu. Modele matematyczne zjawisk przyrodniczych, ekonomicznych i innych *zawsze* są uproszczone w stosunku do tych zjawisk (to jest ich sedno, ponieważ opis „dokładny" dowolnego zjawiska musiałby być ogólną teorią wszystkiego, a więc byłby całkowicie bezużyteczy). Na przykład często używane modele populacji dowolnych organizmów lub zachowania przepływającego gazu są układami równań różniczkowych, których rozwiązaniami są funkcje rzeczywiste o wartościach ułamkowych, choć w każdej chwili liczba osobników w populacji lub liczba cząsteczek gazu w naczyniu jest całkowita. Wyniki obliczeń przeprowadzonych przy użyciu dowolnego modelu (nawet jeśli nie ma błędów w samych obliczeniach) mogą zatem opisywać dane zjawisko tylko w przybliżeniu.

Błędy reprezentacji i danych wejściowych. Tylko skończenie wiele liczb rzeczywistych ma dokładne reprezentacje zmiennopozycyjne. Każda inna liczba rzeczywista, która jest daną dla obliczeń numerycznych, musi być zastąpiona przez jedną z liczb należących do tego skończonego zbioru. Skutkiem tego jest usiłowanie obliczenia wartości funkcji $f_1, \ldots, f_m$ w innym punkcie ich dziedziny (a czasem w punkcie nienależącym do dziedziny którejś z nich).

Dane do obliczeń bardzo często pochodzą z pomiarów. Nawet najdokładniejsze pomiary fizyczne dają najwyżej kilkanaście poprawnych cyfr dziesiętnych, a w wielu przypadkach dokładność jest znacznie mniejsza (błąd pomiaru może dochodzić do kilkudziesięciu procent). Skutki takich błędów (w postaci zmiany wyniku) są oczywiście znacznie większe niż skutki błędów reprezentacji.

Błędy aproksymacji. Definicje funkcji f_i, których wartości należy obliczyć, mogą nie być wygodnymi, ani nawet możliwymi do użycia algorytmami obliczania ich wartości. Przypomnijmy, że w obliczeniach numerycznych mamy do dyspozycji

[8]Pierwszy rozdział *Księcia* Niccolò Machiavellego ma tytuł *Quali sono i generi dei principati e come si acquistino.*

cztery działania arytmetyczne oraz pierwiastek kwadratowy i kilka elementarnych funkcji przestępnych (logarytmy, funkcje wykładnicze, trygonometryczne), których przybliżone wartości w rzeczywistości są obliczane za pomocą działań arytmetycznych (nawet jeśli istnieje odpowiedni rozkaz procesora — odpowiednie działania wykonuje tzw. mikroprogram tego procesora). Dlatego często jedynym sposobem rozwiązania zadania jest zastąpienie funkcji f_i przez inną funkcję, której wartości różnią się dostatecznie mało (to trzeba określić i sprawdzić!).

Błędy aproksymacji pojawiają się na przykład wtedy, gdy

- Zamiast granicy nieskończonego zbieżnego ciągu rzeczywistego $(a_i)_{i=1}^{\infty}$, za wynik obliczeń przyjmujemy element a_n dla jakiegoś (dostatecznie dużego?) n.
- Zamiast sumy szeregu zbieżnego $\sum_{i=1}^{\infty} a_i$, bierzemy sumę pierwszych jego n wyrazów.
- Zamiast obliczać wartość funkcji f w pewnym punkcie x, konstruujemy inną funkcję, która w pewnych punktach ma takie same wartości jak f, ale obliczenie jej wartości wymaga wykonania znacznie mniejszej liczby działań (albo: jest w ogóle wykonalne).
- Zamiast całki oznaczonej zadowalamy się kwadraturą, czyli pewną kombinacją liniową wartości funkcji podcałkowej i ewentualnie jej pochodnych w skończenie wielu punktach.
- Zamiast równania różniczkowego rozwiązujemy równanie różnicowe. W ten sposób możemy otrzymać liczby, które przybliżają wartości rozwiązania wyjściowego równania różniczkowego w pewnym skończonym zbiorze punktów.

Błędy zaokrągleń. Chodzi o błędy spowodowane przez wyznaczanie reprezentacji zmiennopozycyjnej w ustalonym formacie (np. `float`, `double`, `long double`) wyniku każdego działania arytmetycznego. W pewnych przypadkach zaokrąglenie nie wprowadza błędu (np. w mnożeniu i dzieleniu przez całkowite potęgi liczby 2, o ile nie ma nadmiaru ani niedomiaru), ale w ogólności zaokrąglenia powodują powstawanie błędów, których skutki bywają bolesne (tym bardziej bolesne im mniej spodziewane).

Istnienie błędów zaokrągleń powoduje brak łączności mnożenia i dodawania zmiennopozycyjnego. W konsekwencji, wzory równoważne z punktu widzenia algebry, nie muszą być takie w obliczeniach numerycznych (przykład był w poprzednim wykładzie). Dlatego za algorytm numeryczny (w sensie: jednoznaczny opis postępowania prowadzący do otrzymania wyniku na podstawie danych, zobacz definicję na str. 1.1) można uważać tylko takie wyrażenie lub ciąg wyrażeń, który jednoznacznie określa argumenty wszystkich kolejno wykonywanych działań (np. suma więcej niż dwóch składników powinna być zaopatrzona w odpowiednie nawiasy). Różne algorytmy numeryczne dla tych samych danych mogą dawać zupełnie różne wyniki.

Błędy grube. Błędy tego rodzaju teoretycznie nie powinny mieć miejsca, ale byłaby to teoria zbyt abstrakcyjna. Błędy grube to skutki błędów ludzkich (czyli na przykład pomyłek, przeoczeń, błędów w algorytmie lub programie), oraz wszelkich awarii i czasem (niestety) działań celowych.

Wbrew temu, co sugeruje nazwa, błędy grube nie muszą być bardzo duże. Na przykład, błędem grubym jest napisanie w programie liczby `3.14` zamiast znanego skądinąd dokładniej ilorazu długości i średnicy okręgu. Program z błędami grubymi może przypadkowo dawać dobre wyniki (oczywiście tylko do czasu, aż nabierzemy do tego programu pełnego zaufania).

Uwarunkowanie zadań

Ze względu na postać reprezentacji zmiennopozycyjnej, bardziej od błędów bezwzględnych użyteczne bywają oszacowania błędów względnych przetwarzanych liczb rzeczywistych. Podstawowa miara podatności wyniku na zaburzenie danych, zwana wskaźnikiem uwarunkowania zadania (ang. *condition number*), jest zdefiniowana jako największy możliwy iloraz błędu względnego wyniku do względnego zaburzenia danych, które powstanie wyniku z takim błędem spowodowało, przy założeniu odpowiedniego ograniczenia wielkości zaburzenia danej.

Przypuśćmy, że wynik w jest jedną liczbą rzeczywistą, która zależy od jednej liczby danej d. Zależność wyniku od danej opisuje pewna funkcja f, o ktorej założymy, że jest ciągła. Mamy zatem $w = f(d)$ oraz $\tilde{w} = f(\tilde{d})$, gdzie dana $\tilde{d}$ powstała przez zaburzenie d o nie więcej niż pewne ε. Wskaźnik uwarunkowania tego zadania jest określony wzorem

$$\operatorname{cond}_w d = \sup_{0<|\tilde{d}-d|\leq\varepsilon} \left(\frac{|\tilde{w}-w|}{|w|} \Big/ \frac{|\tilde{d}-d|}{|d|} \right).$$

Ograniczenie zaburzenia ustala się, co tu ukrywać, arbitralnie; może być np. $\varepsilon = |d|\nu$ (gdzie ν jest błędem względnym reprezentacji), można przyjąć ε na podstawie przewidywanych błędów pomiarów, z których pochodzą dane, ale najczęściej (bo najłatwiej) przyjmuje się granicę powyższego wyrażenia

przy $\varepsilon \to 0$. Jeśli funkcja f ma pochodną w punkcie d, to prowadzi to do wzoru

$$\operatorname{cond}_w d = \left| \frac{d}{w} f'(d) \right|. \qquad (*)$$

Jeśli dane i wynik składają się z wielu liczb, to oczywiście można wprowadzić wskaźniki uwarunkowania każdego wyniku ze względu na każdą daną, ale zwykle to jest niewygodne. Zamiast tego, dane i wyniki traktuje się jak wektory w $\mathbb{R}^n$ i $\mathbb{R}^m$ i ich zaburzenia mierzy się za pomocą odpowiednich norm. Dla $\mathbf{d}, \tilde{\mathbf{d}} \in \mathbb{R}^n$, $\boldsymbol{w}, \tilde{\boldsymbol{w}} \in \mathbb{R}^m$, $\boldsymbol{w} = f(\mathbf{d})$, $\tilde{\boldsymbol{w}} = f(\tilde{\mathbf{d}})$, mamy zatem

$$\operatorname{cond}_w \mathbf{d} = \sup_{0<\|\tilde{\mathbf{d}}-\mathbf{d}\|\leq\varepsilon} \left(\frac{\|\tilde{\boldsymbol{w}}-\boldsymbol{w}\|}{\|\boldsymbol{w}\|} \Big/ \frac{\|\tilde{\mathbf{d}}-\mathbf{d}\|}{\|\mathbf{d}\|} \right).$$

Warto zwrócić uwagę, że zaburzenia względne indywidualnych danych i wyników w tym przypadku mogą być dowolnie duże (nieskończone dla 0).

Znajomość wskaźnika uwarunkowania zadania i dokładności danych daje możliwość dowiedzenia się, na jaką dokładność wyników można liczyć. Jeśli znamy dane z dokładnością 0.1%, a wskaźnik uwarunkowania jest równy 10, to mamy w zasadzie szansę dostać wynik z dokładnością 1%, ale należy pamiętać, że algorytm numeryczny wprowadzi jeszcze błędy zaokrągleń. Jeśli jednak wskaźnik uwarunkowania zadania jest większy niż 1000, to znając dane z dokładnością 0.1%, należy to zadanie omijać z daleka. Błąd wyniku może wtedy przekroczyć 100%, a to oznacza, że nie możemy być nawet pewni, jaki znak ma wynik. W takiej sytuacji trzeba postarać się o dokładniejsze dane, albo przeformułować zadanie.

O zadaniach z małym wskaźnikiem uwarunkowania (bliskim 0, rzędu 1, lub rzędu n, gdzie n oznacza np. rozmiar zadania) mówi się, że są dobrze uwarunkowane. Zadania źle uwarunkowane mają wskaźniki uwarunkowania duże. Jeśli wskaźnik uwarunkowania zadania jest większy niż ν^{-1}, to nie ma sensu rozwiązywanie go przy użyciu arytmetyki zmiennopozycyjnej, która dopuszcza błąd względny ν (samo reprezentowanie danych z takim błędem wyklucza sukces).

Na podstawie wzoru (*), łatwo jest obliczyć wskaźniki uwarunkowania zadań obliczania sumy, różnicy, iloczynu i ilorazu:

$$\operatorname{cond}_{a+b} a = \left|\frac{a}{a+b}\right|, \quad \operatorname{cond}_{a-b} b = \left|\frac{b}{a-b}\right|,$$
$$\operatorname{cond}_{ab} a = 1, \qquad \operatorname{cond}_{a/b} b = 1.$$

Jak widać z powyższych wzorów, niezależnie od wartości argumentów mnożenie i dzielenie są bardzo dobrze uwarunkowane. Dodawanie i odejmowanie są bardzo dobrze uwarunkowane wtedy, gdy wartość bezwzględna wyniku nie jest znacznie mniejsza niż wartości bezwzględne danych. Zatem, na przykład odejmując dwie liczby bardzo bliskie siebie, możemy otrzymać wynik o bardzo dużym błędzie względnym. To zjawisko nazywa się znoszeniem się składników. Powinien o nim pamiętać każdy, kto chciałby obliczyć pochodną funkcji na podstawie wzoru

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

i jest przekonany (na pozór słusznie), że im mniejsze $|h|$, tym mniejszy błąd. Mniejszy jest tylko błąd aproksymacji, za to błąd zaokrągleń dla $h \to 0$ rośnie do 100% (gdy $\mathrm{fl}(x+h) = x$, otrzymamy licznik ilorazu różnicowego równy 0, czyli pochodna dowolnej funkcji obliczona w ten sposób jest równa 0). Czyli nie tędy droga.

Trzeba podkreślić, że uwarunkowanie zadania jest własnością zadania i nie ma nic wspólnego z algorytmami jego rozwiązywania. Mając zadanie dobrze uwarunkowane, możemy otrzymać wynik z małym błędem, jeśli użyjemy dobrego algorytmu. Gorszy algorytm może wytworzyć błędy znacznie większe niż to wynika z oszacowań uwarunkowania. Mając zadanie źle uwarunkowane trzeba z uwagą wybierać algorytmy, ponieważ na złe uwarunkowanie algorytm nie może pomóc. Może tylko wyrządzić jak najmniej dodatkowych szkód.

Przykład. Rozważmy zadanie polegające na znalezieniu punktu przecięcia dwóch prostych, z których każda jest określona za pomocą dwóch punktów. Dane są współrzędne kartezjańskie tych punktów, mamy obliczyć współrzędne kartezjańskie punktu przecięcia. Prosta ℓ_1, przechodząca przez punkty $[0,0]^T$ i $[3.141, 2.718]^T$, z prostą ℓ_2, przechodzącą przez $[-1,1]^T$ i $[-0.961, 1.033]^T$, przecina się w punkcie $[94.23, 81.54]^T$ (jego współrzędne są podane z błędem mniejszym niż 10^{-3}). Zaburzenie ostatniej współrzędnej ostatniego punktu, zmieniające ją na 1.034 (mniejsze niż 0.1%), powoduje zmianę wyniku na $[-288.972, -250.056]^T$, czyli większą niż 400%.

Numeryczna poprawność i stabilność algorytmów

O algorytmie mówi się, że jest poprawny, jeśli dla dowolnych poprawnych danych algorytm produkuje poprawny wynik (a dla niepoprawnych czym prędzej kończy działanie, sygnalizując błąd). Algorytmy numeryczne, wskutek błędów zaokrągleń, dają prawie zawsze błędne odpowiedzi, tak więc można od nich oczekiwać znacznie mniej. Zamiast ostrego rozróżnienia między algorytmami poprawnymi i błędnymi, mamy możność porównywania algorytmów numerycznych pod względem dokładności dostarczanych przez nie wyników.

Podstawowy pomysł w badaniu jakości algorytmów numerycznych polega na interpretowaniu otrzymanego wyniku (który jest obciążony błędem) jako wyniku dokładnego (ewentualnie z dokładnością do błędów reprezentacji), który odpowiada zmienionym danym. Jeśli dane, którym odpowiada otrzymany wynik, różnią się od danych, na których działał algorytm, na poziomie błędu reprezentacji, to taki algorytm uznaje się za numerycznie poprawny i jest to w praktyce najlepsza własność, jakiej po algorytmie numerycznym można się spodziewać.

Na przykład, wyniki dodawania i mnożenia zmiennopozycyjnego, które pod warunkiem niewystąpienia nadmiaru i niedomiaru są równe

$$\mathrm{fl}(a+b) = (a+b)(1+\varepsilon_1), \quad \mathrm{fl}(cd) = cd(1+\varepsilon_2)$$

dla pewnych liczb $|\varepsilon_1|, |\varepsilon_2| \le \nu$, możemy zinterpretować jako dokładną sumę i iloczyn liczb $\tilde{a} = a(1+\varepsilon_1)$, $\tilde{b} = b(1+\varepsilon_1)$, $\tilde{c} = c(1+\varepsilon_2)$, $\tilde{d} = d$.

Dla zadania obliczania wartości funkcji $f\colon D \subset \mathbb{R}^n \to \mathbb{R}^m$ za pomocą algorytmu A (tu patrzymy na algorytm jak na funkcję, która przyporządkowuje danym obliczone numerycznie wyniki), numeryczna poprawność jest zdefiniowana tak: Istnieją stałe K_d, K_w, niezbyt duże i niezależne od ν (ale stałe te mogą zależeć od liczb n i m, opisujących rozmiar zadania), takie że dla dowolnych danych $\mathbf{d} \in D$ istnieją dane $\tilde{\mathbf{d}}$, takie że

$$\frac{\|\tilde{\mathbf{d}} - \mathbf{d}\|}{\|\mathbf{d}\|} \le K_d \nu, \quad \frac{\|A(\mathbf{d}) - f(\tilde{\mathbf{d}})\|}{\|f(\tilde{\mathbf{d}})\|} \le K_w \nu.$$

Udowodnienie, że jakiś algorytm jest numerycznie poprawny i obliczenie stałych kumulacji K_d i K_w bywa trudne i nie zawsze jest możliwe — nie każdy algorytm jest numerycznie poprawny i jest wiele zadań, dla których algorytmy numerycznie poprawne nie są znane. Własnością słabszą (ale też nie każdy algorytm ją ma i też nie zawsze jest łatwo ją wykazać) jest numeryczna stabilność. Algorytm A jest numerycznie stabilny, jeśli istnieje stała K, niezależna od ν i od danych, taka że

$$\frac{\|A(\mathbf{d}) - f(\mathbf{d})\|}{\|f(\mathbf{d})\|} \le K \cdot (1 + \mathrm{cond}\,\mathbf{d})\nu.$$

Dla algorytmu numerycznie stabilnego, błąd otrzymanego wyniku niekoniecznie może być zinterpretowany jako skutek małego zaburzenia danych. Natomiast tzw. optymalny poziom błędu, $(1 + \mathrm{cond}\,\mathbf{d})\nu$, który wynika z uwarunkowania zadania i konieczności reprezentowania danych i wyniku w stosowanej arytmetyce, jest w takim algorytmie powiększony o niewielki (?) czynnik stały K. Mając algorytm numerycznie stabilny, możemy zmniejszyć błąd (wniesiony przez algorytm) odpowiednio zmniejszając parametr ν, czyli np. zmieniając reprezentację liczb z float na double.

Przykład. Zbadamy algorytm rozwiązywania układu dwóch równań liniowych przy użyciu wzorów Cramera. Mamy układ

$$\begin{cases} ax + by &= e \\ cx + dy &= f \end{cases}$$

Algorytm polega na obliczeniu wyznaczników $d_0 = ad - bc$, $d_1 = ed - bf$, $d_2 = af - ec$, a następnie $x = d_1/d_0$ i $y = d_2/d_0$. Wykonując obliczenia w arytmetyce zmiennopozycyjnej, otrzymamy

$$\begin{aligned}
\tilde{d}_0 &= \big(ad(1+\varepsilon_1) - bc(1+\varepsilon_2)\big)(1+\varepsilon_3),\\
\tilde{d}_1 &= \big(ed(1+\varepsilon_4) - bf(1+\varepsilon_5)\big)(1+\varepsilon_6),\\
\tilde{d}_2 &= \big(af(1+\varepsilon_7) - ec(1+\varepsilon_8)\big)(1+\varepsilon_9),\\
\tilde{x} &= (\tilde{d}_1/\tilde{d}_0)(1+\varepsilon_{10}),\\
\tilde{y} &= (\tilde{d}_2/\tilde{d}_0)(1+\varepsilon_{11}).
\end{aligned}$$

O liczbach $\varepsilon_1, \dots, \varepsilon_{11}$ wiemy tylko tyle, że ich wartości bezwzględne nie są większe niż ν. Aby wykazać numeryczną poprawność, należy błędy zaokrągleń „doczepić” do poszczególnych danych i wyników. Możemy na przykład określić liczby

$$\begin{aligned}
\tilde{a} &= a(1+\varepsilon_1)(1+\varepsilon_3), & \tilde{b} &= b,\\
\tilde{c} &= c(1+\varepsilon_2)(1+\varepsilon_3), & \tilde{d} &= d,\\
\tilde{e} &= e(1+\varepsilon_4)(1+\varepsilon_6)(1+\varepsilon_{10}), & \tilde{f} &= f(1+\varepsilon_5)(1+\varepsilon_6)(1+\varepsilon_{10}).
\end{aligned}$$

Wtedy liczba $\tilde{x}$ jest dokładną wartością pierwszej niewiadomej w rozwiązaniu układu równań o współczynnikach $\tilde{a}, \dots, \tilde{f}$, które przybliżają dane oryginalne $a, \dots, f$ z błędem względnym nie przekraczającym błędu reprezentacji. Mamy stałą kumulacji danych równą 3 i stałą kumulacji wyniku równą 0 — to jest bardzo dobra wiadomość. Niestety, otrzymana liczba $\tilde{y}$ na ogół *nie jest* poprawną wartością drugiej niewiadomej w rozwiązaniu układu określonego przez liczby $\tilde{a}, \dots, \tilde{f}$. Liczba $\tilde{y}$ jest poprawną wartością drugiej niewiadomej w rozwiązaniu układu o innych współczynnikach (też niewiele różniących się od $a, \dots, f$).

W ogólności nie istnieją takie dane $\tilde{a}, \dots, \tilde{f}$, różniące się na poziomie błędu reprezentacji od $a, \dots, f$, dla których liczby $\tilde{x}$ i $\tilde{y}$ są rozwiązaniem (nawet zaburzonym na poziomie błędu reprezentacji). Algorytm oparty na wzorach

Cramera nie jest więc numerycznie poprawny, ale można wykazać jego numeryczną stabilność. Dla układów z więcej niż dwiema niewiadomymi nie mamy nawet tego.

Podstawowe algorytmy numeryczne

Sumowanie

Najprostszy algorytm obliczania sumy n liczb, $s = \sum_{i=0}^{n-1} a_i$, ma postać

```
s = a[0];
for ( i = 1; i < n; i++ )
  s += a[i];
```

Obliczona suma jest końcową wartością zmiennej s, zaś jej obliczanie jest realizacją wzoru

$$s = (\dots((a_0 + a_1) + a_2) + \cdots + a_{n-2}) + a_{n-1}.$$

Użycie arytmetyki zmiennopozycyjnej powoduje, że zamiast dokładnej sumy s otrzymamy liczbę

$$\begin{aligned}
\tilde{s} &= \big((\dots((a_0 + a_1)(1+\varepsilon_1) + a_2)(1+\varepsilon_2) + \cdots + a_{n-2})(1+\varepsilon_{n-2}) + a_{n-1}\big)\cdot\\
&\quad (1+\varepsilon_{n-1})\\
&= a_0(1+\varepsilon_1)\dots(1+\varepsilon_{n-1}) + a_1(1+\varepsilon_1)\dots(1+\varepsilon_{n-1}) + \cdots + a_{n-1}(1+\varepsilon_{n-1})\\
&= \tilde{a}_0 + \tilde{a}_1 + \cdots + \tilde{a}_{n-1}.
\end{aligned}$$

Otrzymana liczba jest więc dokładną sumą zmienionych danych: $\tilde{a}_i = a_i(1+\gamma_i)$. Algorytm jest numerycznie poprawny, jeśli zaburzenia względne danych, γ_i, dają się oszacować za pomocą parametru ν i stałej kumulacji. Jeśli nie było nadmiaru ani niedomiaru, to dla każdego i jest $|\varepsilon_i| < \nu$. Możemy przyjąć, że

$$1 + \gamma_i = (1+\varepsilon_i)\dots(1+\varepsilon_{n-1}) \approx 1 + \varepsilon_i + \cdots \varepsilon_{n-1},$$

Mamy stąd oszacowanie (przybliżone — nie uwzględniające iloczynów dwóch i więcej epsilonów, prawie zawsze się tak robi) $|\gamma_i| \leq (n-i)\nu$, czyli stała kumulacji dla każdego składnika jest nie większa niż $n-1$.

Zauważmy, że składniki dodane na początku mogą mieć znacznie większe zaburzenie, zaś ostatni składnik ma zaburzenie na poziomie co najwyżej ν. Drzewo binarne opisujące ten sposób obliczania sumy (ze składnikami w liściach) ma postać listy liniowej; zaburzenie składnika ma oszacowanie proporcjonalne do odległości jego liścia od korzenia. Dla dużych n możemy budować znacznie niższe drzewa z n liśćmi. Drzewo binarne zrównoważone (w którym wszystkie liście są na jednym lub dwóch poziomach) ma wysokość $\lceil \log_2 n \rceil + 1$. Algorytm sumowania, któremu odpowiada takie drzewo, nazywa się algorytmem sumowania parami. Dodajemy w nim pary kolejnych składników, następnie sumy par itd., dochodząc do sumy wszystkich składników. Stała kumulacji w tym algorytmie jest równa $\lceil \log_2 n \rceil$. Dla $n = 1000$ różnica (w porównaniu ze zwykłym sumowaniem „po kolei") jest znaczna.

Inne podejście jest stosowane w algorytmie Kahana, w którym para zmiennych, s i c, jest używana do zasymulowania zmiennej o większej precyzji; dodając kolejny składnik, algorytm oblicza odpowiednią poprawkę:

```
s = a[0];
c = 0.0;
for ( i = 1; i < n; i++ ) {
  y = a[i]-c;
  t = s+y;
  c = (t-s)-y;
  s = t;
}
```

Schemat Hornera

Obliczenie wartości wielomianu $w(x) = \sum_{k=0}^{n} a_k x^k$ na podstawie danych współczynników $a_0, \ldots, a_n$ i liczby x, może być dokonane na podstawie wzoru

$$w(x) = (\ldots(a_n x + a_{n-1})x + \cdots + a_1)x + a_0.$$

Oparty na tym wzorze algorytm

```
w = a[n];
for ( k = n-1; k >= 0; k-- )
  w = w*x + a[k];
```

wykonuje minimalną liczbę działań arytmetycznych (n mnożeń i dodawań) i jest numerycznie poprawny, co wykażemy. Zamiast wartości wielomianu w,

w arytmetyce zmiennopozycyjnej otrzymamy liczbę

$$\begin{aligned}\tilde{w} &= ((\ldots(a_n x(1+\varepsilon_n) + a_{n-1})(1+\delta_{n-1})x(1+\varepsilon_{n-1}) + \\ &\quad \cdots + a_1)(1+\delta_1)x(1+\varepsilon_1) + a_0)(1+\delta_0) \\ &= (\ldots(\tilde{a}_n x + \tilde{a}_{n-1})x + \cdots + \tilde{a}_1)x + \tilde{a}_0.\end{aligned}$$

Mamy przy tym

$$\tilde{a}_k = a_k(1+\delta_k)(1+\varepsilon_k)(1+\delta_{k-1})\ldots(1+\delta_0) = a_k(1+\gamma_k)$$

(przyjmujemy $\delta_n = \varepsilon_0 = 0$). Liczbę $|\gamma_k|$ możemy oszacować z góry przez $(2k+1)\nu$, a zatem otrzymana liczba jest wartością wielomianu o współczynnikach zmienionych na poziomie błędu reprezentacji (ale x jest niezmienione).

Powyższy algorytm, zwany schematem Hornera, ma wiele odmian i zastosowań. Po pierwsze, można go rozszerzyć w celu jednoczesnego obliczenia wartości pochodnej wielomianu. Zmienna w przyjmuje kolejno wartości a_n, $a_n x + a_{n-1}$, $a_n x^2 + a_{n-1}x + a_{n-2}$ itd. Przedostatnia przypisana jej wartość to $\sum_{k=1}^{n} a_k x^{k-1}$. Gdybyśmy pierwszą z tych liczb pomnożyli przez x^{n-1}, drugą przez x^{n-2} itd., ostatnią przez x^0 i zsumowali, to otrzymamy $\sum_{k=1}^{n} k a_k x^{k-1}$, czyli wartość $w'(x)$. To właśnie robi algorytm

```
p = 0.0;
w = a[n];
for ( k = n-1; k >= 0; k-- ) {
  p = p*x + w;
  w = w*x + a[k];
}
```

Po drugie, schemat Hornera można dostosować do obliczania wartości wielomianów danych w różnych bazach innych niż potęgowa. Baza potęgowa jest określona prostym wzorem, który ma wiele zalet w algebrze, ale w obliczeniach z błędami zaokrągleń zwykle zachowuje się fatalnie (już dla n rzędu kilkanaście) i dlatego lepiej jej unikać.

Wreszcie, zmienna x nie musi być liczbowa; może to być np. macierz kwadratowa (ale w tym przypadku numeryczna poprawność może nie mieć miejsca). Obliczanie (macierzowej) wartości wielomianu lub funkcji wymiernej, której argumentem jest macierz, występuje w wielu zastosowaniach.

Obliczanie wartości funkcji wykładniczej

Aby dla danego x obliczyć e^x, można posłużyć się standardową funkcją exp, ale przypuśćmy, że nie możemy jej użyć. Szereg nieskończony

$$e^x = \frac{1}{0!} + \frac{1}{1!}x + \frac{1}{2!}x^2 + \cdots$$

możemy obciąć do pierwszych kilku lub kilkunastu wyrazów. Jeśli $x \in [0, 1]$, to pierwszych 10 wyrazów wystarczy do otrzymania wyniku z błędem mniejszym niż $3 \cdot 10^{-7}$ (bez uwzględnienia błędów zaokrągleń). Obliczenie można wykonać przy użyciu schematu Hornera:

$$e^x \approx 1 + x\Big(1 + \frac{x}{2}\Big(1 + \frac{x}{3}\Big(\dots\Big(1 + \frac{x}{k}\Big)\dots\Big)\Big)\Big)$$

Zauważmy, że we wzorze tym dla nieujemnego x nie występuje znoszenie się składników — oba argumenty każdego dodawania mają ten sam znak. Poza tym błąd dowolnego kroku przenosi się na wynik z czynnikiem mniejszym niż 1. Dlatego wynik tego obliczenia przy użyciu arytmetyki zmiennopozycyjnej jest bardzo dokładny[9].

Dla $x \gg 1$ trzeba by uwzględnić dużo wyrazów szeregu, aby otrzymać mały błąd aproksymacji, a ponadto błędy zaokrągleń w ostatnich krokach są „wzmacniane". Natomiast dla $x < 0$ mamy znoszenie się składników. Dlatego zamiast podstawiać $x \notin [0, 1]$ do schematu Hornera, lepiej jest obliczyć e^y, gdzie $y = \frac{x}{n}$, dla $n \in \mathbb{Z}$ dobranego tak, aby było $y \in [0, 1]$ (wygodnie jest przyjąć za n pewną potęgę dwójki). Następnie możemy obliczyć $e^x = \big(e^y\big)^n$.

W różnych zastosowaniach trzeba obliczyć macierz

$$e^A = \frac{1}{0!}I + \frac{1}{1!}A + \frac{1}{2!}A^2 + \cdots$$

dla ustalonej macierzy kwadratowej A. Właśnie w tym przypadku funkcja exp nie rozwiązuje problemu, ale możmy użyć schematu Hornera. Niestety, w przypadku macierzy nie możemy uniknąć znoszenia się składników. Możemy jednak wybrać liczbę całkowitą n tak, aby macierz $B = \frac{1}{n}A$ miała sumę wartości bezwzględnych współczynników w każdym wierszu mniejszą od 1. Następnie obliczamy macierz e^B za pomocą schematu Hornera, a następnie $e^A = \big(e^B\big)^n$. Dokładny rachunek błędów zaokrągleń tymczasem pominę.

[9] Różnica pojęć „dokładny" i „bardzo dokładny" jest mniej więcej taka, jak pojęć „fotel" i „fotel dentystyczny".

Błędy w obliczeniach rekurencyjnych

Rozważmy przykład numerycznego obliczania elementów ciągu

$$a_k = \int_0^1 \frac{x^k}{x+7}\,dx.$$

Dla $k \to \infty$ ciąg ten monotonicznie dąży do zera (wszystkie jego wyrazy są dodatnie). Obliczając dla $k > 0$

$$a_k + 7a_{k-1} = \int_0^1 \frac{x^k + 7x^{k-1}}{x+7}\,dx = \int_0^1 x^{k-1}\,dx = \frac{1}{k}$$

otrzymujemy równanie różnicowe

$$a_k = -7a_{k-1} + \frac{1}{k},$$

które możemy rozwiązać, aby znaleźć rozpatrywany ciąg. Warunek początkowy ma postać $a_0 = \ln 8 - \ln 7$.

Program, który przeprowadza to obliczenie przy użyciu arytmetyki pojedynczej precyzji, produkuje liczby

$$\begin{aligned}
&a_0 = 0.133531, \quad a_1 = 0.065280, \quad a_2 = 0.043038, \quad a_3 = 0.032066,\\
&a_4 = 0.025538, \quad a_5 = 0.021234, \quad a_6 = 0.018027, \quad a_7 = 0.016671,\\
&a_8 = 0.008300, \quad a_9 = 0.053008, \quad a_{10} = -0.271055.
\end{aligned}$$

Jak widać, obliczone a_8 jest mniejsze niż a_9, zaś obliczona liczba a_{10} jest ujemna. Oczywiście, jest to skutek błędów zaokrągleń, które w kolejnych iteracjach są „wzmacniane", tj. w tym przypadku mnożone przez 7. Użycie podwójnej precyzji „odsuwa" wykrycie tych błędów do wyrazów a_{17} i a_{18}, ale np. element a_5 obliczony w pojedynczej precyzji ma już tylko 3 poprawne cyfry dziesiętne.

Można powyższy algorytm ulepszyć, przez odwrócenie kolejności obliczania wyrazów ciągu. Jeśli przyjmiemy $a_{12} = 0$ (uwaga: liczba a_{12} jest przyjęta z błędem 100%), a następnie dla $k = 11, \dots, 0$ obliczymy

$$a_{k-1} = \frac{1}{7k} - \frac{a_k}{7},$$

to w arytmetyce pojedynczej precyzji już po trzech krokach mamy a_9 z poprawnymi trzema cyframi dziesiętnymi. Błąd, wprowadzony w dowolnym kroku wskutek zaokrąglenia, w każdym następnym kroku jest dzielony przez 7.

Trudniejsza sytuacja występuje w numerycznym rozwiązywaniu równań wyższego rzędu. Rozważmy równanie

$$a_k = \frac{4}{3}a_{k-1} - \frac{32}{81}a_{k-2} + 1.$$

Dla dowolnych warunków początkowych to równanie można rozwiązywać bezpiecznie. Błąd dowolnego elementu ciągu, który jest rozwiązaniem, możemy zinterpretować jako zmianę (zaburzenie) warunku początkowego dla wyrazów następnych. Ponieważ oba rozwiązania równania charakterystycznego, $\lambda_1 = \frac{4}{9}$ i $\lambda_2 = \frac{8}{9}$, mają wartości bezwzględne mniejsze niż 1, więc skutek takiego zaburzenia dla każdego kolejnego wyrazu jest mniejszy co najmniej o czynnik $\frac{8}{9}$.

Gdyby wartości bezwzględne obu rozwiązań równania charakterystycznego były większe niż 1, to można byłoby „odwrócić" kolejność obliczeń (ale trzeba by dobrać odpowiedni warunek początkowy). Natomiast jeśli jedno rozwiązanie ma wartość bezwzględną większą, a drugie mniejszą niż 1, to obie kolejności obliczeń wiążą się ze wzrostem błędu.

Zadania i problemy

1. Skonstruuj takie dane $\tilde{a}, \ldots, \tilde{f}$, aby liczba $\tilde{y}$, obliczona na podstawie wzoru Cramera (tak jak w wykładzie), była dokładną wartością drugiej niewiadomej w rozwiązaniu układu równań liniowych z tymi danymi.
2. Pokaż, w jaki sposób można użyć stosu i kolejki do zaimplementowania algorytmu sumowania parami.
3. Napisz procedurę realizującą schemat Hornera dla wielomianu danego w bazie Newtona, określonej wzorem

$$\begin{aligned} p_0(x) &= 1, \\ p_1(x) &= (x - b_0), \\ &\vdots \\ p_n(x) &= (x - b_0)\ldots(x - b_{n-1}), \end{aligned}$$

   dla pewnych ustalonych liczb $b_0, \ldots, b_n$.
4. Napisz procedurę realizującą uogólnienie schematu Hornera dla wielomianu danego w bazie Czebyszewa, określonej wzorem rekurencyjnym

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_k(x) &= 2xT_{k-1}(x) - T_{k-2}(x), \quad \text{dla } k > 1 \end{aligned}$$

   (ten algorytm nazywa się algorytmem Clenshawa).
5. Napisz procedurę obliczającą przybliżenie macierzy $\sin A$ dla danej kwadratowej macierzy A. Skorzystaj z „gotowych" procedur mnożenia i dodawania macierzy $n \times n$ oraz z rozwinięcia funkcji sinus w szereg:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Zadania do zaprogramowania (II semestr)

1. Napisz procedury wstawiania i usuwania wierzchołków drzew binarnych wyszukiwań, „zwykłego” i AVL. Użyj tych procedur w programie, który czyta z pliku liczby rzeczywiste i wstawia je do zwykłego drzewa BST i do drzewa AVL. Następnie program ma wyprowadzić ciąg posortowany, otrzymany przez przeglądanie drzewa.

   Operacje porównywania liczb (kluczy) do obu drzew należy policzyć, a następnie wyprowadzić te dwie liczby. Ponadto, należy wyprowadzić wysokości obu drzew.
2. Napisz program, który tworzy kod Huffmana dla ustalonego pliku tekstowego (program musi w tym celu policzyć wystąpienia poszczególnych znaków), a następnie koduje ten plik. Napisz program, który dekoduje zakodowany plik.

   Przetestuj działanie kodera i dekodera na tekstach źródłowych tych programów.
3. Napisz program, który znajduje minimalne drzewo rozpinające graf reprezentowany za pomocą list sąsiedztwa, przy użyciu algorytmu Prima.
4. Napisz program, który znajduje minimalne drzewo rozpinające graf reprezentowany za pomocą macierzy sąsiedztwa, przy użyciu algorytmu Prima.
5. Napisz program, który znajduje minimalne drzewo rozpinające graf reprezentowany za pomocą list sąsiedztwa, przy użyciu algorytmu Kruskala.
6. Napisz program, który dokonuje sortowania topologicznego grafu skierowanego za pomocą algorytmu DFS przeglądania grafu. Graf należy reprezentować w postaci list sąsiedztwa.
7. Napisz program, który dokonuje sortowania topologicznego grafu skierowanego, przy użyciu kolejki. Graf należy reprezentować w postaci list sąsiedztwa.
8. Napisz program, który znajduje składowe silnie spójne grafu skierowanego. Graf należy reprezentować w postaci list sąsiedztwa.
9. Napisz program, który znajduje najkrótsze ścieżki w grafie skierowanym przy użyciu algorytmu Dijkstry. Po utworzeniu reprezentacji grafu program powinien umożliwiać użytkownikowi podanie numeru wierzchołka, z którego najkrótsze ścieżki mają być znalezione i ma je wypisywać. Graf należy reprezentować w postaci list sąsiedztwa.
10. Napisz program, który znajduje najkrótsze ścieżki w grafie skierowanym przy użyciu algorytmu Floyda. Po znalezieniu macierzy będących wynikiem działania tego algorytmu, program ma umożliwiać wprowadzanie par numerów wierzchołków i wyprowadzać ciąg wierzchołków odpowiedniej ścieżki.
11. Napisz program, który wyszukuje nakrótszą drogę w labiryncie między dwoma wskazanymi miejscami. Labirynt jest prostokątną macierzą miejsc. Z każdego miejsca można przejść do (co najwyżej czterech) sąsiadów, pod warunkiem, że nie ma tam ściany.

    Opis labiryntu ma być czytany z pliku tekstowego, który, oglądany w edytorze tekstowym ma uwdoczniać labirynt (można w nim np. używać znaków „`|`” i „`-`” oraz spacji.)
12. Napisz program, który znajduje najkrótsze drogi w labiryncie trójwymiarowym; w tym przypadku każde miejsce ma co najwyżej sześciu sąsiadów, do których można przechodzić.
13. Napisz program, będący kalkulatorem zespolonym; powinien on czytać napisy takie jak `[1,2]*([0,1.5]+3.14/[-1,1])`, w których np. `[1,2]` oznacza liczbę zespoloną $1 + 2i$ i w których mogą występować cztery działania arytmetyczne, i który powinien obliczać i wypisywać wartości wyrażeń reprezentowanych przez takie napisy.
14. Napisz program do różniczkowania symbolicznego. Program czyta napis reprezentujący wyrażenie algebraiczne określające pewną funkcję i produkuje napis reprezentujący wyrażenie opisujące pochodną tej funkcji ze względu na wskazaną zmienną.

Wskaźnikowe struktury danych

Przypomnijmy, że zmienna wskaźnikowa przechowuje adres (identyfikator miejsca w pamięci operacyjnej), pod którym znajduje się pewien obiekt ustalonego typu — najczęściej jest nim *zmienna wskazywana* (ale może też być procedura). Zmienne wskaźnikowe wykorzystywaliśmy dotąd w celu wyprowadzenia wyników działania procedury poza tę procedurę; w tym celu używaliśmy parametrów typu wskaźnikowego, które określały miejsca w pamięci (tj. zmienne zadeklarowane poza procedurą), do których procedura miała przypisać odpowiednie wartości.

Drugie dotychczas przedstawione zastosowanie wskaźników wiąże się z tablicami — nazwa tablicy może być użyta jako wskaźnik pierwszego jej elementu (o indeksie `0`). W szczególności parametr procedury opisany jako tablica jest wskaźnikiem, zaś dowolny parametr wskaźnikowy, a także zmienna wskaźnikowa, może być traktowany (w tym indeksowany) jak tablica. Uwaga: „Zwykła” zmienna tablicowa (nie będąca parametrem procedury) *nie jest* zmienną wskaźnikową, nie można zatem przypisać jej wartości.

Zajmiemy się teraz trzecim zastosowaniem wskaźników. Mianowicie będziemy pewne obiekty reprezentować za pomocą struktur (tj. zmiennych typu `struct {...}`) zawierających między innymi pola wskaźnikowe. Wartości tych pól wskazują inne obiekty tego samego lub innego typu, reprezentując w ten sposób powiązania między obiektami. Powiązania te mogą być dowolnie skomplikowane.

Zwróćmy uwagę, że można ten sam efekt osiągnąć, przechowując wszystkie obiekty w tablicy i używając indeksów do tablicy zamiast wskaźników. Sposób ze wskaźnikami jest jednak bardziej elastyczny; nie musimy mieć jednej tablicy z obiektami, mając adresy (tj. wartości pól wskaźnikowych) nie tracimy czasu na ich obliczanie za pomocą indeksów tablicy, można budować wskaźnikowe struktury zbudowane z obiektów różnych rodzajów, wreszcie możemy używać tzw. zmiennych dynamicznych, których liczba jest ustalana podczas wykonywania programu i nie musi być znana podczas kompilacji.

Zmienne dynamiczne

Zmienne, które występowały we wszystkich dotychczas rozpatrywanych przykładach są statyczne[10], tj. miejsce w pamięci każdej takiej zmiennej jest

[10] Nie należy mylić pojęcia zmiennej statycznej ze zmienną opatrzoną atrybutem `static` w deklaracji; zmienna `static` jest to zmienna utworzona jednorazowo w statycznym obszarze danych programu, zamiast w rekordzie aktywacji procedury. Zmienne `static` nie „znikają” na końcu działania procedury i przy następnym jej wywołaniu mają zachowaną wartość. Zmiennych `static` można używać, jeśli się ma naprawdę dobry powód.

przydzielane przez kompilator. W dotychczasowych przykładach każda zmienna, do której odwołuje się pewna instrukcja, jest zmienną globalną — zadeklarowaną poza treścią wszystkich procedur, zmieną lokalną, tj. zadeklarowaną w procedurze z tą instrukcją (zmiennymi lokalnymi są w szczególności parametry procedury), lub zmienną (globalną albo lokalną zadeklarowaną w dowolnej procedurze) wskazywaną przez parametr wskaźnikowy.

W przypadku zmiennych statycznych istotny jest fakt, że one wszystkie są jawnie zadeklarowane w pewnych miejscach programu i podczas wykonania program może się odwoływać tylko do zestawu zmiennych ustalonego podczas pisania programu. Wiąże się z tym następujący problem: są zadania, dla których nie sposób przewidzieć liczby zmiennych faktycznie potrzebnych do przeprowadzenia obliczeń (i napisać w programie deklaracji tablicy o odpowiedniej długości). Nie można przy tym ustalić górnych ograniczeń, ponieważ mogą być potrzebne zmienne różnych typów — raz będzie trzeba więcej zmiennych jednego typu, a innym razem innego, natomiast ilość pamięci dla wszystkich zmiennych może być ograniczona wspólnie.

Do rozwiązania tego problemu służą zmienne dynamiczne — w tekście programu jest podany ich typ, ale nie mają one z góry ustalonego miejsca w pamięci i nie jest ustalona z góry liczba tych zmiennych. Zamiast tego istnieje pewna pula pamięci, zwana zwykle stertą (ang. *heap*), w której program, w miarę potrzeb, rezerwuje miejsce na zmienne dynamiczne. Rezerwacja miejsca jest równoznaczna z utworzeniem takiej zmiennej, a zwolnienie tego miejsca oznacza jej zlikwidowanie. W czasie między jej utworzeniem a likwidacją zmienna dynamiczna, tak jak każda zmienna, zajmuje stałe miejsce w pamięci. Program ma do niej dostęp za pomocą wskaźników.

Przypomnienie o wskaźnikach

Przypomnijmy na przykładzie składnię deklaracji i sposób użycia zmiennych wskaźnikowych

```
float a, b, *c, *d;
```

Zmienne `a` i `b` są typu zmiennopozycyjnego, zmienne `c` i `d` są wskaźnikami do typu zmiennopozycyjnego. Po wykonaniu instrukcji

```
c = d = &b;
a = 3.14;
```

```
*c = 2.0*a;
```

zmienna b, wskazywana przez c i d ma wartość 6.28 (z dokładnością do błędu reprezentacji).

Przypomnijmy, że słowo kluczowe void oznacza typ, którego zbiór wartości jest pusty i którego elementy nie zajmują miejsca w pamięci. W szczególności używamy tego typu dla procedur, które swoje wyniki przekazują tylko za pomocą parametrów lub efektów ubocznych. Typ void* jest „uniwersalnym” typem wskaźnikowym służącym do przechowywania adresów w pamięci operacyjnej bez wyszczególniania typu wskazywanego obiektu. Jeśli w programie byłyby deklaracje

```
int *a;  double *b;  void *c;
```

a dalej byłaby instrukcja

```
a = b;
```

to kompilator wypisałby co najmniej ostrzeżenie, z powodu niezgodności typów zmiennych w tej instrukcji; wartością każdej z nich jest adres maszynowy i może to być ten sam adres, ale jeśli wskazywana zmienna reprezentuje poprawną (z punktu widzenia sensu programu) wartość typu int, to ciąg bitów znajdujący się pod tym adresem prawie na pewno nie reprezentuje sensownej wartości typu double (zresztą ciągi bitów dla typów int i double mają zwykle inne długości). Natomiast typ void* jest traktowany jak zgodny z wszystkimi typami wskaźnikowymi. Kompilator C nie będzie więc narzekał na takie instrukcje[11]:

```
c = b;  a = c;
```

Można też użyć jawnej konwersji, pisząc

```
a = (int*)b;
```

co oznacza, że odpowiedzialność za sensowność takiego przypisania wziął na siebie autor programu.

[11] Jeśli program napisany w C kompilujemy przy użyciu kompilatora języka C++ (może być to ten sam kompilator; język jest rozpoznawany po rozszerzeniu nazwy pliku źródłowego, tj. .c albo .cpp), to kompilator uzna to za błąd. W tym przypadku konieczna jest jawna konwersja typów wskaźnikowych. Najlepiej jest więc zawsze jej używać.

Procedury obsługi sterty

Teraz możemy przedstawić procedury tworzenia i likwidacji zmiennych dynamicznych. Mają one nagłówki opisane w pliku stdlib.h. Nazwa size_t jest synonimem typu całkowitego (np. unsigned int), odpowiedniego do reprezentowania długości (w bajtach) obszarów pamięci w danej instalacji języka C.

```
void *malloc ( size_t size );
void *calloc ( size_t nmemb, size_t size );
```

Procedura malloc rezerwuje w stercie blok pamięci o długości podanej jako parametr i zwraca wartość będącą adresem pierwszego bajtu w tym bloku. Dopóki rezerwacja nie zostanie odwołana, procedury malloc i calloc nie mogą zarezerwować żadnego bloku pamięci mającego z nim niepuste przecięcie. Zawartość zarezerwowanego bloku jest nieokreślona. Możemy zatem (mając deklaracje zmiennych jak wyżej) napisać instrukcje

```
a = malloc ( sizeof(int) );
b = malloc ( sizeof(double) );
```

albo lepiej

```
a = (int*)malloc ( sizeof(int) );
b = (double*)malloc ( sizeof(double) );
```

Procedura calloc służy do rezerwowania tablicy; długość (w bajtach) rezerwowanego bloku jest iloczynem długości dwóch jej parametrów. Dodatkowo blok pamięci zarezerwowany przez calloc jest wypełniany zerami. Możemy na przykład napisać instrukcję

```
b = (double*)calloc ( 1000, sizeof(double) );
```

i dalej pisać w programie b[i], gdzie indeks i może przyjmować wartości od 0 do 999 (uwaga: to na autorze programu spoczywa odpowiedzialność za to, żeby wartości indeksów do tablicy podczas działania programu faktycznie przyjmowały wartości tylko z dozwolonego zbioru; błędy przekroczenia zakresu indeksów należą do najbardziej wrednych i trudnych do poprawienia).

Procedury rezerwacji pamięci mogą zawieść, jeśli zabrakło miejsca w stercie.

W takim przypadku zwrócona wartość jest adresem zerowym, oznaczanym przez nazwę NULL. Po wywołaniu takiej procedury wypada sprawdzić, czy akcja się powiodła, na przykład w ten sposób

```
if ( (b = (double*)calloc ( 1000, sizeof(double) )) ) {
  ...  /* wykonaj obliczenia */
}
else {
  printf ( "Przepraszam\n" );
  exit ( 1 );
}
```

Jeśli zmienna b otrzyma wartość NULL, czyli 0, to zamiast przeprowadzania niewykonalnych obliczeń program przeprosi za sprawiony zawód.

```
void free ( void *ptr );
```

Procedura free zwalnia rezerwację wcześniej zarezerwowanego i niezwolnionego bloku pamięci. Parmetr musi być adresem początku tego bloku, czyli wartością zwróconą wcześniej przez malloc lub calloc.

Struktury ze wskaźnikami

Jeszcze jeden problem wymaga uwzględnienia w języku. Można definiować wskaźniki do struktur, na przykład

```
typedef struct {
          float x, y;
        } complex, *pcomplex;
```

Mamy tu wprowadzone dwie nazwy, complex jest nazwą typu strukturalnego z dwoma polami zmiennopozycyjnymi, a pcomplex jest nazwą typu wskaźnikowego do takich struktur. Typ wskazywany został opisany wcześniej, zatem w chwili tłumaczenia nazwy pcomplex wiadomo, co ma być wskazywane. Dla wskaźnikowych struktur danych, takich jak listy i drzewa, konieczne jest jednak deklarowanie w strukturze pól wskaźnikowych do takich struktur — ale opis struktury jeszcze nie istnieje.

Rozwiązanie tego problemu polega na użyciu tzw. etykietki struktury; to jest dodatkowa nazwa podana po słowie kluczowym struct. Na przykład


```
typedef struct lista_zesp {
          complex z;
          struct lista_zesp *nast;
        } lista_zesp, *plista_zesp;
```

Identyfikator lista_zesp występuje najpierw dwa razy jako etykietka; pole nast jest typu wskaźnikowego i w chwili tłumaczenia przez kompilator opisu tego pola jest dostępna cała potrzebna w tej chwili informacja — że jest to pole wskaźnikowe do jakiejś struktury (może to być też struktura opisana dalej w programie, musi tylko być wyposażona w odpowiednią etykietkę). Trzecie wystąpienie nazwy lista_zesp ma na celu nazwanie typu strukturalnego, którego opis jest już ukończony. Etykietka nie musi być identyczna z nazwą, którą nadamy strukturze.

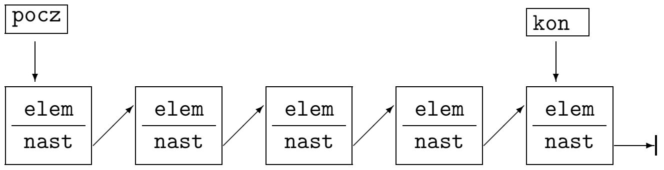
Listy jednokierunkowe

Przypuśćmy, że elementy dowolnego typu (np. liczby całkowite) chcemy ustawić w ciąg. Nie znamy z góry liczby elementów tego ciągu. Kolejne elementy, w miarę ich pojawiania się (np. czytania z pliku lub obliczania) możemy chcieć dołączać na początku lub na końcu ciągu. W tym celu możemy napisać tak:

```
typedef struct lista {
    int elem;
    struct lista *nast;
  } lista, *plista;
```

Typ strukturalny lista jest typem zmiennych dynamicznych, które będziemy tworzyć (ale moglibyśmy też zadeklarować zmienne statyczne tego typu). Typ plista jest typem wskaźnikowym. Struktura typu lista ma tu dwa pola; wartością pierwszego (elem) będzie element naszego ciągu, a drugie pole (nast) jest zmienną wskaźnikową — jego wartość posłuży do dostępu do następnego elementu ciągu.

Aby móc tworzyć zmienne dynamiczne, potrzebna jest co najmniej jedna statyczna zmienna wskaźnikowa, która będzie w tym przypadku dawać dostęp do początku listy. Nazwiemy ją pocz. Druga zmienna statyczna, kon, będzie wskazywać element ostatni i ułatwi dołączanie nowych elementów na koniec listy. Schemat budowy takiej listy wygodnie jest przedstawić na rysunku.




```
plista pocz, kon;

void InicjalizujListe ( void )
{
  pocz = kon = NULL;
} /*InicjalizujListe*/

void WstawNaPoczatek ( int x )
{
  plista p;
  if ( !(p = malloc ( sizeof(lista) )) )
    Error ();
  p->elem = x;
  p->nast = pocz;
  if ( !pocz ) kon = p;
  pocz = p;
} /*WstawNaPoczatek*/

void WstawNaKoniec ( int x )
{
  plista p;
  if ( !(p = malloc ( sizeof(lista) )) )
    Error ();
  p->elem = x;
  p->nast = NULL;
  if ( !pocz ) pocz = p;
             else kon->nast = p;
  kon = p;
} /*WstawNaKoniec*/
```

Powyższe procedury mogą posłużyć do zbudowania listy. Zmienne statyczne `pocz` i `kon` wskazują odpowiednio pierwszy i ostatni element listy (jeśli lista zawiera tylko jeden element, to obie wskazują go jednocześnie). Jeśli lista jest pusta, to obie te zmienne mają wartość wskaźnika pustego (`NULL`). Zwróćmy uwagę na pewien element dobrego stylu programowania: zmienna dynamiczna *natychmiast*

po utworzeniu przez procedurę `malloc` ma inicjalizowane wszystkie pola. Poza tym od razu jest włączana do naszej listy. Gdybyśmy tego nie zrobili, a ponadto „zapomnieli” jej adresu (zmienna `p` przestanie istnieć po zakończeniu działania procedury, w której jest zadeklarowana), to nie moglibyśmy odwołać rezerwacji obszaru przydzielonego przez procedurę `malloc` i do końca działania programu ten obszar nie mógłby już być użyty. Procedura `Error` jest wywoływana w sytuacji braku miejsca w stercie; jej zadaniem jest wypisanie stosownego komunikatu (pewnie należałoby dodać odpowiedni parametr) i zatrzymanie programu, przez wywołanie procedury `exit`.

```
int UsunZPoczatku ( void )
{
  plista p;
  int wynik;
  if ( pocz != NULL ) {
    p = pocz;
    pocz = p->nast;
    if ( pocz == NULL ) kon = NULL;
    wynik = p->elem;
    free ( p );
    return wynik;
  }
  else Error ();   /* lista pusta */
} /*UsunZPoczatku*/

int UsunZKonca ( void )
{
  plista p;
  int wynik;
  if ( pocz != NULL ) {
    wynik = kon->elem;
    if ( pocz == kon ) {
      free ( kon );
      pocz = kon = NULL;
    }
    else {
      p = pocz;
      while ( p->nast != kon )
        p = p->nast;
      free ( kon );
      kon = p;
      kon->nast = NULL;
```

```
    }
    return wynik;
  }
  else Error ();    /* lista pusta */
} /*UsunZKonca*/
```

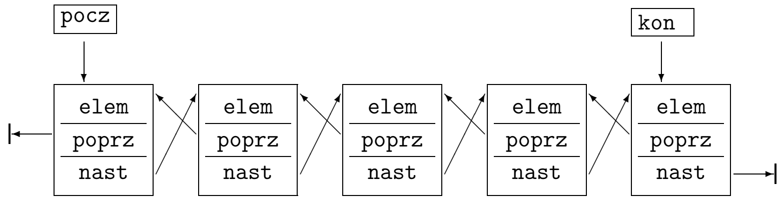
Powyższe dwie procedury usuwają odpowiednio pierwszy i ostatni element listy, podając wartość przechowywanej w nim danej. Zauważmy, że o ile dołączenie elementu na początek i na koniec listy, a także usunięcie elementu z początku listy odbywa się przy użyciu stałej liczby operacji, to aby usunąć element z jej końca, trzeba przejrzeć całą listę — po to, aby odnaleźć element przedostatni, który stanie się ostatnim. Jedyna droga do niego wiedzie od początku listy.

Przy użyciu podprogramów przedstawionych wyżej możemy zaimplementować zarówno stos, jak i kolejkę. Operacja wstawiania elementu na stos (`Push`) jest realizowana przez podprogram `WstawNaPoczatek`, a zdejmowania (`Pop`) przez podprogram `UsunZPoczatku`. Operację wstawiania do kolejki wykona procedura `WstawNaKoniec`, a usuwa element z kolejki funkcja `UsunZPoczatku`. Moglibyśmy użyć procedur odwrotnie (tj. np. element wstawiany na stos byłby dołączany na koniec listy), ale ponieważ usuwanie elementu z końca listy wymaga przejrzenia jej całej, więc nie należy tego robić.

Listy dwukierunkowe

Opisane poprzednio listy jednokierunkowe mają zasadniczą cechę „użytkową”: można je efektywnie przetwarzać „do przodu”. Jeśli mamy wskaźnik pewnego elementu takiej listy, to łatwo znajdziemy element następny. Jednak znalezienie elementu poprzedniego wymaga znacznie więcej pracy — trzeba zaczynając od początku listy przeglądać jej elementy kolejno, aż do znalezienia elementu, którego następnym jest element dany. Czas działania tej procedury może być zatem proporcjonalny do długości listy. Kosztem wprowadzenia dodatkowego wskaźnika dla każdego elementu możemy otrzymać listę, po której łatwo jest „poruszać się” w obie strony.

```
typedef struct dlista {
    int elem;
    struct dlista *poprz, *nast;
  } dlista, *pdlista;

 pdlista pocz, kon;
```

Procedura inicjalizacji listy jest taka sama jak poprzednio — oba wskaźniki statyczne otrzymują wartość `NULL`. Do wstawiania nowego elementu na początek i na koniec listy dwukierunkowej służą procedury

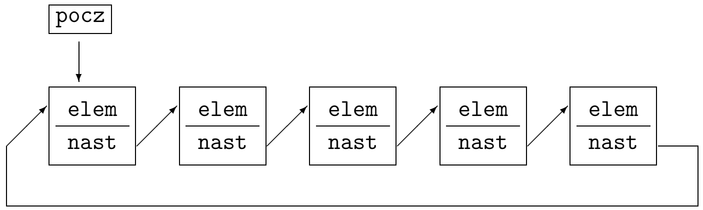
```
void WstawNaPoczatek ( int x )
{
  pdlista p;

  if ( !(p = malloc ( sizeof(dlista) )) )
    Error ();
  p->elem = x;
  p->poprz = NULL;
  p->nast = pocz;
  if ( !pocz )
    kon = p;
  else pocz->poprz = p;
  pocz = p;
} /*WstawNaPoczatek*/

void WstawNaKoniec ( int x )
{   pdlista p;

  if ( !(p = malloc ( sizeof(pdlista) )) )
    Error ();
  p->elem = x;
  p->nast = NULL;
  p->poprz = kon;
  if ( !pocz )
    pocz = p;
  else kon->nast = p;
  kon = p;
} /*WstawNaKoniec*/
```

Zauważmy, że powyższe dwie procedury są „symetryczne”, tj. są w nich tylko zamienione role wskaźników pocz i kon oraz poprz i nast w każdym elemencie. Możemy teraz napisać procedurę usuwania dowolnego elementu z listy dwukierunkowej. Wymagamy, aby parametr procedury wskazywał element obecny w liście. Jeśli lista jest niepusta, to w szczególności procedurę tę będziemy mogli wywołać z parametrem pocz albo kon.

```
void Usun ( pdlista pel )
{
  if ( pocz == kon ) {
    if ( pocz == pel )
      pocz = kon = NULL;
    else Error (); /* elementu nie ma w liście */
  }
  else {
    if ( pel == pocz ) {
      pocz = pocz->nast;
      pocz->poprz = NULL;
    }
    else if ( pel == kon ) {
      kon = kon->poprz;
      kon->nast = NULL;
    }
    else {
      pel->nast->poprz = pel->poprz;
      pel->poprz->nast = pel->nast;
    }
  }
  free ( pel );
} /*Usun*/
```

Listy cykliczne

Ostatni element listy ma wskaźnik elementu następnego równy NULL. Dla listy dwukierunkowej wskaźnik elementu poprzedzającego pierwszy element też ma wartość NULL. Można „zamknąć” takie listy, tj. np. elementowi ostatniemu przypisać jako element następny element pierwszy.



Zmienna wskaźnikowa pocz wskazuje na *któryś* z elementów takiej listy. W pewnych zastosowaniach takie rozwiązanie jest naturalne. Tymczasem zwróćmy uwagę, że z definicji typu zmiennych dynamicznych nie wynika, jaka struktura zostanie z tych zmiennych utworzona. Dlatego konieczne jest ustalanie i opisywanie (w dokumentacji programu i w komentarzach) sposobów wykorzystania dynamicznych struktur danych.

Zadania i problemy

1. Napisz procedurę dołączania elementu na koniec listy jednokierunkowej, dla której istnieje tylko zmienna wskazująca początek listy.
2. Napisz procedurę, która przetwarza listę jednokierunkową w ten sposób, że kolejność elementów zostaje odwrócona.
3. Napisz procedurę, która wyszukuje w liście jednokierunkowej wszystkie elementy spełniające pewien warunek (który jest sprawdzany przez pewną osobną procedurę) i usuwa te elementy z listy.
4. Napisz procedurę, która otrzymuje dwie listy liniowe (parametrami procedury są wskaźniki początków obu list, nie używamy wskaźników końcowych elementów list) i która łączy te listy w jedną, tak, aby elementy z poszczególnych list występowały na przemian. Jeśli jedna z list jest krótsza, to „ogon” drugiej listy będzie „ogonem” listy połączonej.
5. Przerabiając nieco poprzednie zadanie, napisz procedurę realizującą algorytm sortowania przez scalanie.
6. Napisz procedurę, która wyszukuje w liście jednokierunkowej element, którego pole `el` jest równe liczbie podanej jako parametr. Wartością procedury ma być wskaźnik tego elementu listy. Jeśli takiego elementu w liście nie ma, to wartością procedury jest `NULL`.

   Procedurę napisz dla listy „zwykłej”, a także dla listy ze strażnikiem, tj. dodatkowym elementem wskazywanym przez element ostatni. Przed przeszukaniem listy do strażnika wpisuje się wyszukiwaną liczbę, dzięki czemu nie trzeba sprawdzać, czy wskaźnik kolejnego elementu listy jest różny od `NULL`.
7. Zaimplementuj procedury, które rezerwują pamięć na macierz o wymiarach $n \times n$ (gdzie n jest dość duże), reprezentowaną za pomocą tablicy wskaźników wierszy; wiersze te mają być tablicami jednowymiarowymi rezerwowanymi każdy osobno za pomocą procedury `calloc`, zwalniają pamięć zajmowaną przez taką macierz, przestawiają dwa wskazane wiersze i obliczają iloczyn dwóch macierzy reprezentowanych w taki sposób.

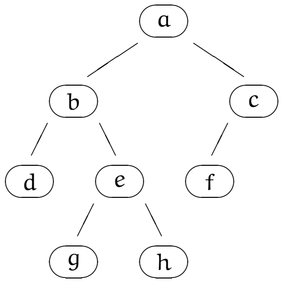
   Jaki jest koszt przestawienia wierszy? Porównaj go z kosztem przestawienia wierszy w statycznej tablicy dwuwymiarowej.

Drzewa binarne

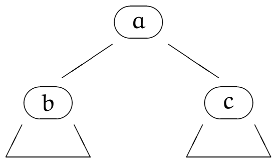
Drzewo binarne jest sposobem częściowego uporządkowania pewnych danych, określonym rekurencyjnie, w następujący sposób: może ono być puste (tj. zbiór danych w drzewie jest pusty), albo może składać się z wierzchołka zwanego korzeniem, oraz z dwóch poddrzew (które są drzewami i mogą w szczególności być puste). Z wierzchołkami wiążemy poszczególne dane, których rodzaj może być dowolny, zależnie od zastosowania. Przykładem drzewa binarnego jest rozpatrywany wcześniej kopiec.

„Botaniczna” terminologia drzew zawiera jeszcze słowo liść, jako określenie wierzchołka, którego oba poddrzewa są puste. Oprócz tej terminologii jest jeszcze terminologia „rodzinna” (inspirowana pojęciem drzewa genealogicznego). Jeśli wierzchołek w_1 ma poddrzewa, których korzeniami są wierzchołki w_2 i w_3, to wierzchołki te są synami (lewym i prawym) wierzchołka w_1, który z kolei jest ich ojcem.

Zobaczmy dwa podstawowe sposoby przedstawiania drzew binarnych: rysunkowy i łańcuchowy. Tradycyjnie drzewa rysuje się korzeniem do góry, liśćmi do dołu:



Korzeń narysowanego wyżej drzewa zawiera daną a, a liście zawierają dane d, f, g i h. Jeśli chcemy skupić uwagę na pewnym wierzchołku drzewa (bo interesują nas tylko dane „w jego bezpośrednim sąsiedztwie”), to możemy na rysunku zastosować uproszczenie:



Łańcuch przedstawiający drzewo binarne jest ciągiem symboli, którego składnię można zdefiniować za pomocą notacji podobnej do opisu składni języka C; definicja taka jest w istocie podaną na wstępie definicją drzewa binarnego. Jeśli przyjmiemy, że dane reprezentujemy przy użyciu liter, to mamy

dana: jeden z
  a b ... z

wierzchołek:
  dana

drzewo:
  (*drzewo*) *wierzchołek* (*drzewo*)
  (*drzewo*) *wierzchołek*
  wierzchołek (*drzewo*)
  wierzchołek
  puste

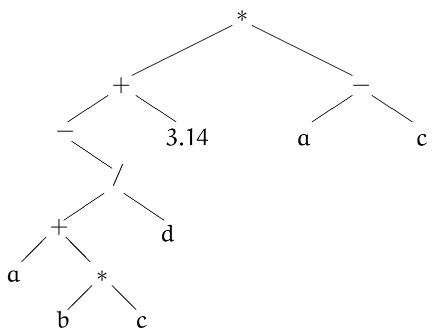
Łańcuch reprezentujący zgodnie z tą składnią drzewo pokazane wcześniej na rysunku jest następujący:

((d) b ((g) e (h))) a ((f) c)

Wyrażenia arytmetyczne składają się z argumentów, operatorów (tj. symboli działań) i z nawiasów. Zauważmy, że każde takie wyrażenie (z operatorami jedno- i dwuargumentowymi) możemy reprezentować za pomocą drzewa, np.

$$(-(a + b * c)/d + 3.14) * (a - c)$$

ma postać



Argumenty występują w liściach tego drzewa, a w pozostałych wierzchołkach występują operatory. W korzeniu jest operator, który reprezentuje ostatnie działanie wykonywane podczas obliczania wartości tego wyrażenia. Jedna z metod kompilacji programów polega na utworzeniu drzew reprezentujących wyrażenia występujące w tekście źródłowym. Na podstawie drzew kompilator generuje rozkazy dla procesora.

Drzewa binarnego wyszukiwania

Inna dziedzina zastosowań drzew to wszelkiego rodzaju słowniki, czyli struktury danych, które służą do przechowywania i sprawnego wyszukiwania danych, na podstawie tzw. kluczy. Zakładamy, że zbiór wszystkich możliwych kluczy jest liniowo uporządkowany (np. alfabetycznie lub numerycznie).

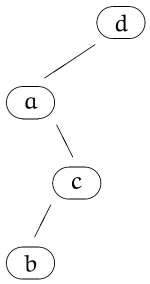
Def. Drzewo binarnego wyszukiwania (ang. *binary search tree*, BST) jest to drzewo binarne, którego każdy wierzchołek zawiera klucz i związane z nim informacje dodatkowe, przy czym dla dowolnego wierzchołka klucze wszystkich wierzchołków w lewym poddrzewie są nie większe, a klucze wszystkich wierzchołków w prawym poddrzewie są nie mniejsze niż klucz w tym wierzchołku.

Podstawową cechą struktury danych, będącej implementacją słownika, jest złożoność (pesymistyczna, optymistyczna lub średnia) odnajdywania potrzebnej informacji na podstawie klucza. Jeśli słownik jest zaimplementowany w postaci drzewa binarnego wyszukiwania, to informację wyszukuje się, zaczynając od korzenia. Koszt jest więc zależny od długości drogi, którą trzeba przebyć od korzenia do odpowiedniego wierzchołka.

Poziom wierzchołka w drzewie jest długością drogi, którą należy przebyć od korzenia tego drzewa do tego wierzchołka, np. korzeń ma poziom 0, jego synowie — 1 itd. Drzewo binarne jest całkowicie wyważone, jeśli jego wszystkie liście mają poziom $h-1$ lub $h-2$, a wszystkie wierzchołki o poziomie mniejszym niż $h-2$ mają oba poddrzewa niepuste[12]. Kopiec, który posłużył do sprawnego zaimplementowania kolejki priorytetowej, jest przykładem drzewa binarnego całkowicie wyważonego (uwaga: to nie jest drzewo binarnego wyszukiwania). O takich drzewach wiemy, że nie istnieje drzewo niższe z tą samą liczbą wierzchołków. Dlatego dostęp do dowolnej danej w drzewie binarnym, który polega na dojściu do odpowiedniego wierzchołka zaczynając od korzenia, ma najmniejszą pesymistyczną złożoność obliczeniową wtedy, gdy to drzewo jest całkowicie wyważone. Złożoność ta ma rząd $O(\log n)$.

[12] Litera h oznacza wysokość drzewa, czyli liczbę poziomów. W całym wykładzie przyjmiemy umowę, że poziomy te są numerowane od 0, zatem ostatni poziom ma numer $h-1$.

Przeciwieństwem drzewa całkowicie wyważonego jest takie drzewo, które ma tylko jeden liść. Każdy pozostały wierzchołek musi mieć wtedy tylko jedno poddrzewo niepuste. Dostęp do informacji w liściu wymaga przejścia przez wszystkie wierzchołki drzewa, zajmuje więc $O(n)$ operacji.



W związku z powyższym należy dążyć do ograniczania wysokości drzew reprezentujących słowniki; idealnie by było, gdyby zawsze były one całkowicie wyważone. Jeśli dane w drzewie są ustalone (i dostęp do nich odbywa się wielokrotnie), to warto zbudować takie drzewo BST. Jeśli jednak w trakcie obliczeń należy wstawiać i usuwać pewne dane (tj. tworzyć i niszczyć wierzchołki), to zwykle nie należy tego robić, ponieważ wstawienie lub usunięcie danej, zależnie od porządku, którego odzwierciedleniem jest drzewo, mogłoby spowodować konieczność przebudowania całego drzewa (tj. wykonania pewnych czynności dla wszystkich jego wierzchołków). Dlatego w praktyce stosuje się rozwiązania kompromisowe. Jedno z nich dokładnie omówimy później.

Wskaźnikowa implementacja drzew binarnych wyszukiwań

Jeśli drzewo jest całkowicie wyważone, to najlepsza dla jego reprezentacji jest tablica, taka jaka była użyta do implementacji kopca. Jej zaletą jest to, że aby uzyskać dostęp do synów lub ojca dowolnego wierzchołka, można użyć prostego wzoru, dzięki czemu informacja o budowie drzewa nie wymaga dodatkowej pamięci (w tablicy wystarczy przechowywać tylko dane, bez indywidualnych informacji potrzebnych do uzyskania dostępu do synów lub ojca danego wierzchołka). Jeśli jednak mamy drzewo, które ma tylko jeden liść, przy czym każdy wierzchołek ma lewe poddrzewo puste, to jeśli w sumie jest n wierzchołków, to liść trzeba by umieścić w tablicy w pozycji 2^n-1. Do przechowania 20 wierzchołków takiego drzewa trzeba by użyć tablicy o długości ponad milion.

Implementacja wskaźnikowa drzewa binarnego wyszukiwania może być zrealizowana przy użyciu następujących typów:

```
typedef ... typklucza;
typedef ... typdanej;
```

```
typedef struct vertex {
          typklucza klucz;
          typdanej dana;
          struct vertex *lewy, *prawy;
        } vertex, *pvertex;
```

W pewnych zastosowaniach może być potrzebne wprowadzenie jeszcze jednego wskaźnika, o nazwie gora lub `ojciec`, którego wartość umożliwi dostęp do ojca danego wierzchołka. W większości zastosowań konieczne są tylko wskaźniki do korzeni poddrzew.

Drzewo binarnego wyszukiwania jest przetwarzane przy użyciu procedury inicjalizacji, wstawiania elementu, wyszukiwania elementu i niszczenia drzewa. Korzeń drzewa jest statyczną globalną (tj. zadeklarowaną poza procedurami) zmienną wskaźnikową o nazwie `root`. Założymy, że klucze nie powtarzają się w wierzchołkach drzewa.

```
pvertex root;

void InitBST ( void )
{
  root = NULL;
} /*InitBST*/
```

Procedurę `InitBST` należy oczywiście wywołać tylko raz, na początku działania programu. Ponowne jej wywołanie wprawdzie „opróżni” drzewo, ale nie odwoła rezerwacji pamięci zajętej przez wierzchołki. Jeśli zatem drzewo po wykorzystaniu należy opróżnić (w celu ewentualnego budowania od początku), to trzeba użyć podanej dalej procedury `BSTDestroy`.

```
pvertex BSTNewLeaf ( typklucza klucz, typdanej dana )
{
  pvertex p;

  if ( (p = malloc ( sizeof(vertex) )) ) {
    p->klucz = klucz;
    p->dana = dana;
    p->lewy = p->prawy = NULL;
  }
  return p;
} /*BSTNewLeaf*/
```


Procedura `BSTNewLeaf` ma za zadanie utworzenie zmiennej dynamicznej, która będzie reprezentowała nowy lisc drzewa i przypisanie odpowiednich wartości wszystkim polom tej zmiennej. Jeśli procedura `malloc` nie może zarezerwować odpowiedniego bloku pamięci, to również wartością procedury `BSTNewLeaf` jest wskaźnik pusty (`NULL`).

```
void BSTInsert ( typklucza klucz, typdanej dana )
{
  pvertex *p;

  p = &root;
  while ( *p ) {
    if ( (*p)->klucz == klucz ) return;
    else if ( (*p)->klucz > klucz ) p = &(*p)->lewy;
    else p = &(*p)->prawy;
  }
  *p = BSTNewLeaf ( klucz, dana );
} /*BSTInsert*/
```

Zwróćmy, uwagę, jak w powyższej procedurze jest wykorzystany *wskaźnik do wskaźnika*; początkową wartością zmiennej `p` jest adres zmiennej `root`. Każda wartość przypisana dalej, w pętli, jest adresem pola wskaźnikowego (`lewy` lub `prawy`) w wierzchołku, które to pole wskazuje następny wierzchołek w drodze do miejsca w drzewie, w którym ma zostać wstawiony nowy wierzchołek.

```
char BSTFind ( typklucza klucz, typdanej *dana )
{
  pvertex p;

  p = root;
  while ( p ) {
    if ( p->klucz == klucz ) {
      *dana = p->dana;
      return 1;
    }
    else if ( p->klucz > klucz ) p = p->lewy;
    else p = p->prawy;
  }
  return 0;
} /*BSTFind*/
```

Powyższa procedura wyszukiwania danej w słowniku zaimplementowanym w postaci drzewa BST zwraca wartość różną od zera, jeśli słownik zawiera wierzchołek o podanym kluczu.

Procedurę niszczenia całego drzewa, która ma zwolnić pamięć zajmowaną przez wszystkie zmienne dynamiczne, które reprezentowały jego wierzchołki, najprościej jest zrealizować jako podprogram rekurencyjny:

```
void BSTDestroy ( pvertex *tree )
{
  if ( *tree ) {
    BSTDestroy ( &(*tree)->lewy );
    BSTDestroy ( &(*tree)->prawy );
    free ( *tree );
    *tree = NULL;
  }
} /*BSTDestroy*/
```

Zwróćmy uwagę, że parametrem tej procedury jest wskaźnik do zmiennej wskazującej wierzchołek (korzeń) niszczonego drzewa. Po zniszczeniu drzewa zmiennej tej jest przypisywana wartość `NULL`, która umożliwia rozpoznanie, że drzewo jest puste. Alternatywne rozwiązanie wykorzystuje parametr typu `pvertex`; przypisywanie mu wartości `NULL` po wywołaniu procedury `free` jest oczywiście bezprzedmiotowe, ale przy takim rozwiązaniu po wywołaniu procedury niszczenia drzewa należy przypisać wartość `NULL` zmiennej `root` (najbardziej elegancko byłoby wywołać w tym celu procedurę `InitBST`).

Kolejne zadanie, które możemy łatwo rozwiązać, mając słownik w postaci drzewa BST, polega na wyprowadzeniu (np. wypisaniu do pliku, ale niekoniecznie) wszystkich danych ze słownika w kolejności uporządkowania kluczy. Najprostszy algorytm rozwiązywania tego zadania jest również rekurencyjny. Załóżmy, że wyprowadzanie danych (cokolwiek miałoby się z nimi stać) wykonuje procedura `Wyprowadz`.

```
void Wyprowadz ( typklucza klucz, typdanej dana )
{
  ...
} /*Wyprowadz*/
```

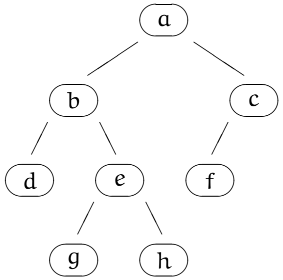


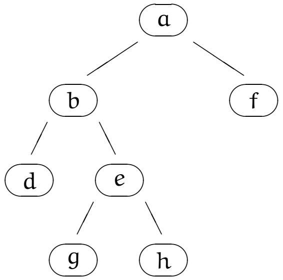
```
void BSTOutputAscending ( pvertex tree )
{
  if ( tree ) {
    BSTOutputAscending ( tree->lewy );
    Wyprowadz ( tree->klucz, tree->dana );
    BSTOutputAscending ( tree->prawy );
  }
} /*BSTOutputAscending*/

void BSTOutputDescending ( pvertex tree )
{
  if ( tree ) {
    BSTOutputDescending ( tree->prawy );
    Wyprowadz ( tree->klucz, tree->dana );
    BSTOutputDescending ( tree->lewy );
  }
} /*BSTOutputDescending*/
```

Istotnie, pierwsza z tych procedur przed wyprowadzeniem danej przechowywanej w wierzchołku drzewa wskazywanym przez parametr wyprowadza dane z lewego poddrzewa (a więc dane, których klucze są mniejsze niż klucz w tym wierzchołku), a dane z prawego poddrzewa (których klucze są większe) są wyprowadzane później. Kolejność wyprowadzania danych przez drugą procedurę jest odwrotna.

Możemy zatem drzewo binarnych wyszukiwań zastosować do posortowania danych; w tym celu tworzymy drzewo, a następnie używamy jednej z powyższych procedur. Zauważmy jednak, że koszt wstawiania wierzchołka do drzewa jest proporcjonalny do poziomu w drzewie, na który ten wierzchołek trafi. Jeśli ciąg danych, które kolejno wstawiamy do drzewa, jest uporządkowany, to wyhodujemy drzewo o wysokości równej liczbie elementów, a to oznacza, że koszt sortowania w tym przypadku jest rzędu n^2, gdzie n jest długością ciągu.

Najbardziej skomplikowaną operacją jest usuwanie wierzchołka drzewa BST, które może być potrzebne w pewnych zastosowaniach słownika. Najprostszy przypadek mamy wtedy, gdy trzeba usunąć liść, lub wierzchołek, którego jedno z poddrzew jest puste. Na przykład, usunięcie wierzchołka c z drzewa na rysunku wiąże się z „przemieszczeniem” wierzchołka f o jeden poziom bliżej korzenia.





Aby usunąć wierzchołek, którego oba poddrzewa są niepuste (np. korzeń drzewa na rysunku), trzeba na jego miejsce wprowadzić inny wierzchołek, do którego można „podczepić” oba poddrzewa. Mamy dwóch „kandydatów” do tej roli: są nimi wierzchołek z największym kluczem w poddrzewie lewym, oraz wierzchołek z najmniejszym kluczem w poddrzewie prawym. Każdy z tych wierzchołków ma co najmniej jedno poddrzewo puste (a zatem stosunkowo łatwo jest go „wyjąć” z drzewa) i umieszczenie dowolnego z nich w korzeniu zapewnia właściwe uporządkowanie kluczy w drzewie.

```
pvertex BSTExtractMax ( pvertex *tree )
{
  pvertex p;

  while ( (*tree)->prawy )
    tree = &(*tree)->prawy;
  p = *tree;
  *tree = p->lewy;
  return p;
} /*BSTExtractMax*/
```

Zadaniem procedury `BSTExtractMax` jest wyjęcie z drzewa, do korzenia którego wskaźnik jest wskazywany przez parametr (drzewo nie może być puste!), wierzchołka z największym kluczem. Droga od korzenia do tego wierzchołka prowadzi zawsze w prawo. Zauważmy, że ten wierzchołek może być korzeniem drzewa i w takiej sytuacji korzeń musi być zmieniony (to dlatego parametrem procedury jest wskaźnik do wskaźnika korzenia). Ponieważ procedura `BSTExtractMax` jest pomocnicza, tj. ma być wywoływana tylko przez `BSTDelete`, nie modyfikuje (na `NULL`) wskaźników w wierzchołku usuniętym z drzewa.


```
void BSTDelete ( typklucza klucz )
{
  pvertex p, q, *r;

  r = &root;
  while ( *r ) {
        /* w pętli wyszukujemy wierzchołek o podanym kluczu */
    p = *r;
    if ( klucz < p->klucz )
      r = &p->lewy;
    else if ( klucz > p->klucz )
      r = &p->prawy;
    else {
        /* wierzchołek został znaleziony; */
        /* najpierw obsługa przypadków, gdy jedno poddrzewo puste */
      if ( !p->lewy ) *r = p->prawy;
      else if ( !p->prawy ) *r = p->lewy;
      else {
        /* oba poddrzewa niepuste, należy w miejsce usuwanego */
        /* wierzchołka "wmontować" wierzchołek wyjęty z poddrzewa */
        q = BSTExtractMax ( &p->lewy );
        q->lewy = p->lewy;
        q->prawy = p->prawy;
        *r = q;
      }
      free ( p );
      return;  /* zadanie wykonane, koniec */
    }
  }
        /* jeśli wierzchołek z podanym kluczem nie istnieje, */
        /* to pętla zakończy się z powodu niespełnienia warunku, */
        /* ze wskaźnik *r jest różny od NULL */
} /*BSTDelete*/
```

Zauważmy, że jeśli maksymalny klucz w lewym poddrzewie jest korzeniem tego poddrzewa (tj. `p->lewy` początkowo wskazuje na wierzchołek o największym kluczu w lewym poddrzewie), to procedura `BSTExtractMax` odpowiednio ten wskaźnik zmieni. Dzięki temu przypisanie `q->lewy = p->lewy` przypisuje wskaźnikowi lewego poddrzewa wierzchołka, który zastąpił wierzchołek usunięty, poprawną wartość.

Zadania i problemy

1. Napisz procedury obsługi drzewa binarnych wyszukiwań ze strażnikiem. Jest to zmienna typu takiego, jak wierzchołki drzewa, wskazywana przez dodatkową wskaźnikową zmienną statyczną `straznik`. Wskaźniki `lewy` i `prawy` do nieistniejących wierzchołków (np. w liściach) mają wskazywać strażnika. Jeśli drzewo jest puste, to zmienna `root` wskazuje strażnika.

   Przed wyszukiwaniem wierzchołka o danym kluczu należy ten klucz przypisać do pola `klucz` w strażniku, dzięki czemu można nie sprawdzać, czy wierzchołek istnieje (tzn. czy wskaźnik wierzchołka ma wartość różną od `NULL`) — klucz zawsze zostanie znaleziony, jeśli w strażniku, to znaczy, że w drzewie go nie ma.
2. Napisz procedurę `BSTExtractMin`, która wymontowuje z drzewa BST wierzchołek o kluczu najmniejszym w poddrzewie, którego wskaźnik wskaźnika korzenia jest podany jako parametr. Napisz wariant procedury `BSTDelete`, która usuwa wierzchołek o podanym kluczu, posługując się w tym celu procedurą `BSTExtractMin` zamiast `BSTExtractMax`.
3. Napisz procedury obsługi drzewa binarnych wyszukiwań zbudowanego z wierzchołków, króre oprócz wskaźników do lewego i prawego poddrzewa mają też wskaźnik do góry (tj. do „ojca” — w korzeniu ten wskaźnik ma mieć wartość `NULL`).
4. Znajdź rząd złożoności optymistycznej sortowania przy użyciu drzewa BST. Przypadek optymistyczny jest wtedy, gdy początkowe uporządkowanie danych prowadzi do otrzymania drzewa całkowicie wyważonego (to stwierdzenie należy uzasadnić).
5. Do struktury `vertex` dodaj pola o nazwach `poziom` i `numer`. Następnie napisz procedurę, która polu `poziom` każdego wierzchołka w drzewie BST przypisze odległość tego wierzchołka od korzenia (korzeń ma poziom 0, korzenie jego poddrzew 1 itd.), a pole `numer` ma otrzymać wartość będącą numerem danego wierzchołka w ramach poziomu (licząc po kolei od 0).

   Do rozwiązania tego zadania użyj pomocniczej listy, której elementy zawierają liczniki elementów w obrębie poziomu.

Drzewa AVL

Drzewa binarnych wyszukiwań umożliwiają wstawianie, wyszukiwanie i usuwanie elementów w czasie proporcjonalnym do wysokości drzewa. Jeśli drzewo zostało zbudowane przez wstawienie n danych z kluczami uporządkowanymi w szczególnie niekorzystny sposób, to wysokość drzewa może być równa n, a w takiej sytuacji zastosowanie drzewa nie daje żadnych korzyści w porównaniu z np. listą nieuporządkowaną z tymi danymi.

Oczywiście, maksymalną korzyść z zastosowania drzewa BST będziemy mieć, jeśli jego wysokość będzie jak najmniejsza. Drzewo binarne, które ma n wierzchołków, musi mieć wysokość co najmniej $\lfloor \log_2 n \rfloor + 1$. Taką minimalną wysokość drzewo miałoby, gdyby było <u>całkowicie zrównoważone</u>, tj. gdyby wszystkie jego liście były na dwóch najwyższych poziomach, a wszystkie wierzchołki na poziomach pozostałych miały oba poddrzewa niepuste.

Jeśli drzewo służy do zbudowania statycznego słownika, tj. zbiór danych w drzewie jest ustalony, a jedyną operacją (która będzie wykonywana wielokrotnie) jest wyszukiwanie, to można i warto zbudować takie drzewo. Jeśli jednak zawartość słownika zmienia się, to po wstawieniu lub usunięciu wierzchołka należałoby przywrócić zrównoważenie drzewa. Okazuje się, że koszt uporządkowania drzewa, który przywraca jego całkowite zrównoważenie, może być proporcjonalny do liczby wierzchołków. Takie postępowanie jest więc nieopłacalne.

Rozwiązania kompromisowe dopuszczają drzewa niezrównoważone, ale takie, których wysokość jest równa $O(\log n)$. Jednym z takich rozwiązań są drzewa AVL. Nazwa tej struktury danych pochodzi od nazwisk jej wynalazców (Adelson-Velski, Landis).

<u>Def.</u> <u>Drzewem AVL</u> nazywamy drzewo binarnych wyszukiwań, które spełnia następujący warunek: dla każdego wierzchołka różnica wysokości poddrzew tego wierzchołka nie może być większa niż 1.

Okazuje się, że powyższy warunek zapewnia, że wysokość drzewa jest rzędu $\log n$, a ponadto umożliwia wstawianie i usuwanie wierzchołków w czasie proporcjonalnym do wysokości. Aby warunek był spełniony, wstawiając lub usuwając wierzchołek nie musimy nigdy przebudowywać całego drzewa; wystarczy tylko wykonać pewne operacje wzdłuż ścieżki od korzenia do odpowiedniego wierzchołka.

Udowodnimy, że wysokość drzewa AVL z n wierzchołkami jest rzędu $\log n$. Drzewo o wysokości 1 ma zawsze 1 wierzchołek (tj. korzeń). Drzewo o wysokości 2 ma 2 lub 3 wierzchołki. Drzewo o wysokości h ma co najmniej $C(h)$ wierzchołków, przy czym dla $h > 2$

$$C(h) = C(h-1) + C(h-2) + 1.$$

Powyższy wzór wziął się stąd, że przyjmujemy, że wysokości poddrzew korzenia różnią się o 1 (tj. drzewo w korzeniu jest tak niezrównoważone, jak tylko dopuszcza to definicja drzewa AVL). Każde z poddrzew zawiera najmniejszą dopuszczalną liczbę wierzchołków, a ostatni składnik, tj. 1, uwzględnia wierzchołek w korzeniu.

Jawną postać funkcji $C(h)$ otrzymamy rozwiązując równanie różnicowe. Rozwiązaniem równania jednorodnego $a_h = a_{h-1} + a_{h-2}$ jest ciąg

$$a_h = e\lambda_1^h + f\lambda_2^h,$$

gdzie $\lambda_1 = \frac{1}{2}(1+\sqrt{5})$, $\lambda_2 = \frac{1}{2}(1-\sqrt{5})$ są pierwiastkami wielomianu charakterystycznego $\lambda^2 - \lambda - 1$, a e i f są dowolnymi liczbami. Ponieważ liczba $\mu = 1$ nie jest pierwiastkiem tego wielomianu, więc istnieje rozwiązanie szczególne równania niejednorodnego, które jest wielomianem stopnia 0 (tj. ciągiem stałym, $C(h) = c$). Po podstawieniu do tego równania otrzymujemy

$$c = 2c + 1, \quad \text{czyli} \quad c = -1,$$

zatem ogólne rozwiązanie równania niejednorodnego ma postać

$$C(h) = e\lambda_1^h + f\lambda_2^h - 1.$$

Liczby e i f znajdziemy na podstawie warunków początkowych, $C(1) = 1$, $C(2) = 2$. Mamy stąd układ równań liniowych,

$$\lambda_1 e + \lambda_2 f = 2,$$
$$\lambda_1^2 e + \lambda_2^2 f = 3,$$

którego rozwiązanie daje nam $e = \frac{1}{10}(5+3\sqrt{5})$, $f = \frac{1}{10}(5-3\sqrt{5})$. Zauważmy, że $\lambda_1 > 1$, $|\lambda_2| < \lambda_1$, oraz $e > -f > 0$. Liczbę n wierzchołków w drzewie AVL o wysokości h oszacujemy tak:

$$n \geq C(h) = e\lambda_1^h + f\lambda_2^h - 1 > (e+f)\lambda_1^h - 1.$$

Ponieważ $e + f = 1$, więc mamy stąd

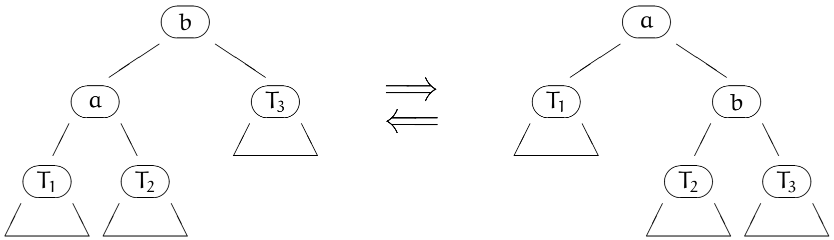
$$\lambda_1^h < n + 1,$$

czyli

$$h < \log_{\lambda_1}(n+1) = O(\log n).$$

Skoro udowodniliśmy, że wysokość drzewa AVL jest rzędu $\log n$, pozostaje wykazać, że wstawianie i usuwanie wierzchołka może być wykonane w czasie proporcjonalnym do wysokości drzewa. Wstawianie wierzchołka do drzewa AVL zaczyna się od wstawienia wierzchołka w taki sam sposób, jak do zwykłego drzewa BST, a następnie należy przywrócić zrównoważenie drzewa, jeśli zostało ono naruszone. Podobnie, usunięcie wierzchołka może doprowadzić do niespełnienia warunku definiującego drzewo AVL, a zatem należy potem tak przekształcić drzewo, aby ten warunek przywrócić.

Elementarną operacją, która służy do przebudowywania drzewa jest tzw. rotacja. Najlepiej wyjaśnia ją rysunek:



Przypuśćmy, że drzewo na początku ma wierzchołki i poddrzewa takie, jak na rysunku z lewej strony. Poddrzewa T_1, T_2 i T_3 mogą być dowolne, także puste. Rotacja w prawo przebuduje to drzewo tak, że powstanie układ z prawej strony. Odwrotną operacją jest rotacja w lewo. Zauważmy, że jeśli przed wykonaniem rotacji klucze w wierzchołkach drzewa są uporządkowane zgodnie z zasadą budowy drzewa BST, to po wykonaniu rotacji drzewo dalej będzie poprawnym drzewem BST.

Każdy wierzchołek drzewa AVL ma zwykłe atrybuty wierzchołka drzewa binarnych wyszukiwań, tj. klucz, ewentualne informacje związane z kluczem i wskaźniki lewego i prawego poddrzewa (może też być wskaźnik „do góry”). Do wykrywania sytuacji, gdy rotacje są potrzebne, służy dodatkowy atrybut wierzchołka, będący wskaźnikiem zrównoważenia. Przypuśćmy, że atrybut ten ma nazwę `eqi`. Jego wartością jest różnica wysokości lewego i prawego poddrzewa. Zatem, w drzewie AVL wartość tego atrybutu jest równa 1, jeśli lewe poddrzewo danego wierzchołka jest wyższe, 0 jeśli oba poddrzewa mają tę samą wysokość i -1, jeśli wyższe jest poddrzewo prawe.

Wyprowadzimy wzory, umożliwiające obliczenie wskaźników zrównoważenia po wykonaniu rotacji w prawo na podstawie wskaźników zrównoważenia przed rotacją, a następnie napiszemy procedurę rotacji. Wysokości poddrzew T_1, T_2 i T_3 oznaczymy odpowiednio h_1, h_2, h_3. Wskaźniki zrównoważenia wierzchołków a i b w początkowym drzewie oznaczymy symbolami a_0 i b_0, a w końcowym drzewie symbolami a_1 i b_1.

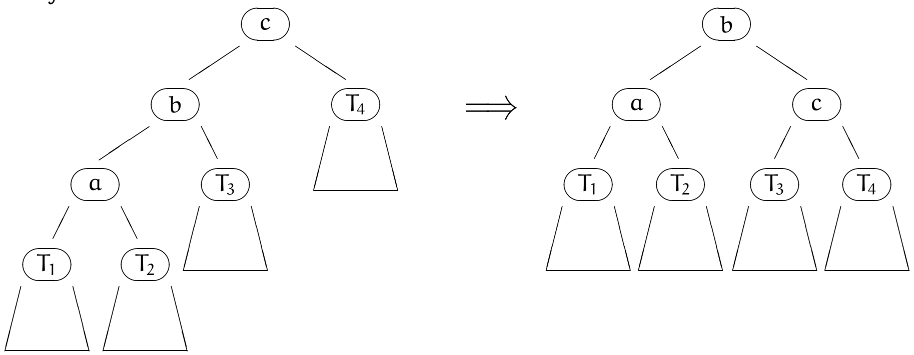
Jeśli $a_0 \geq 0$, to $h_1 = h_2 + a_0 = h_3 + b_0 - 1$, a stąd $b_1 = h_2 - h_3 = b_0 - a_0 - 1$. Przypuśćmy, że $b_1 \geq 0$. Wtedy po rotacji poddrzewo b ma wysokość $h_2 + 1$, a zatem $a_1 = a_0 - 1$. Jeśli zaś $b_1 < 0$, to $a_1 = h_1 - h_3 - 1$. Podstawiając $h_1 - h_3 = b_0 - 1$ dostajemy $a_1 = b_0 - 2$.

Przypuśćmy, że $a_0 < 0$. Wtedy $h_2 = h_3 + b_0 - 1$, a zatem $b_1 = h_2 - h_3 = b_0 - 1$. Jeśli $b_1 \geq 0$, to tak samo jak poprzednio, $a_1 = a_0 - 1$, a w przeciwnym razie $a_1 = h_1 - (h_3 + 1)$. Podstawiając $h_1 = h_2 + a_0$ oraz $h_2 = h_3 + b_1$, po uproszczeniu otrzymujemy $a_1 = a_0 + b_0 - 2$.

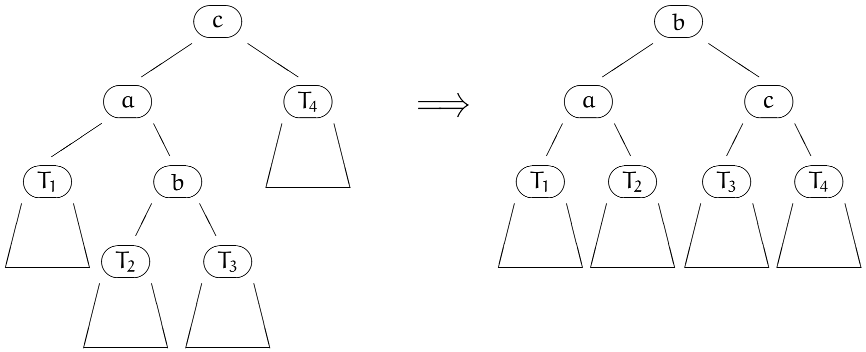
```
void RotacjaWPrawo ( pvertex *root )
{
  pvertex a, b;
  int     a0, b0, b1;

  b = *root;  a = b->lewy;  /* zakładamy, że a != NULL */
  b->lewy = a->prawy;  a->prawy = b;
  *root = a;     /* nowym korzeniem jest wierzchołek a */
  a0 = a->eqi;  b0 = b->eqi;
  if ( a0 >= 0 ) {
    b1 = b0 - a0 - 1;
    if ( b1 >= 0 ) a->eqi = a0 - 1; else a->eqi = b0 - 2;
  }
  else {
    b1 = b0 - 1;
    if ( b1 >= 0 ) a->eqi = a0 - 1; else a->eqi = a0 + b0 - 2;
  }
  b->eqi = b1;
} /*RotacjaWPrawo*/
```

Procedura `RotacjaWLewo` wygląda podobnie, w związku z czym wyprowadzenie potrzebnych wzorów i jej napisanie pozostawiam jako ćwiczenie. Obie procedury zastosujemy teraz do wstawiania wierzchołka do drzewa AVL.

Przypuśćmy, że w pewnym wierzchołku drzewa BST jedno z poddrzew pewnego wierzchołka (dla ustalenia uwagi lewe) ma wysokość większą o 2 od drugiego (prawego) poddrzewa, przy czym oba te poddrzewa są drzewami AVL. Wtedy, zależnie od budowy lewego poddrzewa, należy dokonać jednej albo dwóch rotacji, aby otrzymać drzewo AVL. Pierwsza sytuacja (gdy wystarczy jedna rotacja) jest na rysunku.



Jak widać, wystarczyło dokonać jednej rotacji w prawo, w wierzchołku c; nowym korzeniem stał się wierzchołek b. Jeśli lewe poddrzewo ma prawe poddrzewo wyższe, to potrzebna jest podwójna rotacja; najpierw trzeba dokonać rotacji lewego poddrzewa w lewo, co spowoduje powstanie drzewa o budowie jak z lewej strony na rysunku wyżej, a następnie można dokonać rotacji w prawo.



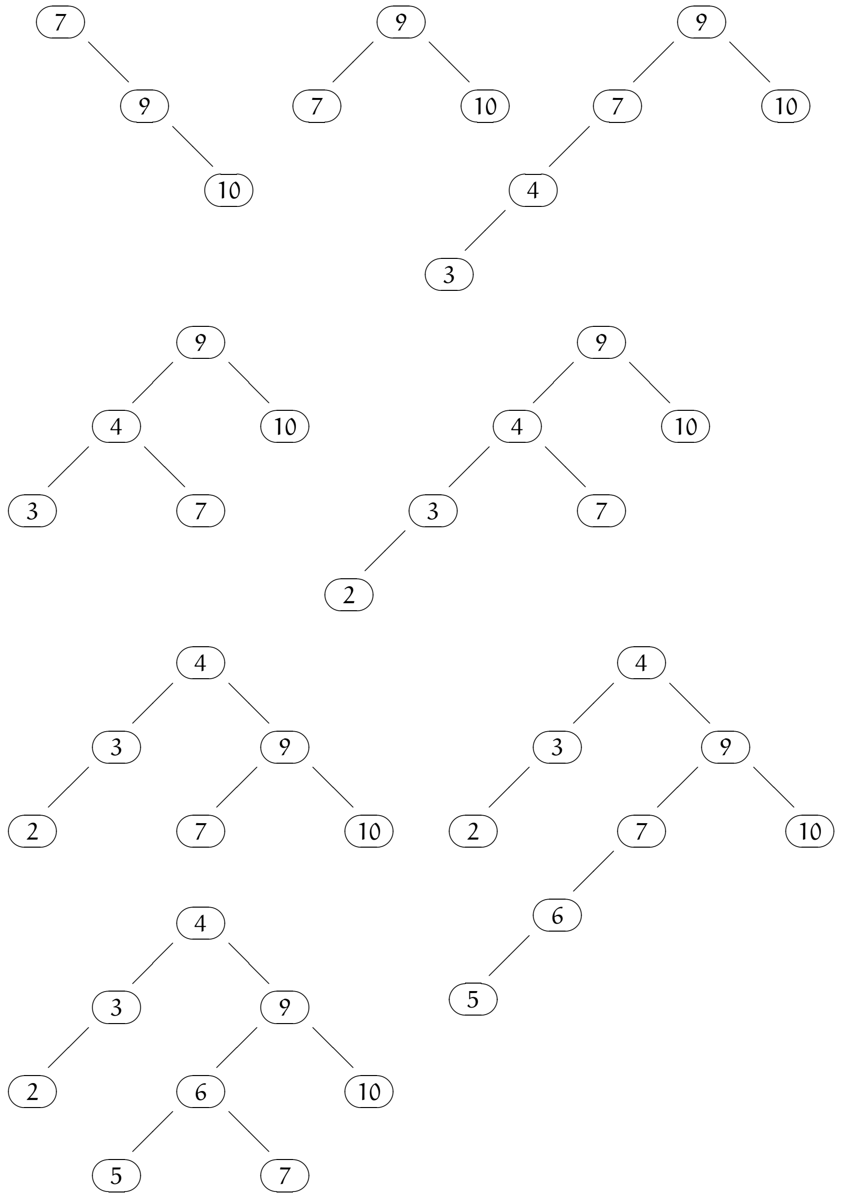
Zauważmy, że w pierwszym przypadku nie ma znaczenia, czy poddrzewa T_1 i T_2 mają jednakową wysokość, czy jedno z nich jest o 1 wyższe. Podobnie, w drugim przypadku poddrzewa T_2 i T_3 mogą mieć jednakową wysokość, lub jedno z nich może być wyższe o 1, a końcowy wynik będzie poprawny. Jeśli prawe poddrzewo ma wysokość większą o 2 od poddrzewa lewego, to też mamy dwa sposoby przywracania zrównoważenia, które trzeba stosować w sytuacjach symetrycznych do narysowanych wyżej dwóch.

Podprogram wstawiania wierzchołka do drzewa AVL jest rekurencyjną procedurą, której wartość niezerowa oznacza, że wysokość drzewa wzrosła (oczywiście o 1).

```
char AVLInsert ( typklucza k, typdanej d, pvertex *root )
{
  if ( !(*root) ) {
    *root = malloc ( sizeof(vertex) );
    (*root)->klucz = k;  (*root)->dana = d;
    (*root)->lewy = (*root)->prawy = NULL;  (*root)->eqi = 0;
    return 1;
  }
  else if ( k < (*root)->klucz ) {
    if ( AVLInsert ( k, d, &(*root)->lewy ) ) {
      (*root)->eqi ++;
      if ( (*root)->eqi == 2 ) {
        if ( (*root)->lewy->eqi < 0 )  /* potrzebna jest podwójna */
                                       /* rotacja */
          RotacjaWLewo ( &(*root)->lewy );
        RotacjaWPrawo ( root );
        return 0;  /* rotacja przywróciła początkową wysokość */
      }
      else return (*root)->eqi == 1;
    }
    else return 0;
  }
  else {
    ...  /* tu wstawiamy wierzchołek do prawego poddrzewa, */
         /* co wymaga "symetrycznych" działań */
  }
} /*AVLInsert*/
```

Podprogram usuwania wierzchołka z drzewa AVL jest bardziej skomplikowany (zobacz Uzupełnienia). Odnotujmy, że jeśli potrzebujemy „wyjąć” z poddrzewa wierzchołek z największym kluczem w tym poddrzewie (po to, aby stał się on korzeniem poddrzewa w miejsce usuwanego korzenia), to wierzchołek ten albo będzie liściem, albo będzie miał niepuste lewe poddrzewo złożone z jednego liścia. Usunięcie wierzchołka może obniżyć wysokość poddrzewa (tylko o 1), dlatego też może zajść konieczność przywrócenia zrównoważenia drzewa we wszystkich wierzchołkach po drodze do korzenia całego drzewa.

Przykład. Do początkowo pustego drzewa AVL wstawimy wierzchołki z kluczami 7, 9, 10, 4, 3, 2, 6, 5. Na rysunku jest pokazana struktura drzewa przed i po każdej rotacji.



Zadania i problemy

1. Napisz procedurę rotacji w lewo, która odpowiednio uaktualni wskaźniki zrównoważenia.
2. Napisz brakującą część podprogramu `AVLInsert`, tj. kod wstawiający wierzchołek do prawego poddrzewa i przywracający w razie potrzeby zrównoważenie drzewa.
3. Uzupełnij procedury rotacji tak, aby nadawały się do obsługi drzew, w których oprócz wskaźników `lewy` i `prawy` (do poddrzew), każdy wierzchołek ma wskaźnik `gora`, którego wartością jest najbliższy wierzchołek na drodze do korzenia (w korzeniu jego wartością ma być `NULL`).
4. Dla drzewa AVL, którego wierzchołki zawierają wskaźniki do góry, napisz nierekurencyjną procedurę wstawiania nowego wierzchołka.
5. Wykaż, że jeśli do początkowo pustego drzewa AVL wstawimy $n = 2^h - 1$ wierzchołków, których klucze będą uporządkowane rosnąco, to otrzymane na końcu drzewo będzie całkowicie zrównoważone i jego wysokość będzie równa h.
6. Do drzewa, które było przykładem na wykładzie, został wstawiony klucz 8. Jakie rotacje muszą być wykonane, aby przywrócić zrównoważenie drzewa?
7. Narysuj drzewa AVL, które powstaną w wyniku usuwania z drzewa w poprzednim zadaniu kolejno wierzchołków zawierających najmniejszy w danej chwili klucz.
8. Pewne drzewo binarne spełnia następujący warunek: dla każdego wierzchołka wartość bezwzględna różnicy wysokości poddrzew tego wierzchołka jest nie większa niż 2. Czy stąd wynika, że istnieje stała c, taka że wysokość takiego drzewa o n wierzchołkach jest nie większa niż $c\log_2 n$?.

Uzupełnienia

Dla osób zainteresowanych zamieszczam kod procedury usuwania wierzchołka z drzewa AVL. Procedura rekurencyjna, która wyszukuje w drzewie wierzchołek z podanym kluczem i usuwa go, jest zadeklarowana z atrybutem static, który sprawia, że nie można jej wywołać bezpośrednio z procedury w innym pliku źródłowym; można ją wywołać tylko za pośrednictwem procedury AVLDelete podanej dalej. Jeśli wierzchołek, który ma być usunięty, ma oba poddrzewa niepuste, to w jego lewym poddrzewie wyszukuje się wierzchołek z największym kluczem, który jest „wyjmowany” z drzewa i wstawiany w miejsce wierzchołka do usunięcia. Parametr swl jest adresem zmiennej, w której jest zapamiętywany adres wskaźnika lewego poddrzewa wierzchołka „zastępującego” wierzchołek usuwany. Zmienna ta jest zadeklarowana w procedurze AVLDelete.

Procedura _AVLDelete zwraca wartość 0 jeśli wysokość poddrzewa o wierzchołku wskazywanym przez *node nie zmieniła się, albo 1, jeśli zmalała.
Parametr todelete wskazuje wskaźnik do usuwanego wierzchołka, jeśli ten wierzchołek ma oba poddrzewa niepuste; wskaźnik ten (zmienna wskazywana przez parametr root albo wskaźnik w odpowiednim wierzchołku drzewa) musi ulec zmianie, aby wskazywać na wierzchołek „zastępujący”.

```
static char _AVLDelete ( pvertex *node, pvertex *todelete,
                         pvertex **swl, typklucza k )
{
  char     c;
  pvertex tn, a;

  if ( !(*node) )
    return 0; /* nie znaleziony klucz, niczego nie usuwamy */
  tn = *node;
  if ( todelete ) {  /* poszukujemy "zastępcy"*/
    if ( !tn->prawy ) { /* jest "zastępca"; */
      a = *todelete;     /* zamień wierzchołki i usuń */
      *node = tn->lewy;
      *todelete = tn;
      tn->lewy = a->lewy; tn->prawy = a->prawy;
      tn->eqi = a->eqi;
      free ( a );
      *swl = &tn->lewy;
      return 1;
```



```
    }
    else if ( _AVLDelete ( &tn->prawy, todelete, swl, k ) )
      goto the_right;
    else
      return 0;
  }
  else { /* poszukujemy wierzchołka do usunięcia */
    if ( tn->klucz == k ) { /* znaleźliśmy */
      todelete = node;
      if ( !tn->prawy )
        *node = tn->lewy;
      else if ( !tn->lewy )
        *node = tn->prawy;
      else
        goto tough_case;
      free ( tn );
      return 1;
    }
tough_case:  /* usuwany wierzchołek ma oba poddrzewa niepuste, */
             /* szukamy "zastępcy" */
    if ( tn->klucz >= k ) {
      if ( _AVLDelete ( &tn->lewy, todelete, swl, k ) ) {
        tn = *node;
        tn->eqi --;
        if ( tn->eqi == -2 ) {
          if ( tn->prawy->eqi > 0 )
            RotacjaWPrawo ( &tn->prawy );
          RotacjaWLewo ( node );
        }
        return (*node)->eqi == 0;
      }
      else return 0;
    }
    else {
      if ( _AVLDelete ( &tn->prawy, todelete, swl, k ) ) {
the_right:
        if ( *swl ) {
          if ( **swl == tn ) { node = *swl; tn = *node; }
        }
        tn->eqi ++;
```

```
        if ( tn->eqi == +2 ) {
          if ( tn->lewy->eqi < 0 )
            RotacjaWLewo ( &tn->lewy );
          RotacjaWPrawo ( node );
        }
        return (*node)->eqi == 0;
      }
      else return 0;
    }
  }
} /*_AVLDelete*/

boolean AVLDelete ( pvertex *root, typklucza k )
{
  pvertex *swl;

  swl = NULL;
  return _AVLDelete ( root, NULL, &swl, k );
} /*AVLDelete*/
```

Kodowanie Huffmana

Uwagi o kompresji danych

Zadaniem o ogromnym znaczeniu praktycznym jest kompresja danych; ilość danych magazynowanych i przesyłanych jest ogromna i wciąż rośnie, a koszty przechowywania np. jednego megabajta danych wprawdzie maleją, ale i tak są znaczne. Jeszcze większy problem jest z przesyłaniem danych, ponieważ konkurencja między właścicielami łączy komunikacyjnych jest znacznie mniejsza niż między producentami dysków i płyt. W pewnych przypadkach koszt przesłania jednego bitu danych jest astronomiczny (np. w transmisji obrazów zarejestrowanych przez sondy kosmiczne) i każda oszczędność jest niesłychanie pożądana.

Kompresja danych występuje w dwóch odmianach: może być bezstratna, albo stratna. W pierwszym przypadku zamieniamy oryginalny zbiór danych (np. ciąg bajtów) na inny (krótszy) ciąg, który możemy zmagazynować lub przesłać. Na podstawie tego ciągu jest możliwe dokładne odtworzenie danych oryginalnych. Ten sposób kompresji jest stosowany wtedy, gdy odtworzone dane muszą być identyczne z oryginalnymi. Dotyczy to przede wszystkim danych tekstowych, np. programów (źródłowych i skompilowanych) lub tekstów literackich i baz danych.

Kompresja stratna jest dopuszczalna wtedy, gdy odtworzone dane nie muszą być identyczne z oryginalnymi, co ma miejsce najczęściej dla obrazów i dźwięków. W tym przypadku określamy dopuszczalny błąd, tj. różnicę między danymi odtworzonymi i oryginalnymi. Poziom dopuszczalnego błędu jest zwykle taki, aby różnica między obrazem lub dźwiękiem oryginalnym i odtworzonym była niedostrzegalna lub niesłyszalna dla przeciętnego człowieka. Dzięki dopuszczeniu niezerowego błędu można osiągnąć znacznie większy poziom kompresji danych (przykładowo, kompresja bezstratna obrazu umożliwia „zmniejszenie jego objętości" dwu- lub trzykrotne, podczas gdy kompresja stratna umożliwia otrzymanie danych kilkadziesiąt razy krótszych niż dane oryginalne). W pewnych przypadkach (zwłaszcza w medycynie) stratna kompresja obrazów nie jest jednak dopuszczalna ze względów prawnych.

Warto zauważyć, że nie istnieje algorytm kompresji bezstratnej, który dla każdego zbioru danych (dla ustalenia uwagi: ciągu bajtów o długości N) wytworzyłby zakodowany ciąg krótszy. Gdyby taki algorytm istniał, to długość M najdłuższego ciągu zakodowanego byłaby mniejsza niż N. Mielibyśmy wtedy co najwyżej 256^M różnych ciągów zakodowanych. Ale kompresja bezstratna musi być przekształceniem różnowartościowym zbioru wszystkich możliwych ciągów N bajtów, czyli różnowartościowym przekształceniem zbioru 256^N elementowego w zbiór, który ma mniej niż 256^N elementów. Takie przekształcenie nie istnieje. Zatem kompresja bezstratna jest skuteczna tylko dla danych mających pewne szczególne własności. Dane, z jakimi mamy do czynienia, bardzo często mają takie własności.

Drzewa Huffmana

Przypuśćmy, że w oryginalnym ciągu danych (np. bajtów) $d_1, \ldots, d_N$ pewne elementy występują częściej niż inne. Naturalnym pomysłem jest zastąpienie każdego elementu ciągu kodem przypisanym mu w taki sposób, aby elementy, które występują w ciągu najczęściej, miały kody krótsze (tj. złożone z mniejszej liczby bitów). Dane zakodowane są ciągiem bitów. Dekodowanie polega na wyodrębnieniu w tym ciągu grup bitów składających się na poszczególne kody, a następnie zastąpieniu każdego kodu odpowiednim elementem oryginalnego ciągu. Oczywiście, w tym celu oprócz zakodowanych danych należy przesłać informację o odwzorowaniu kodów na elementy.

Niech $s_0, \ldots, s_{n-1}$ będzie zbiorem wartości elementów ciągu danych (jeśli jest to ciąg bajtów, to mamy zbiór liczb $\{0, \ldots, 255\}$ lub jego podzbiór). Informacja o powiązaniu kodów z elementami $s_0, \ldots, s_{n-1}$ ma postać drzewa binarnego. Aby zdekodować kolejny element oryginalnego ciągu, ustawiamy się w korzeniu i przetwarzamy kolejne bity, aż dojdziemy do liścia. Jeśli mamy bit 0, przechodzimy do korzenia lewego poddrzewa, a jeśli 1, to do prawego. W liściu drzewa mamy element s_i, którego kod składa się z bitów przetworzonych po drodze od korzenia; oczywiście drzewo ma n liści i różnym sposobom kodowania odpowiadają drzewa o różnej budowie.

Symbolami $p_0, \ldots, p_{n-1}$ określimy prawdopodobieństwa wystąpienia elementów $s_0, \ldots, s_{n-1}$ w ciągu $d_1, \ldots, d_N$. Jeśli mamy zakodować tylko jeden ciąg, to możemy utworzyć histogram, tj. dla każdego elementu policzyć jego wystąpienia w ciągu danym i podzielić ich liczbę przez długość ciągu. W ogólności możemy projektować kod nie znając danych, ale znając rozkład prawdopodobieństwa występowania poszczególnych znaków w danych należących do interesującej nas klasy. Dalej zamiast o prawdopodobieństwach będziemy mówić o „wagach" wierzchołków.

Pomysł Huffmana, który zapewnia otrzymanie kodu optymalnego (tj. takiego, przy którym wartość oczekiwana długości ciągu zakodowanego jest najmniejsza), polega na budowaniu drzewa „od dołu”, tj. zaczynając od liści. Liście tworzymy dla wszystkich elementów $s_0, \dots, s_{n-1}$, przy czym w liściu dla elementu s_i zapamiętujemy (jako dodatkowy atrybut) wagę p_i. W każdym kolejno tworzonym wierzchołku wewnętrznym drzewa zapamiętamy wagę, która jest prawdopodobieństwem wystąpienia w ciągu danych wszystkich elementów związanych z liśćmi poddrzewa, którego korzeniem jest ten wierzchołek.

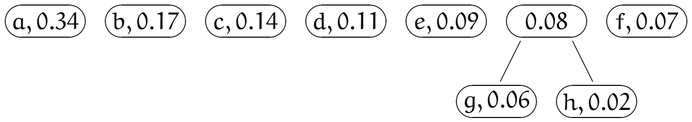
Liście sortujemy w kolejności rosnących wag i wstawiamy (w tej kolejności) do kolejki (możemy też użyć kolejki priorytetowej, w której priorytet jest największy dla elementu o najmniejszej wadze). Tworzymy drugą, początkowo pustą kolejkę (zwykłą), w której będziemy umieszczać wierzchołki wewnętrzne tworzonego drzewa.

Następnie wykonujemy $n - 1$ razy następujące czynności: z kolejek wyjmujemy dwa wierzchołki (każdy z nich jest liściem lub wierzchołkiem wewnętrznym), o najmniejszych wagach. Wierzchołków tych należy poszukiwać na początku kolejek, ponieważ w kolejce z liśćmi wagi wstawionych do niej elementów są uporządkowane rosnąco, zaś w kolejce z wierzchołkami wewnętrznymi elementy, które będziemy wstawiać kolejno, również będą miały coraz większe wagi (to będzie dalej uzasadnione). Zatem, aby wyjąć każdy z potrzebnych wierzchołków, jeśli obie kolejki są niepuste, to porównujemy wagi wierzchołków na początku tych kolejek i wyjmujemy wierzchołek o wadze mniejszej.

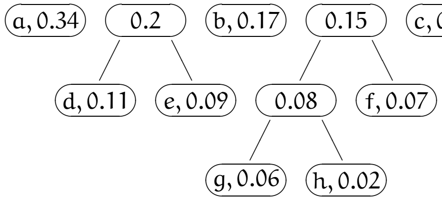
Mając dwa wierzchołki wyjęte z kolejek, tworzymy nowy wierzchołek wewnętrzny. Korzeniami jego poddrzew czynimy wierzchołki wyjęte z kolejek. Nowemu wierzchołkowi przypisujemy wagę równą sumie wag korzeni poddrzew. Zauważmy, że początkowe posortowanie liści w kolejności niemalejących wag zapewnia, że waga nowego wierzchołka rzeczywiście nie może być mniejsza od wag wierzchołków wewnętrznych utworzonych wcześniej. Nowy wierzchołek wstawiamy na koniec kolejki wierzchołków wewnętrznych, a zatem w każdej chwili zawartość tej kolejki jest uporządkowana w kolejności rosnących wag.

Za każdym razem łączna liczba elementów w obu kolejkach zmniejsza się o 1, a zatem po wykonaniu opisanych wyżej czynności w kolejce wierzchołków wewnętrznych mamy jeden wierzchołek drzewa; wierzchołek ten jest korzeniem drzewa Huffmana, które możemy użyć w celu zakodowania i zdekodowania danych. Aby móc zdekodować dane, nie trzeba przesyłać drzewa. Wystarczy przesłać informację o wagach (prawdopodobieństwach) $p_0, \dots, p_{n-1}$ wystąpienia elementów $s_0, \dots, s_1$ (uwaga: jak poradzić sobie z sytuacją, w której pewne dwa znaki (lub więcej) występują z tym samym prawdopodobieństwem?).

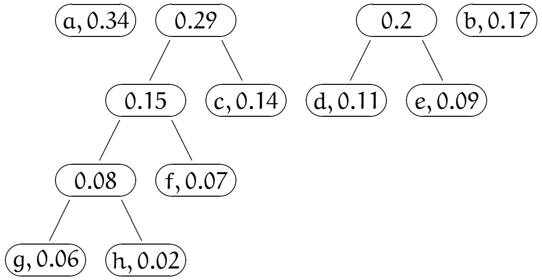
Przykład. Zbiór wartości danego ciągu składa się z ośmiu elementów, a, b, c, d, e, f, g, h. Przypuśćmy, że prawdopodobieństwa ich wystąpienia są odpowiednio równe $p_a = 0.34$, $p_b = 0.17$, $p_c = 0.14$, $p_d = 0.11$, $p_e = 0.09$, $p_f = 0.07$, $p_g = 0.06$, $p_h = 0.02$. Pierwszy wierzchołek wewnętrzny będzie wskazywał na liście z elementami g i h. Zauważmy, że mając dwa poddrzewa, które „podczepimy” pod nowy wierzchołek, możemy dowolnie wybrać spośród nich lewy i prawy.
W przykładzie konsekwentnie przyjmujemy, że korzeń prawego poddrzewa ma mniejszą wagę.



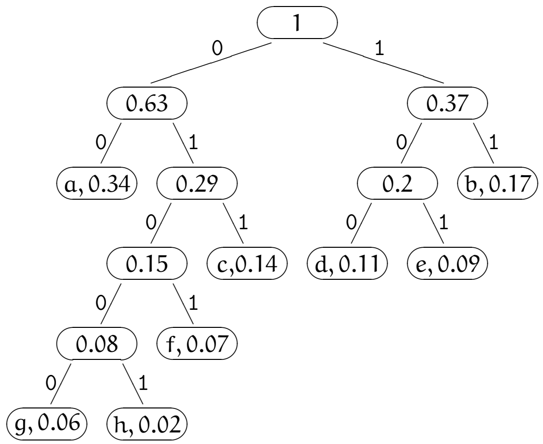
Drugi wierzchołek wewnętrzny wskazuje na liść f i na poprzednio utworzony wierzchołek wewnętrzny. Trzeci wierzchołek wewnętrzny wskazuje na liście d i e.



Czwarty wierzchołek wewnętrzny wskazuje na drugi utworzony wierzchołek wewnętrzny i na liść c.



Po utworzeniu pozostałych wierzchołków otrzymujemy kompletne drzewo Huffmana, które wygląda następująco:



Na podstawie drzewa możemy podać kody przyporządkowane wszystkim elementom: $a \to 00$, $b \to 11$, $c \to 011$, $d \to 100$, $e \to 101$, $f \to 0101$, $g \to 01000$, $h \to 01001$. Zobaczmy, jaka jest efektywność tego kodu. Jeśli mamy ciąg o długości 1000, w którym 340 razy występuje element a, 170 razy element b itd. (zgodnie z rozkładem przyjętym do określenia kodu), to w wyniku zakodowania tego ciągu otrzymamy $(340 + 170) \cdot 2 + (140 + 110 + 90) \cdot 3 + 70 \cdot 4 + (60 + 20) \cdot 5 =$ 2720 bitów.

Jeśli elementy oryginalnego ciągu są kodowane jako bajty, to ciąg ten składa się oczywiście z 8000 bitów. Ponieważ zbiór wartości ciągu jest ośmioelementowy, więc moglibyśmy do wszystkim elementom przyporządkować kody trzybitowe. Wtedy mielibyśmy 3000 bitów, czyli więcej, niż w przypadku użycia kodu Huffmana. W naszym przykładzie kod Huffmana jest lepszy od kodu trzybitowego o niecałe 10%, ale w bardziej „życiowych” przypadkach może być znacznie lepiej. Jeśli chcemy zakodować tekst napisany np. w języku polskim, to większość bajtów reprezentacji tego tekstu odpowiada literom, spacjom i znakom interpunkcyjnym. Zamiana bajtów na kody Huffmana może zmniejszyć objętość takich danych o kilkanaście procent, czyli niewiele. Algorytmy „prawdziwej” kompresji działają w ten sposób, że wyszukują w tekście powtarzające się słowa, składające się z wielu bajtów. Kody przyporządkowuje się tym słowom (trzeba jednak przyznać, że nie zawsze w takiej kompresji stosuje się kodowanie optymalne, czyli Huffmana).

Udowodnimy, że kod Huffmana jest optymalny, tj. nie istnieje kod lepszy, czyli dający zakodowaną reprezentację ciągu danego o mniejszej oczekiwanej długości.

Każdemu kodowi odpowiada pewne drzewo binarne. Jeśli symbolem l_i oznaczymy liczbę bitów przyporządkowanych elementowi s_i (jest to odległość w drzewie od korzenia do odpowiedniego liścia), to wartość oczekiwana liczby bitów zakodowanego ciągu jest równa

$$E = N \sum_{i=0}^{n-1} p_i l_i.$$

Zauważmy, że optymalny kod istnieje. Ponieważ liczba liści (czyli elementów $s_0, \ldots, s_{n-1}$) jest ustalona, więc drzew binarnych o takiej liczbie liści jest skończenie wiele i któreś z nich reprezentuje kod optymalny. Weźmy teraz dowolne drzewo binarne T reprezentujące pewien kod. Rozważmy wierzchołek w, który jest wewnętrznym wierzchołkiem najbardziej oddalonym od korzenia (w przykładzie jest to wierzchołek z wagą 0.08; w ogólnym przypadku może być więcej niż jeden taki wierzchołek). Jeśli dwa liście „podczepione” do tego wierzchołka nie mają najmniejszych dwóch wag, to zamienienie ich z liśćmi elementów o najmniejszych wagach, położonych bliżej korzenia, da drzewo reprezentujące kod o mniejszej wartości E.

Dalej postępujemy indukcyjnie; niech w drzewie T dwa liście o najmniejszych wagach będą podczepione do wierzchołka wewnętrznego w, najbardziej odległego od korzenia (liście te wyznaczają najdłuższe kody). Rozważmy drzewo T', powstałe z drzewa T przez zastąpienie wierzchołka w i jego dwóch liści przez liść w', którego waga jest równa sumie wag korzeni poddrzew wierzchołka w. Odpowiada to zastąpieniu dwu elementów, s_i i s_j, reprezentowanych przez liście usunięte z T, jednym (nowym) elementem s, którego prawdopodobieństwo wystąpienia w ciągu danym jest równe $p_i + p_j$. Niech E' oznacza oczekiwaną długość zakodowanego ciągu danego z elementami s_i i s_j zastąpionymi przez s. Ponieważ $E = E' + N(p_i + p_j)$, więc liczba E jest najmniejsza (czyli kod reprezentowany przez drzewo T jest optymalny) wtedy i tylko wtedy, gdy liczba E' jest najmniejsza. $\Box$

Bufory bitowe

Do implementacji kodowania potrzebny jest jeden element „techniczny”, mianowicie możliwość pisania do pliku i czytania z pliku pojedynczych bitów. Najmniejszą jednostką pamięci, którą można jednorazowo przesyłać w większości systemów komputerowych, jest jeden bajt, tj. zespół ośmiu bitów. Ponieważ chcemy dokonać kompresji, konieczne jest „pakowanie” bitów w bajt, oraz ich „rozpakowywanie”.

Rozwiązanie problemu polega na zastosowaniu bufora, tj. zmiennej pomocniczej, w której możemy gromadzić kolejno wyprowadzane bity, i którą po zapełnieniu możemy wyprowadzić do pliku i „opróżnić”. Różne bufory mają znacznie więcej zastosowań; system operacyjny tworzy odpowiedni bufor dla każdego pliku otwartego przez program, ponieważ dane są czytane i pisane na dysku „porcjami” znacznie większymi niż jeden bajt — zwykle to jest od 512 bajtów do kilku kilobajtów (obsługa buforów systemowych jest niewidoczna dla programów). Do obsługi buforów bitowych odpowiednich na potrzeby kodowania i dekodowania, możemy użyć następujących podprogramów:

```
FILE *f;
unsigned char bufor, maska;
char koniec; /* zmienna koniec jest potrzebna podczas czytania */

void ZacznijPisanie ( char nazwa[] )
{
  f = fopen ( nazwa, "w+");
  maska = 1;
  bufor = 0;  /* ustaw na zero wszystkie 8 bitów */
} /*ZacznijPisanie*/

void ZapiszBit ( char b )
{
  if ( b ) bufor += maska;
  maska *= 2;
  if ( !maska ) {  /* maska == 0 po wystąpieniu nadmiaru */
                   /* czyli po zapisaniu w buforze ośmiu bitów */
    fwrite ( &bufor, 1, 1, f );
    bufor = 0;
    maska = 1;
  }
} /*ZapiszBit*/

void ZakonczPisanie ( void )
{
  if ( maska != 1 )
    fwrite ( &bufor, 1, 1, f );
  fclose ( f );
} /*ZakonczPisanie*/
```



```
void ZacznijCzytanie ( char nazwa[] )
{
  f = fopen ( nazwa, "r+");
  maska = koniec = 0;
} /*ZacznijCzytanie*/

char CzytajBit ( void )
{
  char bit;

  if ( !maska ) {
    if ( !fread ( &bufor, 1, 1, f ) )
      koniec = 1;
    else maska = 1;
  }
  bit = bufor & 1;
  bufor /= 2;
  maska *= 2;
  return bit;
} /*CzytajBit*/

void ZakonczCzytanie ( void )
{
  fclose ( f );
} /*ZakonczCzytanie*/
```

Wyjaśnienie, jak to działa: typ char jest typem liczbowym całkowitym; liczby tego typu są przechowywane w bajtach, tj. w ośmiobitowych komórkach pamięci. Zmienna `maska` kolejno przyjmuje wartości 0, 1, 2, 4, 8, 16, 32, 64, -128, a potem to samo od początku; z wyjątkiem zera każda z tych liczb jest reprezentowana przez ciąg bitów, w którym jest 7 zer i jedynka — dla kolejnych liczb ta jedynka znajduje się na kolejnej pozycji w bajcie. W każdym wywołaniu procedury `ZapiszBit` wartość zmiennej `maska` jest mnożona przez 2, przy czym wynik mnożenia $2 \cdot 64$ z powodu nadmiaru jest równy -128, a wynik mnożenia $2 \cdot -128$ jest równy 0. W ten sposób w zmiennej `maska` zawsze mamy tylko 1 bit niezerowy. Jeśli parametr `b` ma wartość niezerową (czyli wypisujemy bit o wartości 1), to wykonujemy dodawanie `bufor += maska;`, co powoduje przypisanie wartości 1 odpowiedniego bitu w buforze. Jeśli `maska` ma wartość -128, to po pomnożeniu przez 2 następuje nadmiar, wskutek czego `maska` otrzymuje wartość 0. Wskazuje to, że bufor został zapełniony (wpisaliśmy do niego 8 bitów), w związku z czym należy go wypisać do pliku i wyczyścić. Kasujemy wszystkie bity, przez przypisanie `bufor = 0;`, a następnie ustawiamy maskę tak, aby jej najmniej znaczący bit (i tylko on) miał wartość 1.

Procedura `ZakonczCzytanie` tylko zamyka plik, z którego były odczytywane bity, natomiast procedura `ZakonczPisanie` musi wykonać jeszcze jedną czynność, tj. opróżnienie bufora. Rzecz w tym, że jeśli bufor nie jest pełny, to zgromadzone w nim bity też trzeba wypisać, bez czego zakodowany ciąg byłby niekompletny. Ale to prowadzi do następnego problemu. Podczas czytania nie wiadomo, ile bitów ostatniego bajtu w pliku należy do zakodowanego ciągu, a ile z nich tylko „dopełnia" ostatni bajt. Można by sobie z tym poradzić, dostarczając osobno informację o długości (liczbie bitów) zakodowanego ciągu. Innym sposobem jest rozszerzenie zbioru $s_0, \ldots, s_{n-1}$ o jeszcze jeden element, który oznacza koniec ciągu. Elementowi temu trzeba przyporządkować odpowiedni kod, który dołączymy na końcu zakodowanego ciągu.

Zmienna `koniec` służy do zasygnalizowania, że wszystkie bajty z pliku zostały przeczytane. Po wywołaniu funkcji `CzytajBit` powinniśmy zbadać, czy zmienna ta ma wartość niezerową, co oznacza, że wartość funkcji nie opisuje bitu należącego do ciągu. Niestety, zanim zmienna `koniec` otrzyma wartość `1`, możemy odczytać nawet 7 bitów „dołożonych" do ciągu. Właśnie dlatego trzeba zastosować jeden z opisanych wyżej sposobów przekazania informacji o długości ciągu bitów.

W procedurze `CzytajBit` maska działa podobnie jak w `ZapiszBit`, ale jest używana tylko jako licznik bitów odczytanych z bufora. Odczytanie bitu z bufora polega na zbadaniu, czy wartość zmiennej `bufor` jest nieparzysta (jeśli tak, to odczytujemy bit 1, w przeciwnym razie 0; służy do tego operator `&`, który umożliwia sprawdzenie, czy dany bit (w tym przypadku najmniej znaczący) zmiennej `bufor` ma wartość `1`. Po odczytaniu bitu dzielimy wartość zmiennej `bufor` przez 2 (z odrzuceniem reszty), co „przesuwa" następny bit do odczytania na najmniej znaczącą pozycję. Po odczytaniu ośmiu bitów czytamy kolejny bajt z pliku (konieczność czytania jest sygnalizowana przez wartość 0 zmiennej `maska`).

Procedury `fwrite` i `fread` (których nagłówki są zamieszczone w pliku `stdio.h`) służą do pisania i czytania plików, przy czym dane w plikach są przechowywane w postaci binarnej (bez konwersji liczb na ciągi cyfr). Parametry tych procedur opisują wskaźnik bufora (tj. adres obszaru pamięci z danymi do zapisania lub miejsca, do którego przeczytane dane należy wpisać), wielkość elementu danych (w podanych procedurach 1 bajt), liczbę elementów (tu też 1) i wskaźnik struktury typu `FILE`, opisującej plik. Wartości zwracane przez te procedury to odpowiednio liczba zapisanych albo przeczytanych elementów (wartość 0 procedury `fread` oznacza błąd czytania lub dojście do końca pliku, co wykorzystujemy w procedurze czytania bitu).

Zadania i problemy

1. Użyj drzewa Huffmana przedstawionego na wykładzie do rozkodowania ciągu bitów `110001100010000010000010011010111 01.`
2. Opisz, jak można zbudować drzewo Huffmana przy użyciu jednej kolejki priorytetowej, zamiast dwóch kolejek, których sposób użycia był na wykładzie. Który sposób działa szybciej (należy zbadać liczby potrzebnych porównań wag w obu sposobach)? Czy sposób szybszy ma mniejszy rząd złożoności obliczeniowej?
3. Napisz w C procedurę budowania drzewa Huffmana (na podstawie tablicy prawdopodobieństw wystąpienia poszczególnych znaków; typem indeksu tej tablicy jest <u>char</u>). Skorzystaj w tym celu z gotowych procedur sortowania i obsługi kolejek (należy napisać tylko nagłówki procedur i w napisanym kodzie odpowiednio je wywoływać).
4. Napisz procedury kodowania i dekodowania przy użyciu drzewa kodu. Do wprowadzania i wyprowadzania kolejnych bitów użyj procedur przedstawionych na wykładzie.
5. Co można powiedzieć o drzewie Huffmana i o efektywności kodu, jeśli wszystkie znaki występują w ciągu z jednakowym prawdopodobieństwem?

Grafy i ich reprezentacje

Pojęcia grafu i grafu skierowaniego

Def. Niech A oznacza dowolny zbiór. Multizbiorem elementów zbioru A nazywamy dowolną funkcję $f\colon A \to \mathbb{N}$.

Def. Podmultizbiór zbioru f zdefiniowanego wyżej jest to funkcja $g\colon B \to \mathbb{N}$, taka że $B \subset A$ oraz $g(x) \le f(x)$ dla każdego $x \in B$. Możemy używać oznaczenie $g \subset f$.

Intuicyjnie, multizbiór jest to zbiór, którego elementy mogą w nim występować „więcej niż raz”. Uznajemy, że multizbiór jest tożsamy ze zbiorem A (dziedziną funkcji f), jeśli zbiorem wartości funkcji f jest $\{1\}$. Jeśli dla pewnego elementu a zbioru A jest $f(a) = k$, to możemy utworzyć pary $(a, 1), \dots, (a, k)$. Zbiór E takich par utworzonych dla wszystkich elementów dziedziny A jest pewną reprezentacją multizbioru. W takiej reprezentacji każdy element zbioru E, tj. „egzemplarz” elementu zbioru A, ma swój numer, który umożliwia jego identyfikację.

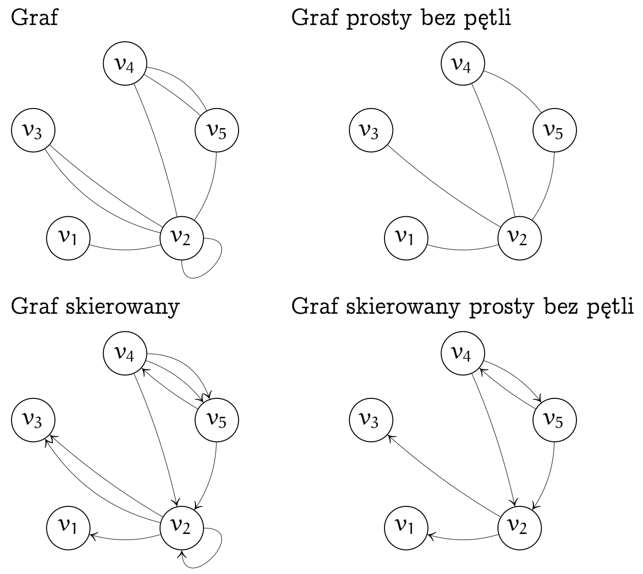
Def. Niech V oznacza dowolny zbiór skończony. Jego elementy będą nazywane wierzchołkami. Grafem nazywamy parę (V, E), gdzie E jest pewnym multizbiorem krawędzi, tj. par *nieuporządkowanych* wierzchołków.

Intuicyjnie, krawędź grafu interpretujemy jako połączenie między wierzchołkami. Oparcie definicji grafu o pojęcie multizbioru dopuszcza istnienie więcej niż jednego połączenia danych dwóch wierzchołków. Jeśli dopuszczamy tylko jedną krawędź między wierzchołkami, to mamy tzw. graf prosty. Jest on parą (V, E), złożoną ze zbioru wierzchołków V i ze *zbioru* krawędzi E.

Def. Graf skierowany jest to para (V, E) złożona ze zbioru wierzchołków V i multizbioru krawędzi E; każda krawędź jest parą *uporządkowaną* wierzchołków. Jeśli zamiast *multizbioru* mamy *zbiór* krawędzi (tj. od dowolnego wierzchołka do tego samego lub innego wierzchołka może być tylko jedna krawędź), to mamy graf skierowany prosty.

Def. Mówimy, że graf $G' = (V', E')$ jest podgrafem grafu $G = (V, E)$, jeśli $V' \subset V$ oraz $E' \subset E$. Analogicznie wprowadzamy pojęcie podgrafu grafu skierowanego.

Def. Pętlą w grafie (odpowiednio: w grafie skierowanym) nazywamy krawędź $\{v, v\}$ (odpowiednio: (v, v)).



Dla dowolnego grafu skierowanego istnieje odpowiadający mu graf nieskierowany, który ma ten sam zbiór wierzchołków. Jego krawędzie otrzymujemy „zapominając” o uporządkowaniu wierzchołków w każdej krawędzi grafu skierowanego (czyli o tzw. orientacji krawędzi). Podobnie, dla grafu (skierowanego lub nie), który nie jest prosty, jest określony odpowiedni graf prosty; otrzymujemy go odrzucając wszystkie krawędzie między dowolnymi dwoma wierzchołkami oprócz jednej. W wielu zastosowaniach mamy do czynienia z grafami prostymi bez pętli, skierowanymi lub nie.

Def. Mówimy, że krawędź $e_k = \{v_i, v_j\}$ jest incydentna z każdym z wierzchołków v_i, v_j, a także, że każdy z tych wierzchołków jest incydentny z krawędzią e_k.

Def. Stopniem wierzchołka v w grafie jest liczba incydentnych z nim krawędzi.

Def. Drogą w grafie $G = (V, E)$ nazywamy dowolny ciąg $v_0, e_1, e_2, \dots, e_m$, gdzie v_0 jest wierzchołkiem, a $e_1, \dots, e_m$ są krawędziami, przy czym istnieje ciąg wierzchołków $v_1, \dots, v_m$, taki że dla każdego $i \in \{1, \dots, m\}$ krawędź e_i jest incydentna z wierzchołkami v_{i-1} oraz v_i. Wierzchołki v_0 i v_m to odpowiednio początek i koniec drogi. Droga może składać się tylko z jednego wierzchołka, v_0, który jest wtedy jednocześnie początkiem i końcem drogi.

Definicja drogi w grafie jest oparta o ciąg krawędzi, ponieważ definicja grafu dopuszcza istnienie więcej niż jednej krawędzi między dwoma wierzchołkami. Znacznie prostsza jest definicja drogi w grafie prostym:

Def. Drogą w grafie prostym $G = (V, E)$ nazywamy dowolny ciąg wierzchołków $v_0, \ldots, v_m$, taki że $\{v_{i-1}, v_i\} \in E$ dla każdego $i \in \{1, \ldots, m\}$.

Def. Cykl w grafie jest to droga, której początek i koniec jest tym samym wierzchołkiem i w której żadna krawędź nie występuje więcej niż raz.

W szczególności cykl może przechodzić przez pewne wierzchołki wielokrotnie.

Def. Ścieżką w grafie jest droga, której odpowiada różnowartościowy ciąg wierzchołków.

Def. Długość ścieżki jest to liczba krawędzi, z których ta ścieżka się składa. Odległość wierzchołków v_1 i v_2 w grafie jest to długość najkrótszej (tj. składającej się z najmniejszej liczby krawędzi) ścieżki, której początkiem i końcem są te dwa wierzchołki.

Zauważmy, że nie każde dwa wierzchołki muszą mieć określoną odległość — warunkiem jest istnienie przynajmniej jednej ścieżki między tymi wierzchołkami.

Def. Graf G jest spójny, jeśli każde dwa jego wierzchołki są połączone przynajmniej jedną drogą.

Def. Składowa spójna grafu $G = (V, E)$ jest to spójny podgraf $G' = (V', E')$ grafu G, taki że żadna krawędź $e \in E$ incydentna z wierzchołkiem należącym do zbioru V' nie jest incydentna z żadnym wierzchołkiem należącym do $V \setminus V'$.

Zauważmy, że dowolny graf określa pewną relację równoważności w zbiorze wierzchołków. Dwa wierzchołki są w tej relacji wtedy i tylko wtedy, gdy istnieje między nimi droga. Istotnie, łatwo jest sprawdzić, że tak określona relacja jest zwrotna (z każdego wierzchołka prowadzi do tego samego wierzchołka ścieżka o długości 0), symetryczna i przechodnia. Zbiór wierzchołków dowolnej składowej spójnej grafu jest klasą abstrakcji tej relacji równoważności.

Odpowiedniki wprowadzonych wyżej pojęć dla grafów skierowanych znacznie się różnią. O ile drogę w zwykłym grafie można odwrócić (tj. wypisać wierzchołki i krawędzie w odwrotnej kolejności, otrzymując drogę należącą do tego samego grafu), to w grafie skierowanym nie jest to zwykle możliwe, jako że z tego, że

$(v_i, v_j) \in E$ *nie wynika*, że $(v_j, v_i) \in E$. Zatem możliwość dojścia z pewnego wierzchołka do innego wzdłuż krawędzi nie oznacza istnienia możliwości powrotu. Pojęcie incydencji krawędzi z wierzchołkiem jest nieprecyzyjne, bo nie określa, czy jest to krawędź wychodząca z, czy krawędź wchodząca do rozpatrywanego wierzchołka. Również pojęcie stopnia, który jest jedną liczbą, nie daje pełnej informacji o otoczeniu wierzchołka grafu skierowanego. Taką informacją powinny być dwie liczby — krawędzi wychodzących z i krawędzi wchodzących do tego wierzchołka.

Def. Drogą w grafie skierowanym $G = (V, E)$ nazywamy dowolny ciąg $v_0, e_1, \ldots, e_m$, taki że istnieje ciąg wierzchołków $v_1, \ldots, v_m$, taki że dla $i \in \{1, \ldots, m\}$ krawędź $e_i = (v_{i-1}, v_i)$. Dla grafu skierowanego *prostego* możemy zdefiniować drogę jako ciąg wierzchołków $v_0, \ldots, v_m$, taki że $(v_{i-1}, v_i) \in E$ dla $i \in \{1, \ldots, m\}$. Wierzchołki v_0 i v_m to początek i koniec drogi.

Def. Cykl w grafie skierowanym jest to droga w grafie skierowanym, której początek jest identyczny z końcem i w której żadna krawędź nie występuje więcej niż raz.

Def. Ścieżka w grafie skierowanym jest to droga w grafie skierowanym, której odpowiada różnowartościowy ciąg wierzchołków.

Długość ścieżki w grafie skierowanym możemy określić tak samo jak w grafie nieskierowanym, ale pojęcie odległości wierzchołków się nie przenosi. W spójnym grafie nieskierowanym funkcja, która parze wierzchołków przyporządkowuje ich odległość, jest metryką (w szczególności jest symetryczna), natomiast w grafie skierowanym długość najkrótszej ścieżki od wierzchołka v_i do v_j może być inna niż długość najkrótszej ścieżki od v_j do v_i (na przykład, od poniedziałku do soboty jest pięć dni, a od soboty do poniedziałku tylko dwa).

Def. Graf skierowany jest silnie spójny, jeśli dla dowolnych dwóch jego wierzchołków, v_i, v_j, istnieje ścieżka, której początkiem jest v_i, a końcem v_j.

Z definicji *nie wynika*, że w skierowanym grafie silnie spójnym każde dwa wierzchołki należą do pewnego cyklu (bo ścieżki łączące wierzchołki mogą mieć wspólną krawędź).

Def. Podgraf $G' = (V', E')$ grafu skierowanego $G = (V, E)$ nazywamy składową silnie spójną, jeśli graf G' jest silnie spójny, oraz dla dowolnego wierzchołka $v \in V \setminus V'$ nie istnieje w grafie G cykl, który przechodzi przez wierzchołek v i przez dowolny wierzchołek podgrafu G'.

Łatwo jest zauważyć, że każdy wierzchołek grafu skierowanego należy do pewnej składowej silnie spójnej. Dlatego dowolny graf skierowany również określa pewną relację równoważności w zbiorze wierzchołków. W szczególności graf, który nie ma cykli (tzw. bezcyklowy graf skierowany, DAG od ang. *directed acyclic graph*) określa relację minimalną — nie są w niej żadne dwa różne wierzchołki — zaś graf silnie spójny określa relację pełną — są w niej każde dwa wierzchołki.

Lasy i drzewa

Def. Lasem nazywamy graf (nieskierowany), który nie ma cykli.

Def. Drzewo jest to las, który jest grafem spójnym.

Twierdzenie. Graf $G = (V, E)$ jest drzewem wtedy i tylko wtedy, gdy ma dowolną z wymienionych niżej własności:

1. Każde dwa wierzchołki są połączone dokładnie jedną ścieżką.
2. Graf jest spójny, ale usunięcie dowolnej krawędzi powoduje powstanie grafu niespójnego.
3. Graf jest bezcyklowy, ale dołączenie dowolnej krawędzi powoduje powstanie cyklu.
4. Graf jest spójny i $|E| = |V| - 1$ (symbol $|A|$ oznacza liczbę elementów zbioru A).
5. Graf jest bezcyklowy i $|E| = |V| - 1$.

Dowód. 1. Jeśli graf G jest drzewem, to jest spójny, a zatem każde dwa wierzchołki są połączone pewną drogą. Gdyby istniały dwie różne ścieżki między pewnymi dwoma wierzchołkami, to w ogólności mogłyby mieć na początku i na końcu pewną liczbę wspólnych krawędzi. Po odrzuceniu tych krawędzi mielibyśmy dwie ścieżki między pewnymi wierzchołkami v_i i v_j, których początkowe i końcowe krawędzie są różne. Niech v_k oznacza wierzchołek różny od v_i, należący do obu ścieżek i położony najbliżej v_i na pierwszej ścieżce (taki wierzchołek istnieje, w braku innego jest nim v_j). Wtedy z krawędzi należących do dwóch różnych ścieżek od v_i do v_k można zbudować cykl.

2. Jeśli usunięcie krawędzi $\{v_i, v_j\}$ z grafu spójnego nie psuje spójności grafu, to istnieje ścieżka między wierzchołkami v_i i v_j złożona z innych krawędzi. Po dołączeniu usuniętej krawędzi do tej ścieżki otrzymujemy cykl, a zatem graf nie jest drzewem.

3., 4., 5. — dowody pozostawiam na ćwiczenia. □

Wyróżniając pewien wierzchołek drzewa, otrzymujemy tzw. drzewo ukorzenione. Ponieważ z każdego wierzchołka (w tym wierzchołka wyróżnionego, czyli korzenia) do dowolnego innego jest tylko jedna ścieżka, więc naturalne jest wprowadzenie orientacji krawędzi takiego drzewa, zgodnie z którą korzeń lub wierzchołek bliższy korzenia jest w krawędzi pierwszy. Tak więc drzewa ukorzenione są w istocie grafami skierowanymi.

Reprezentacje grafów

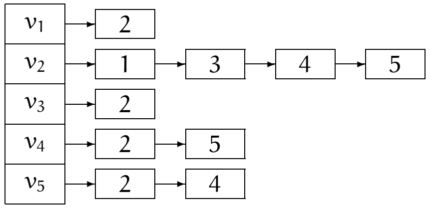
Struktury danych używane do reprezentowania grafów muszą być dostosowane do algorytmu przetwarzania danych, zaś algorytm jest dobierany do konkretnego zadania. Sposób reprezentowania drzew ukorzenionych (za pomocą zmiennych dynamicznych i wskaźników) nadaje się także do reprezentowania bezcyklowych grafów skierowanych. Teraz poznamy dwa sposoby najczęściej używane do reprezentowania grafów w przypadku ogólnym.

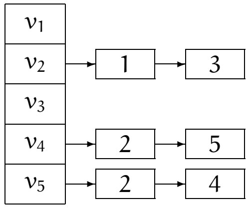
Listy sąsiedztwa. Reprezentacja grafu $G = (V, E)$ składa się z tablicy, której każdy element odpowiada jednemu wierzchołkowi. Elementy te mogą być strukturami zawierającymi różne dane związane z wierzchołkami (mogą też nie zawierać żadnych danych). Każdy element zawiera wskaźnik początku listy sąsiedztwa. Jest to lista liniowa, której elementy odpowiadają krawędziom incydentnym z danym wierzchołkiem (element listy zawiera numer drugiego wierzchołka grafu, ewentualnie dane dodatkowe, oraz wskaźnik następnego elementu listy).

Listy sąsiedztwa w naturalny sposób mogą reprezentować grafy skierowane. Element listy sąsiedztwa odpowiada krawędzi wychodzącej z wierzchołka, w którego liście element ten się znajduje, i zawiera numer wierzchołka, do którego krawędź ta wchodzi. W przypadku grafu nieskierowanego każdej krawędzi odpowiadają dwa elementy list sąsiedztwa — po jednym w listach obu wierzchołków incydentnych z tą krawędzią.

Reprezentacja grafu za pomocą list sąsiedztwa zajmuje w pamięci $O(|V| + |E|)$ jednostek (np. słów). Rysunki przedstawiają listy sąsiedztwa dla grafu prostego i dla grafu skierowanego prostego, narysowanych na stronie 16.2.

Macierze sąsiedztwa. Ten sam graf możemy reprezentować za pomocą kwadratowej tablicy o wymiarach $|V| \times |V|$. Elementy tablicy są liczbami: $a_{ij} = 0$ jeśli nie ma w grafie krawędzi między wierzchołkami v_i i v_j, oraz 1, jeśli istnieje krawędź incydentna z tymi wierzchołkami. Macierz sąsiedztwa grafu nieskierowanego jest symetryczna. Dla grafu skierowanego, w którym jest krawędź (v_i, v_j), ale nie ma krawędzi (v_j, v_i), mamy $a_{ij} = 1$, $a_{ji} = 0$.





Zauważmy, że łatwo jest użyć macierzy sąsiedztwa do reprezentowania grafu, który nie jest prosty; w tym przypadku element a_{ij} tablicy ma wartość, która jest liczbą krawędzi między wierzchołkami v_i i v_j. Dla grafu skierowanego wartość elementu a_{ij} tablicy jest liczbą krawędzi o początku v_i i końcu v_j. Macierze sąsiedztwa dla grafów i grafów skierowanych na stronie 16.2 są następujące:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 2 & 2 & 1 & 1 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 2 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Uwaga: Jeśli pewna krawędź w grafie nieskierowanym jest pętlą, to możemy w określeniu stopnia wierzchołka liczyć ją pojedynczo lub podwójnie. To jest kwestia umowy, zależnej od zastosowania rozpatrywanego grafu. Umowa ta wpływa na współczynniki diagonalne macierzy sąsiedztwa grafu z pętlami. W pierwszej z pokazanych wyżej macierzy sąsiedztwa pętla $\{v_2, v_2\}$ została uwzględniona podwójnie.

Ponieważ macierz sąsiedztwa dla grafu nieskierowanego jest symetryczna, więc jeśli $n = |V|$, to możemy pamiętać tylko $\frac{1}{2}n(n+1)$ współczynników a_{ij}, zamiast n^2. Bardzo często mamy do czynienia z grafami, których liczba krawędzi jest znacznie mniejsza niż n^2 (np. jest proporcjonalna do n — jest tak między innymi wtedy, gdy stopień żadnego wierzchołka nie jest większy niż ustalona liczba). W takim przypadku reprezentacja grafu w postaci list sąsiedztwa jest znacznie bardziej oszczędna.

Zauważmy, że sprawdzenie, czy istnieje krawędź incydentna z dowolnymi dwoma wierzchołkami grafu przy użyciu tablicy sąsiedztwa jest wykonywane w czasie stałym (wystarczy tylko odczytać jeden element tablicy), natomiast znalezienie tej informacji w liście sąsiedztwa może zająć czas proporcjonalny do stopnia jednego lub drugiego wierzchołka. Z drugiej strony, znalezienie wszystkich krawędzi incydentnych z danym wierzchołkiem, w liście sąsiedztwa zajmuje czas proporcjonalny do stopnia wierzchołka, natomiast w tablicy sąsiedztwa trzeba przejrzeć cały wiersz (czyli wykonać $O(|V|)$ operacji).

Algorytmy grafowe

Przeszukiwanie grafów w głąb

Często spotykane zadanie pomocnicze w rozwiązywaniu zadań grafowych polega na znalezieniu wszystkich wierzchołków należących do tej samej spójnej składowej grafu, co pewien wierzchołek dany, i wykonanie pewnego obliczenia dla wszystkich tych wierzchołków. Istnieją dwa podstawowe podejścia do tego zadania. Różnią się one kolejnością wierzchołków grafu, na których wykonywane są potrzebne operacje.

Sposób pierwszy to tzw. przeszukiwanie grafu w głąb (ang. *depth-first search*, DFS). Każdy wierzchołek ma atrybut, któremu nadamy nazwę „`kolor`". Na początku przypisujemy temu atrybutowi we wszystkich wierzchołkach wartość `biały`. Następnie przygotowujemy stos (początkowo pusty) i wstawiamy nań wierzchołek (możemy wstawiać na stos numer wierzchołka, tj. indeks do tablicy z listami sąsiedztwa), od którego zaczynamy przeszukiwanie.

Następnie w pętli wykonujemy następujące czynności: `kolor` elementu na wierzchołku stosu zmieniamy na `szary` i wykonujemy (jeśli trzeba) wszelkie czynności związane ze wstępnym przetwarzaniem tego wierzchołka (który pozostaje na stosie). Następnie, dopóki wierzchołek ten ma białego sąsiada, wstawiamy tego sąsiada na stos i nim się zajmujemy. Po wykonaniu potrzebnych czynności na wszystkich białych sąsiadach wykonujemy (jeśli trzeba) końcowe czynności z danym wierzchołkiem, zdejmujemy go ze stosu i zmieniamy jego kolor na `czarny`. Algorytm kończy działanie z chwilą, gdy stos jest pusty. Wtedy wszystkie wierzchołki, do których można dojść z pierwszego przetwarzanego wierzchołka są czarne.

Dlaczego ten sposób przeszukiwania grafu nazywa się „w głąb"? Otóż zawsze, jeśli w grafie jest jakiś biały sąsiad wierzchołka danego, to jest on umieszczany na stosie, a następnie jeden z jego białych sąsiadów itd. W ten sposób na stosie znajdują się wierzchołki tworzące możliwie *najdłuższą* ścieżkę (tzn. ścieżka jest wydłużana do chwili, gdy staje się to niemożliwe — to *nie oznacza* znalezienia najdłuższej ścieżki w grafie). Kolejny sąsiedni wierzchołek danego wierzchołka będzie nadal biały tylko wtedy, gdy nie da się do niego dojść z żadnego wierzchołka, który był na stosie wcześniej.

Algorytm DFS w pewnych zastosowaniach można uprościć, nie zmieniając sedna. Zamiast koloru wierzchołek może mieć atrybut `odwiedzony`, o początkowej wartości `0`. Umieszczamy na stosie jeden wierzchołek. Następnie w pętli zdejmujemy wierzchołek v ze stosu i jeśli jest nieodwiedzony, to go przetwarzamy (cokolwiek jest do zrobienia), przypisujemy v`.odwiedzony = 1`, oraz wstawiamy na stos wszystkich nieodwiedzonych sąsiadów wierzchołka v. Czynności te powtarzamy do opróżnienia stosu.

Jeśli graf jest drzewem, to nie ma konieczności zaznaczania odwiedzonych wierzchołków, ponieważ od wierzchołka przetwarzanego najpierw prowadzi do każdego innego tylko jedna ścieżka. Wystarczy wstawiać na stos tylko wierzchołki bardziej odległe od korzenia (wierzchołka, od którego zaczynamy) niż wierzchołek właśnie przetworzony. Zauważmy, że przeszukiwanie drzewa BST, które ma na celu wypisanie wszystkich kluczy w kolejności uporządkowanej (rosnąco lub malejąco, zobacz str. 13.7) jest właśnie przeszukiwaniem w głąb. Stos w tym przypadku jest zorganizowanym przez kompilator języka C (lub innego) stosem rekordów aktywacji wywołanych procedur (w tym procedury rekurencyjnej przetwarzania wierzchołków drzewa).

Przeszukiwanie grafów wszerz

Drugi podstawowy sposób przeszukiwania grafu polega na użyciu kolejki zamiast stosu. Ta metoda nazywa się <u>przeszukiwaniem wszerz</u> (ang. *breadth-first search*, BFS). Z kolejki wyjmujemy najpierw elementy wstawione najdawniej. W przeszukiwaniu grafu oznacza to, że wierzchołki, do których od pierwszego wierzchołka prowadzą ścieżki o długości n będą odwiedzone dopiero po odwiedzeniu wszystkich wierzchołków, do których prowadzą ścieżki o długości $n-1$.

Przeszukiwanie grafu wszerz umożliwia zatem wyznaczenie *najkrótszych* (tj. złożonych z najmniejszej liczby krawędzi) ścieżek z pewnego wierzchołka do wszystkich pozostałych wierzchołków.

Złożoność obliczeniowa obu algorytmów przeszukiwania grafu, tj. w głąb i wszerz, przy zastosowaniu list sąsiedztwa, ma taki sam rząd, $O(|V|+|E|)$. Zależnie od budowy grafu różne są potrzebne pojemności stosu i kolejki.

Zadania i problemy

1. Udowodnij twierdzenie Eulera „o uściskach dłoni”: suma stopni wszystkich wierzchołków w grafie jest liczbą parzystą (pętle liczymy podwójnie!).
2. Udowodnij, że własności grafu podane w punktach 3, 4, 5 twierdzenia na str. 16.5 są równoważne podanej definicji drzewa.
3. Napisz w C procedurę przeszukiwania grafu w głąb, która korzysta z procedur obsługi stosu (`InitStack`, `Push`, `Pop` i `StackEmpty`, takich jak w wykładzie 6). Należy przedtem zadeklarować odpowiednie typy do reprezentowania grafu w postaci list sąsiedztwa i typ elementu odkładanego na stosie.
4. Napisz procedurę rekurencyjną przeszukiwania grafu w głąb, która nie korzysta ze stosu zrealizowanego za pomocą procedur.
5. Napisz funkcję typu <u>`char`</u>, której wartość jest różna od zera wtedy i tylko wtedy, gdy graf dany jako parametr jest spójny.
6. Napisz funkcję, której wartością jest liczba składowych spójnych grafu danego jako parametr. Jaka jest złożoność obliczeniowa algorytmu realizowanego przez tę funkcję?
7. Dla grafów reprezentowanych przez listy sąsiedztwa na str. 16.7 podaj kolejność, w jakiej zostaną odwiedzone wierzchołki przez procedury przeszukiwania w głąb i wszerz.
8. <u>Graf płaski</u> jest to graf narysowany na płaszczyźnie w taki sposób, że wierzchołki są punktami, a krawędzie — łączącymi je ciągłymi krzywymi, które nie mają poza wierzchołkami punktów wspólnych. <u>Ścianą</u> grafu płaskiego nazywamy spójny obszar ograniczony przez krawędzie, wewnątrz którego nie ma żadnych innych krawędzi.

   Udowodnij <u>wzór Eulera</u> dla spójnych grafów płaskich: $|V|-|E|+|F|=2$, gdzie $|V|$ oznacza liczbę wierzchołków, $|E|$ — liczbę krawędzi, a $|F|$ — liczbę ścian.
9. Znajdź wzór opisujący maksymalną liczbę krawędzi w płaskim grafie prostym, który ma n wierzchołków. Skorzystaj w tym celu ze wzoru Eulera.

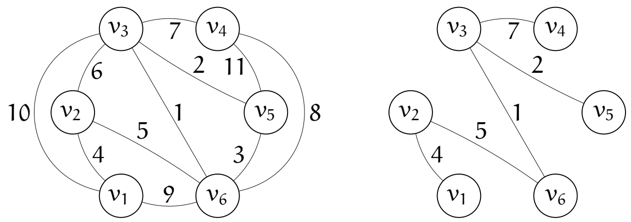
Minimalne drzewa rozpinające

Sformułowanie problemu

Krawędziom grafu prostego $G = (V, E)$ (nieskierowanego) przyporządkujemy liczby rzeczywiste, tak zwane wagi. Wagą grafu możemy wtedy nazwać sumę wag wszystkich jego krawędzi. Zależnie od zastosowania, wagi krawędzi mogą określać pewne koszty (na przykład czasy potrzebne do przebycia krawędzi, wydatki na benzynę, albo też długość kabla potrzebną do połączenia wierzchołków).

Przypuśćmy, że graf G jest spójny. Wtedy istnieje co najmniej jeden podgraf T grafu G, który jest drzewem o tym samym zbiorze wierzchołków; zatem $T = (V, F)$, gdzie $F \subset E$. Każde takie drzewo nazywamy drzewem rozpinającym graf G.

Mając drzewo rozpinające graf G jesteśmy w stanie znaleźć drogę (jednoznacznie określoną) między dowolnymi wierzchołkami grafu. Interpretacja wag jako kosztów (np. wydatków na wybudowanie dróg umożliwiających przedostanie się z każdego wierzchołka do każdego innego; krawędzie grafu G reprezentują wszystkie drogi możliwe do wybudowania) uzasadnia zainteresowanie znalezieniem minimalnego drzewa rozpinającego (ang. *minimal spanning tree*, MST), czyli drzewa rozpinającego graf G, którego waga jest najmniejsza.



Graf z wagami krawędzi i jego minimalne drzewo rozpinające

Jest oczywiste, że dla dowolnego grafu spójnego G, jego minimalne drzewo rozpinające istnieje. Liczba krawędzi grafu G jest większa lub równa $n - 1$, gdzie $n = |V|$ i skończona (dla grafu prostego nie większa niż $\frac{1}{2}n(n-1)$), a zatem istnieje skończenie wiele podzbiorów $n - 1$-elementowych zbioru krawędzi, w tym skończenie wiele podzbiorów określających drzewa. Któreś z tych drzew musi mieć zatem wagę najmniejszą. Jest również oczywiste, że na ogół drzew rozpinających jest za dużo, aby algorytm, który po kolei znajduje wszystkie drzewa rozpinające i oblicza ich wagi, był cokolwiek wart w praktyce.

Dalej, jest oczywiste, że jeśli wszystkie krawędzie mają jednakowe wagi, to wszystkie drzewa rozpinające są minimalne. Wniosek stąd jest taki, że zadanie znalezienia minimalnego drzewa rozpinającego w ogólności może mieć więcej niż jedno rozwiązanie.

Twierdzenie. Niech $G = (V, E)$ będzie grafem spójnym i niech V' będzie podzbiorem V, takim że $V' \neq \emptyset$ oraz $V' \neq V$. Niech podzbiór $E' \subset E$ składa się z tych krawędzi grafu G, których jeden wierzchołek należy do V', a drugi do $V \setminus V'$. Wtedy istnieje minimalne drzewo rozpinające graf G, takie że jedną z jego krawędzi jest krawędź ze zbioru E' o najmniejszej wadze.

Dowód. Zbiór E' jest niepusty, ponieważ w przeciwnym razie graf G byłby niespójny i z tego samego powodu każde drzewo rozpinające musi mieć co najmniej jedną krawędź z E'. Jeśli zbiór E' jest jednoelementowy, to teza jest oczywista (wtedy każde drzewo rozpinające graf G, w tym minimalne, musi mieć krawędź należącą do E'). Niech zatem zbiór E' zawiera co najmniej dwa elementy.

Rozważmy drzewo T rozpinające graf G, do którego nie należy krawędź $e \in E'$ o minimalnej wadze w E' (natomiast należą inne krawędzie z E'). Dołączenie krawędzi e do drzewa T, zgodnie z twierdzeniem z poprzedniego wykładu, powoduje powstanie cyklu. Ponieważ jeden koniec krawędzi e należy do V', a drugi do $V \setminus V'$, więc jeszcze co najmniej jedna krawędź tego cyklu, f, musi należeć do E'. Jej usunięcie likwiduje cykl, dając w wyniku drzewo T', którego waga jest mniejsza od wagi drzewa T o różnicę wag krawędzi f i e. Zatem, drzewo T, do którego nie należy żadna z krawędzi e o minimalnej wadze w E', nie jest minimalnym drzewem rozpinającym graf G. □

Udowodnione wyżej twierdzenie jest podstawą do opracowania algorytmów, które znajdują minimalne drzewo rozpinające (jedno z takich drzew, jeśli jest ich więcej). Są to tzw. *algorytmy zachłanne*, czyli takie, które w każdym kroku wybierają możliwość, która wydaje się być najlepsza w danej chwili (w budowaniu MST algorytm dołącza krawędź o najmniejszej wadze z pewnego zbioru krawędzi, z którego może wybierać). Dzięki twierdzeniu, takie postępowanie daje poprawny wynik. Są jednak zadania, w których najprostsza strategia zachłanna nie musi prowadzić do dobrego wyniku (są też takie zadania, których nie można rozwiązać za pomocą żadnej strategii zachłannej). Rozważmy np. zadanie znajdowania ścieżki o najmniejszej wadze między dwoma ustalonymi wierzchołkami. Pierwsza krawędź tej ścieżki nie musi mieć najmniejszej wagi wśród krawędzi wychodzących z wierzchołka początkowego. Co więcej, ścieżka o najmniejszej wadze między tymi wierzchołkami nie musi zawierać krawędzi o najmniejszej wadze wśród krawędzi należących do tych ścieżek.

Algorytm Prima

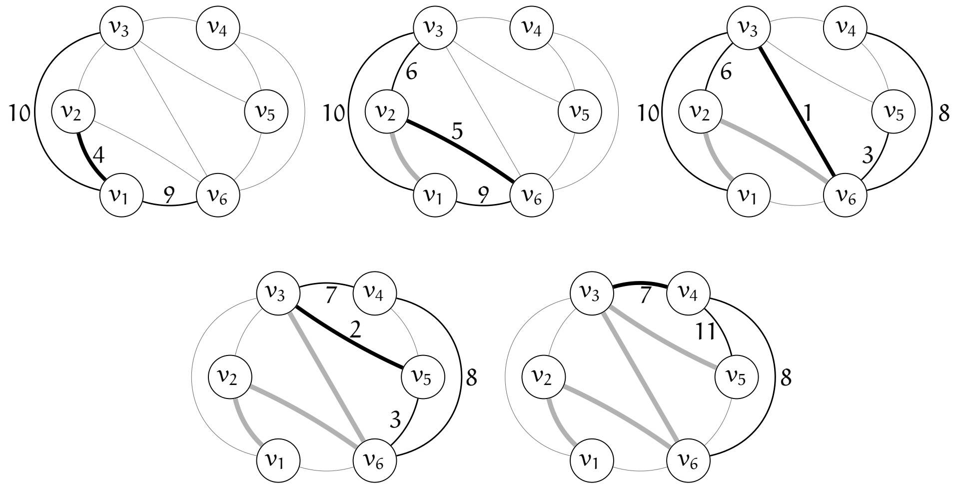
Algorytm Prima buduje minimalne drzewo rozpinające, dołączając w każdym kroku jeden wierzchołek i incydentną z nim krawędź. Początkowo przyjmujemy jednoelementowy zbiór wierzchołków V' (wierzchołek w tym zbiorze dowolny być może). Rozpatrujemy zbiór E_1 krawędzi łączących ten wierzchołek z wszystkimi pozostałymi. Zgodnie ze wskazówką, jaką daje twierdzenie, wybieramy krawędź o najmniejszej wadze i dołączamy ją oraz jej drugi wierzchołek do budowanego drzewa. W każdym następnym (k-tym) kroku rozpatrujemy zbiór E_k krawędzi łączących jeden z wierzchołków dołączonych do budowanego drzewa wcześniej, z pozostałymi wierzchołkami grafu G.

Zbiory E_k, z których możemy wybierać krawędzie w kolejnych krokach, możemy określać indukcyjnie. Zbiór E_1 jest zbiorem wszystkich krawędzi wychodzących z wierzchołka, od którego zaczynamy budowanie drzewa (dla ustalenia uwagi, niech to będzie wierzchołek $v_{i_1} = v_1$). Po dołączeniu wierzchołka v_{i_k} w kroku k-tym możemy otrzymać zbiór E_{k+1} przez usunięcie z E_k wszystkich krawędzi incydentnych z v_{i_k} oraz dołączenie wszystkich krawędzi incydentnych z v_{i_k} oraz z wierzchołkami, które jeszcze nie zostały dołączone.

Słowny opis algorytmu Prima może być taki:

1. Przypisz $V' := \{v_1\}$, $F := \emptyset$, oraz $E' :=$ zbiór wszystkich krawędzi incydentnych z v_1 i z pozostałymi wierzchołkami.
2. Kolejne kroki powtórz $|V| - 1$ razy:
3. Wybierz krawędź $e \in E'$ o najmniejszej wadze.
4. Dołącz krawędź e do F.
5. Dołącz do V' wierzchołek v_{i_k} incydentny z krawędzią e (drugi wierzchołek incydentny z tą krawędzią już jest w V').
6. Usuń z E' wszystkie krawędzie incydentne z v_{i_k} (w tym e).
7. Dołącz do E' wszystkie krawędzie incydentne z v_{i_k}, których drugi wierzchołek nie należy do V'.

Ilustracja działania algorytmu Prima jest na rysunku. Drzewo rozpinające graf o sześciu wierzchołkach ma pięć krawędzi. Pierwszą krawędzią wyznaczaną przez algorytm, incydentną z wierzchołkiem v_1, jest $\{v_1, v_2\}$ o wadze 4. Druga krawędź jest wybierana spośród wszystkich krawędzi incydentnych z v_1 i v_2 *z wyjątkiem* $\{v_1, v_2\}$; to jest krawędź $\{v_2, v_6\}$ o wadze 5. Następna krawędź jest wybierana spośród krawędzi incydentnych z wierzchołkami v_1, v_2 i v_6, z wyjątkiem krawędzi łączących te wierzchołki, itd.



Jak widać, w algorytmie Prima trzeba reprezentować pewne zbiory wierzchołków i krawędzi. Implementacja algorytmu wymaga określenia reprezentacji tych zbiorów, a ta z kolei zależy od działań, które trzeba na nich wykonywać. Zauważmy, że zbiór V' możemy reprezentować w ten sposób, że w tablicy wierzchołków grafu G elementy mogą mieć pole do zaznaczania: początkowa wartość 0 jest zmieniana na 1 w chwili dołączenia wierzchołka do zbioru V'. Elementy V' rozpoznajemy zatem po wartości tego pola; można to uznać za szczególny przypadek kolorowania wierzchołków (podobnego do tego w algorytmach DFS i BFS) — początkowo wszystkie wierzchołki są białe, dołączając wierzchołek do drzewa, malujemy go na czarno.

Zbiór F krawędzi dołączonych do drzewa może być zwykłą tablicą, stosem lub kolejką, ponieważ do tego zbioru tylko dołączamy kolejne krawędzie pojedynczo. Natomiast reprezentacja zbioru E', którego elementami też są krawędzie grafu, ma kluczowe znaczenie dla złożoności obliczeniowej algorytmu.

W kroku 3 należy znaleźć w E' krawędź o najmniejszej wadze. Naturalne jest zatem potraktowanie reprezentacji zbioru E' jako kolejki priorytetowej. Pewne utrudnienie stanowi fakt, że w kroku 6 musimy z E' usunąć wszystkie krawędzie incydentne z pewnym wierzchołkiem, ale krawędzie te nie muszą mieć największych priorytetów — obok zwykłych działań na kolejce priorytetowej jest jeszcze potrzebna operacja usuwania wskazanych elementów. Zbadamy trzy rozwiązania dla reprezentacji zbioru E'.

Pierwsze rozwiązanie polega na użyciu tablicy nieuporządkowanej krawędzi. Wyszukanie krawędzi o najmniejszej wadze polega zatem na przeszukaniu całej tablicy, co odbywa się w czasie proporcjonalnym do liczby elementów przechowywanych w niej. Usunięcie wszystkich krawędzi incydentnych z danym wierzchołkiem może być również wykonane w czasie proporcjonalnym do liczby elementów w tablicy.

Zbadajmy rząd złożoności algorytmu, przy założeniu, że graf G o n wierzchołkach jest prosty, bez pętli i pełny (tj. każde dwa różne wierzchołki są połączone jedną krawędzią). Graf ma zatem $\frac{1}{2}n(n-1)$ krawędzi. Dla takiego grafu w zbiorze E_1 jest $n-1$ krawędzi. Zbiór E_{k+1} otrzymujemy usuwając k krawędzi ze zbioru E_k i dołączając $n-k-1$ nowych krawędzi. Rozwiązując równanie różnicowe

$$\begin{aligned} |E_1| &= n-1, \\ |E_{k+1}| &= |E_k| + n - 2k - 1, \end{aligned}$$

otrzymujemy dla $k = 1, \dots, n$

$$|E_k| = (n-k)k.$$

Koszt wykonywania k-tego kroku jest w przybliżeniu równy $c(n-k)k$ (gdzie c jest pewną stałą), zatem całkowity koszt algorytmu

$$T(n) \approx \sum_{k=1}^{n-1} c(n-k)k = O(n^3).$$

Druga możliwość, jaką zbadamy, polega na użyciu kopca do reprezentowania zbioru E'. Problemem, który trzeba rozwiązać jest znalezienie sposobu wyznaczania elementów kopca, które należy usunąć (*nie wolno* przeglądać wszystkich elementów w kopcu, w celu znalezienia krawędzi incydentnych z danym wierzchołkiem!). Jeśli korzystamy z reprezentacji grafu przy użyciu list sąsiedztwa, to w elementach list (elementy te reprezentują krawędzie) wprowadzimy pole, którego wartość jest indeksem pozycji odpowiedniego elementu w kopcu. W kopcu będziemy przechowywać wskaźniki do elementów list. Aby porównać priorytety elementów kopca, używamy wskaźników do „wyciągnięcia” z elementów list odpowiednich wag. Przestawienie elementów w kopcu polega na przestawieniu wskaźników w tablicy, uzupełnione zamienieniem wartości pól określających pozycję w kopcu w odpowiednich elementach list. W ten sposób operacje porównania i przestawiania elementów w kopcu nadal odbywają się w czasie stałym. Aby usunąć wszystkie krawędzie incydentne z danym wierzchołkiem, wystarczy przejrzeć listę sąsiedztwa tego wierzchołka.

Zarówno koszt wstawiania elementu do kopca, jak i koszt usuwania (dowolnego) elementu, mogą być oszacowane z góry przez wysokość kopca, która jest (z dokładnością do zaokrąglenia) logarytmem liczby przechowywanych w nim elementów. W kroku k-tym mamy usunąć k krawędzi (wykonujemy więc około $k \log\bigl((n-k)k\bigr)$ operacji dominujących, którymi są porównania wag krawędzi) oraz wstawić $(n-k-1)$ krawędzi, kosztem ok. $(n-k-1)\log\bigl((n-k)k\bigr)$ operacji. Liczbę operacji wykonanych przez cały algorytm możemy oszacować następująco:

$$T(n) \approx \sum_{k=1}^{n-1} (n-1)\log\bigl((n-k)k\bigr) \le \sum_{k=1}^{n-1} (n-1)\log\bigl(\frac{1}{4}n^2\bigr) = O(n^2 \log n).$$

Dla dowolnego grafu (niekoniecznie pełnego) każdą krawędź możemy tylko raz wstawić do kopca, po czym też tylko raz ją usuwamy. Rząd złożoności obliczeniowej algorytmu korzystającego z kopca dla grafu o m krawędziach możemy w ogólnym przypadku oszacować jako $O(m \log m)$.

W rozwiązaniu trzecim wykorzystamy dwie tablice, których długość odpowiada wierzchołkom, nazwiemy je `c` i `d`. Jeśli V' oznacza zbiór wierzchołków, które zostały już dołączone do drzewa oraz $v_i \notin V'$, to wartość zmiennej `c[i]` jest numerem wierzchołka w zbiorze V', do którego z wierzchołka v_i prowadzi najkrótsza znaleziona ścieżka (ścieżka ta oczywiście składa się z co najmniej jednej krawędzi). Jeśli ścieżka ta składa się z jednej krawędzi, to wartość zmiennej `d[i]` jest równa wadze tej krawędzi. W przeciwnym razie, albo jeśli $v_i \in V'$, to `d[i]=`∞.

W kroku pierwszym nadajemy wartości początkowe elementom tablicy w sposób, który odpowiada $V' = \{v_1\}$, $E' = \emptyset$:

```
d[1] = ∞;    /* to jest oczywiscie pseudokod, nie język C! */
for ( i = 2; i <= n; i++ ) {
  c[i] = 1;
  d[i] = długość krawędzi (vi,v1), jeśli taka istnieje, albo ∞;
}
```

Następnie w pętli, którą wykonujemy $n-1$ razy, znajdujemy wierzchołek $v_j \notin V'$, z którego do jednego z wierzchołków w V' prowadzi krawędź o najmniejszej wadze:

```
V' = {v1};   F = ∅;
for ( i = 1; i < n; i++ ) {
  jmin = 2;  min = d[2];
  for ( j = 3; j <= n; j++ )
    if ( d[j] < min ) { jmin = j;  min = d[j]; }
```

Teraz znaleziony wierzchołek razem z odpowiednią krawędzią należy dołączyć do drzewa, a następnie uaktualnić tablice.

```
    F = F ∪ {(v_jmin, v_c[jmin])};
    V′ = V′ ∪ {v_jmin};
    d[jmin] = ∞;
    for ( j = 2; j <= n; j++ )
      if ( v_j ∉ V′, (v_j, v_jmin) ∈ E oraz waga((v_j, v_jmin)) < d[j] ) {
        c[j] = jmin;
        d[j] = waga((v_j, v_jmin));
      }
  }
```

Jeśli korzystamy z tablicy sąsiedztwa i wagi krawędzi trzymamy również w kwadratowej tablicy, to stwierdzenie, czy istnieje krawędź i znalezienie jej wagi odbywa się w czasie stałym. Ponieważ ten algorytm jest zrealizowany w postaci pętli zagnieżdżonych do głębokości 2 i każda pętla jest wykonywana w przybliżeniu n razy, więc złożoność obliczeniowa tej wersji algorytmu jest równa $O(n^2)$.

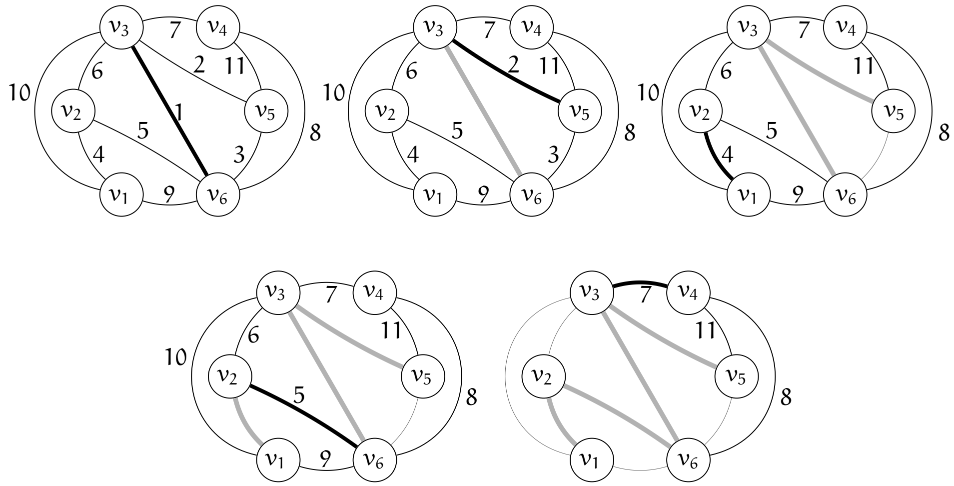
Pora na podsumowanie. Pierwszy sposób zaimplementowania algorytmu jest łatwy, ale zawsze działa najwolniej. Jeśli graf o n wierzchołkach ma niewiele krawędzi (często mamy do czynienia z grafami, których wszystkie wierzchołki mają stopień ograniczony przez niewielką stałą, a wtedy liczba krawędzi jest rzędu n), to najmniejszy rząd złożoności obliczeniowej ma implementacja z kopcem. Jeśli liczba krawędzi jest rzędu n^2, to lepiej jest użyć trzeciego sposobu implementacji. Zauważmy, że w tym przypadku również reprezentowanie grafu za pomocą tablicy sąsiedztwa nie jest marnotrawstwem pamięci.

Algorytm Kruskala

Inny sposób znajdowania MST, który też jest algorytmem zachłannym, nazywa się <u>algorytmem Kruskala</u>. W algorytmie tym dla grafu $G = (V, E)$ o n wierzchołkach tworzymy las $(V, \emptyset)$, który składa się z n składowych spójnych — każda z nich ma jeden wierzchołek. W każdym kolejnym kroku dołączamy jedną krawędź, która łączy dwie *różne* składowe lasu. Po dołączeniu w ten sposób $n-1$ krawędzi otrzymujemy las spójny, czyli drzewo.

Dołączane krawędzie wybieramy ze zbioru E w ten sposób, że wstawiamy je do kolejki priorytetowej. Priorytet krawędzi jest malejącą funkcją wagi. Po wyjęciu

krawędzi z kolejki należy sprawdzić, czy incydentne z nią wierzchołki należą do różnych składowych spójnych lasu, czy do tej samej. W pierwszym przypadku krawędź dołączamy do grafu. W drugim przypadku krawędź odrzucamy i wyjmujemy z kolejki następną, tak długo, aż znajdziemy krawędź łączącą różne składowe. Dołączenie krawędzi oznacza konieczność połączenia dwóch składowych w jedną (jak widać, istotną czynnością w algorytmie jest sprawdzenie, czy dane dwa wierzchołki są w tej samej składowej, czy w różnych).



Przykład jest na rysunku. Na początku mamy las z sześcioma składowymi spójnymi, w każdej jest jeden wierzchołek, a kolejka priorytetowa zawiera wszystkie krawędzie. Krawędź $\{v_3, v_6\}$ ma najmniejszą wagę, tj. największy priorytet i należy ona do poszukiwanego drzewa. Po jej dołączeniu do budowanego drzewa, las składa się ze składowych spójnych $(\{v_1\}, \emptyset)$, $(\{v_2\}, \emptyset)$, $(\{v_3, v_6\}, \{\{v_3, v_6\}\})$, $(\{v_4\}, \emptyset)$, $(\{v_5\}, \emptyset)$. Następna krawędź wyjęta z kolejki to $\{v_3, v_5\}$ o wadze 2. Ponieważ jej wierzchołki należą do różnych składowych lasu, więc dołączamy ją, otrzymując las o składowych $(\{v_1\}, \emptyset)$, $(\{v_2\}, \emptyset)$, $(\{v_3, v_5, v_6\}, \{\{v_3, v_5\}, \{v_3, v_6\}\})$, $(\{v_4\}, \emptyset)$. Kolejną krawędź, $\{v_5, v_6\}$, odrzucamy, bo jej wierzchołki są w jednej składowej. Dalsze działanie algorytmu powinno być jasne.

Część obliczeń jest w istocie sortowaniem krawędzi w kolejności rosnących wag. Jeśli kolejka priorytetowa jest zaimplementowana w postaci kopca, to koszt tej części obliczeń jest rzędu $O(m \log m)$, gdzie m jest liczbą krawędzi. Do tego dochodzi koszt obliczeń, mających na celu identyfikowanie składowych spójnych w przetwarzanym lesie.

Na początku las ma n składowych spójnych. W każdym kroku liczba tych składowych maleje o 1. Składowe te będziemy reprezentować za pomocą list wierzchołków. Zatem, należy zadeklarować tablicę składowych — elementy tej tablicy są wskaźnikami początków list i na początku w każdej liście jest jeden element, zawierający wskaźnik do tablicy wierzchołków. Elementy tablicy wierzchołków mają pole, którego wartość jest numerem składowej, do której dany wierzchołek należy. Zatem, mamy wierzchołki $v_1, \dots, v_n$ oraz początkowy zbiór składowych $s_1, \dots, s_n$; dla każdego i składowa s_i ma tylko wierzchołek v_i.

Aby sprawdzić, czy wierzchołki należą do różnych składowych, wystarczy porównać numery składowych, przechowywane w tablicy wierzchołków. Aby połączyć składowe s_i i s_j, należy listę wierzchołków składowej s_i dołączyć do listy wierzchołków s_j. Trzeba przy tym przejrzeć elementy listy s_i i odpowiednim wierzchołkom przypisać numer składowej j. Składowa s_i staje się przy tym pusta (wskaźnikowi początku listy jej wierzchołków przypisujemy wartość `NULL`).

Dla poprawności algorytmu nie ma znaczenia, którą składową dołączamy do drugiej. Jeśli jednak będziemy wybierać to dowolnie, to możemy otrzymać koszt rzędu n^2. Jeśli bowiem w kroku k-tym będziemy do składowej z jednym wierzchołkiem dołączać składową o k wierzchołkach, to nowy numer składowej trzeba będzie przypisać wierzchołkom $\frac{1}{2}n(n-1)$ razy.

Aby zmniejszyć rząd pesymistycznej złożoności obliczeniowej, wystarczy dołączać zawsze składową o *mniejszej* liczbie wierzchołków do składowej, która ma więcej wierzchołków. W takim przypadku wykonamy co najwyżej $O(n \log n)$ przypisań nowych numerów składowych. Ponieważ w grafie spójnym jest $m \geq n-1$, więc złożoność obliczeniowa całego algorytmu jest rzędu $O(m \log m)$.

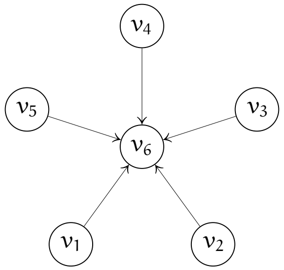
Zadania i problemy

1. Udowodnij, że jeśli wszystkie krawędzie grafu spójnego mają różne wagi, to minimalne drzewo rozpinające jest określone jednoznacznie.
2. W sytuacji, gdy istnieje więcej niż jedno minimalne drzewo rozpinające graf, różne implementacje algorytmu Prima mogą dać różne wyniki. Od czego zależy, które minimalne drzewo rozpinające otrzymamy w wyniku działania algorytmu?
3. Znajdź minimalne drzewo rozpinające graf użyty jako przykład na wykładzie, z wagami wszystkich krawędzi pomnożonymi przez -1. Użyj w tym celu algorytmów Prima i Kruskala, podając kolejność, w jakiej algorytmy te wyznaczą krawędzie drzewa.
4. Napisz w C procedurę łączenia składowych lasu, potrzebną w algorytmie Kruskala, działającą zgodnie z opisem słownym podanym na wykładzie.
5. Ile jest różnych drzew rozpinających graf prosty pełny (tj. taki, którego każde dwa wierzchołki są połączone krawędzią)?

Lasy rozpinające grafy skierowane

Las rozpinający graf *nieskierowany* jest sumą drzew rozpinających poszczególne składowe spójne tego grafu. Drzewa te są grafami nieskierowanymi, a zatem żaden wierzchołek takiego drzewa nie jest wyróżniony przez orientację incydentnych z nim krawędzi (bo orientacja nie istnieje). Zauważmy, że w ogólności las rozpinający nie jest określony jednoznacznie, ale zawsze liczba drzew jest równa liczbie składowych spójnych grafu (i zbiory wierzchołków poszczególnych drzew są jednoznacznie określone). Drzewa rozpinające poszczególne składowe spójne grafu możemy znaleźć przeszukując graf metodą DFS lub BFS.

Dla grafu *skierowanego* las rozpinający składa się z drzew skierowanych; każde z nich ma korzeń, tj. wierzchołek, do którego nie wchodzi żadna krawędź tego drzewa. Podział zbioru wierzchołków grafu skierowanego na zbiory wierzchołków poszczególnych drzew jest niejednoznaczny.



Przykład jest na rysunku. Wierzchołek v_6 może należeć tylko do jednego drzewa w lesie rozpinającym. Korzeniem tego drzewa może być dowolny z wierzchołków $v_1, \ldots, v_5$, przy czym pozostałe wierzchołki są korzeniami drzew o jednym wierzchołku. Ponadto, przeszukiwanie grafu algorytmem DFS albo BFS zaczynając od wierzchołka v_6 spowoduje wykrycie drzewa, którego v_6 jest jedynym wierzchołkiem (i wtedy będziemy mieli las z sześcioma drzewami, mimo że istnieją lasy rozpinające ten graf, złożone z pięciu drzew).

Znajdowanie lasu metodą DFS

Podana niżej procedura przeszukiwania grafu skierowanego metodą DFS określa dla każdego wierzchołka dwie liczby. Pierwsza z nich odpowiada chwili dojścia do wierzchołka podczas przeszukiwania grafu. Druga liczba odpowiada chwili zakończenia przetwarzania danego wierzchołka (w chwili tej przetwarzanie

wszystkich wierzchołków, do których procedura przeszukiwania dotarła z wierzchołka danego, jest już zakończone). Liczby te mogą być zapamiętane w strukturze opisującej wierzchołek (ponieważ mogą być potrzebne dalej), lub nie (ale wiemy, że są określone i możemy ich użyć w rozważaniach teoretycznych).

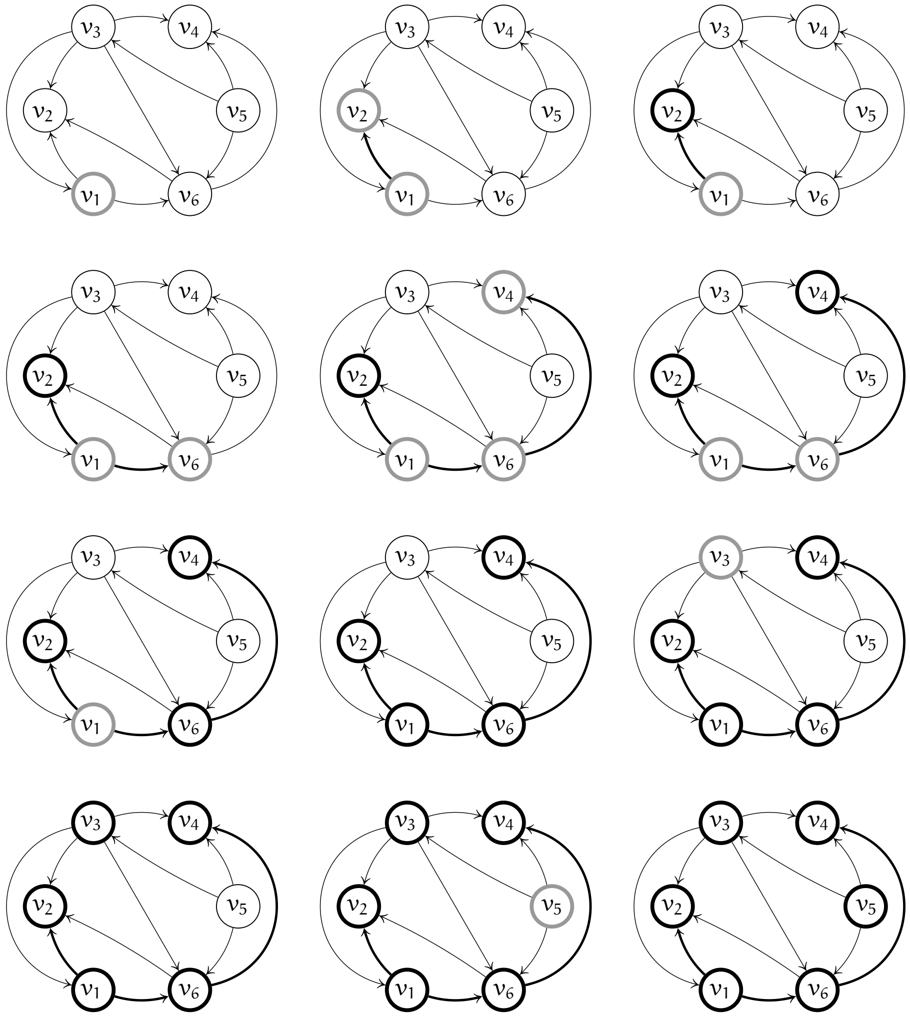
Opis algorytmu nie zakłada korzystania z konkretnej reprezentacji grafu — mogą to być listy sąsiedztwa albo macierz sąsiedztwa. Zakładamy natomiast, że wierzchołki grafu są reprezentowane przez struktury, z których każda zawiera pole o nazwie `kolor`, przyjmujące wartości `biały`, `szary` lub `czarny`, oraz pola `b`, `f` i `poprz`, typu int. Polom `b` i `f` procedura przypisze wspomniane wcześniej liczby. Rola pola `poprz` będzie wyjaśniona dalej. Alternatywnie, można użyć dodatkowych tablic, indeksowanych tak samo jak tablica wierzchołków. Podany niżej zapis procedury w pseudo-języku C pomija szczegóły implementacyjne.

```
int licznik;

void rDFS ( graf *G /* = (V, E) */, wierzchołek *v )
{
  wierzchołek *u;
  v->kolor = szary;
  v->b = licznik ++;
  for ( u = kolejno adres każdego wierzchołka, takiego że (v, u) ∈ E )
    if ( u->kolor == biały ) {
      u->poprz = v;
      rDFS ( G, u );
    }
  v->kolor = czarny;
  v->f = licznik ++;
} /*rDFS*/

void DFS ( graf *G /* = (V, E) */ )
{
  wierzchołek *v;
  for ( v = kolejno adres każdego wierzchołka v ∈ V ) {
    v->kolor = biały;
    v->poprz = NULL;
  }
  licznik = 0;
  for ( v = kolejno adres każdego wierzchołka v ∈ V ) {
    if ( v->kolor == biały ) rDFS ( G, v );
} /*DFS*/
```

Pole `poprz` wierzchołka v otrzymuje wartość, która wskazuje wierzchołek, z którego procedura „przyszła” do tego wierzchołka. Dla korzeni drzew znalezionych przez procedurę wartość tego pola jest równa `NULL`. Zauważmy, że jeśli $v.\mathtt{b} < u.\mathtt{b}$, to mamy jedną z dwóch możliwości:

$$\begin{aligned} & v.\mathtt{b} < v.\mathtt{f} < u.\mathtt{b} < u.\mathtt{f}, \\ \text{albo } & v.\mathtt{b} < u.\mathtt{b} < u.\mathtt{f} < v.\mathtt{f}. \end{aligned}$$

Istotnie, przypadek $v.\mathtt{b} < u.\mathtt{b} < v.\mathtt{f} < u.\mathtt{f}$ jest wykluczony. Jeśli wierzchołek u



został odnaleziony *po* wierzchołku v, ale *przed* zakończeniem przetwarzania wierzchołka v, to w chwili przypisania wartości polu $v.\mathtt{f}$ pole $u.\mathtt{f}$ ma już przypisaną (w rekurencyjnym wywołaniu `rDFS`) wartość, która jest mniejsza. Ta własność algorytmu DFS ma nazwę własności nawiasów.

Prześledźmy działanie algorytmu na przykładzie. Graf przedstawiony na rysunku przeszukujemy, zaczynając od wierzchołka v_1, który zaznaczamy jako szary. Wychodzące z tego wierzchołka krawędzie dochodzą do v_2 i v_6; pierwszy z tych wierzchołków, v_2, przetwarzamy najpierw. Ponieważ żadne krawędzie z niego nie wychodzą, więc zaraz po zaznaczeniu go jako odnalezionego (tj. po nadaniu mu koloru szarego), kończymy jego przetwarzanie, nadając mu kolor czarny. Następnie odwiedzamy wierzchołek v_6. Jedyna krawędź, która z niego wychodzi, prowadzi do v_4. Jest to ostatni wierzchołek pierwszego znalezionego drzewa w lesie rozpinającym graf. Następne dwa drzewa mają po jednym wierzchołku (i nie mają krawędzi), są to v_3 i v_5; zauważmy, że gdyby wierzchołki te były odwrotnie ponumerowane, to v_5 byłby korzeniem znalezionego drzewa o dwóch wierzchołkach.

Po zakończeniu działania procedury pola poszczególnych wierzchołków mają następujące wartości:

	v_1	v_2	v_3	v_4	v_5	v_6
`b` :	0	1	8	4	10	3
`f` :	7	2	9	5	11	6
`poprz` :	`NULL`	$\&v_1$	`NULL`	$\&v_6$	`NULL`	$\&v_1$

Rozpatrywany tu algorytm w różny sposób przetwarza różne krawędzie, które można podzielić na cztery rodzaje (uwaga: podział ten zależy od algorytmu, a nie tylko od samego grafu). Są to

- Krawędzie drzewowe. To są krawędzie należące do znalezionych drzew. W chwili wywołania rekurencyjnego procedury `rDFS` wierzchołek końcowy takiej krawędzi jest biały.
- Krawędzie powrotne. To są krawędzie, których końcowy wierzchołek jest szary, (i dlatego nie wywołujemy dla tego wierzchołka rekurencyjnie procedury `rDFS`). Od wierzchołka tego do początku krawędzi powrotnej istnieje ścieżka, z którą krawędź powrotna tworzy cykl.
- Krawędzie skierowane w przód. To są krawędzie, które od danego wierzchołka prowadzą do innych, już odnalezionych i przetworzonych (czyli czarnych) wierzchołków tego samego drzewa.

- <u>Krawędzie boczne</u>. To są krawędzie, których wierzchołki należą do różnych drzew. Podczas przetwarzania takiej krawędzi wierzchołek końcowy jest czarny (bo drzewo, do którego należy ten wierzchołek już zostało znalezione; gdyby był biały, to dołączylibyśmy go do tego samego drzewa, a gdyby był szary, to mielibyśmy krawędź powrotną).

Wykrywanie cykli w grafie

Aby stwierdzić, że graf skierowany nie zawiera cykli, wystarczy wykonać opisaną wcześniej procedurę DFS, znajdującą las rozpinający. Graf nie zawiera cykli wtedy i tylko wtedy, gdy żadna krawędź grafu nie jest powrotna.

Udowodnimy to. Przypuśćmy, że pewna krawędź grafu, (v, u), jest powrotna, tj. w chwili dojścia algorytmu do jej przetwarzania, jej koniec (wierzchołek u), jest szary. To oznacza, że wierzchołek u należy do tego samego drzewa, co v, a ponadto istnieje ścieżka od u do v, której wszystkie wierzchołki są szare (każdy z tych wierzchołków może stać się czarny dopiero po przetworzeniu wierzchołka v). Dołączenie krawędzi (v, u) do tej ścieżki daje cykl.

Jeśli graf zawiera cykl, to pewien wierzchołek tego cyklu będzie odwiedzony (tj. pomalowany na szaro) najwcześniej. Procedura DFS, przed pomalowaniem tego wierzchołka na czarno, odwiedzi wszystkie wierzchołki osiągalne z niego, w tym wierzchołek, z którego wychodzi krawędź zamykająca cykl. Koniec dowodu.

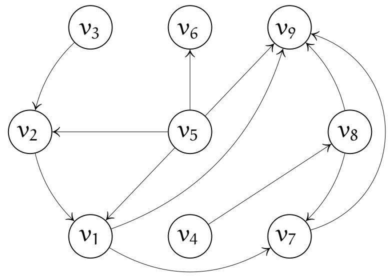
Sortowanie topologiczne

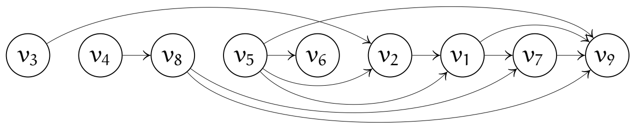
Każdy bezcyklowy graf skierowany opisuje pewien porządek częściowy w zbiorze swoich wierzchołków. Zadanie sortowania topologicznego polega na znalezieniu takiej kolejności wierzchołków, w której dla dowolnej krawędzi grafu, (v_i, v_j), zachodzi nierówność $i < j$. Udowodnimy, że istnienie takiego uporządkowania wierzchołków jest równoważne brakowi cykli w grafie.

Istotnie, jeśli jest cykl, to uporządkowanie topologiczne wierzchołków nie istnieje, bo przynajmniej jedna krawędź tego cyklu musi mieć koniec o numerze (w kolejności uporządkowania) mniejszym niż początek. Jeśli natomiast graf nie zawiera cykli, to co najmniej z jednego wierzchołka nie wychodzi ani jedna krawędź. Gdyby z każdego wierzchołka wychodziła co najmniej jedna krawędź, to zaczynając od dowolnego wierzchołka skonstruujemy maksymalnie długą ścieżkę, która musi mieć mniej krawędzi, niż liczba wierzchołków w grafie. Krawędź wychodząca z końcowego wierzchołka tej ścieżki wchodzi do jednego

z wierzchołków na tej ścieżce, tworząc cykl (gdyby końcowy wierzchołek tej krawędzi nie należał do ścieżki, to moglibyśmy krawędź dołączyć do ścieżki, a tymczasem założyliśmy, że ścieżki tej nie da się wydłużyć).

Jeśli graf bezcyklowy ma $n > 1$ wierzchołków, to wierzchołkowi v, z którego nie wychodzi żadna krawędź (udowodniliśmy, że taki istnieje) nadajemy numer n, a następnie usuwamy z grafu ten wierzchołek i wszystkie krawędzie wchodzące do niego. Otrzymujemy graf skierowany G' o $n - 1$ wierzchołkach, który jest oczywiście bezcyklowy. Z założenia indukcyjnego, że dla każdego bezcyklowego grafu skierowanego o $n - 1$ wierzchołkach istnieje uporządkowanie topologiczne wierzchołków (to jest oczywiste dla $n = 1$) wynika istnienie takiego porządku dla grafu o n wierzchołkach — wystarczy dołączyć wierzchołek v na końcu posortowanego topologicznie ciągu wierzchołków $v_{i_1}, \ldots, v_{i_{n-1}}$ grafu G'.





Sortowanie topologiczne ma wiele praktycznych zastosowań, zwłaszcza w organizacji pracy. W celu jej wykonania trzeba wykonać wiele czynności, przy czym zbiór tych czynności jest częściowo uporządkowany. Relacja $a < b$ zachodzi wtedy, gdy nie można wykonać b, jeśli wcześniej nie zrobiło się a (np. nie można pomalować pewnego przedmiotu przed jego wyprodukowaniem). Znalezienie porządku topologicznego umożliwia wykonanie tych czynności.

Sortowanie topologiczne — DFS

Aby posortować topologicznie wierzchołki bezcyklowego grafu skierowanego, wystarczy wykonać procedurę DFS opisaną wcześniej. Kolejne wierzchołki należy wstawiać na początek listy wierzchołków. W ten sposób kolejność wstawiania jest odwrotna do kolejności wierzchołków w znalezionym porządku. Wierzchołek należy wstawić *po* przeszukaniu wierzchołków sąsiednich (czyli wstawiamy wierzchołek wtedy, gdy zostaje zamalowany na czarno — do podanej procedury DFS wystarczy dopisać w tym miejscu wywołanie procedury, która dołącza wierzchołek do listy), zatem uporządkowanie wierzchołków odpowiadające porządkowi topologicznemu jest w kolejności malejących atrybutów f przypisanych wierzchołkom. Zamiast listy możemy użyć stosu. Po zakończeniu procedury możemy pozdejmować wierzchołki ze stosu, otrzymując właściwą kolejność.

Upewnijmy się, że ten algorytm jest poprawny. Jeśli w grafie nie ma cyklu, to algorytm DFS nie wykryje żadnej krawędzi powrotnej (wykrycie takiej krawędzi powinno spowodować przerwanie obliczeń i wyprowadzenie informacji, że wierzchołków grafu nie można posortować topologicznie). Wszystkie krawędzie są zatem drzewowe, skierowane w przód, albo boczne. W przypadku krawędzi drzewowej algorytm najpierw wyprowadzi wierzchołek końcowy tej krawędzi, a później wierzchołek na jej początku. W przypadku krawędzi skierowanej w przód lub bocznej, koniec krawędzi został odwiedzony (i wyprowadzony) przed dojściem algorytmu do początku tej krawędzi. Zatem, kolejność malowania wierzchołków na czarno przez algorytm DFS jest odwrotna do właściwego uporządkowania topologicznego.

Sortowanie topologiczne — zastosowanie kolejki

Inna metoda sortowania topologicznego polega na użyciu kolejki. W grafie bezcyklowym istnieje wierzchołek, do którego nie wchodzi żadna krawędź (jeśli graf jest bezcyklowy, to graf G^T, który powstaje z G przez odwrócenie wszystkich krawędzi, też jest bezcyklowy). Algorytm sortowania topologicznego przy użyciu kolejki jest taki:

1. Dla każdego wierzchołka znajdź liczbę krawędzi, które do niego wchodzą.
2. Wstaw do (początkowo pustej) kolejki wszystkie wierzchołki, do których wchodzi 0 krawędzi.
3. Następne kroki powtarzaj tak długo, aż kolejka stanie się pusta.
4. Wyjmij z kolejki wierzchołek v i nadaj mu kolejny numer, który określa porządek topologiczny.
5. Dla każdej krawędzi (v, u) wychodzącej z v zmniejsz licznik krawędzi wchodzących do u o 1. Jeśli licznik wierzchołka u otrzymał wartość 0, to wstaw wierzchołek u do kolejki.

Graf zawiera cykl (czyli porządek topologiczny nie istnieje), jeśli po opróżnieniu kolejki istnieją wierzchołki, które nigdy nie były do niej wstawione (czyli nie mają nadanych numerów).

Zadania i problemy

1. Napisz w C procedurę, która na podstawie list sąsiedztwa reprezentujących graf skierowany G, znajduje reprezentację grafu G^T, który ma ten sam zbiór wierzchołków, a jego zbiór krawędzi powstaje przez zamianę orientacji każdej krawędzi grafu G.

   W jaki sposób można rozwiązać to zadanie dla reprezentacji grafu w postaci tablicy sąsiedztwa?
2. Zidentyfikuj krawędzie drzewowe, powrotne, skierowane w przód i boczne w grafie użytym jako przykład działania algorytmu DFS na wykładzie.
3. Wskaż relację częściowego porządku w zbiorze części ubrania, które ktoś ma na siebie założyć, i przedstaw ją za pomocą grafu skierowanego.
4. Znajdź las rozpinający graf narysowany na stronie 18.6.
5. Udowodnij twierdzenie, że porządek topologiczny w zbiorze wierzchołków grafu skierowanego istnieje wtedy i tylko wtedy, gdy graf jest bezcyklowy, dowodząc najpierw istnienia w takim grafie wierzchołka, do którego nie wchodzi żadna krawędź, a następnie korzystając z indukcji.
6. Jaki jest związek między uporządkowaniami topologicznymi zbioru wierzchołków grafu G i grafu G^T?

Znajdowanie składowych silnie spójnych

Składowa silnie spójna w grafie skierowanym jest to maksymalny podgraf, taki że dowolne dwa jego wierzchołki są połączone ścieżkami (pierwszy wierzchołek jest początkiem jednej ścieżki i końcem drugiej, a drugi odwrotnie). Relacja „dwa wierzchołki należą do tej samej składowej silnie spójnej" jest równoważnością w zbiorze wierzchołków. Graf skierowany bezcyklowy ma tyle składowych, ile wierzchołków. Niepusty graf silnie spójny ma jedną składową. Zajmiemy się zadaniem znajdowania składowych silnie spójnych w danym grafie skierowanym.

Do rozwiązania zadania wystarczy wskazać odpowiednie podzbiory zbioru wierzchołków (wystarczy, aby każdy wierzchołek miał określony atrybut, będący numerem składowej, do której należy), ponieważ krawędzie składowej, której zbiór wierzchołków znamy, są łatwe do zidentyfikowania w zbiorze krawędzi.

Aby znaleźć algorytm znajdowania składowych, rozważmy graf skierowany $G = (V, E)$, dla którego znaleźliśmy las rozpinający metodą DFS. Możemy uporządkować wierzchołki grafu G w kolejności *malejących* numerów f, nadanych wierzchołkom podczas przeszukiwania grafu (numery te odpowiadają chwilom, w których wierzchołki zostały oznaczone jako czarne). Gdyby graf G był bezcyklowy, to otrzymana kolejność wierzchołków byłaby poprawnym wynikiem sortowania topologicznego.

Niech $G^T = (V, E^T)$ oznacza graf skierowany otrzymany przez odwrócenie wszystkich krawędzi (tj. każdej krawędzi $(v_i, v_j) \in E$ odpowiada krawędź $(v_j, v_i) \in E^T$). To oznaczenie wzięło się stąd, że macierz sąsiedztwa grafu G^T jest transpozycją macierzy sąsiedztwa grafu G.

Wykonamy procedurę przeszukiwania w głąb grafu G^T, zaczynając od wierzchołka v, który otrzymał największy numer f. Przekonamy się, że każdy wierzchołek w, należący do drzewa, którego korzeniem jest v, należy razem z v do tej samej silnie spójnej składowej grafu G, a wierzchołki, które do tego drzewa nie należą, należą do innych składowych.

Zauważmy, że każdej składowej silnie spójnej grafu G odpowiada składowa silnie spójna grafu G^T, o tym samym zbiorze wierzchołków. Jeśli wierzchołek $w \neq v$ został dołączony do drzewa (w lesie rozpinającym G^T), którego korzeniem jest v, to oczywiście istnieje w G^T ścieżka prowadząca od v do w. Zatem, w grafie G istnieje ścieżka prowadząca od w do v.

Trzeba wykazać, że w grafie G istnieje również ścieżka prowadząca od v do w. Wierzchołek v ma największy numer f, a zatem $v.f > w.f$. Przypuśćmy, że $w.b < v.b$. Ponieważ istnieje w G ścieżka od w do v, więc podczas przeszukiwania grafu G musiałaby zostać odkryta i z własności nawiasów wynika, że w takim razie byłoby $w.b < v.b < v.f < w.f$, co jest sprzeczne z założeniem, że atrybut f wierzchołka v jest największy.

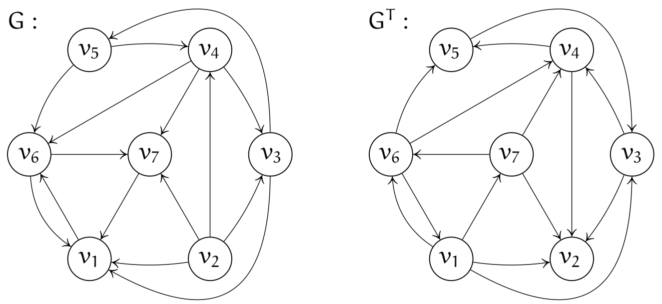
Musi zatem być $v.b < w.b < w.f < v.f$. Stąd jednak wynika, że przetwarzanie wierzchołka w rozpoczęło się i zakończyło po rozpoczęciu i przed zakończeniem przetwarzania v, ale to oznacza istnienie w grafie G ścieżki od v do w.

Jeśli do drzewa o korzeniu v (w lesie rozpinającym graf G^T) zostaną dołączone wierzchołki u i w, to z istnienia ścieżek od u do v, od v do w, od w do v i od v do u w oczywisty sposób wynika, że wierzchołki te należą do tej samej silnie spójnej składowej grafu G. Jeśli wierzchołek w nie należy do składowej z wierzchołkiem v, to któraś ze ścieżek — od w do v lub od v do w — nie istnieje. W takim przypadku wierzchołek w nie może zostać dołączony do drzewa o korzeniu v.

Zauważmy, że algorytm DFS (przeszukiwania grafu G^T) wywołany dla wierzchołka v, takiego że liczba $v.f$ jest największa, zaznaczy (na czarno) wszystkie wierzchołki składowej silnie spójnej grafu G z wierzchołkiem v, a wszystkie pozostałe wierzchołki pozostaną białe. Jeśli odrzucimy wierzchołki czarne (i wszystkie krawędzie z nimi incydentne), to ze zbioru wierzchołków białych (jeśli jest niepusty) możemy wybrać wierzchołek o największej wartości atrybutu f i powtórzyć rozumowanie. Algorytm znajdowania składowych silnie spójnych grafu oparty na tym rozumowaniu składa się z trzech kroków:

1. Wykonaj algorytm DFS, w celu określenia kolejności wierzchołków grafu G zgodnie z malejącymi wartościami atrybutu f (zapamiętywanie tych liczb jest niekonieczne, ważna jest tylko kolejność).
2. Skonstruuj reprezentację grafu G^T.
3. Zaznacz wszystkie wierzchołki jako białe. Zaczynając DFS za każdym razem od wierzchołka o największej wartości f w zbiorze wierzchołków białych, znajdź las rozpinający G^T. Każde drzewo w tym lesie określa jedną składową silnie spójną grafu G (zbiór wierzchołków składowej jest zbiorem wierzchołków drzewa).

Zobaczmy działanie algorytmu na przykładzie grafu przedstawionego na rysunku. Przeszukiwanie grafu G algorytmem DFS przyporządkowuje jego wierzchołkom



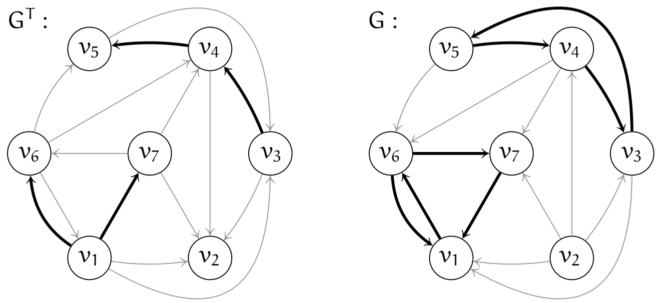
następujące atrybuty b i f:

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
b :	0	6	7	9	8	1	2
f :	5	13	12	10	11	4	3

Po uporządkowaniu wierzchołków w kolejności malejących wartości atrybutu f, tj. $v_2, v_3, v_5, v_4, v_1, v_6, v_7$, znajdujemy las rozpinający graf G^T, za pomocą algorytmu DFS. Podczas tego przeszukiwania wierzchołki otrzymają następujące atrybuty:

	v_2	v_3	v_5	v_4	v_1	v_6	v_7
b :	0	2	4	3	8	9	11
f :	1	7	5	6	13	10	12

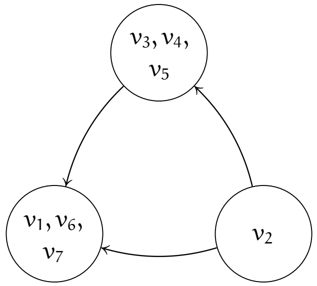
Jak widać z tabeli, pierwsze drzewo lasu ma tylko jeden wierzchołek, v_2. Kolejne drzewo ma wierzchołki v_3, v_5, v_4, a pozostałe należą do ostatniego drzewa. Na rysunku niżej są wyróżnione krawędzie znalezionego lasu rozpinającego graf G^T (tj. krawędzie, które algorytm DFS sklasyfikował jako drzewowe) i krawędzie składowych silnie spójnych grafu G.



Graf zredukowany

Mając dowolny graf skierowany G, możemy określić jego graf zredukowany. Jest to graf skierowany prosty, którego wierzchołkami są składowe silnie spójne grafu G. Oznaczmy je symbolami $s_1, \dots, s_k$. Krawędź (s_i, s_j) należy do grafu zredukowanego, wtedy i tylko wtedy, gdy istnieją wierzchołki $v_l \in s_i$ oraz $v_m \in s_j$, takie że graf G zawiera krawędź (v_l, v_m).

Rysunek poniżej przedstawia graf zredukowany odpowiadający grafowi użytemu do zilustrowania działania algorytmu znajdowania składowych silnie spójnych.



Jest jasne, że graf zredukowany jest bezcyklowy. Istotnie, gdyby w grafie tym był cykl, to z dowolnego wierzchołka grafu G należącego do składowej, przez którą przechodzi ten cykl, istniałaby ścieżka prowadząca do wierzchołków pozostałych składowych w tym cyklu — ale to oznacza istnienie składowej silnie spójnej, której wierzchołkami są wszystkie wierzchołki składowych występujących w cyklu grafu zredukowanego.

Jeśli graf zredukowany ma tylko jeden wierzchołek, to graf G jest silnie spójny. Mając graf zredukowany możemy rozwiązać następujące zadanie: znalezienie krawędzi, których dołączenie do grafu G dałoby w wyniku graf silnie spójny. Znalezienie odpowiedniego algorytmu pozostawiam jako problem do zastanowienia.

Zadania i problemy

1. Prześledź działanie algorytmu znajdowania składowych silnie spójnych grafu skierowanego na innych przykładach.
2. Zaproponuj algorytm znajdowania reprezentacji w postaci list sąsiedztwa grafu zredukowanego dla danego grafu G.
3. Zaproponuj algorytm, który dołącza do grafu skierowanego G krawędzie tak, aby powstał graf silnie spójny.
4. Napisz procedurę, która dla grafu G, reprezentowanego za pomocą list sąsiedztwa, skonstruuje analogiczną reprezentację grafu G^T.
5. Jak poradzić sobie z zadaniem zaimplementowania algorytmu DFS, jeśli chcemy narzucić kolejność, w jakiej mają być wybierane spośród białych w danej chwili wierzchołków grafu korzenie drzew w lesie rozpinającym (np. ma być zawsze wierzchołek o największym atrybucie f)?

Znajdowanie najkrótszych ścieżek

Przypuśćmy, że w grafie skierowanym $G = (V, E)$ każda krawędź ma przyporządkowaną liczbę, tzw. wagę, koszt lub długość. Zajmiemy się zadaniem znajdowania najkrótszych dróg w grafie między ustalonymi wierzchołkami, przy czym przez długość drogi rozumiemy sumę długości krawędzi, z których ta droga się składa. Najbardziej oczywistym (ale nie jedynym) zastosowaniem praktycznym tego zadania jest planowanie podróży — chodzi o wybranie najlepszego (w sensie najmniejszej liczby kilometrów, czasu podróży, ilości zużytego paliwa itp.) sposobu przedostania się z jednej miejscowości do drugiej.

Pod względem ideologicznym zadanie to trochę przypomina zadanie znajdowania minimalnego drzewa rozpinającego graf (nieskierowany). Problem znajdowania najkrótszych dróg stawiamy jednak dla grafu skierowanego, ponieważ aby znaleźć najkrótszą drogę w grafie nieskierowanym G, wystarczy określić graf skierowany G' (o tym samym zbiorze wierzchołków), taki że każdej krawędzi grafu G odpowiadają dwie krawędzie grafu G' między tymi samymi wierzchołkami, zorientowane przeciwnie. Wagi tych krawędzi muszą być równe wadze odpowiedniej krawędzi grafu G (zauważmy, że reprezentacja grafu G w postaci list sąsiedztwa lub macierzy sąsiedztwa jest jednocześnie reprezentacją grafu G'). Ponadto, postawienie problemu dla grafu skierowanego jest bardziej naturalne: wprawdzie jadąc z miejscowości A do B przejedziemy tyle samo kilometrów, co z B do A, ale jeśli w jedną stronę jedzie się z góry, a w drugą trzeba pchać samochód pod górę, to ilości zużytego czasu i benzyny w każdą stronę będą inne.

Zanim zabierzemy się za algorytmy, zbadajmy problem teoretycznie. Po pierwsze, w przypadku ogólnym zadanie może nie mieć jednoznacznego rozwiązania; przykłady każdy łatwo sam wskaże. Jeśli rozwiązań jest więcej niż jedno, to zadowalamy się znalezieniem jednego (dowolnego) rozwiazania.

Rozważmy drogę między wierzchołkami. Jeśli zawiera ona cykl o dodatniej sumie wag, to możemy go usunąć, otrzymując krótszą drogę między tymi samymi wierzchołkami (tj. o mniejszej sumie wag krawędzi). Cykl o zerowej sumie wag możemy też usunąć, otrzymując drogę o tej samej sumie wag. Jeśli nie ma cykli ujemnych, to usunięcie wszystkich cykli z dowolnej drogi daje w wyniku ścieżkę o nie większej długości, a zatem przy braku cykli, których suma wag jest ujemna (tzw. cykli ujemnych) zadanie znalezienia najkrótszej drogi jest dobrze postawione i jego rozwiązania możemy szukać w zbiorze ścieżek. Jeśli graf zawiera cykl ujemny, to zadanie znalezienia najkrótszej drogi może nie mieć rozwiązania, choć zadanie znalezienia najkrótszej ścieżki jest dobrze postawione.

Istnieją algorytmy znajdowania najkrótszych ścieżek, które działają przy założeniu, że nie ma ujemnych cykli, ale poszczególne krawędzie mogą mieć ujemne wagi. Inne algorytmy działają poprawnie przy założeniu, że wszystkie krawędzie mają wagi nieujemne.

Problem można postawić na trzy sposoby: można chcieć znaleźć najkrótszą ścieżkę między dwoma wskazanymi wierzchołkami, można poszukiwać najkrótszych ścieżek z pewnego wierzchołka do wszystkich pozostałych i wreszcie można zażyczyć sobie informacji o najkrótszych ścieżkach między wszystkimi parami wierzchołków.

Jest oczywiste, że najkrótsza ścieżka z dowolnego wierzchołka v do niego samego składa się z 0 krawędzi i ma długość 0. Jeśli najkrótsza ścieżka z wierzchołka u do v przechodzi przez wierzchołek w, to ścieżka ta składa się z najkrótszej ścieżki z u do w i najkrótszej ścieżki z w do v. To spostrzeżenie umożliwia rozwiązanie problemu przy użyciu algorytmów zachłannych.

Możemy wreszcie zauważyć, że problem znalezienia najkrótszej ścieżki między dwoma wskazanymi wierzchołkami ma tę samą złożoność pesymistyczną, co problem znalezienia najkrótszych ścieżek do wszystkich pozostałych wierzchołków. Istotnie, albo najkrótsza ścieżka z u do v prowadzi przez pewien wierzchołek w (i wtedy ją wyznaczymy), albo nie i wtedy musimy wykluczyć możliwość, że istnieje krótsza ścieżka od u do v, przechodząca przez w. Aby to zrobić, może być potrzebne wyznaczenie najkrótszej ścieżki z u do w.

Między dwoma wierzchołkami może być więcej niż jedna najkrótsza ścieżka; nas interesować będzie znalezienie dowolnej ścieżki najkrótszej. Możemy zauważyć, że istnieje drzewo skierowane (którego korzeniem jest s), które zawiera wszystkie najkrótsze ścieżki od wierzchołka s do wszystkich tych wierzchołków, do których dowolne ścieżki wychodzące z s istnieją.

Algorytm Bellmana-Forda

Algorytm Bellmana-Forda znajduje najkrótsze ścieżki z dowolnego wskazanego wierzchołka do wszystkich pozostałych, przy założeniu, że w grafie nie ma cykli ujemnych. Dla każdego wierzchołka mamy dwa atrybuty. Można je pamiętać w dwóch dodatkowych tablicach, których zbiory indeksów są takie jak zbiór indeksów wierzchołków, mogą też to być pola struktury (`struct` w C) opisującej wierzchołek; takiej notacji użyjemy. Wartość atrybutu d jest długością najkrótszej znalezionej w pewnym etapie ścieżki. Wartością atrybutu p jest indeks wierzchołka, z którego do wierzchołka danego prowadzi ostatnia krawędź tej ścieżki.

Algorytm ten korzysta z dwóch procedur pomocniczych (napisanych niżej w cokolwiek rozpaczliwym pseudojęzyku — pisząc w stylu języka C niesposób uniknąć przedstawiania *wszystkich* szczegółów implementacyjnych):

```
void InitSingleSource ( graf G, wierzchołek *u )
{
  for ( v wskazuje kolejno wszystkie wierzchołki grafu ) {
    v->d = ∞;
    v->p = 0;    /* 0 nie jest numerem wierzchołka */
  }
  u->d = 0;
} /*InitSingleSource*/

void Relax ( krawędź *e /* = (v,w) */ )
{
  if ( w.d > v.d + waga ( e ) ) {
    w.d = v.d + waga ( e );
    w.p = indeks ( v );
  }
} /*Relax*/
```

Procedura InitSingleSource ustala sytuację początkową, w której niezbadane są wszystkie ścieżki w grafie, zatem do każdego wierzchołka umiemy dotrzeć „ścieżką" o nieskończonej długości, której ostatnia krawędź wychodzi nie wiadomo skąd. Jedynym wyjątkiem jest wierzchołek u, z którego ścieżki mamy znaleźć; do niego umiemy dotrzeć z u ścieżką pustą o długości 0.

Procedura Relax dokonuje tzw. relaksacji. Jeśli dla wierzchołka u znamy taką ścieżkę z s, że suma jej długości i wagi krawędzi $e = (u, v)$ jest mniejsza niż długość najkrótszej znalezionej dotąd ścieżki z s do v, to właśnie znaleźliśmy krótszą ścieżkę do v, przechodzącą przez u. Algorytm Bellmana-Forda realizuje procedura

```
void BellmanFord ( graf G /* = (V,E) */, wierzchołek *u )
{
  InitSingleSource ( G, u );
  for ( i = 1; i < |V|; i++ )
    for ( e wskazuje kolejno wszystkie krawędzie w E ) Relax ( e );
  for ( e wskazuje kolejno wszystkie krawędzie w E )  /* *e = (v,w) */
    if ( w.d > v.d + waga ( e ) )
      graf G zawiera ujemny cykl;
} /*BellmanFord*/
```


Jest oczywiste, że algorytm Bellmana-Forda wykonuje $\Theta(nm)$ operacji, gdzie $n = |V|$, $m = |E|$. Uzasadnijmy, że daje on poprawny wynik. Zasadnicza część algorytmu to wykonanie relaksacji dla wszystkich krawędzi $n-1$ razy — tyle, z ilu krawędzi maksymalnie może składać się ścieżka w grafie o n wierzchołkach. Zauważmy, że jeśli pewna ścieżka S najkrótsza (w sensie sumy wag) składa się z $k > 1$ krawędzi, to istnieją w tym grafie także ścieżki najkrótsze składające się z $1, \ldots, k-1$ krawędzi — ścieżki te prowadzą z u do wierzchołków leżących na ścieżce S.

Przypuśćmy, że w grafie nie ma cykli ujemnych. Jeśli przyjmiemy założenie indukcyjne, że po zakończeniu $k-1$ przebiegów zewnętrznej pętli algorytm znalazł ścieżki najkrótsze w zbiorze E_{k-1} ścieżek o początku w u, do którego należą wszystkie ścieżki składające się z co najwyżej $k-1$ krawędzi (i być może pewne ścieżki złożone z większej ich liczby), to możemy wykazać, że w k-tym przebiegu tej pętli algorytm znajdzie ścieżki najkrótsze w zbiorze E_k ścieżek, którego elementami są wszystkie ścieżki co najwyżej k-krawędziowe i być może pewne ścieżki dłuższe. Istotnie, każda ścieżka o długości $l > 0$, składa się z początkowych $l-1$ krawędzi i z krawędzi ostatniej, $e = (v, w)$. Ścieżka złożona z początkowych $l-1$ krawędzi jest najkrótszą ścieżką od wierzchołka u do v, złożoną z krawędzi należących do E_{k-1} (z założenia indukcyjnego), natomiast cała ścieżka ma sumę wag minimalną w zbiorze ścieżek z E_{k-1} wydłużonych o jedną krawędź (ten zbiór, E_k, zawiera zatem wszystkie ścieżki o początku w u złożone z k krawędzi).

Na końcu procedury następuje sprawdzenie, czy graf zawiera cykl ujemny. Polega to na zbadaniu, czy istnieje krawędź, której relaksacja spowodowałaby znalezienie krótszej drogi z wierzchołka u do dowolnego innego wierzchołka.

Po wykonaniu algorytmu Bellmana-Forda, a także opisanego dalej algorytmu Dijkstry, dla każdego wierzchołka znamy długość najkrótszej ścieżki z u. Ścieżkę od u do v, tj. ciąg wierzchołków, przez które ta ścieżka przechodzi, możemy otrzymać za pomocą następującej procedury (parametr i jest numerem końcowego wierzchołka ścieżki):

```
void WyprowadźŚcieżkęD ( graf G, int i )
{
  do {
    v = G.wierzchołek[i];
    wyprowadź ( v );
    i = v.p;
  } while ( i );
} /*WyprowadźŚcieżkęD*/
```

Algorytm Dijkstry

Algorytm Dijkstry jest algorytmem zachłannym wyznaczania najkrótszych ścieżek z ustalonego wierzchołka do wszystkich pozostałych, działającym szybciej niż algorytm Bellmana-Forda. Warunkiem poprawnego działania tego algorytmu jest, aby każda krawędź miała nieujemną wagę.

Zasada działania algorytmu Dijkstry jest następująca: niech u oznacza wierzchołek, z którego najkrótsze ścieżki do pozostałych wierzchołków mamy znaleźć; wierzchołek ten bywa nazywany źródłem. Algorytm tworzy reprezentację zbioru $S \subset V$, który początkowo ma tylko jeden element, u. Podczas działania algorytmu zbiór S składa się z tych wierzchołków, do których najkrótsze ścieżki zostały już znalezione. W każdym kroku zbiór ten jest powiększany o jeden element. Jest to wierzchołek v, do którego prowadzi najkrótsza ścieżka z u, której wszystkie wierzchołki oprócz v należą do zbioru S (wybór wierzchołka v następuje w sposób zachłanny — trochę przypomina to działanie algorytmu Prima znajdowania MST).

Algorytm Dijkstry wykorzystuje te same procedury `InitSingleSource` i `Relax`, co algorytm Bellmana-Forda. Wybór wierzchołka v, który w danym kroku zostaje dołączony do zbioru S jest wykonywany w następujący sposób: dla każdego $w \in V \setminus S$ znana jest długość najkrótszej ścieżki z u, której wszystkie wierzchołki oprócz w należą do S (jeśli ścieżka taka nie istnieje, to przyjmujemy, że długością ścieżki jest ∞). Wierzchołek v, który dołączymy do S, leży na końcu najkrótszej z tych ścieżek — długość najkrótszej ścieżki z u prowadzącej do $w \in V \setminus S$ przez wierzchołki należące do S określa priorytet wierzchołka w. Zatem, zbiór $V \setminus S$ może być reprezentowany przez kolejkę priorytetową.

Po dołączeniu wierzchołka v do zbioru S, rozpatrujemy wszystkie krawędzie, których początkiem jest v, a koniec w nie należy do S. Dla każdej takiej krawędzi wykonujemy relaksację.

Reasumując: dla każdego wierzchołka $w \in V$ znamy w danej chwili długość najkrótszej znalezionej dotąd ścieżki z u. Dla wierzchołków należących do S jest to najkrótsza ścieżka istniejąca w grafie G (później to udowodnimy). Dla pozostałych wierzchołków algorytm może jeszcze znaleźć ścieżkę krótszą.


```
void Dijkstra ( graf G, wierzchołek *u )
{
  InitSingleSource ( G, u );
  S = ∅;
  do {
    v = wierzchołek należący do V\S, taki że v.d jest najmniejsze;
    S = S∪{v};
    for ( e wskazuje kolejno wszystkie krawędzie o początku v )
      Relax ( e );
  } while ( S != V );
} /*Dijkstra*/
```

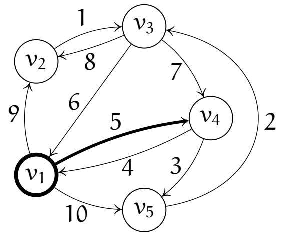
Algorytm działa także wtedy, gdy istnieją wierzchołki, do których nie da się dojść z wierzchołka u. Atrybut d każdego takiego wierzchołka będzie miał wartość ∞, ale praktyczna implementacja algorytmu musi uwzględniać możliwość wystąpienia nieskończoności jako argumentu dodawania (niezależnie od tego, nieskończoność może wystąpić jako argument operacji porównywania). Dlatego należy zachować szczególną ostrożność, jeśli do reprezentowania nieskończoności używamy jakiejś bardzo dużej liczby stało- lub zmiennopozycyjnej.

Udowodnimy, że algorytm Dijkstry daje poprawny wynik. Na początku k-tego kroku (przebiegu zewnętrznej pętli) zbiór S jest $k-1$ elementowy, a po jego wykonaniu zbiór ten ma k elementów. Przyjmijmy założenie indukcyjne, że na początku k-tego kroku dla każdego wierzchołka v wartość atrybutu d jest długością najkrótszej ścieżki wychodzącej z u, której wszystkie wierzchołki (oprócz v, jeśli $v \notin S$) są elementami zbioru S. W pierwszym kroku do zbioru pustego S zostanie dołączony wierzchołek u, z którego wychodzą wszystkie interesujące nas ścieżki. Na początku kroku drugiego będziemy znać wszystkie ścieżki o początku w u złożone z jednej krawędzi, tak więc założenie indukcyjne dla $k = 2$ jest prawdziwe.

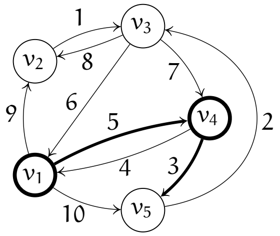
Niech v i w oznaczają wierzchołki, z których pierwszy należy do S, a drugi nie należy, przy czym atrybut d wierzchołka w jest najmniejszy w zbiorze $V \setminus S$. Należy wykazać, że najkrótsza ścieżka z u do w jest połączeniem najkrótszej ścieżki z u do v z krawędzią (v, w), tj. nie istnieje ścieżka krótsza. Z założenia indukcyjnego, wszystkie wierzchołki najkrótszej ścieżki z u do v należą do S. Rozpatrywana ścieżka z u do w jest więc najkrótszą ścieżką, której wszystkie wierzchołki oprócz końcowego są elementami zbioru S. Zatem wystarczy wykazać, że nie istnieje krótsza ścieżka z u do w, która prowadzi przez wierzchołek u nie należący do S.

Przypuśćmy, że taka ścieżka istnieje. Zgodnie z wcześniejszym spostrzeżeniem, ścieżka ta jest połączeniem najkrótszej ścieżki z u do $v \notin S$ oraz najkrótszej ścieżki z v do w. Niech v będzie pierwszym na tej ścieżce wierzchołkiem, który nie należy do S (początek ścieżki, tj. u należy do S). Długość tej ścieżki jest równa sumie długości najkrótszych ścieżek z u do v i z v do w. Ale pierwsza z tych długości jest równa $u.d > w.d$ (bo tak wybieramy wierzchołek w), a ścieżka z v do w ma długość nieujemną (bo wszystkie krawędzie mają wagi nieujemne). Otrzymana sprzeczność dowodzi, że istnieje najkrótsza ścieżka z u do w przechodząca tylko przez wierzchołki należące do S.

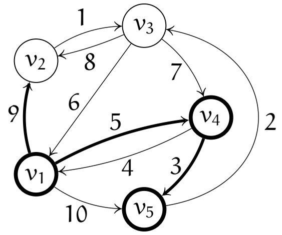
Po dołączeniu wierzchołka w do zbioru S uaktualniamy wartości atrybutu `d` wszystkich wierzchołków, do których z wierzchołka w prowadzi krawędź. W ten sposób dla każdego wierzchołka $v \in V \setminus S$, do którego najkrótsza ścieżka z u przechodzi przez w, atrybut v.`d` staje się równy długości tej ścieżki. Atrybut `d` każdego wierzchołka, do którego najkrótsza znaleziona dotąd ścieżka z u przechodzi tylko przez elementy S inne niż w, ma wartość niezmienioną. W ten sposób po wykonaniu k-tego kroku warunek indukcyjny dla $k+1$ jest spełniony.



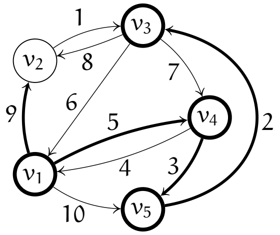
d: 0 9 ∞ 5 10



d: 0 9 ∞ 5 8



d: 0 9 10 5 8



d: 0 9 10 5 8

Przykład. Rysunek przedstawia graf skierowany, dla którego należy znaleźć najkrótsze ścieżki z wierzchołka v_1 do wszystkich pozostałych. Wierzchołki należące do zbioru S w kolejnych krokach i znalezione krawędzie drzewa najkrótszych ścieżek z v_1 są zaznaczone grubą linią. Pod obrazem grafu są wypisane wartości atrybutu d wszystkich wierzchołków.

Po sprawdzeniu, że algorytm jest poprawny, pora ustalić szczegóły implementacji i zbadać złożoność obliczeniową. Do reprezentacji zbioru $V \setminus S$ użyjemy kopca (zrealizowanego w tablicy), w taki sam sposób, jak w jednej z przedstawionych wcześniej na wykładzie implementacji algorytmu Prima. W linii 6 mamy operację wyjęcia z kolejki priorytetowej elementu o największym priorytecie. W linii 9 atrybut `d` otrzymuje nową wartość, co wiąże się z uaktualnieniem priorytetu wierzchołka. W podanym pseudokodzie brakuje tego szczegółu, ale w implementacji musi on być obecny. Ponieważ wartość atrybutu `d` może tylko maleć, czyli priorytet wierzchołka może wzrosnąć, porządkowanie kopca może być wykonane za pomocą procedury `UpHeap`. W tablicy wierzchołków grafu należy pamiętać indeksy pozycji wierzchołków w kopcu i odpowiednio uaktualniać te indeksy podczas przestawiania elementów kopca.

Kopiec początkowo zawiera n wierzchołków grafu, a jego zbudowanie może być wykonane w czasie $O(n)$ (zauważmy, że początkowo tylko jeden wierzchołek, u, ma atrybut $d = 0$, a pozostałe mają $d = \infty$, zatem nie trzeba nawet korzystać ze sposobu opisanego w pierwszym semestrze). W każdym kroku usuwamy jeden wierzchołek z kopca; sumaryczny koszt tych operacji jest taki jak algorytmu HeapSort, tj. $O(n \log n)$. Operacja porządkowania kopca po zmianie priorytetu wierzchołka może zająć co najwyżej $O(\log n)$ operacji, przy czym operacja ta jest wykonywana jeden raz dla każdej krawędzi. Ostatecznie, dla grafu G, który ma n wierzchołków i m krawędzi, złożoność pesymistyczna implementacji algorytmu Dijkstry z użyciem kopca jest równa $O((n+m)\log n)$. Jeśli $m = O(n)$, to złożoność algorytmu Dijkstry nie przewyższa $O(n \log n)$. Dla $m = O(n^2)$ mamy złożoność algorytmu $O(n^2 \log n)$.

Algorytm Floyda

Algorytm Floyda służy do znalezienia najkrótszych ścieżek między wszystkimi parami wierzchołków grafu. Algorytm ten dopuszcza występowanie w grafie krawędzi o ujemnych wagach. Niech n oznacza liczbę wierzchołków grafu G, które ponumerujemy od 1 do n. Dla uproszczenia rozpatrujemy graf skierowany prosty bez pętli; dla dowolnego grafu możemy odrzucić wszystkie pętle oraz krawędzie o tym samym początku i końcu z wyjątkiem jednej, o najmniejszej wadze.

Określimy rekurencyjnie ciąg macierzy $A^{(0)}, \ldots, A^{(n)}$:

$$A^{(0)} = [a_{ij}^{(0)}]_{i,j}: \quad a_{ij}^{(0)} = \begin{cases} 0 & \text{jeśli } i = j, \\ \text{waga krawędzi } (v_i, v_j) & \text{jeśli istnieje taka krawędź,} \\ \infty & \text{w przeciwnym razie.} \end{cases}$$
$$A^{(k)} = [a_{ij}^{(k)}]_{i,j}: \quad a_{ij}^{(k)} = \min\{a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\}, \qquad \text{dla } k = 1, \ldots, n.$$

Współczynniki macierzy $A^{(0)}$ opisują długości wszystkich ścieżek złożonych z jednej krawędzi. Przyjmijmy założenie indukcyjne, że współczynnik $a_{ij}^{(k-1)}$ jest długością najkrótszej ścieżki od wierzchołka v_i do v_j, której wszystkie pozostałe wierzchołki należą do zbioru $\{v_1, \ldots, v_{k-1}\}$ (jeśli taka ścieżka nie istnieje, to $a_{ij}^{(k-1)} = \infty$). Przypuśćmy, że ścieżka od v_i do v_j, która przechodzi tylko przez wierzchołki ze zbioru $\{v_1, \ldots, v_k\}$ istnieje. Wtedy najkrótsza taka ścieżka też istnieje i albo przechodzi przez wierzchołek v_k, albo nie. W pierwszym przypadku długość tej ścieżki jest sumą długości ścieżek od v_i do v_k i od v_k do v_j, a w drugim jej długość jest równa $a_{ij}^{(k-1)}$. Wzór określający współczynnik $a_{ij}^{(k)}$ wyraża zatem długość najkrótszej ścieżki od v_i do v_j, przechodzącej tylko przez wierzchołki należące do zbioru $\{v_1, \ldots, v_k\}$ (i jest poprawny także wtedy, gdy taka ścieżka nie istnieje).

<u>Przykład.</u> Dla grafu, który posłużył jako ilustracja działania algorytmu Dijkstry, otrzymujemy następujące macierze:

$$A^{(0)} = \begin{bmatrix} 0 & 9 & \infty & 5 & 10 \\ \infty & 0 & 1 & \infty & \infty \\ 6 & 8 & 0 & 7 & \infty \\ 4 & \infty & \infty & 0 & 3 \\ \infty & \infty & 2 & \infty & 0 \end{bmatrix}, \quad A^{(1)} = \begin{bmatrix} 0 & 9 & \infty & 5 & 10 \\ \infty & 0 & 1 & \infty & \infty \\ 6 & 8 & 0 & 7 & 16 \\ 4 & 13 & \infty & 0 & 3 \\ \infty & \infty & 2 & \infty & 0 \end{bmatrix},$$
$$A^{(2)} = \begin{bmatrix} 0 & 9 & 10 & 5 & 10 \\ \infty & 0 & 1 & \infty & \infty \\ 6 & 8 & 0 & 7 & 16 \\ 4 & 13 & 14 & 0 & 3 \\ \infty & \infty & 2 & \infty & 0 \end{bmatrix}, \quad A^{(3)} = \begin{bmatrix} 0 & 9 & 10 & 5 & 10 \\ 7 & 0 & 1 & 8 & 17 \\ 6 & 8 & 0 & 7 & 16 \\ 4 & 13 & 14 & 0 & 3 \\ 8 & 10 & 2 & 9 & 0 \end{bmatrix},$$
$$A^{(4)} = \begin{bmatrix} 0 & 9 & 10 & 5 & 8 \\ 7 & 0 & 1 & 8 & 11 \\ 6 & 8 & 0 & 7 & 10 \\ 4 & 13 & 14 & 0 & 3 \\ 8 & 10 & 2 & 9 & 0 \end{bmatrix}, \quad A^{(5)} = \begin{bmatrix} 0 & 9 & 10 & 5 & 8 \\ 7 & 0 & 1 & 8 & 11 \\ 6 & 8 & 0 & 7 & 10 \\ 4 & 13 & 5 & 0 & 3 \\ 8 & 10 & 2 & 9 & 0 \end{bmatrix}.$$

Podstawą algorytmu Floyda jest obliczanie macierzy $A^{(0)}, \ldots, A^{(n)}$; ostatnia z tych macierzy opisuje długości wszystkich najkrótszych ścieżek. Wszystkie te macierze

mogą być przechowywane w jednej tablicy $n \times n$. Możemy bowiem zauważyć, że dla każdego $i, j \in \{1, \ldots, n\}$ jest $a_{ik}^{(k)} = a_{ik}^{(k-1)}$ oraz $a_{kj}^{(k)} = a_{kj}^{(k-1)}$. Do obliczenia współczynnika $a_{ij}^{(k)}$ dla $i \neq k \neq j$ potrzebne są tylko współczynniki $a_{ik}^{(k-1)}$ i $a_{kj}^{(k-1)}$ (które „przechodzą" do macierzy $A^{(k)}$), oraz $a_{ij}^{(k-1)}$, który możemy zastąpić przez $a_{ij}^{(k)}$, bo do niczego później się nie przyda.

Oprócz długości najkrótszych ścieżek ważne jest uzyskanie informacji umożliwiającej odtworzenie najkrótszej ścieżki o wskazanym początku i końcu. Informację taką możemy przechowywać w macierzy $B = [b_{ij}]_{i,j}$, złożonej z liczb od 0 do n, określonych następująco: $b_{ij} = 0$, jeśli najkrótsza ścieżka z v_i do v_j składa się z jednej krawędzi lub nie istnieje, oraz $b_{ij} = k$, gdzie $k \notin \{i, j\}$ jest numerem jednego z wierzchołków, przez które przechodzi najkrótsza ścieżka z v_i do v_j. Zauważmy, że obliczając współczynniki macierzy $A^{(k)}$, możemy przypisać elementowi `b[i][j]` tablicy (zawierającej początkowo same zera) wartość k, jeśli stwierdzimy, że najkrótsza ścieżka znaleziona dotąd przechodzi przez v_k.

```
void Floyd ( graf G, float a[n+1][n+1], int b[n+1][n+1] )
{
  for ( i = 1; i <= n; i++ ) {
    for ( j = 1; j <= n; j++ )
      { a[i][j] = ∞;  b[i][j] = 0; }
    a[i][i] = 0;
  }
  for ( e wskazuje kolejno każdą krawędź (vi, vj) ∈ E )
    a[i][j] = waga krawędzi *e;
  for ( k = 1; k <= n; k++ )
    for ( i = 1; i <= n; i++ )
      for ( j = 1; j <= n; j++ )
        if ( a[i][k]+a[k][j] < a[i][j] ) {
          a[i][j] = a[i][k] + a[k][j];
          b[i][j] = k;
        }
} /*Floyd*/
```

Zgodnie z wcześniejszymi rozważaniami, po wykonaniu podanej wyżej procedury tablica a zawiera długości najkrótszych ścieżek, zaś `b[i][j]`, jeśli jest różne od 0, to jest numerem pewnego wierzchołka, przez który przechodzi ścieżka z v_i do v_j. Do wyprowadzenia wierzchołków najkrótszej ścieżki z v_i do v_j służy następująca procedura rekurencyjna (która wyprowadza numery wierzchołków, z wyjątkiem początku i końca):

```
void WyprowadźŚcieżkęF ( int b[n+1][n+1], int i, int j )
{
  if ( b[i][j] ) {
    WyprowadźŚcieżkęF ( b, i, b[i][j] );
    wyprowadź ( b[i][j] );
    WyprowadźŚcieżkęF ( b, b[i][j], j );
  }
} /*WyprowadźŚcieżkęF*/
```

Złożoność obliczeniowa algorytmu Floyda jest, jak łatwo zauważyć, rzędu $O(n^3+m)$. Jeśli liczba krawędzi grafu $m = O(n^2)$, to zastosowanie algorytmu Dijkstry do znalezienia najkrótszych ścieżek dla wszystkich par wierzchołków wymagałoby wykonania $O(n^3 \log n)$ operacji (algorytm Dijkstry trzeba by wykonać n razy, przyjmując kolejno każdy wierzchołek za źródło). Zatem algorytm Floyda dla grafów (prawie) pełnych ma mniejszy rząd złożoności. Jeśli jednak liczba krawędzi jest rzędu $O(n)$, to n-krotne wykonanie algorytmu Dijkstry jest bardziej opłacalne. Oczywiście, jeśli w grafie występują krawędzie o ujemnych wagach, to nie możemy użyć algorytmu Dijkstry.

Zadania i problemy

1. Prześledź działanie algorytmu Bellmana-Forda na przykładach. Rozważ w nich grafy z krawędziami o ujemnych wagach.
2. Dla bezcyklowego grafu skierowanego można znaleźć najkrótsze ścieżki, dokonując relaksacji tylko raz dla każdej krawędzi, przy założeniu odpowiedniej kolejności krawędzi. Zbadaj, jaka to kolejność i wykaż prawdziwość poprzedniego zdania.

   Wskazówka: jest ono związane z topologicznym uporządkowaniem zbioru wierzchołków grafu.
3. Użyj algorytmu Dijkstry do znalezienia najkrótszych ścieżek w grafie użytym jako przykład na wykładzie, z wierzchołka v_2 do wszystkich pozostałych.
4. Znajdź macierz B (zawierającą informacje o najkrótszych ścieżkach) dla tego samego grafu, przy użyciu algorytmu Floyda.
5. Bardzo podobnie do algorytmu Floyda działa algorytm Warshalla, którego celem jest stwierdzenie dla każdej pary wierzchołków, czy istnieje ścieżka, której pierwszy wierzchołek jest początkiem, a drugi końcem.
   Macierze $A^{(0)} = [a_{ij}^{(0)}]_{i,j}, \ldots, A^{(n)} = [a_{ij}^{(n)}]_{i,j}$ są określone wzorem

   $$a_{ij}^{(0)} = \begin{cases} 0 & \text{jeśli } i = j, \\ 1 & \text{jeśli } (v_i, v_j) \in E, \\ \infty & \text{w przeciwnym razie.} \end{cases}$$
   $$a_{ij}^{(k)} = \min\{a_{ij}^{(k-1)}, (a_{ik}^{(k-1)} + a_{kj}^{(k-1)})/2\}$$

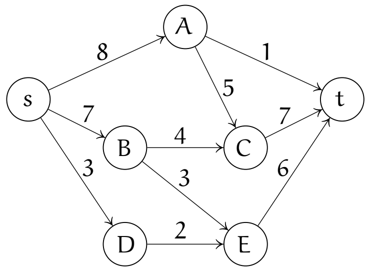
   Wszystkie współczynniki macierzy są zatem równe 0, 1 lub ∞. Udowodnij, że jeśli $i \neq j$, to ścieżka z v_i do v_j istnieje wtedy i tylko wtedy, gdy $a_{ij}^{(n)} = 1$.
6. Bezcyklowy graf skierowany określa częściowy porządek w zbiorze wierzchołków grafu, ale dwa wierzchołki mogą być w tej relacji także wtedy, gdy nie są połączone krawędzią (natomiast istnieje łącząca je ścieżka). Pokaż, w jaki sposób można użyć algorytmu Warshalla do uzyskania pełnej informacji o tej relacji.
7. Niech $G = (V, E)$ oznacza graf skierowany (niekoniecznie prosty), którego macierzą sąsiedztwa jest macierz A. Udowodnij, że dla dowolnej liczby naturalnej k (dopuszczamy także $k = 0$) współczynnik $a_{ij}^{(k)}$ macierzy $A^{(k)} = A^k$ jest liczbą różnych dróg o długości k od wierzchołka v_i do v_j.

   Zaproponuj algorytm o złożoności rzędu $O(n^3 \log k)$ obliczania liczby wszystkich dróg o długości k w danym grafie.

Sieci przepływowe

Niech $G = (V, E)$ oznacza graf skierowany prosty, w którym każdej krawędzi jest przyporządkowana całkowita liczba dodatnia, tzw. przepustowość (albo pojemność). Graf, który nie jest prosty, możemy zastąpić skierowanym grafem prostym, w którym wszystkim krawędziom o tym samym początku i tym samym końcu odpowiada jedna krawędź — jej przepustowość jest sumą przepustowości zastąpionych krawędzi. Założymy, że graf nie ma pętli (jeśli ma, możemy je odrzucić).

Graf skierowany prosty bez pętli G, którego krawędzie mają określoną przepustowość, nazywa się siecią przepływową.



Określimy funkcję $c\colon V \times V \to \mathbb{R}$, tzw. funkcję przepustowości sieci, wzorem

$$c(u, v) = \begin{cases} \text{przepustowość krawędzi } (u, v) & \text{jeśli } (u, v) \in E, \\ 0 & \text{w przeciwnym razie.} \end{cases}$$

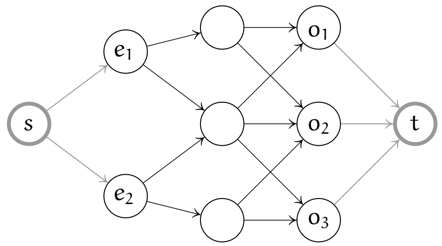
Wyróżnimy dwa wierzchołki sieci: s, zwany źródłem i t, zwany ujściem.

Def. Przepływem w sieci przepływowej G między źródłem s i ujściem t nazywamy dowolną funkcję $f\colon V \times V \to \mathbb{R}$ spełniającą następujące warunki:

- $\forall_{u,v \in V}\ f(u, v) \le c(u, v)$.
- $\forall_{u,v \in V}\ f(u, v) = -f(v, u)$.
- $\forall_{u \in V \setminus \{s,t\}}\ \sum_{v \in V} f(u, v) = 0$.

Przykładem sieci przepływowej jest sieć energetyczna z jedną elektrownią i jednym odbiorcą energii. Przypuśćmy, że istnieje wiele przewodów łączących elektrownię z odbiorcą poprzez wiele stacji transformatorów. Przepływ w takiej sieci opisuje moc przesyłaną przez poszczególne przewody (krawędzie grafu), przy

czym każdy przewód ma ograniczenie mocy, którą można przez niego przesyłać. Podane warunki, które spełnia przepływ, możemy zinterpretować następujaco: przez żaden przewód nie przesyłamy mocy większej od przepustowości (próba przekroczenia tego ograniczenia spowodowałaby awarię lub katastrofę), jeśli z wierzchołka u do v przepływa moc $f(u, v)$, to w drugą stronę przesyłamy $-f(u, v)$ (zamiast o energii elektrycznej można myśleć o pieniądzach: jeśli v dostał od u $f(u, v)$ złotych, to u stał się o tyle złotych uboższy z powodu v) i wreszcie, z wyjątkiem źródła i ujścia, w żadnym wierzchołku energia nie jest gromadzona, marnowana, ani produkowana dodatkowo (taka sieć jest bezstratna, tj. cała moc wysyłana z elektrowni dochodzi do odbiorcy).



Zauważmy, że jeśli mamy wiele elektrowni i odbiorców energii (ta sytuacja jest bardziej życiowa), to do grafu, którego wierzchołkami są elektrownie, stacje transformatorów i odbiorcy, a krawędziami są przewody, możemy dołączyć dwa wierzchołki, które uznamy za źródło i ujście. Dodamy też krawędzie wychodzące ze źródła do wszystkich elektrowni, oraz krawędzie ze wszystkich odbiorców do ujścia. Przyjmiemy, że przepustowość krawędzi łączącej źródło z elektrownią jest równa maksymalnej mocy, jaką ta elektrownia może wytwarzać, i podobnie przepustowość krawędzi od dowolnego odbiorcy do ujścia odpowiada maksymalnemu zapotrzebowaniu na moc tego odbiorcy. Dla tak rozszerzonego grafu możemy określić przepływ ze źródła do ujścia, a następnie zbadać, jaka moc przepływa przez dowolną krawędź reprezentującą istniejący przewód.

Problem maksymalnego przepływu polega na znalezieniu (dla ustalonej sieci przepływowej ze źródłem s i ujściem t) przepływu f, takiego że liczba

$$|f| \stackrel{\text{def}}{=} \sum_{v \in V} f(s, v)$$

jest największa. Taki tzw. maksymalny przepływ oznaczymy symbolem f^*. Dla dowolnego przepływu f liczbę $|f|$ nazwiemy sumą przepływu.

Dla uproszczenia rachunków przyjmiemy następujące oznaczenie: jeśli $X, Y \subset V$, to

$$f(X, Y) \stackrel{\text{def}}{=} \sum_{x \in X} \sum_{y \in Y} f(x, y),$$

a ponadto dla zbiorów jednoelementowych zamiast $\{x\}$ będziemy pisać x. W tej notacji mamy

$$|f| = f(s, V).$$

Podobnie jak dla przepływów, również dla funkcji przepustowości sieci przyjmiemy notację $c(X, Y) = \sum_{x \in X} \sum_{y \in Y} c(x, y)$.

Stwierdzenie. Dowolny przepływ spełnia następujące warunki:

1. $\forall_{u \notin \{s,t\}}\, f(u, V) = 0$.
2. $\forall_{X,Y \subset V}\, f(X, Y) = -f(Y, X)$.
3. $\forall_{X \subset V}\, f(X, X) = 0$.
4. $\forall_{X,Y,Z \subset V}\, f(X, Y \cup Z) = f(X, Y) + f(X, Z) - f(X, Y \cap Z)$.
   W szczególności, jeśli $Y \cap Z = \emptyset$, to $f(X, Y \cup Z) = f(X, Y) + f(X, Z)$.
5. $\forall_{X,Y,Z \subset V}\, f(X \cup Y, Z) = f(X, Z) + f(Y, Z) - f(X \cap Y, Z)$.
   W szczególności, jeśli $X \cap Y = \emptyset$, to $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$.
6. $|f| = f(V, t)$.

Dowód. 1. jest innym sformułowaniem warunku $\sum_{v \in V} f(u, v) = 0$ dla $u \notin \{s, t\}$. 2. i 3. wynikają z antysymetrii, tj. warunku $f(u, v) = -f(v, u)$. Do dowodu 4. weźmy $x \in X$; możemy wtedy obliczyć

$$f(x, Y \cup Z) = \sum_{y \in Y \cup Z} f(x, y) = \sum_{y \in Y} f(x, y) + \sum_{y \in Z} f(x, y) - \sum_{y \in Y \cap Z} f(x, y).$$

Następnie wystarczy dokonać sumowania stron równości po wszystkich $x \in X$. Dowód 5. wygląda analogicznie. Wreszcie, z 1. wynika, że $f(V \setminus \{s, t\}, V) = 0$, zatem

$$\begin{aligned} |f| &= f(s, V) \\ &= f(s, V) + f(V \setminus \{s, t\}, V) \\ &= f(V \setminus \{t\}, V) \\ &= f(V \setminus \{t\}, V) - f(V \setminus \{t\}, V \setminus \{t\}) \\ &= f(V, t). \end{aligned}$$

Udowodniona w ten sposób własność 6. oznacza, że wszystko co opuszcza źródło, dociera do ujścia i tam znika. □

Stwierdzenie. Niech $S, T \subset V$, $S \cup T = V$, $S \cap T = \emptyset$, $s \in S$, $t \in T$. Dla dowolnego przepływu f zachodzi równość $|f| = f(S, T)$.

Dowód.

$$\begin{aligned} f(S, T) &= f(S, V) - f(S, S) \\ &= f(S, V) \\ &= f(s, V) + f(S \setminus \{s\}, V) \\ &= f(s, V) = |f|. \quad \square \end{aligned}$$

Podział zbioru wierzchołków V na dwa zbiory S i T spełniające założenia ostatniego stwierdzenia, będziemy nazywać przekrojem. Jak widać, suma przepływu między wierzchołkami dowolnego przekroju jest taka sama.

Def. Niech f będzie przepływem w ustalonej sieci $G = (V, E)$. Siecią residualną dla przepływu f nazywamy sieć $G_f = (V, E_f)$, której odpowiada funkcja przepustowości sieci $c_f \colon V \times V \to \mathbb{R}$, taka że $\forall_{u,v \in V}\, c_f(u, v) = c(u, v) - f(u, v)$, przy czym $E_f = \{\, (u, v) \in V \times V \colon c_f(u, v) > 0 \,\}$.

Jest jasne, że sieć residualna jest również siecią przepływową.

Uwaga: Funkcja c_f przyjmuje tylko wartości nieujemne, natomiast zbiór E_f *może nie być* podzbiorem E.

Stwierdzenie. Niech f będzie przepływem w sieci G. Jeśli $\tilde{f}$ jest przepływem w sieci residualnej G_f, to funkcja $f + \tilde{f}$ (określona wzorem $(f + \tilde{f})(u, v) = f(u, v) + \tilde{f}(u, v)$ dla każdego $u, v \in V$) jest również przepływem w sieci G, a ponadto $|f + \tilde{f}| = |f| + |\tilde{f}|$.

Dowód. Jeśli $(u, v) \in E \cap E_f$, to

$$\begin{aligned} (f + \tilde{f})(u, v) &= f(u, v) + \tilde{f}(u, v) \\ &\le f(u, v) + c_f(u, v) \\ &= f(u, v) + c(u, v) - f(u, v) = c(u, v). \end{aligned}$$

Dla $(u, v) \notin E_f$ musi być $\tilde{f}(u, v) \le 0$, a zatem $(f + \tilde{f})(u, v) \le f(u, v) \le c(u, v)$. Warunek $(u, v) \notin E$ oraz $(u, v) \in E_f$ jest możliwy wtedy i tylko wtedy, gdy

$(v, u) \in E$ i wtedy $c_f(u, v) = f(v, u) \leq c(v, u)$, zatem $(f + \tilde{f})(v, u) \leq c(v, u)$. Tak więc dla każdej krawędzi (u, v) sieci G mamy $f(u, v) \leq c(u, v)$. Suma $f + \tilde{f}$ antysymetrycznych funkcji f i $\tilde{f}$ jest oczywiście funkcją antysymetryczną; trzeci warunek podany w definicji przepływu jest również spełniony w sposób oczywisty, tak samo jak fakt, że suma przepływu $f + \tilde{f}$ jest sumą sum przepływów f i $\tilde{f}$. □

Def. Ścieżką powiększającą nazywamy dowolną ścieżkę, prowadzącą ze źródła s do ujścia t w sieci residualnej G_f. Przepustowość ścieżki jest to najmniejsza przepustowość krawędzi wchodzącej w skład ścieżki.

Twierdzenie. Następujące warunki są równoważne:

1. Przepływ f jest maksymalnym przepływem w sieci G.
2. Sieć residualna G_f nie zawiera ścieżek powiększających.
3. $|f| = c(S, T)$ dla pewnego przekroju (S, T).

Dowód. (1.⇒2.) Przypuśćmy, że $f = f^*$ (tj. że przepływ f jest maksymalny). Gdyby w sieci G_f istniała ścieżka powiększająca, to odpowiadający jej przepływ spełniałby warunek $|f_p| > 0$, a zatem przepływ $f + f_p$ miałby większą sumę.
(2.⇒3.) Niech

$$S = \{v \in V\colon \text{w } G_f \text{ jest ścieżka z } s \text{ do } v\}, \quad T = V \setminus S.$$

Jeśli w G_f nie ma ścieżki powiększającej, to $t \notin S$, czyli $t \in T$, zatem podział zbioru wierzchołków na podzbiory S i T jest przekrojem, a ponadto zbiór E_f nie zawiera żadnej krawędzi (u, v), takiej że $u \in S$ oraz $v \in T$. Zatem dla każdego $u \in S$ oraz $v \in T$ jest

$$0 = c_f(u, v) = c(u, v) - f(u, v),$$

skąd wynika $|f| = f(S, T) = c(S, T)$.
(3.⇒1.) Dla dowolnego przepływu f mamy

$$|f| = f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) \leq \sum_{u \in S} \sum_{v \in T} c(u, v) = c(S, T).$$

Jeśli więc $|f| = c(S, T)$, to f jest przepływem maksymalnym. □

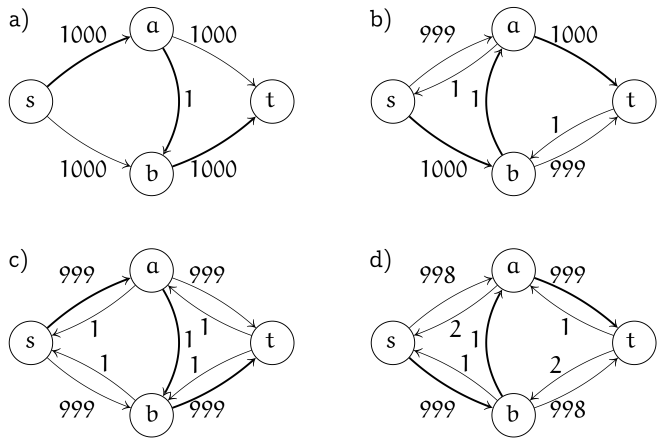
Algorytm Forda-Fulkersona

Przypuśćmy, że znamy pewien przepływ f (przepływem jest w szczególności funkcja zerowa, i możemy od niej zacząć poszukiwanie maksymalnego przepływu).

Możemy znaleźć dowolną ścieżkę powiększającą p w sieci G_f i dodać do f przepływ f_p, który dla każdej krawędzi tej ścieżki przyjmuje wartość równą jej przepustowości (a dla wszystkich pozostałych krawędzi ma wartość 0 — te dwa warunki określają jednoznacznie przepływ f_p, nazwiemy go przepływem przez ścieżkę p). Czynność tę możemy powtarzać iteracyjnie. Zatem, mamy algorytm:

1. Przyjmij $f = 0$ oraz $G_f = G$.
2. Dopóki istnieje ścieżka powiększająca p w G_f, powtarzaj w pętli
3. Znajdź przepływ f_p przez ścieżkę p.
4. Przypisz $f := f + f_p$ i wyznacz sieć residualną G_f.

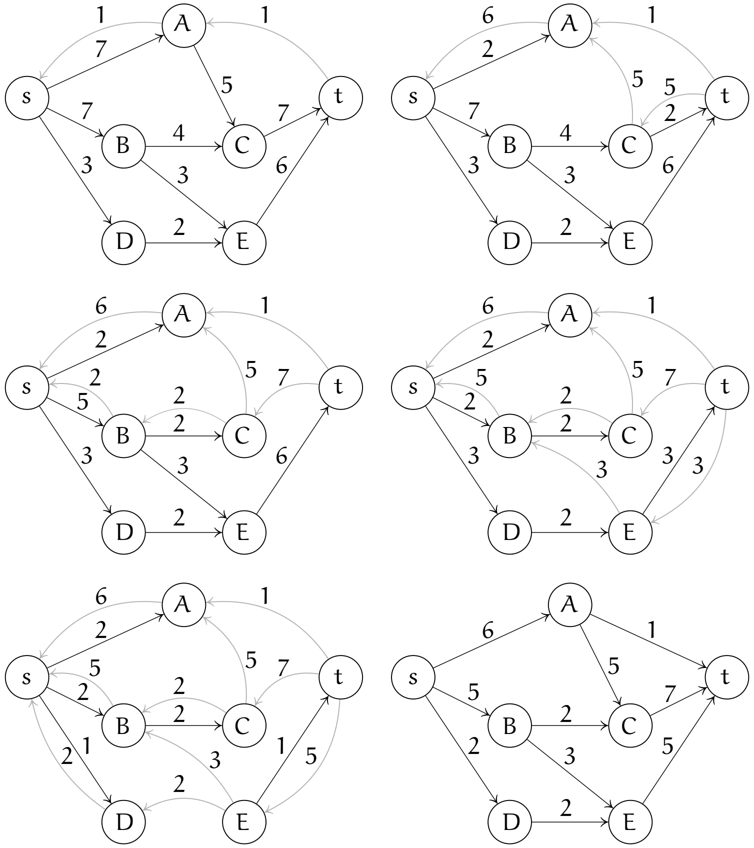
Jeśli algorytm ten zakończy działanie, to z udowodnionego wcześniej twierdzenia wynika, że znajdzie on przepływ maksymalny. Zauważmy, że ciąg sum kolejno wyznaczonych przepływów jest monotonicznie rosnący. Każda ścieżka w sieci residualnej zawiera krawędź należącą do zbioru E krawędzi oryginalnej sieci. Stąd jeszcze nie wynika, że algorytm zakończy działanie po skończenie wielu krokach, ale rozpatrujemy przepustowości o wartościach całkowitych. Suma f_p jest zawsze całkowita, i stąd już własność stopu wynika.



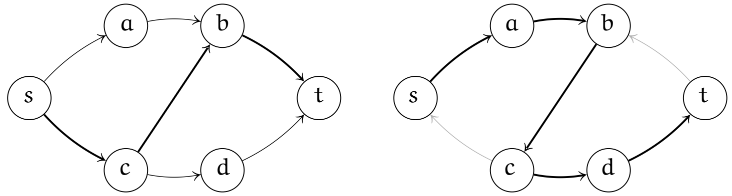
Zobaczmy przykład na rysunku. W sieci na rysunku a) ścieżka $s - a - b - t$ ma przepustowość taką, jak krawędź (a, b), czyli 1. Jeśli algorytm zacznie znajdowanie przepływu maksymalnego od niej, to w sieci residualnej na rysunku b) pojawi się ścieżka $s - b - a - t$ o przepustowości 1, której przetworzenie w następnym kroku da w wyniku sieć residualną c) itd. Przykład ten pokazuje, że nawet w sieci o małej liczbie wierzchołków podstawowy pomysł może doprowadzić do bardzo długich obliczeń.

Aby zminimalizować liczbę kroków algorytmu, należy spośród ścieżek powiększających istniejących w danym kroku wybrać jedną ze ścieżek o najmniejszej liczbie krawędzi. W przykładzie rozpatrywanym wyżej algorytm oparty na tym pomyśle zakończy działanie po dwóch krokach. Można udowodnić, że taki algorytm dla grafu, który ma n wierzchołków i m krawędzi, wykona co najwyżej $O(nm)$ kroków. Można też usprawnić znajdowanie ścieżek (nie musimy w każdej kolejnej sieci residualnej wyszukiwać ścieżki „od zera”).

Działanie algorytmu dla sieci podanej na początku wykładu jest pokazane na rysunku.



Zobaczmy jeszcze jeden przykład, który uwidocznia potrzebę wprowadzenia krawędzi o przeciwnej orientacji w sieci residualnej:



Przypuśćmy, że wszystkie krawędzie mają przepustowość 1. Jeśli w pierwszym kroku została wybrana ścieżka powiększająca $s - c - b - t$, to bez krawędzi (b, c) w sieci residualnej nie można byłoby znaleźć maksymalnego przepływu. Zauważmy, że wszystkie ścieżki powiększające w sieci danej w tym przykładzie mają tę samą długość; algorytm Forda-Fulkersona może w pierwszym kroku wybrać dowolną z nich.

Skojarzenia w grafie dwudzielnym

Def. Graf dwudzielny $G = (V, E)$ jest to graf prosty, taki że $V = S \cup T$, $S \cap T = \emptyset$, oraz każda krawędź $\{u, v\} \in E$ ma jeden wierzchołek ze zbioru S, a drugi należy do T.

Jednym z praktycznych problemów grafowych jest znalezienie skojarzenia w grafie dwudzielnym, tj. podgrafu $G' = (V, E')$, takiego że każdy wierzchołek jest incydentny z dokładnie jedną krawędzią. Można też rozpatrywać „jednostronne” skojarzenie, w którym każdy wierzchołek zbioru S jest incydentny z jedną krawędzią. Oczywiście, nie każdy graf dwudzielny ma skojarzenie, np. nie istnieje skojarzenie w grafie, w którym pewien wierzchołek nie jest incydentny z żadną krawędzią.

O istnieniu skojarzenia orzeka twierdzenie Halla, znane też jako twierdzenie o małżeństwach (interpretujemy wierzchołki należące do zbiorów S i T odpowiednio jako chłopców i dziewczyny, a krawędzie grafu jako znajomości — mamy wszystkich ożenić, ale wykluczone są małżeństwa osób, które się nie znają).

Twierdzenie Halla. Skojarzenie w grafie dwudzielnym istnieje wtedy i tylko wtedy, gdy dla każdego podzbioru S' zbioru S liczba wierzchołków należących do T, połączonych krawędziami z wierzchołkami należącymi do S' jest nie mniejsza niż k, gdzie k jest liczbą elementów zbioru S'.

Inaczej to formułując, wszystkich chłopców można ożenić wtedy, gdy dla każdego $k \leq |S|$ dowolnie wybrani k chłopcy znają w sumie co najmniej k dziewczyn. Niezbyt trudny dowód indukcyjny (ze względu na liczbę wierzchołków zbioru S) pominiemy (ewentualnie zostawimy na ćwiczenia). Zauważmy natomiast, że problematyczna jest konstruktywność tego twierdzenia; mimo, że zbiór wierzchołków jest skończony, to sprawdzenie, czy dany graf spełnia warunek istnienia skojarzenia, może być bardzo czasochłonne (liczba różnowartościowych odwzorowań n-elementowego zbioru T w m-elementowy zbiór S jest równa $\frac{m!}{(m-n)!}$, liczba różnych podzbiorów zbioru n-elementowego jest równa 2^n). Możemy jednak problem skojarzeń sprowadzić do innego problemu, który umiemy rozwiązać w czasie wielomianowym. Tym problemem jest znalezienie maksymalnego przepływu.

Sprowadzenie zadania znalezienia skojarzenia do zadania znalezienia maksymalnego przepływu wygląda tak: Nadajemy krawędziom orientację (tak, aby początek każdej krawędzi należał do S, a koniec do T). Do grafu dodajemy dwa nowe wierzchołki, s i t, które będą źródłem i ujściem. Źródło s łączymy krawędziami ze wszystkimi wierzchołkami zbioru S; podobnie, wszystkie wierzchołki należące do T łączymy krawędziami z ujściem. Każdej krawędzi otrzymanego grafu przypisujemy przepustowość 1 i znajdujemy maksymalny przepływ w powstałej sieci. Jeśli suma przepływu jest równa liczbie wierzchołków w zbiorze S, to każdy taki wierzchołek jest skojarzony z pewnym wierzchołkiem należącym do T, przy czym krawędzie należące do skojarzenia to te, dla których znaleziony przepływ ma wartość 1.

Uwagi o klasach złożoności obliczeniowej zadań

Rozpatrywaliśmy przed chwilą dwa związane ze sobą zadania, z których jedno (znalezienie maksymalnego przepływu) jest optymalizacyjne, a drugie (znalezienie skojarzenia) jest decyzyjne. Rozwiązanie zadania decyzyjnego polega na udzieleniu odpowiedzi na pytanie, czy skojarzenie w grafie dwudzielnym istnieje. Odpowiedź jest twierdząca lub przecząca, przy czym musi istnieć odpowiednie „świadectwo". Takim „świadectwem" dla odpowiedzi twierdzącej jest konkretne skojarzenie, czyli zbiór par odpowiednich wierzchołków. „Świadectwem" dla odpowiedzi przeczącej jest zbiór $S' \subset S$ wierzchołków, które są połączone z mniej licznym zbiorem $T' \subset T$. Jeśli ktoś nam poda odpowiednie „świadectwo", to możemy łatwo sprawdzić, czy jest ono poprawne (co jest dużo łatwiejsze niż znalezienie samemu takiego świadectwa) i udzielić odpowiedzi na problem decyzyjny. W teorii złożoności obliczeniowej istotną rolę odgrywa sprowadzanie jednych problemów do drugich (np. problemów optymalizacyjnych do decyzyjnych).

Wyróżnia się klasy problemów, które dają się rozwiązać w czasie odpowiednio zależnym od rozmiaru n zadania, przy czym z uwagi na możliwość sprowadzenia rozwiązania danego problemu do rozwiązania innego, rozpatruje się (dla wygody) głównie problemy decyzyjne.

Klasa P to problemy, dla których istnieją algorytmy, które w czasie wielomianowym (stopień wielomianu jest dla teorii złożoności mało istotny, choć w praktyce ma spore znaczenie) znajdują „świadectwo" rozwiązania lub „świadectwo" jego nieistnienia. Wszystkie algorytmy omówione wcześniej na tym wykładzie działają w czasie wielomianowym, w związku z czym zadania rozwiązywane przez te algorytmy należą do klasy P. Problemy należące do klasy P można uznać za praktycznie rozwiązywalne. Nazwa klasy, P, pochodzi od słowa *polynomial*, tj. wielomian.

Klasa NP obejmuje problemy, dla których daje się w czasie wielomianowym zweryfikować poprawność „świadectwa" (czyli na przykład sprawdzić, czy podany wynik jest rozwiązaniem). Przypuśćmy, że wynik („świadectwo") jest zakodowany jako pewien ciąg bitów o długości n. Możemy sobie wyobrazić pełne drzewo binarne o wysokości n; każdy kolejny bit określa, czy w drodze od korzenia do liścia mamy pójść w lewo, czy w prawo. Niektóre liście odpowiadają rozwiązaniom zadania (poprawnym „świadectwom"), ale sprawdzenie, czy dany ciąg bitów jest poprawnym „świadectwem" może nastąpić dopiero wtedy, gdy znamy cały ten ciąg. Tymczasem różnych ciągów bitów o długości n jest 2^n i sprawdzenie ich wszystkich już dla n rzędu kilkadziesiąt jest niewykonalne. Na przykład, mając dwie liczby naturalne, łatwo możemy sprawdzić, czy jedna jest dzielnikiem drugiej, ale zupełnie innym problemem jest znalezienie rozkładu liczby danej na czynniki.

Nazwa klasy, NP, pochodzi od określenia *nondeterministic polynomial*. Poszukując „świadectwa" możemy w każdym kroku, w którym należy podjąć decyzję (np. w drzewie opisanym wyżej), wykazywać „niezdecydowanie" i iść we wszystkie strony. Oczywiście, klasa P jest zawarta w NP. Istnieją też problemy nienależące do klasy NP.

W związku z tym, że pewne problemy można sprowadzać do innych, okazuje się, że w poszczególnych klasach występują problemy zupełne w danej klasie; *dowolny* inny problem z danej klasy można sprowadzić (w czasie nieistotnym dla złożoności obliczeniowej) do problemu zupełnego. Przykładem problemu zupełnego w klasie P (P-zupełnego) jest problem obliczania wartości formuły logicznej bez zmiennych (tj. formuły otrzymanej przez zastosowanie koniunkcji, alternatyw i negacji do stałych `true` i `false`). Rozmiarem zadania jest tu długość formuły.

Istnieją też problemy NP-zupełne; najbardziej znane grafowe problemy NP-zupełne to sprawdzenie, czy dane dwa grafy są izomorficzne, czy w danym grafie istnieje cykl Hamiltona (zamknięta ścieżka przechodząca przez wszystkie wierzchołki), problem komiwojażera (znalezienie najkrótszego cyklu Hamiltona w grafie pełnym, którego krawędzie mają określone długości) itd.

Nie-grafowymi problemami NP-zupełnymi są np. problem podziału: mając ciąg $a_1, \ldots, a_n$ liczb całkowitych, należy zbadać, czy daje się on podzielić na dwa rozłączne podciągi o równej sumie, problem spełnialności formuły logicznej: należy zbadać, czy istnieje taki układ wartości zmiennych występujących w wyrażeniu boolowskim, że wartością tej formuły jest `true`, i inne.

Otwartym problemem w teorii złożoności jest pytanie, czy $P = NP$. Jak dotąd nie wiadomo, czy tak jest; równość miałaby miejsce, gdyby dla dowolnego problemu NP-zupełnego istniał algorytm rozwiązujący ten problem w czasie wielomianowym. Póki co, nikt nie opublikował takiego algorytmu, ani dowodu, że klasy te są różne. Podanie odpowiedzi na to pytanie, oprócz nagrody 1000000$, dałoby autorowi prawo do przebierania w najlepszych ofertach pracy na świecie i w razie, gdyby klasy te były identyczne, sprawiłoby potworne kłopoty wszystkim ludziom i instytucjom, które korzystają z kryptografii (np. bankom, wojsku itd). Może ktoś nie opublikował odpowiedzi z tego powodu.

Zadania i problemy

1. Znajdź maksymalny przepływ w sieci otrzymanej z sieci na rysunku na str. 19.1, przez zastąpienie przepustowości c każdej krawędzi przez $10 - c$.
2. Pomnożenie przepustowości wszystkich krawędzi przez stałą s powoduje zwiększenie maksymalnego przepływu o ten sam czynnik. Co można powiedzieć o skutku dodania stałej $s > 0$ do przepustowości wszystkich krawędzi?
3. Udowodnij twierdzenie Halla (przez indukcję ze względu na liczbę wierzchołków zbioru S).

Algorytm FFT

Def. Niech $(a_k)_{k\in\mathbb{Z}}$ będzie okresowym ciągiem liczb zespolonych, o okresie n. Dyskretną transformatą Fouriera tego ciągu nazywamy ciąg liczb zespolonych $(b_j)_{j\in\mathbb{Z}}$ określony wzorem

$$b_j = \sum_{k=0}^{n-1} a_k e^{-2\pi ijk/n}.$$

Odwrotną dyskretną transformatą Fouriera ciągu $(a_k)_k$ nazywamy ciąg $(c_j)_{j\in\mathbb{Z}}$ określony wzorem

$$c_j = \frac{1}{n}\sum_{k=0}^{n-1} a_k e^{2\pi ijk/n}.$$

Uwaga: Czasem wybiera się inne niż 1 i $\frac{1}{n}$ czynniki stałe przed sumami we wzorach definiujących transformaty, np. $\frac{1}{\sqrt{n}}$ i $\frac{1}{\sqrt{n}}$ albo $\frac{1}{n}$ i 1; zawsze iloczyn tych czynników jest równy $\frac{1}{n}$.

Ciąg okresowy możemy utożsamić z podciągiem $a_0, \dots, a_{n-1}$. Dostrzegamy natychmiast, że zarówno transformata dowolnego ciągu, jak i transformata odwrotna, są ciągami okresowymi o okresie n. Co więcej, oba przekształcenia są liniowe i odwrotną transformatą Fouriera transformaty Fouriera ciągu $(a_k)_{k\in\mathbb{Z}}$ jest ten sam ciąg, co usprawiedliwia nazewnictwo. Mamy bowiem

$$d_l = \frac{1}{n}\sum_{j=0}^{n-1}\left(\sum_{k=0}^{n-1} a_k e^{-2\pi ijk/n}\right)e^{2\pi ilj/n} = \frac{1}{n}\sum_{k=0}^{n-1} a_k \sum_{j=0}^{n-1} e^{2\pi i(l-k)j/n}$$

Jeśli $k = l$, to $e^{2\pi i(l-k)j/n} = e^0 = 1$, natomiast jeśli $l \neq k$, to liczby $e^{2\pi i(l-k)j/n}$ są pierwiastkami zespolonymi z liczby 1; dla j przebiegającego zbiór $\{0, \dots, n-1\}$, suma tych liczb jest równa 0. Zatem

$$d_l = \frac{1}{n}\sum_{k=0}^{n-1} a_k n\delta_{lk} = a_l.$$

Dygresja. Dyskretna transformata Fouriera i jej odwrotność mają związek z przekształceniami określonymi za pomocą całek, przy czym funkcje będące argumentami tych przekształceń muszą spełniać pewne warunki, o których nie będzie tu mowy. Transformatą Fouriera funkcji $a\colon \mathbb{R} \to \mathbb{C}$ jest funkcja

$$b(\omega) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} a(t)e^{-i\omega t}\,dt,$$

a odwrotna transformata Fouriera jest określona wzorem

$$c(t) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} b(\omega)e^{i\omega t}\,d\omega,$$

przy czym zwykle funkcja c ma tę samą wartość co a „prawie wszędzie” — zostawimy ten temat.

Zadanie obliczenia dyskretnej transformaty Fouriera występuje w wielu problemach związanych z analizą, transmisją i przetwarzaniem sygnałów (np. akustycznych lub obrazów), a także w rozwiązywaniu równań różniczkowych. Możemy zauważyć, że ciąg, który jest transformatą ciągu $(a_k)_{k\in\mathbb{Z}}$, składa się z wartości wielomianu stopnia $n-1$ o współczynnikach $a_0, \dots, a_{n-1}$ w punktach $e^{-2\pi ij/n}$, $j = 0, \dots, n-1$. Można to zadanie rozwiązać przy użyciu schematu Hornera; wyznaczenie pełnej transformaty kosztowałoby wtedy $n^2 - n$ mnożeń i dodawań zespolonych. Okazuje się, że można to zadanie rozwiązać kosztem $\Theta(n(p_1 + \cdots + p_r))$ działań, gdzie $p_1, \dots, p_r$ są liczbami pierwszymi, takimi że $n = p_1 \cdot \ldots \cdot p_r$. Odkrycia tego dokonali w 1952 r. Cooley i Tukey.

Zauważmy, że ciąg okresowy $(a_k)_{k\in\mathbb{Z}}$ o okresie 1 jest ciągiem stałym i jest on identyczny ze swoją dyskretną transformatą Fouriera. Dalej, przypuśćmy, że liczba n jest podzielna przez $p > 1$. Oznaczmy $w_j = e^{-2\pi ij/n}$. Wtedy wzór definiujący dyskretną transformatę Fouriera można przedstawić w postaci

$$b_j = \sum_{k=0}^{n/p-1} a_{pk}w_j^{pk} + w_j\sum_{k=0}^{n/p-1} a_{pk+1}w_j^{pk} + \cdots + w_j^{p-1}\sum_{k=0}^{n/p-1} a_{pk+p-1}w_j^{pk}.$$

Podzieliliśmy tu ciąg $p_0, \dots, p_{n-1}$ na podciągi n/p-elementowe, wybierając do każdego z nich co p-ty element. Możemy dalej zauważyć, że sumy mnożone przez kolejne potęgi liczby w_j są wyrażeniami opisującymi transformaty tych podciągów, a dokładniej ich obustronnie nieskończonych rozszerzeń o okresie n/p. Obliczenie dyskretnej transformaty Fouriera dla ciągu o okresie n może być zatem wykonane przez następujący algorytm rekurencyjny:

- Jeśli $n = 1$, to przyjmij $b_0 = a_0$ (transformata jest otrzymana za darmo, tj. kosztem 0 działań arytmetycznych).
- Jeśli n jest liczbą pierwszą, to zastosuj wzór podany jako definicja dyskretnej transformaty Fouriera.
- Jeśli $n > 1$ jest podzielne przez liczbę pierwszą $p < n$, to podziel ciąg na p podciągów (zgodnie z opisem wyżej), oblicz transformaty tych podciągów i „scal” je, stosując wzór podany wyżej.

Zarówno „scalanie” transformat podciągów, jak i obliczenia dla liczby pierwszej n mogą być zaimplementowane przy użyciu schematu Hornera.

Wzór opisujący transformatę odwrotną może być przekształcony podobnie; zamiast $w_j = (\cos\frac{2\pi j}{n}, -\sin\frac{2\pi j}{n})$ występuje w nim liczba $\overline{w_j} = (\cos\frac{2\pi j}{n}, \sin\frac{2\pi j}{n})$. Możemy zatem użyć takiego samego algorytmu, zostawiając mnożenie wyniku działania procedury rekurencyjnej przez czynnik $\frac{1}{n}$ na sam koniec. Koszt algorytmu w istotny sposób zależy od możliwości rozłożenia liczby n na czynniki.

Algorytm jest najbardziej efektywny, jeśli liczba n jest potęgą liczby 2 i zwykle w praktyce nazwa FFT (od angielskiego *Fast Fourier Transform*) dotyczy takiego wariantu algorytmu. Zbadamy go dokładniej i zobaczymy implementacje. Dla parzystej liczby n transformatę otrzymamy przez „scalenie” transformat dwóch podciągów, złożonych odpowiednio z elementów parzystych i nieparzystych ciągu danego. Transformaty te oznaczymy symbolami $(p_j)_{j\in\mathbb{Z}}$ i $(q_j)_{j\in\mathbb{Z}}$. Przypomnijmy, że transformaty te są ciągami obustronnie nieskończonymi, o okresie $n/2$, reprezentowanymi przez podciągi $p_0, \ldots, p_{n/2-1}$ i $q_0, \ldots, q_{n/2-1}$. Możemy napisać

$$b_j = \sum_{k=0}^{n/2-1} a_{2k} w_j^{2k} + w_j \sum_{k=0}^{n/2-1} a_{2k+1} w_j^{2k} = p_j + w_j q_j.$$

Podstawiając $j + n/2$ w miejsce j, i biorąc pod uwagę, że $w_{j+n/2} = e^{-2\pi i(j+n/2)/n} = e^{-2\pi ij/n}e^{-2\pi in/(2n)} = -w_j$ oraz $w_{j+n/2}^{2k} = w_j^{2k}$, dostajemy

$$b_{j+n/2} = \sum_{k=0}^{n/2-1} a_{2k} w_{j+n/2}^{2k} + w_{j+n/2} \sum_{k=0}^{n/2-1} a_{2k+1} w_{j+n/2}^{2k} = p_j - w_j q_j.$$

Implementacja algorytmu FFT w postaci procedury rekurencyjnej jest następująca:

```
void rFFT ( int n, complex a[] )
{
  complex *p, *q, u, w, t;  int j;

  if ( n > 1 ) {
    p = malloc ( n/2*sizeof(complex) );
    q = malloc ( n/2*sizeof(complex) );
    for ( j = 0; j < n/2; j++ ) {
      p[j] = a[2*j];
      q[j] = a[2*j+1];
    }
```

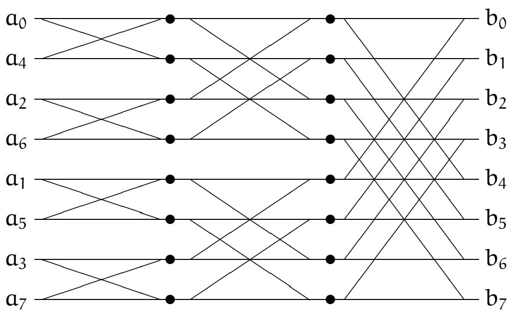


```
    rFFT ( n/2, p );
    rFFT ( n/2, q );
    u = 1;
    w = e^{-2πi/n};
    for ( j = 0; j < n/2; j++ ) {
      t = u*q[j];
      a[j] = p[j] + t;
      a[j+n/2] = p[j] - t;
      u = u*w;
    }
    free ( q );
    free ( p );
  }
} /*rFFT*/
```

Oczywiście, mnożenia i dodawania zespolone w języku C koduje się w trochę bardziej skomplikowany sposób[13], ale podany wyżej kod ma być przede wszystkim zrozumiały dla studentów, czyli ludzi.

Oglądając algorytm, widzimy, że choć procedura umieszcza transformatę w tej samej tablicy, w której były dane liczby $a_0, \ldots, a_{n-1}$, potrzebuje ona sporo pamięci dodatkowej (w rzeczywistości potrzeba dodatkowych tablic o sumarycznej długości $2n$). Można jednak zaprojektować taką implementację, która wszystkie obliczenia wykonuje „w miejscu”, tj. która oprócz tablicy z danymi, które należy zastąpić przez transformatę, potrzebuje tylko niewielkiej ustalonej liczby zmiennych prostych. Aby otrzymać taką procedurę, nierekurencyjną i dodatkowo oszczędzającą pewne operacje, przyjrzymy się „przepływowi danych”, to znaczy zbadamy, od których współczynników zależą transformaty obliczane „po drodze”. Dla $n = 8$ „przepływ danych” jest przedstawiony na rysunku (najlepiej go oglądać od prawej do lewej strony).

[13] W języku C++ można określić operatory „*”, „+” i „-” mnożenia, dodawania i odejmowania liczb zespolonych w taki sposób, że tak zapisany kod już by działał, oczywiście po właściwym zakodowaniu wyrażenia $e^{-2\pi i/n}$.



Krawędzie łączą dane z wynikami, tj. każda liczba (z wyjątkiem danych) jest obliczana na podstawie liczb znajdujących się w kolumnie na lewo od niej, połączonych z nią kreskami. Widzimy, że w każdym przypadku obliczenie polega na zastąpieniu pary liczb przez inną parę, obliczoną tylko na jej podstawie (i dana para liczb nie jest do niczego innego potrzebna). Jeśli zatem ustawimy dane wejściowe w odpowiedniej kolejności, to można całe obliczenie wykonać bez potrzeby rezerwowania dodatkowej tablicy.

Ostatnia transformata powstaje z transformat podciągów „parzystego" i „nieparzystego". Każda z tych dwóch transformat jest obliczana na podstawie transformat „parzystego" i „nieparzystego" podciągu odpowiedniego podciągu itd.; zatem ogólna reguła porządkowania danych wejściowych polega na ustawieniu ich *w kolejności odwróconych bitów*. Jeśli indeks j danego współczynnika a_j przedstawimy w układzie dwójkowym, przy użyciu $l+1 = (\log_2 n) + 1$ cyfr dwójkowych (bitów), to indeks miejsca w tablicy, na którym ma się on znaleźć, otrzymamy wypisując te bity w odwrotnej kolejności. Procedura FFT, która realizuje to obliczenie, ma postać

```
void FFT ( int n, complex a[] )
{
  complex t, u, w;
  int i, j, k, l, m, p;

  l = log2n;  m = n/2;
        /* przestawianie danych w tablicy */
  for ( i = 1, j = m;  i < n-1;  i++ ) {
    if ( i < j ) przestaw ( &a[i], &a[j] );
    k = m;
    while ( k <= j ) { j -= k;  k /= 2; }
    j += k;
  }
```



```
        /* obliczanie transformaty */
  for ( k = 1; k <= l; k++ ) {
    m = 2^k;  p = m/2;
    u = 1;  w = e^(-πi/p);
    for ( j = 0; j < p; j++ ) {
      i = j;
      do {
        t = a[i+p]*u;
        a[i+p] = a[i]-t;  a[i] = a[i]+t;
        i += m;
      } while ( i <= n );
      u *= w;
    }
  }
} /*FFT*/
```

Algorytm ten opublikowali w 1965 r. Cooley, Lewis i Welch. Pierwsza pętla, `for ( i = ... ) ...`, dokonuje przestawienia elementów w tablicy zgodnie z kolejnością odwróconych bitów. Kolejne przebiegi drugiej pętli, `for ( k = ... ) ...`, mają na celu obliczenie $n/2$ transformat podciągów o okresie 2, $n/4$ transformat podciągów o okresie 4, itd. Liczba $e^{-2\pi i/n}$ (wartość zmiennej `w`) i jej potęgi, czyli liczby w_j (kolejne wartości zmiennej `u`) są obliczane tylko raz dla wszystkich transformat podciągów o tym samym okresie. W każdym przebiegu pętli `for ( j = ... ) ...` obliczane są pary współczynników o numerach j oraz $j+p$ we wszystkich transformatach podciągów o okresie $m = 2p$, ponieważ pętla najbardziej wewnętrzna (`do`...`while`) przebiega przez wszystkie te transformaty.

Można udowodnić, że algorytm FFT, także w wersji ogólnej (dla dowolnego n), jest numerycznie stabilny, tj. istnieje stała K (zależna od n), taka że współczynniki $\tilde{b}_j$ obliczone przy użyciu arytmetyki zmiennopozycyjnej przybliżają dokładne współczynniki b_j dyskretnej transformaty Fouriera z błędem spełniającym nierówność

$$\max_j |\tilde{b}_j - b_j| \le K\nu \max_j |b_j|,$$

gdzie $\nu = 2^{-t}$. Dla liczby n będącej potęgą 2 jest

$$K = \big(\sqrt{2}\,\log_2 n + (\log_2 n - 1)(3+2\varepsilon)\big)\sqrt{n},$$

gdzie ε jest oszacowaniem błędu bezwzględnego obliczonych cosinusów i sinusów, tj. części rzeczywistych i urojonych liczb w_j.

Zadanie interpolacyjne Lagrange'a

Aby pokazać jedno z zastosowań algorytmu FFT, sformułujemy zadanie interpolacyjne Lagrange'a: mając dane liczby $x_0, \ldots, x_n$, parami różne, i liczby $y_0, \ldots, y_n$ (dowolne), należy skonstruować wielomian w stopnia nie większego niż n, taki że $w(x_i) = y_i$ dla $i = 0, \ldots, n$.

Zadanie to ma jednoznaczne rozwiązanie; jest ono opisane wzorem

$$w(x) = \sum_{i=0}^{n} y_i \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}.$$

Poprawność tego wzoru jest łatwa do sprawdzenia, a z niej wynika istnienie rozwiązania. Gdyby zaś istniało więcej niż jedno rozwiązanie, to odejmując dwa różne rozwiązania, otrzymalibyśmy wielomian stopnia co najwyżej n, który ma co najmniej $n+1$ miejsc zerowych. Nie będziemy się teraz zajmować praktycznymi algorytmami konstruowania rozwiązania, czyli obliczania współczynników wielomianu w w jakichś wygodnych do użycia bazach, natomiast korzystając z istnienia i jednoznaczności rozwiązania zadania interpolacyjnego, pokażemy, jak można szybko mnożyć wielomiany.

Zajmiemy się następującym zadaniem: dane są współczynniki $a_0, \ldots, a_n$ i $b_0, \ldots, b_m$ wielomianów $a(x) = \sum_{k=0}^{n} a_k x^k$ i $b(x) = \sum_{k=0}^{m} b_k x^k$. Należy obliczyć współczynniki $c_0, \ldots, c_{n+m}$ wielomianu $c(x) = \sum_{k=0}^{n+m} c_k x^k = a(x)b(x)$. „Zwykły” algorytm mnożenia wielomianów można zrealizować za pomocą podprogramu

```
for ( k = 0; k <= n+m; k++ ) c[k] = 0;
for ( i = 0; i <= n; i++ )
  for ( j = 0; j <= m; j++ ) c[i+j] += a[i]*b[j];
```

Operacją dominującą w tym algorytmie jest mnożenie współczynników; operacji tych należy wykonać $(n+1)(m+1)$; jeśli $m \approx n$, to złożoność obliczeniowa ma rząd $O(n^2)$, choć zarówno danych, jak i wyników jest $O(n)$.

Alternatywny sposób rozwiązywania tego zadania polega na wybraniu liczb $x_0, \ldots, x_{n+m}$, obliczeniu wartości wielomianów a i b, obliczeniu wartości $c(x_j) = a(x_j)b(x_j)$ wielomianu c i znalezieniu jego współczynników w bazie potęgowej, przez rozwiązanie zadania interpolacyjnego. Mnożenie wielomianów — w postaci mnożenia ich wartości w wybranych punktach — wymaga wykonania tylko $n+m+1$ mnożeń. Trzeba tylko umieć szybko obliczyć wartości wielomianów a i b i szybko rozwiązać zadanie interpolacyjne Lagrange'a.

Do tego celu możemy użyć algorytmu FFT; jeśli przyjmiemy, że $x_j = e^{-2\pi i j/N}$, gdzie liczba N jest najmniejszą całkowitą potęgą liczby 2 większą niż $n+m$, to ciąg wartości wielomianu a w tych punktach jest dyskretną transformatą Fouriera ciągu współczynników $a_0, \ldots, a_n, 0, \ldots, 0$ o długości (a raczej okresie) N. Mając wartości wielomianu c w punktach x_j, możemy obliczyć jego współczynniki w bazie potęgowej, wyznaczając odwrotną dyskretną transformatę Fouriera. Całe to obliczenie jest wykonalne kosztem $O(N \log N)$ działań zmiennopozycyjnych.