1 Algorytm

1.1 Wstęp do zadania — rozważania ogólne

Wpierw zastanówmy się, jak można opisać zjawisko rzucania cienia. Przyjmujemy milcząco założenie nieskończenie dalekiego, punktowego słońca. Przy takim założeniu godzina wyznacza po prostu kierunek, z którego padają równomiernie z całego nieba promienie słoneczne. To, że dany punkt jest oświetlony oznacza po prostu tyle, że półprosta wychodząca z tego punktu "w stronę słońca" nie przecina żadnego budynku. Zauważmy, że jeśli dany punkt jest oświetlony, to wszystkie znajdujące się bezpośrednio nad nim również są oświetlone, budynki bowiem są "spójne w dół". W szczególności oznacza to, że badając, czy dany apartament jest oświetlony możemy badać tylko najniższe punkty na jego wschodniej i zachodniej ścianie, a nie całe ściany.

Słońce w Szanghaju stoi w zenicie o godzinie 11:57. Przed tą godziną promienie słońca biegną "z góry i wschodu", a po tej godzinie — "z góry i zachodu". Zauważmy, że jeśli jakiś punkt jest oświetlony o pewnej godzinie g przed 11:57, to jest też oświetlony w dowolnej godzinie pomiędzy g a 11:57, bo im później, tym promienie słońca padają bardziej z góry, zatem prosta z naszego punktu "w stronę słońca" biegnie ponad prostą wyznaczoną w godzinie g, a zatem gdyby jakiś budynek przecinał naszą prostą, to musiałby też przeciąć prostą z godziny g, co jednak jest sprzeczne z założeniem, że o godzinie g nasz punkt był oświetlony. Sytuacja po godzinie 11:57 oczywiście jest symetryczna. Zatem każdy punkt nie znajdujący się wewnątrz budynku jest oświetlony w pewnym przedziale czasowym zawierającym godzinę 11:57.

Dolny koniec wschodniej ściany danego apartamentu po godzinie 11:57 znajduje się w cieniu rzucanym przez ten budynek, zaś dolny koniec zachodniej znajduje się w cieniu do godziny 11:57. Zatem, aby dowiedzieć się, kiedy budynek jest oświetlony, wystarczy sprawdzić, od której chwili oświetlony jest dolny koniec wschodniej ściany i do której chwili oświetlony jest dolny koniec zachodniej ściany. Obydwa te problemy są symetryczne, zatem skoncentrujemy się na jednym z nich, powiedzmy pierwszym, a drugi uznamy, że robi się tak samo. Rozważmy pewien konkretny apartament w budynku A, oznaczmy dolny koniec wschodniej ściany tego apartamentu przez p.

W kluczowej chwili prosta idąca od p "w stronę słońca" przechodzi przez zachodni górny róg pewnego budynku leżącego na wschód od naszego. Faktycznie, łatwo sprawdzić, że to ten punkt każdego budynku rzuca cień na dowolny punkt najpóźniej. Niech (a_n) oznacza ciąg zachodnich górnych rogów budynków leżących na wschód od naszego. Spośród wszystkich punktów a_n najpóźniej cień rzuca ten, dla którego kąt między prostą łączącą a_n z p a gruntem jest największy. Stąd pierwszy, najprostszy pomysł na algorytm:

1.2 Algorytm bezpośredni — rozwiązanie niewzorcowe

Autor: Jakub Wojtaszczyk

Dla każdego zgrupowania budynków wczytujemy wszystkie budynki i zapamiętujemy w tablicy ich wysokości oraz położenia ich wschodnich brzegów (przyjmujemy, że wschodni brzeg najdalej na wschód położonego budynku leży w punkcie 0). Położenia co prawda nie są podane na wejściu, ale możemy je obliczać dla kolejnych wczytywanych budynków, dodając $w + d(i)$ do położenia poprzedniego budynku. Następnie dla każdego apartamentu, o który jesteśmy pytani, sprawdzamy kąt między p a a_n dla wszystkich budynków na wschód od p, zapamiętując największy (w praktyce łatwiej liczyć i pamiętać tangens kąta, niż sam kąt), następnie na podstawie kąta obliczamy godzinę (kąty liniowo przekładają się na godziny w przedziale 5:37 — 11:57), i podajemy tę godzinę. Taką samą operacje przeprowadzamy na zachód od budynku, by dowiedzieć się, kiedy przestaje być on oświetlony.

1.2.1 Analiza złożoności

Dla każdego zapytania przeglądamy wszystkie budynki, zatem złożoność czasowa naszego algorytmu to $O(ln)$, gdzie l to liczba zapytań, zaś n to liczba budynków. Złożoność pamięciowa to $O(n)$ na tablice, w których przechowujemy wysokości i położenia budynków.

1.2.2 Uwagi

Autor zadania nie wysilił nadmiernie intelektu wybierając zakres możliwych n do zadania (oraz w ogóle nie podał ograniczenia na l). Później podamy algorytm wzorcowy, który działa w czasie $O(n^2 + l \cdot \log n)$. Możliwy jest też algorytm działający w czasie $O(n + l \log^2 n)$ oraz algorytm w czasie $O(n + l(\log l + \log n)$. Niestety, przy $n \leqslant 100$ dobrze napisany algorytm bezpośredni nie działa na tyle wolniej od bardziej skomplikowanych algorytmów o lepszych złożonościach, by dało się to wychwycić na testach, szczególnie, że z każdym zapytaniem związane jest wypisanie wyjścia, które jest operacją czasochłonną w porównaniu z obliczeniami. Gdyby w danych zadania było podane $n \leqslant 1000$ i $l \leqslant 100000$, to algorytm wzorcowy o złożoności $O(n^2 + l \log n)$ byłby już istotnie (i wychwytywalne przez testy) lepszy od bezpośredniego. Gdyby było $n \leqslant 10000$ i $l \leqslant 10000$, to akceptowalne byłyby jedynie dwa ostatnie algorytmy. Niestety, dane zadania są jakie są, i dlatego na konkursie najlepszym (bo najprostszym) rozwiązaniem było szybkie napisanie algorytmu bezpośredniego. Ten algorytm (bez żadnych przyspieszających udziwnień) jest zaimplementowany jako **brute.cpp**.

1.3 Rozważania ciut bardziej zaawansowane

By wypracować rozwiązanie wzorcowe będziemy musieli jeszcze trochę popracować. Zauważmy, że jeśli pewien róg a_l leży pomiędzy (w linii wschód–zachód) a_k i a_m oraz poniżej linii, która je łączy, to na pewno nie jest on ostatnim punktem rzucającym cień na p, bo albo linia łącząca p z a_k, albo z a_m przebiega ponad a_l, a zatem ponad linią łączącą p z a_l. Zatem możemy wyrzucić z ciągu a_n te punkty, które leżą pod linią łączącą dwa inne.

Niech teraz a_k i a_{k+1} będą dwoma kolejnymi punktami w naszym już oczyszczonym ciągu, i załóżmy, że linia łącząca p z a_k przebiega ponad a_{k+1}. Z tego już wynika, że dla $l > k$ linia łącząca p z a_k przebiega również ponad a_l. Gdyby było inaczej, to a_l leżałoby ponad linią od p do a_k, a zatem linia od a_k do a_l leżałaby jeszcze wyżej, czyli w szczególności nad a_{k+1}, a zatem a_{k+1} nie byłoby w naszym oczyszczonym ciągu, co jest sprzeczne z założeniem. Analogicznie jeśli a_{k+1} leży ponad linią łączącą p i a_k, to możemy już nie rozważać punktów a_l dla $l < k$.

Widać zatem, że kształtuje nam się pomysł wyszukiwania binarnego. Potrzeba nam jeszcze ostatniego spostrzeżenia — "oczyszczony" ciąg a_n nie zależy od konkretnego apartamentu, a tylko od budynku, na wschód od którego ma się znajdować. Zatem możemy policzyć go na początku dla każdego z budynków, a potem dla konkretnego zapytania wyszukiwać binarnie w oczyszczonym ciągu.

1.4 Rozwiązanie wzorcowe 1

Autor: Jakub Wojtaszczyk

Dla każdego zgrupowania budynków jak poprzednio wczytujemy wszystkie budynki. Teraz jednak, zanim zaczniemy wczytywać zapytania tworzymy "ciągi oczyszczone". Dla każdego budynku przechodzimy budynki na wschód od niego, idąc od wschodu, i każdy kolejny budynek dodając do ciągu. Następnie, jeśli dodaliśmy go na pozycji a_k, póki a_{k-1} leży pod linią łączącą a_k i a_{k-2}, wyrzucamy a_{k-1} z ciągu i na jego miejsce przestawiamy nasze a_k. Te ciągi trzymamy w tablicach (jedna tablica dla każdego budynku). Analogicznie tworzymy "zachodnie ciągi oczyszczone".

Dla każdego zapytania patrzymy na "wschodni ciąg oczyszczony" dla budynku, w którym leży apartament, o który pytamy i w tym ciągu binarnie wyszukujemy największe takie k, że linia łącząca a_{k-1} z a_k przebiega ponad p. Liczymy kąt między prostą łączącą p i a_k a gruntem i przekształcamy go (jak w poprzednim rozwiązaniu) na godzinę. Analogicznie postępujemy po zachodniej stronie.

1.4.1 Analiza poprawności

Wpierw zastanówmy się, dlaczego nasz algorytm faktycznie znajduje ciąg oczyszczony. Do tego nam się przyda matematyczny model tego ciągu. Dorzućmy jeszcze dwa punkty, które znajdują się nieskończenie nisko, jeden, S pod zachodnim brzegiem najbardziej wschodniego budynku, a drugi, T pod zachodnim brzegiem wschodniego sąsiada naszego budynku A. Twierdzimy, że w ciągu oczyszczonym znajdują się te i tylko te punkty a_i, które leżą na brzegu najmniejszego zbioru wypukłego zawierającego wszystkie a_i oraz S i T (tzw. otoczki wypukłej).

Z jednej strony, jeśli jakiś punkt a_l leży pod pewną linią łączącą dwa punkty a_k i a_m, to leży we wnętrzu czworokąta S, a_k, a_m, T, a zatem tym bardziej leży we wnętrzu, a nie na brzegu otoczki wypukłej. Z drugiej strony zauważmy, że brzeg otoczki wypukłej będzie się składał z dwóch półprostych pionowych od S do a_1 i od a_v do T, leżącego nieskończenie nisko odcinka ST oraz górnych boków, będących odcinkami łączącymi pary a_i, a_j. Zatem jeśli jakiś punkt nie leży na brzegu, to w szczególności musi znajdować się pod którymś z górnych odcinków, a zatem nie należy do ciągu oczyszczonego.

Teraz już możemy udowodnić poprawność znajdowania ciągu oczyszczonego. Jeśli pewien element należy do ciągu oczyszczonego, to oczywiście w toku konstrukcji nie zostanie wyrzucony, bo wyrzucamy tylko te punkty, które leżą pod pewną prostą. Z drugiej strony indukcyjnie widać, że na k-tym kroku mamy stworzony ciąg punktów na brzegu otoczki wypukłej $S, a_1, a_2, \ldots, a_k, T_k$, gdzie T_k to nieskończenie daleki punkt pod a_k. Gdy dodamy do tego ciągu a_{k+1}, znajdziemy (to robi nasz algorytm) pierwszy punkt a_l taki, że a_{l-1} leży pod linią łączącą a_l i a_{k+1}, to figura o wierzchołkach $S, a_1, a_2, \ldots, a_l, a_{k+1}, T_{k+1}$ będzie wypukła, czyli te wierzchołki faktycznie będą stanowiły $k+1$-wszy ciąg oczyszczony.

Na ciągu oczyszczonym, czyli na otoczce wypukłej wyszukiwanie binarne działa, bo nachylenia kolejnych boków są uporządkowane malejąco. Już dowodziliśmy, że wystarczy rozważać wierzchołki otoczki wypukłej jako kandydatów na punkty, które ostatnie rzucają cień, czyli nasz algorytm faktycznie zwróci ostatni punkt rzucający cień.

1.4.2 Analiza złożoności

Złożoność pamięciowa jest rzędu $O(n^2)$, bo tyle miejsca zajmują tablice, w których pamiętamy otoczki wypukłe. Złożoność konstrukcji ciągów oczyszczonych to też $O(n^2)$. Faktycznie, dla każdego budynku A konstruując jego wschodni ciąg oczyszczony, każdy budynek na wschód od A będziemy procesowali co najwyżej dwa razy — raz dokładając go do ciągu, a raz wyrzucając go z ciągu. Zatem budowa pojedynczej otoczki działa w czasie $O(n)$, a budujemy n otoczek. Otoczka ma najwyżej n elementów, więc wyszukiwanie binarne działa na niej w czasie $O(\log n)$. Zatem cały algorytm ma złożoność czasową $O(n^2 + l \log n)$.

1.4.3 Uwagi

Algorytm ten został zaimplementowany jako **subtle.cpp**. Jak już było wspomniane w uwagach do algorytmu bezpośredniego, ograniczenie $n \leqslant 100$ w praktyce uniemożliwia odróżnienie czasowe przyzwoicie napisanego rozwiązania logarytmicznego od dobrze napisanego rozwiązania liniowego. Przy tym ograniczeniu również nie ma sensu wysiłek mający na celu zredukowanie czasu n^2 na konstrukcję tablicy otoczek wypukłych. Dla ciekawości można jednak zastanowić się, czy można ten czas poprawić. Gdyby dane zadania przewidywały jakieś sensowne n, powiedzmy $n < 10000$, to czas n^2 zacząłby być nieco za duży. Tu naszkicujemy zatem dwa alternatywne algorytmy.

Zauważmy, że otoczkę wypukła dla budynku k konstruujemy indukcyjnie modyfikując otoczkę dla budynku $k-1$. Jeśli więc posortujemy zapytania po numerze budynku, to będziemy mogli w momencie, gdy mamy skonstruowaną otoczkę dla budynku $k-1$ odpowiedzieć na pytania o "wschodnie strony" dla tego budynku, a następnie zapomnieć o otoczce dla tego budynku i konstruować otoczkę dla dalszych budynków indukcyjnie. Podobnie potem konstruujemy od zachodu zachodnią otoczkę, i odpowiadamy na zachodnie strony pytań o budynek k, gdy przy nim jesteśmy. Następnie z powrotem sortujemy odpowiedzi po kolejnym numerze pytania, i wypisujemy wyniki. Złożoność czasowa takiego algorytmu to $O(n + l \log l + l \log n)$ (pierwsza część odpowiada za budowę otoczek, druga za posortowanie pytań i odpowiedzi, a trzecia za wyszukiwanie binarne dla każdego pytania), co jest równe $O(n + l \log l)$. Jego wadą jest złożoność pamięciowa rzędu $O(l + n)$, która dla $l >> n$ może być nie do zaakceptowania.

Drugie rozwiązanie jest trochę bardziej trikowe. Jest to drugie rozwiązanie wzorcowe.

1.5 Rozwiązanie wzorcowe 2

Autorzy: Wojciech Czerwiski, Jakub Wojtaszczyk

1.5.1 Algorytm

Dla każdego budynku będziemy pamiętać zakresy pięter, dla których cień rzuca ten sam budynek. Dla budynku k jesteśmy w stanie uzyskać te zakresy analizując otoczkę dla budynku k. Patrzymy gdzie linia łącząca prawe górne rogi dwóch sąsiednich budynków na otoczce budynku k trafia w ten budynek. Powyżej miejsca trafienia cień rzuca lewy z tej pary budynków, lub jakiś budynek jeszcze bardziej na lewo od niego. W ten sposób mając zakresy pieter, będziemy mogli mając zapytanie wyszukiwać dla danego budynku binarnie. Zauważmy teraz, że jeśli mając otoczkę budynku k będziemy rozważać pary budynków tworzących linię od par po prawej stronie (tych bliżej budynku k), to linie łączące rogi bu-

dynków z pary będą trafiać coraz wyżej w budynek k. Zatem w momencie, gdy linia będzie przelatywać nad budynkiem k możemy przestać sprawdzać pary bardziej na lewo, gdyż dla tamtych par linia będzie jeszcze wyżej i nie jest istotna dla zapytań o budynek k.

1.5.2 Analiza złożoności

Zauważmy, że postępując w ten sposób zbudujemy strukturę zakresów na budynkach w czasie $O(n)$. Te budynki, które wygenerują linie trafiające w budynek zostaną usunięte z następnej otoczki (dla budynku $k+1$), gdyż budynek k je zasłoni. Takźe operacji wrzucenia do otoczki bedzie tyle co operacji zbadania linii z dokładnością do stałej liczby operacji na budynek, czyli do $O(n)$. Przypomnijmy jednak, że otoczkę buduję się w czasie $O(n)$, czyli cała budowa struktury zajmie faktycznie $O(n)$ czasu. Rozpatrywanie zapytań zajmie $O(l \log(n))$, czyli cały algorytm będzie działał w czasie $O(n + l \log(n))$, co jest najlepszym z dotychczasowych wyników.

1.5.3 Implementacja

Podany algorytm został zaimplementowany w C++ w pliku **sunlight.cpp** oraz w Javie w pliku **LotsOfSunlight.java**.

2 Testy

Test 0 to standardowo test przykładowy.
Testy 1 oraz 2 to niewielkie testy losowe sprawdzające elementarną poprawność napisanego programu bez przypadków szczególnych.
Test 3 to test obejmujący przypadki budynków coraz wyższych, coraz niższych oraz równych i przy okazji sprawdzający poprawność odpowiedzi na pytanie o nieistniejące piętro.
Test 4 sprawdza przypadki szczególne, budynki malejące w tym samym tempie, budynki, które maleją coraz bardziej lub coraz mniej. Poza tym w tym zestawie testów jest tez kolejny test na nieistniejące piętro (w budynku o numerze o 1 większym niż największy numer budynku).
Test 5 to pojedynczy zestaw sprawdzający tylko, czy rozważone zostały duże liczby, czy wysokośc budynku w piętrach pomnożona przez wysokość piętra nie wychodzi poza zakres.
Testy 6 i 7 to sredniej wielkości testy.
Testy 8 i 9 to duże testy wydajnościowe.