1 Sortowanie liczb całkowitych

W poprzednich dwóch wykładach zajmowaliśmy się problemem poprzednika. Pokazaliśmy, że można wszystkie wymagane operacje wykonywać w zamortyzowanym czasie $O(\sqrt{\lg n})$, przy czym jeśli zależy nam na złożoności pamięciowej $O(n)$, to musimy użyć randomizacji.

Problem poprzednika jest uogólnieniem problemu kolejki priorytetowej, a zatem

Twierdzenie 1. *n liczb całkowitych można posortować w czasie $O(n\sqrt{\lg n})$.*

W tym wykładzie zastanowimy się, czy można uzyskać złożoność lepszą niż $O(n\sqrt{\lg n})$. Mogłoby się wydawać, że sortowanie niewiele ma wspólnego ze strukturami danych, czyli tematem tych zajęć. Jest to jednak dalekie od prawdy. Po pierwsze, właśnie użyliśmy struktur danych do sortowania, a po drugie, za jakiś czas zobaczymy, że można też pójść w drugą stronę.

1.1 Algorytm Range Reduction, czyli lepszy Radix Sort

Zanim zaczniemy omawiać bardziej zaawansowane algorytmy, zastanówmy się jaką złożoność ma w modelu WordRAM standardowy "szybki" algorytm sortowania liczb całkowitych – Radix Sort.

Ćwiczenie 1. *Jaką złożonośc ma algorytm Radix Sort?*

Rozwiązanie. Sortowanie n liczb w-bitowym na w-bitowym WordRAM-ie wymaga czasu $O(n\frac{w}{\lg n})$ – każda faza w czasie $O(n)$ sortuje kubełkowo względem kolejnego bloku $\lg n$ bitów.

Zwróćmy uwagę, że jeśli $w = \omega(\lg^2 n)$, to ten algorytm jest gorszy nawet od algorytmów porównaniowych, np. od algorytmu Heap Sort.

Pokażemy, że można w bardzo podobny sposób uzyskać dużo lepszą złożoność.

Twierdzenie 2. *n liczb całkowitych można posortować w czasie $O(n \lg \frac{w}{\lg n})$.*

Twierdzenie 2 jest natychmiastowym wnioskiem z następującego lematu, który w „zastosowaniach teoretycznych" jest dużo ważniejszy niż samo twierdzenie.

Lemat 3 (Range reduction)**.** *Sortowanie n liczb b-bitowych można w w-bitowym modelu WordRAM ($b \leq w$) sprowadzić w czasie $O(n)$ do sortowania n liczb $b/2$-bitowych.*

Dowód. Niech $S = \{x_1, \ldots, x_n\}$ będzie zbiorem do posortowania. Najprostszym pomysłem byłoby posortowanie S po $\mathrm{low}(x_i)$, a następnie rozrzucenie sortowanych elementów po kubełkach indeksowanych $\mathrm{high}(x_i)$. Problem polega na tym, że aby uzyskać ciąg posortowany, musimy umieć przejść niepuste kubełki w rosnącej kolejności czyli de facto posortować zbiór $\mathrm{high}(S)$. W ten sposób sprowadzilibyśmy sortowanie liczb b-bitowych do dwóch (!) sortowań liczb $b/2$-bitowych.

Oczywiście nie musimy wykonywać dwóch sortowań, wystarczy do każdej połówki dodać informacje, czy jest to high czy low, i posortować wszystko razem. Wtedy jednak sortujemy zbiór rozmiaru $2n$, znów za dużo.

Użyjemy następującego tricku. Wrzucamy wszystkie liczby do kubełków indeksowanych $\mathrm{high}(x_i)$ i znajdujemy przy okazji minima każdego z kubełków. Następnie sortujemy jednocześnie wszystkie $\mathrm{low}(x_i)$ dla x_i nie będącymi minimami kubełków oraz wszystkie $\mathrm{high}(x_i)$.

W ten sposób sortujemy tylko n liczb $b/2$-bitowych.

$\square$

1.2 Sortowanie bitoniczne w wydaniu WordRAM

Pokazaliśmy, że można w czasie liniowym dwukrotnie zmniejszać rozmiar sortowanych liczb. Pokażemy teraz, że jeśli sortowane liczby są wystarczająco małe względem w to można je posortować bardzo szybko. W połączeniu z Lematem 3 uzyskamy w ten sposób algorytm sortowania w czasie $O(n \lg\lg n)$.

Algorytm sortowania małych liczb jest WordRAM-ową wersją algorytmu sortowania bitonicznego przy pomocy sieci sortujących (patrz Cormen, Leiserson, Rivest „Wprowadzenie do algorytmów").

Lemat 4 (Scalanie bitoniczne)**.** *Niech $b \le w$ i $k = O(w/b)$. Dwa spakowane posortowane k-elementowe ciągi liczb b-bitowych można scalić w czasie $O(\lg k)$.*

Dowód tego lematu (dość techniczny) odłożymy na później. Najpierw zobaczmy co z niego wynika.

Twierdzenie 5 (Spakowane sortowanie (Albers, Hagerup))**.** *Niech $b \le w$ i $k = O(w/b)$. Sortowanie n-elementowego ciągu liczb b-bitowych można wykonać w czasie $O(n\frac{\lg k}{k})$. W szczególności, jeśli $k = \Omega(\lg n \lg\lg n)$, to sortowanie zajmuje czas liniowy.*

Ćwiczenie 2. *Udowodnij Twierdzenie 5.*

Wskazówka. Użyj algorytmu Merge Sort.

Twierdzenie 6 (Andersson,Hagerup,Nilsson,Raman)**.** *n liczb można posortować w czasie $O(n \lg\lg n)$.*

Ćwiczenie 3. *Udowodnij to twierdzenie.*

Wskazówka. Połącz Lemat 3 i Twierdzenie 5.

Pozostaje nam udowodnić Lemat 4. Przypomnijmy krótko jak działa scalanie bitoniczne.

Ciąg $x_1, \ldots, x_n$ jest *b*itoniczny jeśli pewne jego przesunięcie cykliczne do pewnego miejsca rośnie, a potem maleje.

Lemat 7. *Niech $x_1, \ldots, x_n$ będzie ciągiem bitonicznym. Jeśli ustawimy w rosnącej kolejności elementy każdej z par x_i,$x_{i+n/2}$, to wszystkie elementy w pierwszej połówce będą $\leq$ od wszystkich elementów w drugiej połówce. Ponadto, obie połówki będą bitoniczne.*

A zatem aby scalić dwa posortowane ciągi $x_1, \ldots, x_n$, $y_1, \ldots, y_n$ należy odwrócić drugi ciąg i dopisać go na końcu pierwszego. Otrzymujemy w ten sposób ciąg bitoniczny, który można posortować stosując rekurencyjnie lemat 7. Głębokość rekursji jest $O(\lg n)$.

Dowód Lematu 4. Aby scalić dwa spakowane posortowane ciągi długości k w czasie $\lg k$ użyjemy algorytmu scalania bitonicznego. Musimy pokazać, że da się w czasie $O(\lg k)$:

- odwrócić spakowany ciąg,
- ustawić w dobrej kolejności każdą z par elementów ciągu odległych o k.

i to wszystko w sposób, który można „zrównoleglić".

Ćwiczenie 4. *Uzupełnij szczegóły dowodu.*

□

1.3 Funkcje haszujące kontratakują, czyli sortowanie sygnaturowe

Pokażemy teraz, że

Twierdzenie 8 (Andersson,Hagerup,Nilsson,Raman)**.** *Jeśli $w \geq \lg^{2+\epsilon} n$, to n-elementowy ciąg liczb w-bitowych można posortować w czasie $O(n)$.*

Niech $x_1, \ldots, x_n$ będzie ciągiem liczb w-bitowych. Główny pomysł tego algorytmu jest następujący. Jeśli w jest duże, jak w założeniach twierdzenia, to większość bitów jest w jakimś sensie „niepotrzebna" do odróżniania liczb x_i między sobą. Można mianowicie znaleźć funkcję haszującą h taką, że liczby $h(x_i)$ mają $O(\lg n)$ bitów i są wszystkie różne. Wystarczy wybrać losową funkcję z uniwersalnej rodziny funkcji haszujących. Niestety w ten sposób całkowicie tracimy strukturę porządku. Chcielibyśmy wykorzystać redukcję rozmiaru liczb jaką daje haszowanie, ale bez straty informacji o porządku i w ten sposób uzyskać redukcję lepszą niż w Lemacie 3.

Lemat 9 (Andersson,Hagerup,Nilsson,Raman)**.** *Niech $w \geq (\lg^{2+\epsilon} n) \lg\lg n = \lg^{2+\epsilon'} n$ i niech $\lg^{1+\epsilon} n \leq b \leq w$. Wtedy można w czasie $O(n)$ sprowadzić sortowanie n liczb b-bitowych do sortowania n liczb $b/\lg^{\epsilon} n$-bitowych.*

Ćwiczenie 5. *Pokaż, że z Lematu 9 wynika Twierdzenie 8.*

Wskazówka. Skorzystaj z bitonicznego algorytmu Albers i Hagerupa.

Dowód Lematu 9. Niech $x_1, \ldots, x_n$ będzie ciągiem liczb, który chcemy posortować. Dzielimy każdą z liczb x_i na $\lg^\epsilon n$ kawałków i każdy z tych kawałków haszujemy (można to zrobić w czasie stałym). Z dużym prawdopodobieństwem różnym kawałkom zostaną przypisane różne wartości W ten sposób każdej z liczb x_i przypisujemy jej $b' = O(\lg^{1+\epsilon} n)$-bitową $s$ygnaturę.

Następnie:

- sortujemy sygnatury algorytmem Albers i Hagerupa w czasie $O(n)$,
- budujemy skompresowane drzewo trie w czasie $O(n)$ – naszym alfabetem są tu hasze kawałków,
- sortujemy synów każdego wierzchołka (sortowanie jest względem prawdziwych kawałków a nie ich haszów),
- wypisujemy wierzchołki w porządku inorder.

W jaki sposób sortować synów? Łącznie mamy $n - 1$ synów. Każdego syna można opisać jednym $b/\lg^\epsilon n$-bitowym kawałkiem. Wszystkich synów można posortować jednym sortowaniem (pamiętając odpowiednie dodatkowe informacje). W ten sposób sprowadziliśmy sortowanie liczb b-bitowych do sortowania liczb $b/\lg^\epsilon n$-bitowych. □

Jaki ładny algorytm.