Tomasz Waleń
Treść zadania, Opracowanie

Krzysztof Dulęba, Jakub Radoszewski
Program

OI, Etap I, 22.10–19.11.2007

Klocki

Bajtazar jako małe dziecko uwielbiał bawić się klockami. Jego zabawa polegała na układaniu z klocków n kolumn o losowo wybranych wysokościach, a następnie ich porządkowaniu. Bajtazar wybierał liczbę k, a następnie starał się w minimalnej liczbie ruchów tak uporządkować klocki, by pewne k kolejnych kolumn klocków miało tę samą wysokość. Pojedynczy ruch polega na:

- *położeniu jednego klocka na szczycie wybranej kolumny klocków (Bajtazar posiadał ogromne pudło z zapasowymi klockami, więc ten ruch jest zawsze możliwy), lub*
- *zdjęciu jednego klocka ze szczytu wybranej kolumny.*

Bajtazar nigdy nie był pewien, czy wybrane przez niego rozwiązanie było optymalne, i poprosił Cię o napisanie programu, który pomoże mu rozwiązywać ten problem.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczbę k i opis początkowego układu klocków,*
- *wyznaczy rozwiązanie wymagające minimalnej liczby ruchów,*
- *wypisze otrzymane rozwiązanie na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie liczby całkowite n oraz k ($1 \leqslant k \leqslant n \leqslant 100\,000$), oddzielone pojedynczym odstępem. W kolejnych n wierszach zapisane są początkowe wysokości kolumn klocków; wiersz $(i+1)$-szy zawiera jedną liczbę całkowitą $0 \leqslant h_i \leqslant 1\,000\,000$ — wysokość i-tej kolumny klocków, czyli liczbę klocków, z których się ona składa.

Wyjście

Na standardowe wyjście należy wypisać optymalne rozwiązanie, to jest układ klocków, który:

- *zawiera k kolejnych kolumn o tej samej wysokości,*
- *można otrzymać z początkowego układu w minimalnej liczbie ruchów.*

Wyjście powinno składać się z $n+1$ wierszy, a każdy z nich powinien zawierać jedną liczbę całkowitą. W pierwszym wierszu należy wypisać minimalną liczbę ruchów, potrzebnych do uzyskania żądanego układu. W $(i+1)$-szym wierszu (dla $1 \leqslant i \leqslant n$) należy wypisać liczbę h'_i — końcową wysokość i-tej kolumny klocków. W przypadku, gdy istnieje wiele rozwiązań, należy podać dowolne z nich.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`5 3`	`2`
`3`	`3`
`9`	`9`
`2`	`2`
`3`	`2`
`1`	`2`

Rozwiązanie

Problem możemy podzielić na dwie części:

- *uporządkowanie* — dla wybranych, dowolnych k kolejnych kolumn możemy obliczyć minimalny koszt wyrównania ich wysokości,
- *wybór* — znając powyższe, możemy spośród wszystkich kolumn wybrać te, których uporządkowanie będzie miało najniższy koszt.

Dalsza część rozwiązania będzie przebiegać zgodnie z nakreślonym powyżej planem.

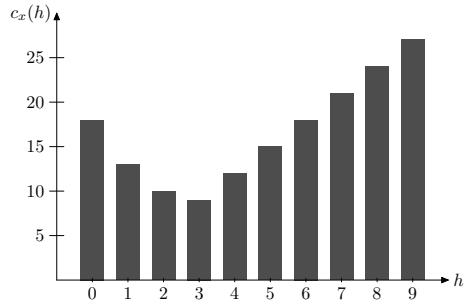
Uporządkowanie kolumn

Zajmijmy się ustalonymi k kolumnami klocków o wysokościach równych: $x_1, \ldots, x_k$. Zamierzamy je uporządkować, dodając i/lub zdejmując po jednym klocku tak, by ostatecznie były tej samej wysokości.

Niech $c_x(h)$ dla $x = (x_1, \ldots, x_k)$ oznacza koszt wyrównania rozważanych kolumn do wysokości h. Funkcję tę możemy wyrazić następującym wzorem:

$$c_x(h) = \sum_{i=1}^{k} |h - x_i|.$$

Przykład 1. Rozważmy konfigurację klocków podaną w przykładowych danych do zadania, czyli ciąg $x = (3,9,2,3,1)$. Na rys. 1 przedstawiony jest wykres funkcji $c_x(h)$ dla tego ciągu.



Rys. 1: Wykres wartości funkcji $c_x(h)$ dla ciągu $x = (3,9,2,3,1)$.

Na podstawie wykresu możemy zaobserwować, że funkcja $c_x(h)$ najpierw maleje, osiągając w pewnym punkcie minimum, a następnie rośnie. To, co nas interesuje, to wysokość h, dla której to minimum jest osiągane.

Poszukując „optymalnego" h, przyjrzyjmy się bliżej naszej funkcji. Najpierw zauważmy, że dla $h = 0$ wartością $c_x(0)$ jest:

$$c_x(0) = \sum_{i=1}^{k} x_i.$$

Następnie zobaczmy, jak zmienia się funkcja po zwiększeniu wysokości o 1. Otóż mając wartość $c_x(h-1)$ i chcąc obliczyć wartość $c_x(h)$:

- musimy dołożyć po jednym klocku do każdej kolumny, której wysokość jest mniejsza lub równa $h-1$,
- dla każdej kolumny, która początkowo miała wysokość większą lub równą h, oszczędzamy jeden ruch w porównaniu z uporządkowaniem jej na wysokości $h-1$ (musimy zdjąć jeden klocek mniej).

Bardziej formalnie:

$$\begin{aligned} c_x(h) &= c_x(h-1) + |\{x_i : x_i \leqslant h-1\}| - |\{x_i : x_i \geqslant h\}| \\ &= c_x(h-1) + |\{x_i : x_i \leqslant h-1\}| - (k - |\{x_i : x_i \leqslant h-1\}|) \\ &= c_x(h-1) + 2|\{x_i : x_i \leqslant h-1\}| - k. \end{aligned}$$

Niech $p_x(h)$ oznacza liczbę elementów ciągu x o wartości mniejszej lub równej h, czyli $p_x(h) = |\{x_i : x_i \leqslant h\}|$. Teraz wartości funkcji $c_x(h)$ możemy zapisać jako:

$$\begin{aligned} c_x(h) &= c_x(h-1) + 2p_x(h-1) - k \\ &= c_x(0) + \sum_{i=0}^{h-1} (2p_x(i) - k). \end{aligned}$$

Ponieważ funkcja $p_x(h)$ jest niemalejąca, to przy poszukiwaniu minimalnej wartości $c_x(h)$ opłaca się nam zwiększać h tak długo, jak długo składniki sumy w powyższym wzorze są ujemne. To oznacza, że $c_x(h)$ osiąga minimalną wartość dla wysokości $h = H$, takiej że

$$2p_x(H-1) - k < 0 \quad \text{oraz} \quad 2p_x(H) - k \geqslant 0. \tag{1}$$

Wartość H spełniająca nierówności (1) zawsze istnieje, ponieważ funkcja $p_x(i)$ zmienia wartości od 0 do k dla $i = -1, \ldots, \max\{x_j : 1 \leqslant j \leqslant k\}$, co oznacza, że funkcja $2p_x(i) - k$ dla tych samych argumentów przyjmuje wartości od $-k$ do k.

Spójrzmy raz jeszcze na nierówności (1). Dla poszukiwanej wysokości H chcemy mieć w ciągu x:

- mniej niż $\lceil k/2 \rceil$ kolumn o wysokościach mniejszych bądź równych $H-1$ oraz

- co najmniej $\lceil k/2 \rceil$ kolumn o wysokościach mniejszych bądź równych H.

Podstawiając za H wysokość $\lceil k/2 \rceil$-tej co do wysokości kolumny ciągu x, spełniamy oba powyższe warunki (a wyznaczony w ten sposób element nazywamy *medianą* ciągu x).

Wniosek 1. Funkcja $c_x(h)$ osiąga minimum dla H równego wartości $\lceil k/2 \rceil$-tego co do wielkości elementu ciągu x, gdzie k jest długością ciągu x.

Wybór kolumn

Korzystając z wniosku 1, możemy zapisać wstępny szkic algorytmu znajdowania najłatwiejszych do uporządkowania kolumn w całym ciągu $a_1, \ldots, a_n$:

```
1:  wynik := +∞;
2:  for i := 1,...,n+1−k do begin
3:    wyznacz medianę H ciągu x = (a_i,...,a_{i+k−1});
4:    oblicz optymalny koszt c_x(H) dla tego ciągu;
5:    wynik := min(wynik, c_x(H));
6:  end
```

Medianę ciągu k–elementowego potrafimy wyznaczyć w oczekiwanym czasie $O(k)$, korzystając z algorytmu Hoare'a (niestety pesymistyczny koszt tego algorytmu to $O(k^2)$). Możemy również zastosować algorytm Magicznych Piątek, którego pesymistyczny koszt wynosi $O(k)$ (niestety jest on trudniejszy w implementacji, a jego złożoność, choć teoretycznie dobra, jest opatrzona dużą stałą; przez to algorytm jest raczej mało praktyczny). Opisy obu algorytmów można odnaleźć w książkach [17, 20]. Wartości $c_x(H)$ można łatwo obliczyć (wprost z definicji) w czasie $O(k)$. To oznacza, że na razie mamy rozwiązanie zadania, którego złożoność czasowa wynosi $O(nk)$. Jeśli k jest małą stałą, to złożoność $O(nk)$ w zupełności nam wystarczy. Jednak według warunków zadania parametr k może osiągać wartości rzędu $\Theta(n)$, czyli złożoność czasową całego algorytmu musimy oszacować jako $O(n^2)$. W takim razie nasz algorytm wymaga jeszcze usprawnień.

Zauważmy, że kolejne ciągi x, dla których wyznaczamy medianę w kroku 3. algorytmu, są do siebie bardzo podobne. W i-tej iteracji analizujemy ciąg $(a_i, a_{i+1}, \ldots, a_{i+k-1})$, a w kolejnej — ciąg $(a_{i+1}, \ldots, a_{i+k-1}, a_{i+k})$. Ciągi te różnią się tylko dwoma elementami. Korzystając z tej obserwacji, możemy stworzyć wydajniejszy algorytm, jeśli tylko znajdziemy strukturę danych S, dla której będziemy potrafili efektywnie wykonać operacje:

- $przygotuj(x_1, x_2, \ldots, x_k)$ — umieszczenie w strukturze ciągu x o długości k,
- $dodaj(y)$ — dodanie do struktury elementu o wartości y,
- $usun(y)$ — usunięcie ze struktury elementu o wartości y,
- $mediana()$ — wyznaczenie mediany ciągu przechowywanego w strukturze,
- $minKoszt()$ — wyznaczenie wartości funkcji $c_x(mediana())$ dla ciągu przechowywanego w strukturze.

Używając struktury S, możemy zapisać następujący algorytm:

```
1:  S.przygotuj(a_1, ..., a_k);
2:  wynik := S.minKoszt();
3:  for i := 1, ..., n-k do begin
4:    S.usun(a_i);
5:    S.dodaj(a_{i+k});
6:    wynik := min(wynik, S.minKoszt());
7:  end
```

Pierwsza implementacja struktury — ulepszamy drzewa czerwono-czarne. Poszukajmy efektywnej implementacji struktury S. Jednym z możliwych rozwiązań jest zastosowanie struktury słownikowej (np. drzew czerwono-czarnych) wzbogaconej o dodatkowe możliwości. Sama struktura słownikowa pozwala nam efektywnie wyszukiwać, dodawać i usuwać elementy. W razie potrzeby możemy jednak do każdego węzła v dodać informacje typu: liczba czy suma elementów w poddrzewie ukorzenionym w v, minimalny lub maksymalny element w poddrzewie v. Informacje te stosunkowo łatwo można uaktualniać w czasie modyfikacji wykonywanych na strukturze (na przykład dodawania czy usuwania węzłów, rotacji itp.) bez zwiększania złożoności tych operacji. Dla naszych celów przydatne będzie zapisanie w każdym węźle v drzewa czerwono-czarnego:

- liczby elementów w poddrzewie ukorzenionym w v,
- sumy elementów w poddrzewie ukorzenionym w v.

Operacje dodawania i usuwania elementów wymagają oczywiście czasu $O(\log k)$. Korzystając z atrybutu *liczba elementów w poddrzewie*, możemy w czasie $O(\log k)$ odnaleźć medianę ciągu. Funkcję *minKoszt*() możemy również obliczyć w czasie $O(\log k)$. Wymaga to jedynie:

- odnalezienia węzła zawierającego medianę,
- wyznaczenia sumy oraz liczby kluczy leżących w drzewie na lewo od mediany (będą to elementy ciągu o wartości nie większej od mediany) — nazwijmy ten multizbiór[1] kluczy M,
- wyznaczenia sumy oraz liczby kluczy leżących w drzewie na prawo od mediany (będą to elementy ciągu o wartości nie mniejszej od mediany) — nazwijmy ten multizbiór kluczy W,
- obliczenia funkcji $c_x(mediana)$ za pomocą wzoru:

$$c_x(mediana) = (|M| \cdot mediana - \sum_{x \in M} x) + (\sum_{x \in W} x - |W| \cdot mediana).$$

Opisana implementacja struktury S pozwala zatem rozwiązać nasze zadanie w złożoności czasowej $O(n \log k)$. Niestety ma ona pewną dość istotną wadę — wymaga samodzielnej implementacji drzew czerwono-czarnych. Co prawda biblioteka STL (dla języka C++)

[1] Multizbiór to zbiór, w którym elementy o takiej samej wartości mogą występować wielokrotnie.

oferuje struktury słownikowe, jednak w chwili obecnej nie umożliwia wzbogacenia ich o wybrane atrybuty.

W przypadku naszego zadania można jednakże zaimplementować opisaną strukturę S znacznie prościej i w nieznacznie gorszej złożoności czasowej — $O(n \log n)$. W tym celu wystarczy na samym początku algorytmu posortować wszystkie wysokości kolumn klocków i zbudować statyczną, zrównoważoną strukturę drzewa — w postaci drzewa BST lub w postaci drzewa przedziałowego[2]. Wówczas wszystkie opisane operacje można zrealizować za pomocą analogicznego jak w powyższym opisie wzbogacenia tego drzewa. Struktura statyczna jest dużo przyjemniejsza w implementacji — można ją utrzymywać bezpośrednio w tablicy, a ponadto przy wstawianiu i usuwaniu elementów nie trzeba się martwić o operacje równoważące — i jedynie nieznacznie wolniejsza — wzrost kosztu czasowego wynika z konieczności wstępnego posortowania danych oraz stąd, że wysokość drzewa jest w tym przypadku rzędu $\Theta(\log n)$, a nie $\Theta(\log k)$ jak poprzednio. Implementację rozwiązania wykorzystującego statyczne drzewo zrównoważone można znaleźć w plikach `klo.cpp` oraz `klo1.pas`.

Druga implementacja struktury — po rozbiciu na części jest łatwiej. Strukturę S możemy także zaimplementować mniej pracochłonnie. Dla zbioru $(x_1, \ldots, x_k)$ stworzymy następujące „składowe" struktury:

- *mediana* — element stanowiący medianę ciągu,
- *mniejsze* — multizbiór elementów mniejszych bądź równych medianie,
- *wieksze* — multizbiór elementów większych bądź równych medianie,
- *sumaMniejsze*, *sumaWieksze* — odpowiednio suma elementów w multizbiorze *mniejsze*/*wieksze*.

Będziemy przy tym dbać, by w zbiorach *mniejsze* i *wieksze* było po tyle samo (z dokładnością do 1) elementów, czyli aby spełniony był niezmiennik:

$$|mniejsze| = \left\lceil \frac{|mniejsze| + |wieksze| + 1}{2} \right\rceil - 1.$$

Multizbiory *mniejsze*/*wieksze* możemy zaimplementować za pomocą struktury `multiset` z biblioteki STL.

Powracając do zadania, które musimy rozwiązać, pokażmy, jak za pomocą struktury S wykonać wszystkie potrzebne w algorytmie operacje:

- *przygotuj*(x) — możemy posortować zbiór x, wybrać jego medianę, a elementy od niej mniejsze (odpowiednio, większe) umieścić w multizbiorze *mniejsze* (odpowiednio, w multizbiorze *wieksze*), przy okazji zliczając sumy elementów;
- *mediana*$()$ — na to pytanie odpowiadamy w czasie stałym, podając wartość pola *mediana*;

[2] O statycznych drzewach przedziałowych można poczytać np. w opisie rozwiązania zadania Tetris 3D w [13].

- *minKoszt*() — wyznaczamy w czasie stałym wartość funkcji $c_x(mediana())$, korzystając z wartości *mediana*, *sumaMniejsze*, *sumaWieksze* oraz z liczby elementów w multizbiorach *mniejsze*/*wieksze*;

- *dodaj*(y) — najpierw dodajemy element do jednego z multizbiorów: jeśli $y \leqslant mediana$, to dodajemy element y do *mniejsze*, w przeciwnym przypadku dodajemy ten element do *wieksze*. Niestety, w ten sposób możemy zaburzyć niezmiennik:

$$|mniejsze| = \left\lceil \frac{|mniejsze| + |wieksze| + 1}{2} \right\rceil - 1.$$

  Aby go przywrócić, porównujemy moce zbiorów *mniejsze* i *wieksze*:

  – jeśli w zbiorze *mniejsze* brakuje jednego elementu, to dodajemy wartość *mediana* do zbioru *mniejsze*, ze zbioru *wieksze* usuwamy najmniejszą wartość i zapisujemy ją w polu *mediana*;
  – analogicznie, jeśli zbiór *mniejsze* zawiera o jeden element za dużo, to dodajemy wartość *mediana* do zbioru *wieksze*, ze zbioru *mniejsze* usuwamy największą wartość i zapisujemy ją w polu *mediana*.

  Struktura `multiset` z biblioteki STL pozwala wykonywać operacje dodawania, usuwania oraz znajdowania największego i najmniejszego elementu w czasie $O(\log k)$. Samodzielnie możemy ją także zaimplementować za pomocą drzew zrównoważonych lub odpowiednio wzbogaconych (np. podobnie jak w algorytmie Dijkstry) kopców[3].

- *usun*(y) — jeśli $y < mediana$, to usuwamy element y z *mniejsze*, jeśli $y > mediana$, to usuwamy element y z *wieksze*; w ostatnim możliwym przypadku mamy $y = mediana$ — usuwamy więc element y z pola *mediana*, a za nową wartość mediany przyjmujemy np. najmniejszy element ze zbioru *wieksze*. Na koniec równoważymy zbiory *mniejsze*/*wieksze*, podobnie jak to robiliśmy przy operacji *dodaj*.

W rozwiązaniu wzorcowym zastosowaliśmy wspomnianą strukturę `multiset` — jest ono zaimplementowane w pliku `klo2.cpp`. Ponieważ pesymistyczny koszt operacji *dodaj*/*usun* wynosi $O(\log k)$, to złożoność czasowa całego algorytmu to $O(n \log k)$.

Testy

Rozwiązania zawodników były sprawdzane na 28 testach, zgrupowanych w 11 zestawów.

Nazwa	n	k	Opis
klo1a.in	10	5	test poprawnościowy małego rozmiaru
klo1b.in	10	5	test poprawnościowy małego rozmiaru
klo1c.in	1	1	test poprawnościowy małego rozmiaru

[3] O wielu rodzajach drzew zrównoważonych i o kopcach można poczytać w podstawowych podręcznikach z algorytmiki: [15, 20].

Nazwa	n	k	Opis
klo1d.in	100	10	test poprawnościowy małego rozmiaru
klo2a.in	100	20	test poprawnościowy małego rozmiaru
klo2b.in	100	50	test poprawnościowy małego rozmiaru
klo2c.in	100	100	test poprawnościowy małego rozmiaru
klo3a.in	99	15	test poprawnościowy małego rozmiaru
klo3b.in	100	50	test poprawnościowy małego rozmiaru
klo4a.in	10 000	101	test poprawnościowy średniego rozmiaru
klo4b.in	10 000	5 000	test poprawnościowy średniego rozmiaru
klo5a.in	100 000	1	test poprawnościowy średniego rozmiaru
klo5b.in	100 000	1 000	test poprawnościowy średniego rozmiaru
klo5c.in	100 000	1 003	test poprawnościowy średniego rozmiaru
klo6a.in	100 000	2	test wydajnościowy
klo6b.in	100 000	10 001	test wydajnościowy
klo6c.in	99 889	1 000	test wydajnościowy
klo7a.in	100 000	10 000	test wydajnościowy
klo7b.in	100 000	10 000	test wydajnościowy
klo8a.in	100 000	10 000	test wydajnościowy
klo8b.in	100 000	10 000	test wydajnościowy
klo9a.in	90 000	90 000	test wydajnościowy
klo9b.in	100 000	10 000	test wydajnościowy
klo10a.in	99 999	101	test wydajnościowy
klo10b.in	100 000	10 000	test wydajnościowy
klo11a.in	95 000	1 001	test wydajnościowy
klo11b.in	100 000	10 000	test wydajnościowy
klo11c.in	100 000	10 000	test wydajnościowy