Jakub Radoszewski
Treść zadania, Opracowanie

Paweł Parys
Program

OI, Etap III, dzień drugi, 29.03.2007

Klocki

Agatka dostała na urodziny komplet klocków. Klocki mają kształt sześcianów i są wszystkie tej samej wielkości. Na każdym z klocków jest napisana jedna dodatnia liczba całkowita. Agatce klocki bardzo się spodobały i natychmiast ustawiła z nich wszystkich jedną wysoką wieżę.

Mama powiedziała Agatce, że celem zabawy klockami jest ustawienie wieży, w której jak najwięcej klocków znajdzie się na swoich miejscach. Klocek, na którym jest napisana liczba i, jest na swoim miejscu, jeżeli znajduje się w wieży na wysokości i (klocek na samym dole wieży jest na wysokości 1, klocek stojący na nim jest na wysokości 2 itd.). Agatka postanowiła ostrożnie pousuwać z wieży pewne klocki (starając się, żeby wieża się nie przewróciła), tak aby jak najwięcej klocków znalazło się w rezultacie na swoich miejscach. Doradź Agatce, które klocki najlepiej usunąć.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis wieży, jaką na początku ustawiła Agatka,*
- *wyznaczy, które klocki Agatka ma usunąć,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 100\,000$), oznaczającą początkową wysokość wieży klocków. Drugi wiersz wejścia zawiera n dodatnich liczb całkowitych: $a_1, a_2, \ldots, a_n$ ($1 \leqslant a_i \leqslant 1\,000\,000$), pooddzielanych pojedynczymi odstępami i oznaczających liczby napisane na klockach. Liczby te są podane w kolejności od klocka położonego najniżej do klocka położonego najwyżej.

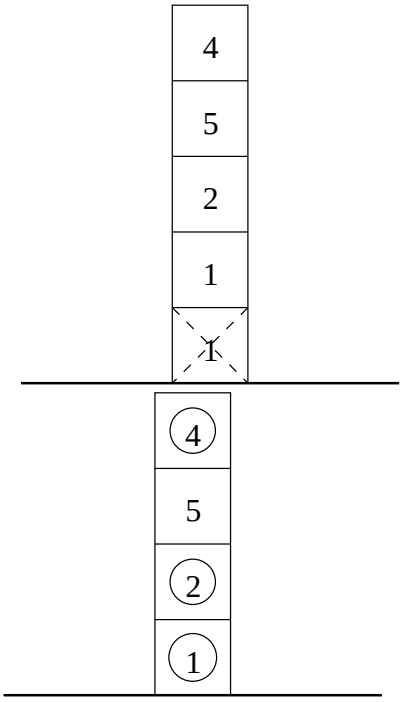
Wyjście

Twój program powinien wypisać w pierwszym wierszu liczbę klocków, które należy usunąć z wieży, aby zmaksymalizować liczbę klocków, które znajdą się na swoich miejscach. Drugi wiersz powinien zawierać numery usuwanych klocków (pooddzielane pojedynczymi odstępami). Klocki są ponumerowane kolejnymi liczbami od 1 do n w porządku od najniżej do najwyżej położonego klocka w początkowej wieży. Jeżeli istnieje więcej niż jedno rozwiązanie, Twój program powinien wypisać dowolne z nich.

Przykład

Dla danych wejściowych:

```
5
1 1 2 5 4
```



poprawnym wynikiem jest:

```
1
1
```


Rozwiązanie

Programowanie dynamiczne

Zadanie można rozwiązać na wiele sposobów z wykorzystaniem techniki programowania dynamicznego. Otrzymane rozwiązania mają w większości złożoność czasową $O(n^3)$ lub, te lepsze, $O(n^2)$. My także rozpoczniemy od rozwiązania o złożoności $O(n^2)$, a następnie je przyspieszymy.

W dalszej części *rozwiązaniem* będziemy nazywać początkową wieżę z usuniętymi dowolnymi klockami. *Rozwiązaniem optymalnym* nazwiemy rozwiązanie, w którym najwięcej klocków jest na swoich miejscach. Okazuje się, że poszukując rozwiązania optymalnego, możemy brać pod uwagę tylko takie rozwiązania, w których ostatni klocek (na szczycie wieży) jest na swoim miejscu. Każde inne rozwiązanie (także optymalne) można bowiem „poprawić", usuwając z niego wszystkie zbędne klocki z góry wieży, które nie są na swoim miejscu i nie mają wpływu na jakość rozwiązania. *Rozwiązaniem i-optymalnym* nazwiemy rozwiązanie optymalne wśród rozwiązań, w których na szczycie wieży jest klocek i-ty i jest on na swoim miejscu. Przez t_i (dla $1 \leqslant i \leqslant n$) oznaczymy liczbę klocków znajdujących się na swoich miejscach w rozwiązaniu i-optymalnym. Oczywiście wartością poszukiwaną w zadaniu jest maksimum z wartości t_i.

Pozostaje zatem problem, jak wyliczyć wartości t_i. Rozważmy klocek o numerze i dla pewnego $i \in \{1,\ldots,n\}$. Jeśli $a_i > i$, to nie istnieje rozwiązanie i-optymalne, gdyż a_i już na początku jest poniżej „swojej" docelowej pozycji, a usuwanie klocków tylko tę sytuację pogarsza. Przyjmujemy wówczas, że $t_i = 0$. Jeśli $a_i \leqslant i$, to rozwiązanie i-optymalne istnieje. Poszukując dokładnej wartości t_i zauważmy, że możliwe są sytuacje:

- $t_i = 1$, gdy w rozwiązaniu i-optymalnym tylko klocek i-ty jest na swoim miejscu;
- $t_i = 1 + t_j$, gdy w rozwiązaniu i-optymalnym klocek j-ty jest najwyższym spośród klocków znajdujących się pod klockiem i-tym, który znalazł się na swoim miejscu.

Wartość t_i obliczamy zatem ze wzoru:

$$t_i = \max(1, \max\{t_j + 1 : \text{dla niektórych } 1 \leqslant j < i\}),$$

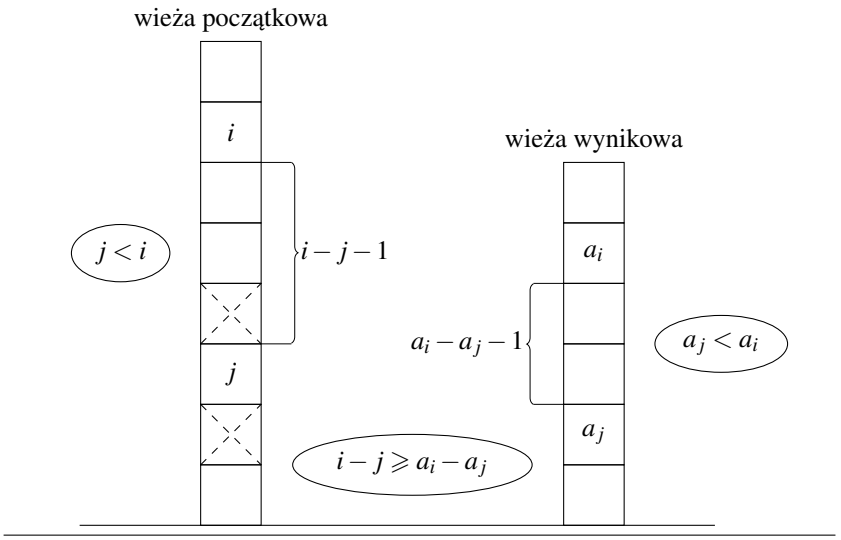
gdzie wewnętrzne maksimum jest wybierane po tych wartościach $j < i$, dla których klocek j-ty może być na swoim miejscu w rozwiązaniu, w którym na szczycie pozostaje klocek i-ty, także zajmujący swoje miejsce. Spróbujmy dokładniej określić warunki, jakie musi spełniać j:

$$\begin{aligned} j &< i, && (1)\\ a_j &< a_i, && (2)\\ j - a_j &\leqslant i - a_i. && (3) \end{aligned}$$

Pierwszy warunek jest oczywisty — klocek j-ty musi znajdować się poniżej klocka i-tego w wieży początkowej. Drugi warunek to analogiczny fakt dla wieży wynikowej — klocek j-ty, który trafi na pozycję a_j, musi znajdować się poniżej klocka i-tego, który trafi na pozycję a_i. Skąd jednak bierze się warunek 3? Zapisawszy go inaczej: $i - j + 1 \geqslant a_i - a_j + 1$, widzimy, iż oznacza on, że liczba klocków pomiędzy klockiem j-tym a i-tym w wieży wynikowej nie może być większa niż w początkowej (Agatka może wyłącznie usuwać klocki).

Na rys. 1 jest przedstawiona ilustracja warunków (1)–(3). Widać z niego także, że warunki te są wystarczające, by przy wyznaczaniu t_i skorzystać z wartości $t_j + 1$.
Możemy teraz zapisać procedurę wyznaczania t_i dla $1 \leqslant i \leqslant n$:

```
1:  for i := 1 to n do
2:  begin
3:    if a_i > i then
4:      t_i := 0
5:    else
6:    begin
7:      t_i := 1;
8:      for j := 1 to i-1 do
9:        if (a_j < a_i) and (j - a_j ⩽ i - a_i) then
10:         { Oczywiście zachodzi warunek 1: j < i. }
11:       t_i = max(t_i, t_j + 1);
12:   end
13: end
```



Rys. 1: Ilustracja nierówności (1)–(3).

Liczbę klocków znajdujących się na swoim miejscu w rozwiązaniu optymalnym otrzymujemy, wyliczając t_k, dla którego $t_k = \max(t_1, \ldots, t_n)$. Zadanie *Klocki* polega jednak na skonstruowaniu listy klocków, które należy usunąć, aby to rozwiązanie uzyskać. W tym celu uzupełnimy powyższy algorytm. Dla każdego i ($1 \leqslant i \leqslant n$) zapamiętamy wartość p_i, oznaczającą indeks j, dla którego ostatecznie $t_i = t_j + 1$ (czyli indeks znaleziony w pętli 8–11). Znając k oraz p_i dla wszystkich $i \in \{1, \ldots, n\}$, będziemy mogli odtworzyć sposób usuwania klocków, zgodnie z poniższym pseudokodem:

```
1: i := k;
2: a_0 := 0; { żeby dać poprawną odpowiedź w przypadku, gdy t_i = 1 }
3: while t_i > 0 do
4: begin
5:     j := p_i;
6:     Wybierz dowolnie (i − j) − (a_i − a_j) klocków spośród j + 1, ..., i − 1;
7:     Usuń wybrane klocki z wieży;
8:     i := j;
9: end
```

O poprawności zaproponowanej powyżej konstrukcji wieży można się przekonać, powracając do rys. 1 oraz uważnie analizując postępowanie w przypadku, gdy $t_i = 1$ (usuniętych zostaje $i - a_i$ klocków).

Złożoność całego rozwiązania to czas $O(n^2)$, potrzebny na wyznaczenie wartości t_i, oraz czas $O(n)$, konieczny do odtworzenia wyniku, czyli razem — zgodnie z zapowiedzią — $O(n^2)$. Rozwiązanie to zostało zaimplementowane w pliku `klos1.cpp`. Na zawodach można było za nie zdobyć (podobnie jak za inne rozwiązania o złożoności czasowej kwadratowej względem n) niecałe 50% punktów.

Próba usprawnienia poprzedniego rozwiązania

Najwięcej czasu w przedstawionym rozwiązaniu zajmuje wyznaczanie wartości t_i. Gdybyśmy efektywniej wyliczali $\max(t_j + 1 : j < i$ oraz $a_j < a_i$ oraz $j - a_j \leqslant i - a_i)$ dla zadanego i, to moglibyśmy istotnie przyspieszyć nasze rozwiązanie. Spróbujmy w tym celu zastosować odpowiednią strukturę danych — taką, w której będziemy mogli:

- przechowywać pary postaci (*element*, *wartość*) oraz
- efektywnie odpowiadać na pytania, jaka jest maksymalna *wartość* dla *elementów*, zawartych w wybranym podzbiorze zbioru wszystkich elementów.

Efektywność poszukiwanej struktury danych zależy znacząco od tego, jakiego typu elementy zamierzamy przechowywać i o jakiego typu zbiory elementów planujemy pytać. W interesującym nas przypadku elementy to *krotki liczb* (inaczej *wektory*), a zbiory specyfikujemy, podając dla każdej współrzędnej krotki zakres wartości (przedział). Dokładniej:

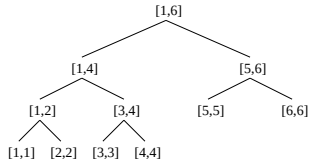
- j-ty klocek możemy reprezentować jako wektor $(j, a_j, j - a_j)$ o wartości $t_j + 1$,
- wartość t_i otrzymujemy jako odpowiedź na pytanie o maksimum z wartości elementów, których:
  - pierwsza współrzędna należy do przedziału $[1, i)$,
  - druga — do przedziału $[1, a_i)$,
  - trzecia — do przedziału $[0, i - a_i]$ (jeżeli $j - a_j < 0$, to i tak $t_j = 0$).

W jaki sposób można zrealizować opisaną wyżej strukturę danych? Rozważmy bardziej ogólny przypadek — załóżmy, że elementy to k-elementowe krotki (wektory k-wymiarowe). Pokażemy, że wówczas wstawienie elementu do struktury lub znalezienie odpowiedzi na pytanie o maksimum można wykonać w czasie $O(\log^k n)$.

Jeżeli $k = 1$, to mamy do czynienia z elementami-liczbami z przypisanymi wartościami liczbowymi, a zapytania dotyczą maksimum wartości przypisanych elementom z pewnych przedziałów. Dobrą strukturą dla tego problemu jest odpowiednio wzbogacone zrównoważone drzewo poszukiwań binarnych (na przykład AVL). Dla każdego elementu mamy w drzewie jeden liść, w którym jest zapisany element i jego wartość. Natomiast w każdym z węzłów wewnętrznych znajduje się zbiorcza informacja o poddrzewie zakorzenionym w tym węźle:

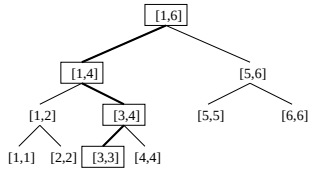
- maksimum z wartości przypisanych liściom z jego poddrzewa oraz
- granice przedziału, w którym zawierają się wszystkie elementy zapisane w jego liściach.

Omawiane drzewo najłatwiej zaimplementować, jeżeli z góry znamy wszystkie możliwe elementy (na przykład są to liczby naturalne z niedużego przedziału). W takim przypadku możemy zarezerwować po jednym liściu na każdy potencjalny element i na samym początku zbudować dobrze wyważone drzewo (tzw. drzewo statyczne). W trakcie działania algorytmu pozostanie nam jedynie „uaktywnianie" elementów, zamiast ich fizycznego wstawiania do drzewa, oraz aktualizacja informacji w wierzchołkach wewnętrznych. Strukturę taką



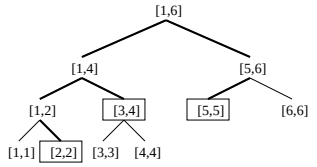
Rys. 2: Przykładowe drzewo statyczne dla elementów $1, 2, 3, 4, 5, 6$.

nazywamy *drzewem licznikowym* lub *drzewem przedziałowym*. Przykład dla elementów będących liczbami od 1 do 6 jest przedstawiony na rys. 2. Wstawienie nowego elementu odpowiada przejściu w drzewie od odpowiadającego mu liścia do korzenia i aktualizacji informacji we wszystkich węzłach na tej ścieżce (patrz rys. 3). Natomiast



Rys. 3: „Wstawienie" elementu 3 do drzewa z poprzedniego rysunku.

odpowiedź na pytanie o maksimum wartości elementów z danego przedziału znajdujemy, rozkładając ten przedział na *przedziały bazowe* (czyli przedziały związane z węzłami drzewa) i wyznaczając maksimum z wartości w węzłach im odpowiadających (patrz rys. 4). Dokładniejsze omówienie implementacji podobnej struktury można znaleźć na przykład



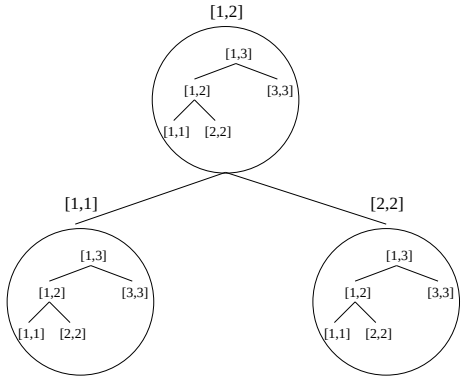
Rys. 4: Rozkład przedziału $[2, 5]$ na przedziały bazowe.

w opisie rozwiązania zadania *Tetris 3D* w książeczce XIII Olimpiady Informatycznej.

Dla bardziej złożonych elementów, gdy $k > 1$, możemy zbudować strukturę rekurencyjną — drzewo zrównoważone, w którego wierzchołkach znajdują się struktury wymiaru $k - 1$. Drzewo to jest skonstruowane według pierwszych współrzędnych elementów i w każdym wierzchołku zawiera:

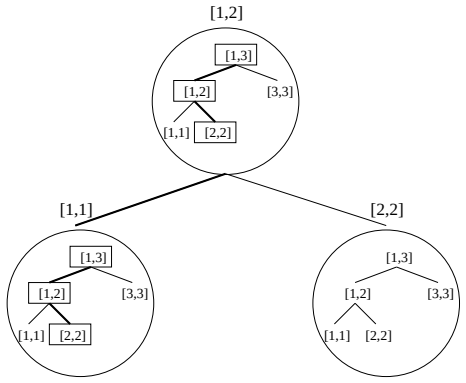
- granice przedziału dla pierwszej współrzędnej elementów wyznaczone analogicznie, jak w przypadku $k = 1$;
- strukturę wymiaru $k - 1$, zawierającą wszystkie elementy, których pierwsza współrzędna należy do przedziału związanego z danym wierzchołkiem.

Podobnie, jak w przypadku jednowymiarowym, i tym razem całą strukturę można zbudować na początku. Przykład struktury dla $k = 2$ i dla elementów, których pierwsza współrzędna należy do przedziału $[1, 2]$, a druga do przedziału $[1, 3]$, obrazuje rys. 5. Wstawienie elementu



Rys. 5: Przykład drzewa drzew dla elementów ze zbioru $[1,2] \times [1,3]$.

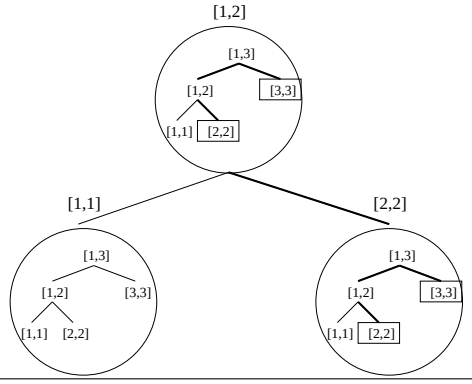
do struktury k-wymiarowej polega na wstawieniu go (rekurencyjnie) do wszystkich struktur $(k-1)$-wymiarowych, znajdujących się na ścieżce od odpowiedniego liścia drzewa do jego korzenia — patrz rys. 6. Proste rozumowanie indukcyjne pozwala wykazać, że złożoność



Rys. 6: Wstawienie elementu $(1,2)$ do struktury z poprzedniego rysunku.

czasowa operacji wstawienia elementu do struktury wynosi $O(\log^k n)$. W tym samym czasie możemy znaleźć maksimum wartości dla zbioru elementów, określonego poprzez zadanie przedziału dla każdej współrzędnej. Pytanie to sprowadzamy bowiem do $O(\log n)$ zapytań dla struktur $(k-1)$-wymiarowych (patrz rys. 7).

Powróćmy do problemu z zadania. Na początku rozdziału przedstawiliśmy klocki jako krotki 3-wymiarowe. Zastosowanie dla nich opisanej powyżej struktury pozwala więc rozwiązać zadanie w czasie $O(n \log^3 n)$. Proste spostrzeżenie pozwala zredukować ten czas do $O(n \log^2 n)$. Wystarczy zauważyć, że klocki rozważamy w kolejności rosnących pierwszych współrzędnych i, gdy pytamy o klocki, dla których pierwsza współrzędna należy do przedziału $[1,i)$, to pytamy o wszystkie dotychczas przetworzone klocki. To oznacza, że pierwszą współrzędną możemy pominąć i rozważać wszystkie przeanalizowane dotychczas klocki, dla których jedna współrzędna (początkowo druga) należy do przedziału $[1,a_i)$, a druga (początkowo trzecia) — do przedziału $[0,i-a_i]$.



Rys. 7: Węzły analizowane w trakcie zapytania o maksimum wartości elementów, których pierwsza współrzędna należy do przedziału $[2,2]$, a druga do przedziału $[2,3]$. Pytamy się zatem o maksimum wartości elementów należących do prostokąta $[2,2] \times [2,3]$.

Pomimo kolejnego usprawnienia nie będzie chyba dla nikogo niespodzianką, że żaden z zawodników, podobnie jak i autor programu wzorcowego, nie pokusił się o implementację opisanego tu rozwiązania. Problem tkwi w tym, że praktycznie nie ma szans, by implementacja struktury w wersji statycznej zmieściła się w zadanych ograniczeniach pamięciowych. Natomiast zaprogramowanie dynamicznie aktualizowanego drzewa drzew zrównoważonych (czyli struktury dla $k = 2$) jest praktycznie niemożliwe w trakcie pięciogodzinnej sesji (nawet samodzielna implementacja zwykłego drzewa AVL nie jest ani łatwa, ani przyjemna). Dodatkowo rozwiązanie to charakteryzuje się dużą stałą w złożoności czasowej, co uniemożliwiłoby zapewne zmieszczenie się w limicie czasowym. Z powyższych powodów zamieszczony opis struktury danych ma jedynie charakter szkicu. Natomiast zainteresowanym zawodnikom niewątpliwie warto polecić dokładniejszą analizę oraz implementację tego rozwiązania — można ją zrealizować szczególnie elegancko w funkcyjnym języku programowania, na przykład Ocamlu.

Rozwiązanie wzorcowe

Pod koniec poprzedniego rozdziału zredukowaliśmy złożoność naszego problemu, dzięki temu, że jeden z warunków (1)–(3) w naturalny sposób wynikał z porządku, w jakim rozważaliśmy klocki. Udało nam się w ten sposób wyeliminować sprawdzanie warunku (1). Zauważmy, że zmieniając porządek przeglądania klocków na kolejność według wartości a_i albo $i - a_i$, moglibyśmy zamiast warunku (1) wyeliminować równie dobrze warunek (2) albo (3). Spostrzeżenie to daje nam pewną swobodę wyboru, która bardzo nam się przyda, gdyż chcielibyśmy zredukować rozmiar struktury danych do przechowywania klocków o jeszcze jeden stopień, osiągając $k = 1$. Wówczas pozostanie nam zaimplementowanie zwykłego drzewa licznikowego.

W pliku `klob3.cpp` znajduje się *niepoprawne* rozwiązanie, w którym przy wyznaczaniu wartości t_i pominięty został warunek (3). *Nie dostaniemy* także dobrego rozwiązania, gdy pominiemy warunek (2). Okazuje się jednak, że możemy zrezygnować z warunku (1)! Jeśli bowiem przy wyznaczaniu wartości t_i ograniczymy się do j, dla których zachodzą warunki

(2)–(3), czyli $a_j < a_i$ oraz $j - a_j \leqslant i - a_i$, to automatycznie mamy zagwarantowaną własność $j < i$, która wynika z dwu pozostałych. Możemy więc wybrać porządek rozpatrywania klocków według wartości a_i i zbudować drzewo licznikowe dla elementów $j - a_j$, dla których będziemy pamiętać wartości t_j. W ten sposób podczas rozważania klocka i-tego:

- warunek (2), czyli ograniczenie się do klocków, dla których $a_j < a_i$, wyniknie z kolejności rozpatrywania klocków,
- warunek (3) spełnimy, wyszukując w strukturze elementy z przedziału $[0, i - a_i]$,
- warunek (1) wyniknie sam z pozostałych dwu.

Tym samym otrzymaliśmy eleganckie rozwiązanie o złożoności czasowej $O(n \log n)$, które zostało zaimplementowane w pliku `klob8.cpp` i ... jest rozwiązaniem błędnym! Rozwiązanie to pozwalało zdobyć na zawodach nieco ponad połowę punktów.

Zastanówmy się, gdzie tkwi błąd. Otóż zapomnieliśmy, że nierówności z warunków (1) oraz (2) muszą być *ostre*. Dla warunku (1) nie musimy się tym przejmować — i tak każdy klocek ma inny numer w początkowej wieży, więc nierówności automatycznie są ostre. Jednak rozważając klocki uporządkowane według wartości a_i, musimy jakoś rozstrzygać remisy. Gdybyśmy się nimi nie przejmowali, to dla $j < i$ oraz $a_j = a_i$, licząc t_i, moglibyśmy wziąć pod uwagę $t_j + 1$. Jest to błąd, bo przecież nie możemy jednocześnie w rozwiązaniu mieć obu klocków: i-tego oraz j-tego, zajmujących w wieży wynikowej tę samą pozycję $a_i = a_j$. Najłatwiejszym sposobem wybrnięcia z tej kłopotliwej sytuacji jest uważne posortowanie klocków według wartości a_i — w taki sposób, by klocki z takimi samymi numerami były rozważane w kolejności *od góry do dołu*. Wówczas, jeżeli $j < i$ oraz $a_j = a_i$, to najpierw rozważymy klocek i-ty, a rozpatrując go, nie będziemy jeszcze mieli w drzewie klocka j-tego. Przy liczeniu t_i nie uwzględnimy więc $t_j + 1$, czyli unikniemy poprzedniego błędu. W ten sposób otrzymujemy poprawny sposób wyznaczania wartości t_i, zaimplementowany w plikach `klo.cpp` oraz `klo2.pas`.

Pozostaje jeszcze jeden szczegół, o którym wypada przypomnieć — poszukiwanym rozwiązaniem nie jest t_i, lecz numery klocków, które należy usunąć. Aby je wyznaczyć, posłużymy się, podobnie jak w rozwiązaniu o złożoności kwadratowej względem n, wartościami p_i. To oznacza, że w węzłach drzewa licznikowego, oprócz wartości maksymalnych będziemy zapisywać, skąd te maksima pochodzą (z których liści drzewa się wywodzą). Znając wartości p_i, możemy wygenerować rozwiązanie identycznie, jak w poprzednim algorytmie.

Nietrudno zauważyć, że wymaganie w odpowiedzi numerów klocków do usunięcia zamiast wartości t_i nie ma wpływu na trudność problemu. Można by więc spytać, dlaczego autorom zadania nie wystarcza, by program wyznaczał optymalną liczbę klocków pozostających na swoim miejscu po przebudowie. Odpowiedź na to pytanie jest prosta: znajomość samego wyniku nie pomogłaby Agatce w pousuwaniu z wieży odpowiednich klocków!

Inne spojrzenie na rozwiązanie wzorcowe

Problem wyznaczania wartości t_i możemy sformułować także trochę inaczej. Przedstawmy ponownie każdy klocek jako trójwymiarową krotkę o współrzędnych $(i, a_i, i - a_i)$. Licząc

t_i, bierzemy pod uwagę wartość $t_j + 1$, jeżeli krotka odpowiadająca j-temu klockowi ma pierwsze dwie współrzędne mniejsze od krotki odpowiadającej klockowi i-temu, a trzecią — nie większą. Wynika to wprost z warunków (1)–(3). To oznacza, że klocki, które w skonstruowanym przez nas rozwiązaniu optymalnym znajdą się na swoich miejscach, będą tworzyły ciąg punktów z przestrzeni trójwymiarowej:

- rosnący ze względu na pierwszą i drugą współrzędną oraz
- niemalejący ze względu na trzecią współrzędną.

Podobnie jak poprzednio, możemy zauważyć, że ciąg właściwie uporządkowany ze względu na drugą i trzecią współrzędną, musi być rosnący ze względu na pierwszą współrzędną. To oznacza, że wystarczy reprezentować klocki jako krotki dwuwymiarowe postaci $(a_i, i - a_i)$, a konstruowany ciąg będzie rosnący względem pierwszej współrzędnej i zarazem niemalejący względem drugiej (taki ciąg nazwiemy *rosnąco-niemalejącym*).

W jaki sposób taki ciąg wyznaczyć? Można, na przykład, ustawić wszystkie krotki w ciąg S, sortując je niemalejąco względem drugiej współrzędnej $(i - a_i)$, a ewentualne remisy rozstrzygając na korzyść krotek o mniejszej pierwszej współrzędnej a_i. Okazuje się, że wybierając podciąg ciągu S rosnący ze względu na pierwszą współrzędną (a_i), dostajemy rosnąco-niemalejący ciąg krotek, jeśli weźmiemy pod uwagę obie współrzędne $(a_i, i - a_i)$. Zachodzi także stwierdzenie odwrotne — każdy rosnąco-niemalejący ciąg krotek odpowiada podciągowi rosnącemu ze względu na pierwszą współrzędną w ciągu S. W ten sposób sprowadziliśmy rozważany problem do wyznaczania (długości) najdłuższego podciągu rosnącego ciągu liczbowego. Jest to klasyczne zadanie, które można wykonać w złożoności czasowej $O(n \log n)$ — na przykład za pomocą drzewa przedziałowego. Istnieje także inny algorytm, niewykorzystujący złożonych struktur danych, a oparty jedynie na wyszukiwaniu binarnym (jego opis można znaleźć w opracowaniu zadania *Egzamin na prawo jazdy* w niniejszej książeczce). Duża grupa zawodników, którzy rozwiązali niniejsze zadanie, uzyskując maksymalną liczbę punktów, wykorzystała metodę podobną do powyższej.

Na koniec warto jeszcze chwilę zastanowić się, czy przy konstrukcji ciągu S wybór współrzędnej, według której sortujemy, jest istotny. Przypomnijmy, że chodzi nam o znalezienie podciągu właściwie uporządkowanego względem obu współrzędnych:

- dla drugiej $(i - a_i)$ porządek niemalejący gwarantujemy sobie, sortując elementy,
- dla pierwszej (a_i) porządek rosnący uzyskujemy, wybierając podciąg rosnący.

Czy coś stoi na przeszkodzie, by porządek dla pierwszej współrzędnej zapewnić w trakcie sortowania (budowy ciągu S), a dla drugiej — w trakcie wybierania podciągu (tym razem niemalejącego) z ciągu S? Owszem, jest pewien drobny problem, na który natknęliśmy się już wcześniej. Powiedzmy, że posortowaliśmy punkty niemalejąco względem pierwszej współrzędnej (a_i) i szukamy podciągu niemalejącego względem drugiej $(i - a_i)$, a w danych trafiły się nam dwa klocki, dla których $a_i = a_j$ oraz $j < i$. Wówczas może się zdarzyć, że do ciągu wynikowego weźmiemy oba te klocki (bo $j - a_j < i - a_i$), a przecież nie mogą one równocześnie być na swoim miejscu w rozwiązaniu wynikowym. Wcześniej opisany wybór porządku sortowania powoduje natomiast, że możemy poszukiwać podciągu *rosnącego*, a nie *niemalejącego* i problem klocków o równych wartościach a_i po prostu nie występuje.

Inne rozwiązania

Jeszcze inne rozwiązanie poprawne

Na początku opracowania stwierdziliśmy, że istnieje wiele poprawnych rozwiązań opartych na technice programowania dynamicznego. Aby nie być gołosłownym, zaprezentujemy krótko jeszcze jedno podejście.

Rozwiązanie nazwiemy (h,c)-rozwiązaniem, jeżeli wieża wynikowa ma h klocków, z których dokładnie c znajduje się na swoim miejscu. Zdefiniujmy $T_{h,c}$ jako minimalną wysokość dowolnego fragmentu wieży początkowej, z którego można, usuwając klocki, uzyskać (h,c)-rozwiązanie. Dla $h > 0$ oraz $c > 0$ mamy następujący wzór:

$$T_{h,c} = \min(T_{h-1,c} + 1, \min\{i : a_i = h \text{ oraz } T_{h-1,c-1} < i\}),$$

przy czym, jeżeli dla pewnych h oraz c nie istnieje (h,c)-rozwiązanie, to $T_{h,c} = \infty$. Aby przekonać się o poprawności podanego wzoru, przeanalizujmy konstrukcję (h,c)-rozwiązania. Chcąc skonstruować wieżę o wysokości h i c klockach zajmujących swoje pozycje:

- możemy wziąć niższą o jeden klocek wieżę już zawierającą c klocków na swoich miejscach ($(h-1,c)$-rozwiązanie) i na górę dostawić najniższy z pozostałych po jej budowie klocków (stąd we wzorze wyrażenie $T_{h-1,c}+1$) albo
- jeśli $a_i = h$ oraz z klocków o numerach $\{1,2,\ldots,i-1\}$ da się zbudować $(h-1,c-1)$-rozwiązanie (czyli $T_{h-1,c-1} < i$), to możemy dołożyć na jego szczycie pasujący tam klocek i-ty.

W drugim z przypadków, numer i dostawianego klocka decyduje o wartości $T_{h,c}$, a zatem chcemy wybrać najniższy tego typu klocek (stąd minimum po i we wzorze).

Uzupełniając podany wyżej wzór dla przypadków skrajnych (kiedy $h = 0$, $c = 0$ lub $T_{h-1,c} = n$), otrzymujemy metodę wyznaczenia całej tablicy T. Maksymalna wartość c, dla której istnieje wartość h, taka że $T_{h,c} \neq \infty$, jest wówczas poszukiwanym rozwiązaniem — maksymalną liczbą klocków, które mogą znaleźć się na swoich miejscach w wieży wynikowej. W najprostszej implementacji rozwiązanie to ma złożoność czasową $O(n^3)$ (trzeba wyliczyć wartości $O(n^2)$ komórek tablicy, a wyznaczenie każdej z nich wymaga czasu $O(n)$). Sprytniejsze przeanalizowanie wszystkich klocków, dla których $a_i = h$ (przy ustalonej wartości parametru h), pozwala usprawnić powyższe rozwiązanie i wykonać obliczenia w złożoności czasowej $O(n^2)$. Rozwiązanie można jeszcze bardziej przyspieszyć, otrzymując złożoność czasową $O(n \log n)$. Duży stopień skomplikowania tej metody spowodował jednak, iż zdecydowaliśmy się pominąć jej opis — zainteresowani mogą jednak prześledzić jej implementację w pliku `klo5.cpp`.

Rozwiązania błędne

W pliku `klob4.cpp` znajduje się rozwiązanie zachłanne, w którym budujemy wieżę, wybierając w każdym kroku klocek o najmniejszym numerze i, który możemy umieścić na swoim miejscu w wieży wynikowej. To rozwiązanie przechodziło zaledwie jeden test. Rozwiązanie `klob6.cpp`, także niepoprawne, działa podobnie, jednak jako kolejny do umieszczenia na swoim miejscu wybiera klocek o najmniejszej wartości a_i. Takie rozwiązanie nie pozwalało zdobyć na zawodach żadnych punktów.

Testy

Zadanie było sprawdzane na 11 zestawach danych wejściowych. Wszystkie testy, z wyjątkiem testów z grup b, to testy generowane w pewnym stopniu losowo. Wykorzystane były dwie metody losowej generacji danych:

- losowe rozmieszczenie liczb na klockach — takie testy w poniższej tabelce nazywamy po prostu „losowymi”,
- rozmieszczenie na klockach liczb wzrastających stopniowo, tzn. liczb losowych położonych niedaleko pewnej liniowej funkcji rosnącej (zaletą takich testów jest istnienie rozwiązania, w którym duża liczba klocków znajduje się na swoich miejscach) — testy te nazwaliśmy „podłużnymi”.

W poniższym zestawieniu przez n oznaczono rozmiar testu, a przez w wynik (liczbę klocków, które mogą znaleźć się na swoich miejscach).

Nazwa	n	w	Opis
klo1.in	100	9	test losowy
klo2.in	500	23	test losowy
klo3.in	2 000	43	test losowy
klo4.in	7 000	2 430	test podłużny
klo5.in	8 000	87	test losowy
klo6.in	50 000	218	test losowy
klo7.in	100 000	41 019	test podłużny
klo8.in	100 000	2 206	test podłużny
klo9a.in	100 000	486	test podłużny
klo9b.in	100 000	100 000	test od razu ułożony
klo10a.in	100 000	24 303	test podłużny
klo10b.in	100 000	0	test złożony tylko z dużych liczb
klo11.in	100 000	4 975	test podłużny