Jacek Tomasiewicz	Michał Pilipczuk Jacek Tomasiewicz	Michał Pilipczuk
Treść zadania	Opracowanie	Program

Dostępna pamięć: 64 MB. *OI, Etap II, dzień pierwszy, 10.02.2010*

Klocki

Bajtek dostał na urodziny komplet drewnianych klocków. Klocki są nierozróżnialne, mają kształt jednakowej wielkości sześcianów. Bajtek układa klocki jeden na drugim, tworząc w ten sposób słupki. Zbudował cały rządek takich słupków, jeden obok drugiego, w linii prostej. Słupki mogą mieć różne wysokości.

Tata Bajtka, Bajtazar, zadał mu zagadkę. Podał mu liczbę k i poprosił, żeby tak poprzestawiał klocki, aby jak najwięcej kolejnych słupków miało wysokość przynajmniej k klocków. Przy tym, klocki można przekładać tylko w określony sposób: klocek można wziąć tylko ze słupka, którego wysokość przekracza k, i przełożyć na sąsiedni słupek. Podczas przekładania nie można tworzyć nowych słupków, klocki wolno przekładać tylko pomiędzy już istniejącymi.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite oddzielone pojedynczym odstępem: n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczająca liczbę słupków, oraz m ($1 \leqslant m \leqslant 50$), oznaczająca liczbę pytań Bajtazara. Słupki są ponumerowane od 1 do n. W drugim wierszu znajduje się n liczb całkowitych $x_1, x_2, \ldots, x_n$ pooddzielanych pojedynczymi odstępami ($1 \leqslant x_i \leqslant 1\,000\,000\,000$). Liczba x_i oznacza wysokość i-tego słupka. W trzecim wierszu znajduje się m liczb całkowitych $k_1, k_2, \ldots, k_m$ pooddzielanych pojedynczymi odstępami ($1 \leqslant k_i \leqslant 1\,000\,000\,000$). Są to kolejne liczby k, dla których należy rozwiązać zagadkę, czyli wyznaczyć największą możliwą liczbę kolejnych słupków o wysokości co najmniej k, jakie można uzyskać za pomocą poprawnych przestawień przy tej wartości parametru k.

Wyjście

Twój program powinien wypisać na standardowe wyjście m liczb całkowitych pooddzielanych pojedynczymi odstępami — i-ta z tych liczb powinna być odpowiedzią na zagadkę dla zadanego zestawu słupków oraz parametru k_i.

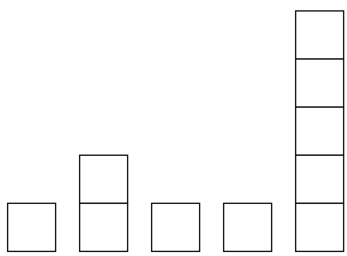
Przykład

Dla danych wejściowych:

```
5 6
1 2 1 1 5
1 2 3 4 5 6
```

poprawnym wynikiem jest:

```
5 5 2 1 1 0
```



Rozwiązanie

Uproszczenie zadania

Rozważmy rządek słupków o wysokościach $x_1, x_2, \ldots, x_n$ oraz jedną, konkretną wartość parametru k. *Fragmentem* ciągu x nazwiemy jego spójny podciąg, czyli podciąg złożony z pewnej liczby kolejnych elementów ciągu x. Powiemy, że fragment $x_i, x_{i+1}, \ldots, x_j$ jest *poprawnie rozmieszczony*, jeżeli każdy słupek w tym fragmencie ma wysokość co najmniej k (takie fragmenty chcemy umieć uzyskiwać). Dalej, fragment $x_i, x_{i+1}, \ldots, x_j$ nazwiemy *dobrym*, jeżeli możemy go sprowadzić do fragmentu poprawnie rozmieszczonego, wykonując (wielokrotnie) w ciągu x operację polegającą na przestawieniu pojedynczego klocka ze słupka o wysokości przekraczającej k na sąsiedni słupek (czyli zgodnie z treścią zadania). W tym zadaniu interesuje nas długość najdłuższego dobrego fragmentu ciągu x.

Niestety, nie jest wcale łatwo wyszukiwać w ciągu dobre fragmenty, gdyż podczas przekształcania takich fragmentów w poprawnie rozmieszczone możemy używać klocków pochodzących z bardzo odległych miejsc w ciągu x. Dlatego też wprowadzimy pojęcie *bardzo dobrego* fragmentu, czyli fragmentu, który można przekształcić w poprawnie rozmieszczony za pomocą operacji wykonywanych wyłącznie na słupkach *z tego fragmentu*. Szczęśliwie okazuje się, że aby rozwiązać zadanie, wystarczy zajmować się bardzo dobrymi fragmentami wyjściowego ciągu, patrz poniższe twierdzenie.

Twierdzenie 1. *Najdłuższy dobry fragment ciągu x jest zarazem najdłuższym bardzo dobrym fragmentem tego ciągu.*

Dowód: Każdy bardzo dobry fragment ciągu jest oczywiście także dobrym fragmentem ciągu. Odwrotna zależność niestety nie zachodzi, jednakże pokażemy, że każdy *najdłuższy* dobry fragment ciągu musi być bardzo dobry.

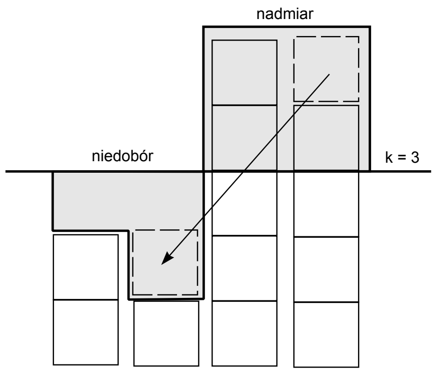
Załóżmy przez sprzeczność, że najdłuższy dobry fragment $x_i, x_{i+1}, \ldots, x_j$ nie jest bardzo dobry. To oznacza, że podczas wykonywania operacji na ciągu x prowadzących do przekształcenia tego fragmentu w poprawnie rozmieszczony, trafił do niego jakiś klocek spoza tego fragmentu. Załóżmy, że był to klocek pochodzący ze słupka x_u dla $u < i$ (przypadek $u > j$ rozpatruje się analogicznie). Twierdzimy, że wówczas po wykonaniu wszystkich operacji wszystkie słupki $x_u, x_{u+1}, \ldots, x_{i-1}$ mają wysokość co najmniej k, co oznacza, że fragment $x_i, x_{i+1}, \ldots, x_j$ bynajmniej nie był najdłuższym dobrym, gdyż dobry jest także fragment $x_u, x_{u+1}, \ldots, x_i, \ldots, x_j$.

No dobrze, a dlaczego wszystkie te słupki mają wysokość co najmniej k? Aby ów klocek ze słupka x_u mógł dostać się do rozważanego fragmentu, z każdego ze słupków $x_u, x_{u+1}, \ldots, x_{i-1}$ musiał zostać co najmniej raz przestawiony na słupek sąsiadujący z nim po prawej. To oznacza, że każdy z tych słupków musiał mieć wówczas wysokość przekraczającą k. Aby zakończyć dowód, wystarczy teraz zauważyć, że w wyniku wykonywania opisanych w zadaniu operacji nie da się obniżyć żadnego słupka o wysokości co najmniej k do wysokości mniejszej niż k. ■

Wprowadźmy dwa następujące parametry związane z fragmentami $x_i, x_{i+1}, \ldots, x_j$ ciągu x:

- liczba brakujących klocków (*niedobór*) — jest to minimalna liczba klocków, jakie musielibyśmy postawić na pewnych słupkach rozważanego fragmentu, aby wszystkie słupki tego fragmentu miały wysokość co najmniej k,
- liczba nadmiarowych klocków (*nadmiar*) — jest to liczba klocków, które znajdują się w rozważanym fragmencie powyżej wysokości k.

Oznaczmy je odpowiednio przez $p(x_i, \ldots, x_j;\ k)$ oraz $q(x_i, \ldots, x_j;\ k)$.



Rys. 1: Rysunek przedstawiający nadmiar q oraz niedobór p klocków. Zobrazowana jest także operacja przeniesienia jednego klocka z nadmiaru do najbliższego słupka o wysokości mniejszej niż k.

Następujące twierdzenie dostarcza wygodnej charakteryzacji bardzo dobrych fragmentów ciągu x, używając wprowadzonych pojęć niedoboru i nadmiaru.

Twierdzenie 2. *Fragment $x_i, x_{i+1}, \ldots, x_j$ ciągu słupków jest bardzo dobry wtedy i tylko wtedy, gdy nadmiar w tym fragmencie jest nie mniejszy od niedoboru, czyli*

$$q(x_i, \ldots, x_j;\ k) \geqslant p(x_i, \ldots, x_j;\ k).$$

Dowód: Jedna strona równoważności zawartej w tezie twierdzenia jest całkiem oczywista: jeśli fragment ciągu jest bardzo dobry, to niedobór nie może w nim przekraczać nadmiaru, gdyż wówczas w żaden sposób nie udałoby się zapełnić w tym fragmencie całego niedoboru.

Udowodnimy teraz implikację w drugą stronę: jeżeli nadmiar (q) jest nie mniejszy od niedoboru (p), to fragment $x_i, x_{i+1}, \ldots, x_j$ jest bardzo dobry. W tym celu pokażemy sekwencję operacji, która sprowadza nasz fragment do poprawnie rozmieszczonego.

Jeżeli $p = 0$, to wiemy, że wszystkie słupki fragmentu mają wysokość co najmniej k, więc nie musimy nic robić. Jeśli nie, to możemy przejść do pewnego stanu wysokości słupków $x'_i, x'_{i+1}, \ldots, x'_j$, w którym p jest mniejsze o 1. Wystarczy, że weźmiemy jakikolwiek słupek, którego wysokość jest większa niż k (wiemy, że taki słupek istnieje, ponieważ $q \geqslant p$), i przeniesiemy jeden klocek do najbliższego słupka, którego wysokość jest mniejsza od k. Ponieważ jest on najbliższy, więc wszystkie słupki pomiędzy nimi mają wysokość co najmniej k, a zatem możemy kolejno przenosić klocek pomiędzy tymi słupkami. W ten sposób zarówno p jak i q zmniejszyły się o 1 (ponieważ jeden nadmiarowy klocek uzupełnił lukę w brakujących klockach). Warunek $q \geqslant p$ nie zmienił się. Schemat ten możemy powtórzyć p razy, dzięki czemu liczba brakujących klocków spadnie do 0. Oznaczać to będzie, że wszystkie słupki będą miały wysokość co najmniej k. ■

Następująca obserwacja jest wnioskiem z dwóch powyższych twierdzeń.

Wniosek 1. Aby rozwiązać zadanie, wystarczy dla każdego zapytania k znaleźć największą długość fragmentu ciągu wysokości słupków x, w którym nadmiar jest co najmniej taki jak niedobór.

Narzucające się rozwiązania

Po uproszczeniu zadania możemy z łatwością wskazać nieoptymalne rozwiązania naszego problemu.

Pierwsze – wolniejsze

Możemy przejrzeć każdy fragment ciągu słupków i wybrać najdłuższy z tych, w których nadmiar jest nie mniejszy od niedoboru. Wartość nadmiaru i niedoboru obliczamy liniowo dla każdego z fragmentów. Fragmentów jest $\frac{n(n+1)}{2}$, zapytań m, więc daje nam to sumaryczny czas $O(mn^3)$.

Za tak rozwiązane zadanie uzyskiwało się na zawodach 20 punktów. Implementacja znajduje się w pliku `klos1.cpp`.

Drugie – szybsze

Możemy poprawić trochę złożoność powyższego rozwiązania. Zauważmy, że nie interesuje nas dokładna wartość nadmiaru oraz niedoboru. Wystarczyłoby nam znać wartość różnicy nadmiar minus niedobór, czyli „$q-p$". Jeśli różnica ta byłaby nieujemna, to liczba klocków nadmiarowych byłaby nie mniejsza od liczby klocków brakujących.

Przyjmijmy, że liczby brakujących klocków trzymamy jako liczby ujemne, a nadmiarowych — jako dodatnie. Wtedy dla pojedynczego, i-tego słupka możemy obliczyć wartość $w_i = x_i - k$, która oznacza różnicę „$q-p$" dla tego właśnie słupka.

A jak to wygląda dla dłuższych fragmentów $(x_i, x_{i+1}, \ldots, x_j)$? Otóż wystarczy zauważyć, że każdy ze słupków zawartych w takim fragmencie dostarcza albo tylko nadmiaru, albo tylko niedoboru (albo ma wysokość dokładnie k, a wówczas możemy przyjąć optymistycznie, że ma nadmiar równy 0). To oznacza, że różnicę „$q-p$" dla fragmentu możemy wyznaczyć, sumując nadmiary z nadmiarowych słupków i odejmując od tego sumę niedoborów z pozostałych słupków. Zauważmy jednak, że dokładnie taki sam wynik uzyskamy, sumując po prostu odpowiednie elementy ciągu w:

$$q(x_i, \ldots, x_j;\ k) - p(x_i, \ldots, x_j;\ k) \ = \ w_i + w_{i+1} + \ldots + w_j. \qquad (1)$$

Aby móc efektywnie obliczać takie sumy, wyznaczymy ciąg sum częściowych ciągu w. Przypomnijmy, że jest on zdefiniowany jako $a_i = w_1 + w_2 + \ldots + w_i$ i możemy go łatwo obliczyć w koszcie czasowym $O(n)$:

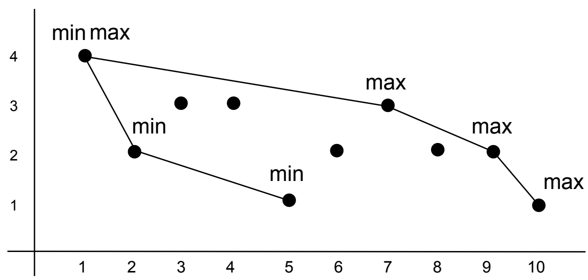
$a[0] := 0;$
for $i := 1$ **to** n **do** $a[i] := a[i-1] + w[i];$

W ten oto sposób wartość (1) możemy wyznaczyć w czasie stałym ze wzoru $a_j - a_{i-1}$. Dzięki temu złożoność czasowa rozwiązania spada do $O(mn^2)$. Za takie rozwiązanie można było uzyskać około 40 punktów. Implementacja znajduje się w pliku `klos2.cpp`.

W poszukiwaniu lepszego algorytmu

W powyższym, drugim rozwiązaniu nieefektywnym sprowadziliśmy wyjściowy problem do następującego: Mamy dany ciąg liczb $a_0, a_1, \ldots, a_n$ i poszukujemy dwóch indeksów l i r, dla których $a_r \geqslant a_l$ oraz $r - l$ jest największe możliwe. Odtąd skupimy się już tylko na tym właśnie sformułowaniu problemu.

Definicja 1. *Minimum prefiksowym* w ciągu a nazwiemy taki indeks l, dla którego wartości $a_0, a_1, \ldots, a_{l-1}$ są większe niż a_l. Analogicznie *maksimum sufiksowym* w ciągu a nazwiemy taki indeks r, dla którego wartości $a_{r+1}, a_{r+2}, \ldots, a_n$ są mniejsze niż a_r.



Rys. 2: Rysunek z minimami prefiksowymi (min) i maksimami sufiksowymi (max) w przykładowym ciągu $a = (4,\ 2,\ 3,\ 3,\ 1,\ 2,\ 3,\ 2,\ 2,\ 1)$.

Zauważmy, że optymalne indeksy l i r muszą być odpowiednio minimum prefiksowym i maksimum sufiksowym. Dlaczego? Gdyby l nie było minimum prefiksowym, to można byłoby je zamienić na największe $l' < l$ będące minimum prefiksowym. Ponieważ $a_{l'}$ byłoby nie większe od a_l, więc warunek $a_r \geqslant a_{l'}$ byłby tym bardziej spełniony, a wartość $r - l'$ byłaby jeszcze większa. Analogicznie wygląda sytuacja z r i maksimami sufiksowymi.

Łatwo zauważyć, że wartości elementów ciągu a odpowiadających minimom prefiksowym i maksimom sufiksowym tworzą ciągi malejące. Jeśli nie jest to dla Czytelnika oczywiste, polecamy ponowne przeczytanie definicji tych pojęć.

Zastanówmy się nad wyznaczaniem minimów prefiksowych. Otóż wystarczy liniowo przeglądać ciąg sum częściowych (a_i), trzymając minimum z aktualnie przejrzanych elementów (akt_min). Jeśli przeglądana wartość jest mniejsza niż akt_min, to aktualny indeks jest minimum prefiksowym i za jego pomocą uaktualniamy akt_min.

W przeciwnym razie dla danego elementu nie wykonujemy żadnych czynności. W algorytmie dla maksimów sufiksowych analogicznie przeglądamy ciąg (a_i) od końca, utrzymując aktualne maksimum akt_max.

Oto pseudokod znajdujący minima prefiksowe oraz maksima sufiksowe i umieszczający je odpowiednio w tablicach $pref$ ($pref[1]$ to pierwsze minimum prefiksowe w ciągu) oraz suf ($suf[1]$ to ostatnie maksimum sufiksowe w ciągu):

```
akt_min := ∞; il_pref := 0;
for i := 0 to n do
  if a[i] < akt_min then begin
    akt_min := a[i];
    il_pref := il_pref + 1;
    pref[il_pref] := i;
  end

akt_max := −∞; il_suf := 0;
for i := n downto 0 do
  if a[i] > akt_max then begin
    akt_max := a[i];
    il_suf := il_suf + 1;
    suf[il_suf] := i;
  end
```

Przykładowo, dla ciągu (4, 2, 3, 3, 1, 2, 3, 2, 2, 1) przedstawionego graficznie na rys. 2 tablice minimów prefiksowych i maksimów sufiksowych mają postać:

$$pref = [1,\ 2,\ 5], \quad suf = [10,\ 9,\ 7,\ 1]. \tag{2}$$

Rozwiązanie wzorcowe

Pokażemy teraz, jak w czasie liniowym dla każdego z minimów prefiksowych znaleźć najdalsze, nie mniejsze co do wartości maksimum sufiksowe.

Wykonujemy algorytm stosowy. Ustawiamy maksima sufiksowe w stos, mający na szczycie najwyższy indeks, i po kolei przeglądamy minima prefiksowe od największych indeksów. Dla danego minimum prefiksowego zdejmujemy ze stosu maksima sufiksowe, aż wartość w maksimum na szczycie będzie nie mniejsza niż wartość w przeglądanym minimum. Znalezione maksimum jest najdalszym, dla przeglądanego minimum, elementem spełniającym $a_l \leqslant a_r$ — wszystkie dalsze maksima zostały zdjęte, więc były mniejsze co do wartości niż przeglądane minimum. Jednocześnie przeglądamy minima w porządku rosnących wartości, więc zdjęte ze stosu maksima nigdy już nie będą przydatne.

Poniżej implementacja tego podejścia, w której stos jest ukryty w tablicy suf.

```
wsk_suf := 1;
wynik := 0;
for wsk_pref := il_pref downto 1 do begin
```

while $a[pref[wsk\_pref]] > a[suf[wsk\_suf]]$ **do**
  $wsk\_suf := wsk\_suf + 1;$
 $wynik := \max(wynik,\ suf[wsk\_suf] - pref[wsk\_pref]);$
end
return $wynik;$

Polecamy Czytelnikowi uzasadnienie poprawności tej implementacji. Pomocne jest w tym zauważenie, że wartość w największym minimum prefiksowym nie może przekroczyć wartości w największym maksimum sufiksowym.

Aby oszacować złożoność czasową powyższego algorytmu, należy zwrócić uwagę na łączną liczbę obrotów pętli **while**. Zauważmy mianowicie, że każdy obrót powoduje zwiększenie wartości zmiennej wsk_suf o jeden. Na początku zmienna ta ma wartość 1 i w żadnym momencie nie może przekroczyć il_suf, które jest nie większe niż n. Ponieważ wszystkich operacji poza pętlą **while** jest wykonywanych $O(n)$, więc pokazaliśmy, że złożoność czasowa powyższego algorytmu to właśnie $O(n)$.

Dla naszego przykładowego ciągu (rys. 2 i tablice minimów prefiksowych i maksimów sufiksowych (2)), powyższy algorytm wyznaczy następujące przypisania: dla minimum z pozycji 5 maksimum z pozycji 10, dla minimum z pozycji 2 maksimum z pozycji 9 (ta para daje największą różnicę indeksów), dla minimum z pozycji 1 maksimum także z pozycji 1.

Podsumowanie

Warto podsumować cały algorytm wzorcowy. Przebiega on w następujących krokach:

1. wczytaj tablicę wejściową $[x_1, x_2, \ldots, x_n]$
2. dla kolejnych zapytań k:
3.   oblicz tablicę $[a_0, a_1, \ldots, a_n]$
4.   oblicz minima prefiksowe dla tablicy $[a_0, a_1, \ldots, a_n]$
5.   oblicz maksima sufiksowe dla tablicy $[a_0, a_1, \ldots, a_n]$
6.   wykonaj algorytm stosowy
7.   wypisz największą odległość wśród par obliczonych w punkcie 6.

Złożoność czasowa algorytmu wynosi $O(nm)$, ponieważ „pętla” w punkcie 2 obraca się m razy, a obliczanie tablicy w punkcie 3 oraz wyznaczanie minimów prefiksowych i maksimów sufiksowych działają w czasie $O(n)$. Pokazaliśmy, że liniowo działa również algorytm stosowy. Złożoność pamięciowa algorytmu wynosi $O(n)$.

Implementacja rozwiązania wzorcowego znajduje się w plikach `klo.cpp` oraz `klo1.pas`.

Rozwiązania trochę wolniejsze

Podczas rozwiązywania zadania niektórzy zawodnicy stwierdzali, że rozwiązania nieznacznie gorsze, czyli o złożoności czasowej $O(mn \log n)$, są wystarczające do uzyskania maksymalnej punktacji, mimo że limity na dane wejściowe zdawały się sugerować wymaganie rozwiązania o złożoności liniowej. Takie rozwiązanie można otrzymać na co najmniej dwa sposoby.

Sortowanie

Tak jak w rozwiązaniu wzorcowym, w liniowej złożoności czasowej sprowadzamy wyjściowy problem do znalezienia najbardziej oddalonych indeksów l, r, dla których $a_r \geqslant a_l$. Następnie sortujemy pary (wartość, indeks): (a_i, i). Będziemy je przeglądać w kolejności od największych a_i, a w przypadku remisu od największych i.

W każdym momencie pamiętamy najbardziej wysunięty na prawo indeks r wśród już przetworzonych. W trakcie przeglądania kolejnych par znajdujemy największą odległość pomiędzy właśnie przeglądanym indeksem l a bieżącym indeksem r. Para l, r indeksów realizujących tę właśnie największą odległość jest właśnie szukaną parą — warunek $a_l \leqslant a_r$ jest zagwarantowany przez kolejność przeglądania par (wartość, indeks), zaś optymalność dzięki temu, że r jest indeksem wysuniętym najbardziej na prawo. Złożoność czasowa tego algorytmu wynosi $O(mn \log n)$.

Za takie rozwiązanie można było otrzymać około 70-80 punktów. Implementacja znajduje się w pliku `klos5.cpp`.

Wyszukiwanie binarne

Tak jak w rozwiązaniu wzorcowym obliczamy minima prefiksowe i maksima sufiksowe. Następnie, zamiast liniowego przeszukiwania dwoma wskaźnikami, dla każdego minimum prefiksowego wyszukujemy binarnie odpowiednie maksimum sufiksowe, korzystając z tego, że minima i maksima tworzą ciągi malejące. Łączna złożoność czasowa $m \cdot n$ takich wyszukiwań binarnych wynosi $O(mn \log n)$.

Za takie rozwiązanie można było otrzymać około 80-90 punktów. Implementacja znajduje się w pliku `klos6.cpp`.

O czym jeszcze należało pamiętać

Podczas rozwiązywania zadania należało pamiętać, aby używać zmiennych całkowitych 64-bitowych, chociażby do przechowywania wyrazów ciągu a. Algorytmy używające tylko zmiennych 32-bitowych uzyskiwały maksymalnie 70 punktów.

Testy

W zestawie były dwa typy testów. W testach pierwszego typu ciąg x składa się z grup wysokich słupków pooddzielanych fragmentami o mniejszej wysokości (testy 1, 2, 3, 4, 5, 9). Testy drugiego typu były generowane poprzez zadanie odpowiednich ciągów sum częściowych (a_i), tak aby charakteryzowały się żądanej postaci ciągami minimów prefiksowych i maksimów sufiksowych (testy 6, 7, 8, 10).

Nazwa	**n**	**m**	**Opis**
klo1.in	25	26	mały test poprawnościowy
klo2.in	70	25	mały test poprawnościowy
klo3.in	799	50	mały test poprawnościowy
klo4.in	1 543	49	mały test poprawnościowy
klo5.in	6 464	50	średni test poprawnościowo-wydajnościowy
klo6.in	50 003	50	średni test poprawnościowo-wydajnościowy
klo7.in	150 000	50	duży test poprawnościowo-wydajnościowy
klo8.in	299 999	50	duży test poprawnościowo-wydajnościowy
klo9.in	919 200	50	duży test poprawnościowo-wydajnościowy
klo10.in	1 000 000	50	duży test wydajnościowy