Marian M. Kędzierski
Treść zadania, Opracowanie

Zbigniew Wojna
Program

Dostępna pamięć: 128 MB.

OI, etap I, 17.10–14.11.2011

Litery

Mały Jaś ma bardzo długie nazwisko. Nie jest jednak jedyną taką osobą w swoim środowisku. Okazało się bowiem, że jedna z jego koleżanek z przedszkola, Małgosia, ma nazwisko dokładnie tej samej długości, chociaż inne. Co więcej, ich nazwiska zawierają dokładnie tyle samo liter każdego rodzaju — tyle samo liter `A`*, tyle samo liter* `B`*, itd.*

Jaś i Małgosia bardzo się polubili i często bawią się razem. Jedną z ich ulubionych zabaw jest zebranie dużej liczby małych karteczek, napisanie na nich kolejnych liter nazwiska Jasia, a następnie przesuwanie karteczek tak, aby powstało z nich nazwisko Małgosi.

Ponieważ Jaś uwielbia łamigłówki, zaczął zastanawiać się, ile co najmniej zamian sąsiednich liter trzeba wykonać, żeby przekształcić jego nazwisko w nazwisko Małgosi. Nie jest to łatwe zadanie dla kilkuletniego dziecka, dlatego Jaś poprosił Ciebie, głównego programistę w przedszkolu, o napisanie programu, który znajdzie odpowiedź na nurtujące go pytanie.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n *(*$2 \leqslant n \leqslant 1\,000\,000$*) oznaczająca liczbę liter w nazwisku Jasia. W drugim wierszu znajduje się* n *kolejnych liter nazwiska Jasia (bez odstępów). W trzecim wierszu znajduje się* n *kolejnych liter nazwiska Małgosi (również bez odstępów). Oba napisy składają się jedynie z wielkich liter alfabetu angielskiego.*

W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n \leqslant 1\,000$*.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą, oznaczającą minimalną liczbę zamian sąsiednich liter, które przekształcają nazwisko Jasia w nazwisko Małgosi.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`3`
`ABC`
`BCA`
	`2`

Rozwiązanie

Analiza problemu

W zadaniu mamy dane dwa słowa $u = u_1u_2 \ldots u_n$ oraz $w = w_1w_2 \ldots w_n$, w których każda litera występuje dokładnie tyle samo razy. Słowa o tej własności określa się czę-

sto mianem *anagramów*. Naszym zadaniem jest znalezienie minimalnej liczby zamian sąsiednich liter pozwalających przekształcić słowo u w słowo w. Takie pojedyncze operacje będziemy odtąd nazywali *przesunięciami*.

Powstaje naturalne pytanie, czy zawsze istnieje sekwencja przesunięć przekształcająca słowo u w słowo w. Na mocy założeń występujących w zadaniu wiemy, że po posortowaniu liter w słowach u oraz w otrzymujemy to samo słowo. A ponieważ przesunięcia pozwalają na posortowanie dowolnego słowa, przykładową sekwencję przesunięć możemy skonstruować w następujący sposób:

- najpierw sortujemy słowo u,
- potem wykonujemy przesunięcia sortujące słowo w, ale w odwrotnej kolejności.

Oczywiście, nie zawsze jest to strategia optymalna (przykładem mogą tu być chociażby słowa „`CBA`" oraz „`BCA`"), jednak jest ona zawsze poprawna.

Dolne ograniczenie

Skoro wiemy już, że przekształcenie słowa u w słowo w przy pomocy przesunięć jest zawsze możliwe, zastanówmy się nad dolnym ograniczeniem na liczbę przesunięć. W tym celu przyda nam się następujące pojęcie:

Definicja 1. Niech u będzie dowolnym anagramem słowa w. *Odległością* słowa u od słowa w będziemy nazywać następującą wartość:

$$d(u,w) = \sum_{a \in A} \sum_{i=1}^{k(a)} |u_{a,i} - w_{a,i}|$$

gdzie $A = \{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}\}$ oznacza alfabet, $k(a)$ oznacza liczbę wystąpień litery a w słowie w (krotność), zaś $u_{a,1}, \ldots, u_{a,k(a)}$ oraz $w_{a,1}, \ldots, w_{a,k(a)}$ to pozycje kolejnych wystąpień tej litery odpowiednio w słowach u oraz w.

Skąd pomysł na taką funkcję? Otóż intuicyjnie ma ona oddawać sumaryczną odległość pomiędzy „odpowiadającymi sobie"[1] literami w słowach u oraz w. Dla pokazania dolnego ograniczenia na wynik nie będzie nam jednak potrzebna wnikliwa analiza funkcji odległości, a jedynie jej następująca prosta własność.

Fakt 1. *Niech u będzie dowolnym anagramem słowa w oraz niech v będzie słowem powstałym z u przez wykonanie dokładnie jednego przesunięcia. Wtedy* $d(u,w) - d(v,w) \in \{-2, 0, 2\}$.

Dowód: Każde przesunięcie zmienia o jeden pozycje dwóch liter, przy czym każdą z nich może albo oddalić od „odpowiadającej jej" litery docelowej, albo ją do niej przybliżyć. Stąd odległość może się zmniejszyć o dwa, zwiększyć o dwa lub pozostać taka sama. ■

[1] Na tym etapie nie jest jeszcze jasne, czy pierwsze wystąpienie litery `A` w słowie u w optymalnej strategii przechodzi na pierwsze wystąpienie litery `A` w słowie w, drugie na drugie, itd. i podobnie dla pozostałych liter.

Prostym wnioskiem z powyższego faktu jest szukane dolne ograniczenie:

Twierdzenie 1. *Minimalna liczba przesunięć przekształcających słowo u w słowo w jest nie mniejsza niż $\frac{1}{2}d(u,w)$.*

Dowód: Wykazaliśmy, że każde przesunięcie zmniejsza odległość słów co najwyżej o dwa. Skoro początkowa odległość słowa u od słowa w wynosi $d(u,w)$, a końcowa — $d(w,w)=0$, to konieczne jest wykonanie co najmniej $\frac{1}{2}d(u,w)$ przesunięć. ■

Rozwiązanie zwracające zawsze wynik $\frac{1}{2}d(u,w)$ okazuje się niepoprawne już dla pary słów „`ABC`" i „`CBA`", ponieważ daje w wyniku 2, podczas gdy poprawnym wynikiem jest 3. Takie rozwiązania na zawodach nie uzyskiwały żadnych punktów (przykładowa implementacja w pliku `litb1.cpp`).

Rozwiązanie poprawne

Tak naprawdę nie udało nam się jeszcze wskazać żadnej konkretnej strategii wykonywania przesunięć: nie wynikła ona ani z analizy dolnego ograniczenia, ani z metody opartej o sortowanie liter w słowach, w której nie widać od razu, jaka jest minimalna liczba przesunięć potrzebnych do posortowania słowa. Czas więc od rozważań czysto teoretycznych przejść do znalezienia pierwszej konkretnej strategii. Narzucającym się pomysłem jest następujące podejście zachłanne.

Niech $a = w_1$ i niech $u_{a,1}$ będzie pierwszym wystąpieniem litery a w słowie u. Wykonajmy $u_{a,1} - 1$ przesunięć w słowie u przemieszczających pierwsze wystąpienie litery a na początek słowa. Następnie odetnijmy pierwsze litery tak zmodyfikowanego słowa u oraz słowa w i całą operację powtórzmy $n-1$ razy.

Czy taka strategia jest optymalna? Nie zawsze algorytmy zachłanne prowadzą do poprawnych albo optymalnych rozwiązań, jednak w tym przypadku odpowiedź jest twierdząca. Musimy to jednak uzasadnić.

Po pierwsze: dlaczego to właśnie litera z pozycji $u_{a,1}$, a nie jakieś inne wystąpienie tej litery w słowie u, ma przejść na pierwszą literę słowa w? Załóżmy przeciwnie, że litera odpowiadająca w_1 w pewnej hipotetycznej strategii optymalnej znajduje się w u na pozycji p, $p > u_{a,1}$. Wtedy w czasie przekształcania słowa u w słowo w musiałby nastąpić moment, w którym litera pochodząca z pozycji $u_{a,1}$ poprzedza bezpośrednio literę z pozycji p, a kolejne przesunięcie zamienia te litery miejscami. Jednak takie przesunięcie nie zmienia w ogóle układu liter w słowie. Rezygnacja z tego jednego przesunięcia da nam zatem strategię lepszą od optymalnej — sprzeczność. Tak więc pierwsza litera słowa w w każdej optymalnej strategii rzeczywiście odpowiada literze z pozycji $u_{a,1}$ w słowie u.

Nazwijmy teraz literę a pochodzącą z pozycji $u_{a,1}$ literą *wyróżnioną*. Przyjrzyjmy się strategii optymalnej pod kątem dwóch niezależnie zmieniających się stanów:

- położenia wyróżnionej litery,
- postaci *reszty* słowa u, tzn. kolejności wszystkich pozostałych liter.

Dlaczego te stany są niezależne? Zauważmy, że mamy dwa rodzaje przesunięć:

1. zmieniające położenie wyróżnionej litery (wtedy kolejność liter w reszcie słowa nie zmienia się),

2. zmieniające resztę słowa (wtedy pozycja wyróżnionej litery nie zmienia się).

Niezależność tych dwóch stanów pozwala nam zmienić kolejność wykonywania przesunięć tak, aby najpierw przeprowadzić wszystkie przesunięcia pierwszego typu, a następnie drugiego. W ten sposób otrzymamy strategię o tej samej liczbie przesunięć, a więc również optymalną. Zarazem pierwsze kroki, przemieszczające literę a z pozycji $u_{a,1}$ na początek, będą identyczne jak w naszym algorytmie zachłannym.

Powtarzając to rozumowanie dla kolejnych liter, pokażemy, że litery uznane wyżej za „odpowiadające sobie" rzeczywiście odpowiadają sobie w każdej strategii optymalnej, a zaproponowany algorytm zachłanny stanowi strategię optymalną.

Bezpośrednia implementacja powyższego algorytmu (patrz pliki `lits1.cpp` i `lits2.pas`) ma złożoność czasową $O(n^2)$. Można ją jednak usprawnić...

Rozwiązanie wzorcowe

Przyspieszenie algorytmu zachłannego wymaga od nas znalezienia struktury danych, która pozwoli efektywnie wyznaczać pierwszą niewykorzystaną pozycję danej litery w zmieniającym się słowie u, tzn. takie wystąpienie tej litery, które nie zostało jeszcze przemieszczone na początek, a następnie odcięte.

Gdyby słowo u nie ulegało zmianom, łatwo byłoby znajdować pierwsze niewykorzystane pozycje poszczególnych liter. Wystarczyłaby do tego tablica stosów *stosy*[] indeksowana literami alfabetu, która pod indeksem $a \in A$ przechowywałaby pozycje kolejnych wystąpień litery a w słowie u, od lewej do prawej. Taką tablicę stosów łatwo zbudować. Wystarczy przejść w pętli po słowie u od końca i każdą napotkaną literę wrzucać na szczyt odpowiadającego jej stosu. Późniejsze korzystanie z tej tablicy również byłoby proste — w każdym kroku algorytmu szukana pozycja litery byłaby na szczycie stosu odpowiadającego tej literze.

Niestety słowo u zmienia się w trakcie działania algorytmu, potrzebujemy więc jakiegoś mechanizmu, który aktualizowałby pozycje zawarte w tablicy *stosy*[] zgodnie ze zmianami kolejności liter. W każdym momencie możemy skupić się jedynie na niewykorzystanych dotąd literach, bo pozycji liter już przesuniętych na początek nie będziemy później badać. A jak zmienia się pozycja niewykorzystanej litery? Otóż w każdym kroku algorytmu będzie ona mniejsza od początkowej o liczbę tych liter, które na początku znajdowały się wcześniej niż rozważana i zostały już wykorzystane.

Musimy zatem zaproponować mechanizm pozwalający szybko znajdować liczbę wykorzystanych liter znajdujących się początkowo w słowie u na pozycji wcześniejszej niż zadana. Do tego służyć może struktura danych zwana *drzewem licznikowym* lub *przedziałowym*[2]. Jest to statyczne drzewo binarne zbudowane nad tablicą liczb $t[1..n]$; pozwala ono na wykonywanie w czasie $O(\log n)$ następujących operacji:

ustaw(i, x) — przypisz $t[i] := x$;

[2]Drzewa przedziałowe pojawiały się już wielokrotnie w rozwiązaniach zadań z Olimpiady Informatycznej, nie będziemy więc ich tutaj szczegółowo opisywać. Można o nich przeczytać np. w opracowaniu zadania Tetris 3D z XIII Olimpiady Informatycznej [13].

$suma(l, r)$ — zwróć sumę $t[l] + t[l + 1] + \ldots + t[r]$, przy czym $l \leqslant r$.

Mając do dyspozycji drzewo licznikowe, możemy już bardzo łatwo utrzymywać informacje o przesunięciach liter w słowie u. Wystarczy na początku, przy budowaniu drzewa, wyzerować tablicę $t[]$, a potem, przesuwając literę z jej początkowej pozycji i na początek słowa, wykonywać operację $ustaw(i, 1)$. Wtedy w dowolnym momencie działania algorytmu wiemy, że pozycja niewykorzystanej litery, która początkowo znajdowała się na pozycji i, od początku algorytmu zmalała o $suma(1, i-1)$. Algorytm wygląda więc następująco:

```
1: wynik := 0;
2: for k := 1 to n do begin
3:    a := m[k];
4:    i := stosy[a].pop();
5:    ustaw(i, 1);
6:    i := i − suma(1, i − 1);
7:    wynik := wynik + i − 1;
8: end
9: return wynik;
```

W ten sposób otrzymujemy rozwiązanie wzorcowe, zaimplementowane w plikach `lit.cpp` i `lit1.pas`.

Nieco inne spojrzenie na zadanie

Pokazaliśmy wcześniej, że dolnym ograniczeniem na wynik jest odległość słów, ale w ogólności konieczna liczba przesunięć może być większa. Istnieje jednak inna miara odległości pomiędzy słowami, którą także można łatwo wyznaczyć, bez symulowania żadnej konkretnej strategii, a która jest już równa szukanemu wynikowi. Aby ją poznać, przyjrzymy się permutacji[3] liter, jaką należy wykonać, by przekształcić słowo u w słowo w. Permutację tę nazwijmy *permutacją przeprowadzającą u na w*.

Wiemy już, że i-te wystąpienie litery `A` w słowie u odpowiada i-temu wystąpieniu litery `A` w słowie w, i-te wystąpienie litery `B` w słowie u odpowiada i-temu wystąpieniu litery `B` w słowie w itd. To pozwala skonstruować permutację przeprowadzającą u na w: otóż jeśli literą odpowiadającą literze u_i w słowie w jest w_k, to mamy $p_i = k$, co oznacza tyle co: „chcemy przestawić literę z pozycji i na pozycję k". Przykładowo, permutacją przeprowadzającą słowo $u =$ `ABAAB` na słowo $w =$ `BABAA` jest $p = (2, 1, 4, 5, 3)$.

Nie jest trudno napisać algorytm konstruujący permutację przeprowadzającą u na w przy użyciu tablicy stosów, wspomnianej w poprzednim rozdziale. Kiedy już będziemy znali tę permutację, możemy obliczyć w niej liczbę inwersji.

Definicja 2. Niech p będzie permutacją n liczb. *Inwersją* w permutacji p nazywamy dowolną parę indeksów $1 \leqslant a < b \leqslant n$, taką że $p_a > p_b$.

[3] *Permutacją* nazywamy operację zmieniającą kolejność liter zadanego słowa. Dowolną permutację n-literowego słowa możemy przedstawić w postaci ciągu $(p_i)_{i=1}^{n}$ liczb całkowitych z zakresu od 1 do n, w którym każda z tych liczb występuje dokładnie raz. Zapis ten oznacza, że permutacja p dla każdego i przemieszcza literę z pozycji i na pozycję p_i.

Innymi słowy, inwersję stanowi każda para elementów permutacji, która jest ustawiona w niewłaściwej kolejności w stosunku do permutacji identycznościowej (posortowanej rosnąco). I tak, na przykład, permutacja $p = (1, 2, \ldots, n)$ nie ma żadnych inwersji, a permutacja $p = (n, (n-1), \ldots, 1)$ ma ich $\frac{n(n-1)}{2}$.

Możemy już teraz poczynić kluczowe spostrzeżenie: otóż szukany przez nas wynik jest równy właśnie liczbie inwersji w permutacji przeprowadzającej u na w! Dlaczego tak jest? Każde przesunięcie w słowie u powoduje zarazem odpowiadające mu przesunięcie (czyli podobnie jak w przypadku słów — zamianę dwóch sąsiednich liczb) w permutacji p, i odwrotnie. Słowo u zostaje ostatecznie przekształcone w słowo w w momencie, w którym permutacja przeprowadzająca u na w jest posortowana. Ponieważ każde przesunięcie zmienia liczbę inwersji dokładnie o jeden (bo zmienia względne położenie dokładnie jednej pary sąsiednich elementów), więc przy przekształcaniu słowa u w słowo w trzeba wykonać co najmniej tyle przesunięć, ile wynosi liczba inwersji w p. Z drugiej strony, każdą permutację można posortować w następujący sposób: dopóki istnieje jakaś para sąsiednich elementów tworząca inwersję, wykonaj na nich przesunięcie. W momencie, w którym nie będzie już żadnej takiej pary, permutacja będzie posortowana[4]. Tak więc liczba inwersji jest nie tylko dolnym ograniczeniem na minimalną liczbę przesunięć przekształcających słowo u w słowo w, lecz także i górnym, czyli jest równa szukanemu wynikowi.

Jak zliczać inwersje w permutacji?

Znanych jest kilka efektywnych algorytmów pozwalających zliczać inwersje w permutacji. Najpopularniejszym rozwiązaniem w programach zawodników była pewna modyfikacja algorytmu sortowania przez scalanie (Mergesort).

Algorytm Mergesort bazuje na operacji *scalenia* (ang. *merge*) dwóch posortowanych ciągów, która łączy je w ciąg posortowany w czasie liniowym. Z jej użyciem można skonstruować następujący algorytm sortowania ($|ciąg|$ oznacza długość ciągu):

```
1: function mergesort(ciąg)
2: begin
3:   if |ciąg| = 1 then return ciąg;
4:   (ciąg1, ciąg2) := podziel_na_połowy(ciąg);
5:   return merge(mergesort(ciąg1), mergesort(ciąg2));
6: end
```

Funkcja *podziel_na_połowy* w powyższym pseudokodzie „łamie ciąg na pół", tzn. długości zwracanych przez nią ciągów różnią się co najwyżej o 1. Przy założeniu, że operacja $merge(ciąg1, ciąg2)$ działa w czasie $O(|ciąg1| + |ciąg2|)$, cały algorytm Mergesort ma złożoność czasową $O(n \log n)$, gdzie n to długość sortowanego ciągu.

Okazuje się, że można tak zmodyfikować operację scalania, żeby oprócz łączenia dwóch posortowanych ciągów w nowy posortowany ciąg, obliczała liczbę takich inwersji (a, b), że a należy do pierwszego scalanego ciągu, zaś b do drugiego. Dla każdej pary wyrazów (a, b) wyjściowej permutacji p będzie dokładnie jedno wywołanie funkcji $merge(ciąg1, ciąg2)$ dla argumentów takich, że a należy do ciągu *ciąg1*, zaś b do ciągu *ciąg2* — nastąpi to w tym wywołaniu funkcji *mergesort*, w którym a i b zostaną

[4]Mowa tu o algorytmie sortowania bąbelkowego (Bubblesort).

rozdzielone do dwóch różnych połówek sortowanego ciągu. Jeżeli więc uda nam się wzbogacić funkcję *merge* w żądany sposób, to w efekcie wyznaczy ona liczbę inwersji w oryginalnym ciągu. Oto pseudokod takiej wzbogaconej operacji scalania:

```
1:  function merge(ciąg1, ciąg2)
2:  begin
3:    k1 := |ciąg1|;   k2 := |ciąg2|;
4:    i1 := 1;   i2 := 1;
5:    while i1 ⩽ k1 or i2 ⩽ k2 do
6:      if i1 > k1 then begin
7:        ciąg[i1 + i2 − 1] := ciąg2[i2];
8:        i2 := i2 + 1;
9:      end else if i2 > k2 then begin
10:       inwersje := inwersje + i2 − 1;   { tu zliczamy inwersje! }
11:       ciąg[i1 + i2 − 1] := ciąg1[i1];
12:       i1 := i1 + 1;
13:     end else if ciąg1[i1] < ciąg2[i2] then begin
14:       inwersje := inwersje + i2 − 1;   { tu zliczamy inwersje! }
15:       ciąg[i1 + i2 − 1] := ciąg1[i1];
16:       i1 := i1 + 1;
17:     end else begin
18:       ciąg[i1 + i2 − 1] := ciąg2[i2];
19:       i2 := i2 + 1;
20:     end
21:   return ciąg;
22: end
```

Dowód poprawności powyższej funkcji pozostawiamy Czytelnikowi jako ćwiczenie. W ten sposób otrzymujemy rozwiązanie alternatywne o koszcie czasowym $O(n \log n)$.

Istnieje też inny, podobny algorytm zliczający inwersje w permutacji, który zasugerował nam Marcin Pilipczuk. Tym razem funkcja dzieląca ciąg dzieli go na pół nie według pozycji, ale według wartości, tzn. jeśli elementy dzielonego ciągu należą do przedziału $[x, y]$ (początkowo: $[1, n]$), to pierwszy ciąg powstały w wyniku podziału składa się z liczb z przedziału $[x, z]$, zaś drugi — z przedziału $[z + 1, y]$, gdzie z jest środkiem przedziału $[x, y]$. Poniżej umieszczamy funkcję rekurencyjną implementującą to podejście. Złożoność czasowa tej metody to także $O(n \log n)$.

```
1:  function inwersje(ciąg, [x, y])
2:  begin
3:    if x = y then return 0;
4:    i1 := 0;   i2 := 0;
5:    wynik := 0;   z := (x + y) div 2;
6:    for i := 1 to |ciąg| do
7:      if ciąg[i] ⩽ z then begin
8:        wynik := wynik + i2;
9:        i1 := i1 + 1;
10:       ciąg1[i1] := ciąg[i];
11:     end else begin
```

```
12:         i2 := i2 + 1;
13:         ciąg2[i2] := ciąg[i];
14:       end
15:     return wynik + inwersje(ciąg1, [x, z]) + inwersje(ciąg2, [z + 1, y]);
16: end
```

Jeszcze inne efektywne rozwiązanie problemu zliczania inwersji w permutacji można znaleźć w opisie rozwiązania zadania *Kodowanie permutacji* w książce [38].

Implementacje rozwiązań opartych na poszczególnych metodach zliczania inwersji można znaleźć w plikach `lit2.cpp` – `lit7.pas`.

Warto na koniec zauważyć, że podany we wcześniejszej sekcji algorytm wzorcowy daje zarazem kolejne rozwiązanie problemu zliczania inwersji w permutacji — również w czasie liniowo-logarytmicznym, ale za pomocą drzew przedziałowych.

Testy

Rozwiązania zawodników były sprawdzane z użyciem 10 zestawów testowych. Zestawy 6–10 wymagały reprezentowania wyniku za pomocą 64-bitowej zmiennej całkowitej.

Nazwa	**n**	*inwersje*	**Opis**
lit1.in	4	4	mały test stworzony ręcznie
lit2.in	7	16	mały test stworzony ręcznie
lit3a.in	1 000	44 599	test losowy
lit3b.in	1 000	480 766	test wymagający dużej liczby przesunięć
lit4a.in	10 000	1 499 627	test losowy
lit4b.in	10 000	48 076 920	test wymagający dużej liczby przesunięć
lit5a.in	50 000	17 536 625	test losowy
lit5b.in	50 000	1 201 923 076	test wymagający dużej liczby przesunięć
lit6a.in	100 000	47 167 845	test losowy
lit6b.in	100 000	4 807 692 306	test wymagający dużej liczby przesunięć
lit7a.in	200 000	130 542 635	test losowy
lit7b.in	200 000	19 230 769 228	test wymagający dużej liczby przesunięć
lit8a.in	500 000	580 934 901	test losowy
lit8b.in	500 000	120 192 307 690	test wymagający dużej liczby przesunięć
lit9a.in	1 000 000	1 482 550 867	test losowy
lit9b.in	1 000 000	480 769 230 766	test wymagający dużej liczby przesunięć
lit10a.in	1 000 000	1 819 136 406	test losowy
lit10b.in	1 000 000	480 769 230 766	test wymagający dużej liczby przesunięć