Jacek Tomasiewicz — Treść zadania
Bartosz Tarnawski — Opracowanie
Michał Adamczyk — Program

Dostępna pamięć: 128 MB. *OI, etap I, 7.10–4.11.2013*

Bar sałatkowy

Bajtotka wybrała się do baru sałatkowego. W barze na ladzie leży n owoców ułożonych w jednym rzędzie. Są to pomarańcze i jabłka. Bajtotka może wybrać pewien spójny fragment rzędu owoców, z którego zostanie przygotowana sałatka owocowa.

Wiadomo, że owoce z wybranego fragmentu będą dodawane do sałatki kolejno od lewej do prawej albo kolejno od prawej do lewej. Bajtotka uwielbia pomarańcze i ma dodatkowe wymaganie, aby w trakcie robienia sałatki liczba dodanych już pomarańczy nigdy nie była mniejsza od liczby dodanych jabłek, niezależnie od tego, czy owoce będą dodawane od lewej do prawej, czy odwrotnie. Pomóż Bajtotce i napisz program, który znajdzie jak najdłuższy fragment rzędu owoców spełniający jej wymagania.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczającą liczbę owoców. Kolejny wiersz zawiera napis złożony z n liter $a_1 a_2 \ldots a_n$ ($a_i \in \{\mathtt{p}, \mathtt{j}\}$). Jeśli $a_i = \mathtt{p}$, to i-tym owocem w rzędzie jest pomarańcza, w przeciwnym przypadku jest to jabłko.

Możesz założyć, że w testach wartych 50% punktów zachodzi $n \leqslant 10\,000$, a w testach wartych 20% punktów zachodzi $n \leqslant 1000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą równą liczbie owoców w najdłuższym spójnym fragmencie rzędu, który spełnia wymagania Bajtotki. Jeśli sałatka dla Bajtotki nie może zostać przyrządzona, prawidłowym wynikiem jest 0.

Przykład

Dla danych wejściowych:

```
6
jpjppj
```

poprawnym wynikiem jest:

```
4
```

Wyjaśnienie do przykładu: *Po odrzuceniu skrajnie lewego i skrajnie prawego jabłka Bajtotka może zamówić sałatkę z pozostałych owoców.*

Rozwiązanie

Analiza problemu

Będziemy rozważać słowa składające się wyłącznie z liter `p` oraz `j`. Długość słowa w oznaczamy przez $|w|$.

Słowo nazwiemy *legalnym*, jeśli każdy jego prefiks i każdy jego sufiks zawiera co najmniej tyle liter `p` co liter `j`. Mając dane słowo $w = a_1 a_2 \ldots a_n$, chcemy znaleźć jego najdłuższe legalne podsłowo.

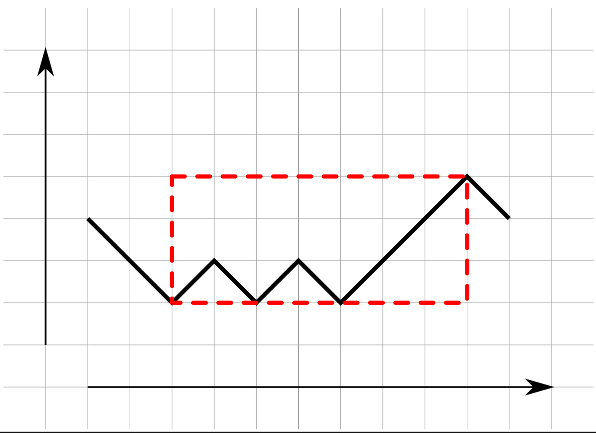
Trochę wygodniej będzie nam myśleć w kategoriach sum prefiksowych. Literze `p` przypiszmy wartość 1, a literze `j` wartość -1. Niech $pre[i]$ $(0 \leqslant i \leqslant n)$ będzie sumą wartości i pierwszych liter słowa w (w szczególności $pre[0] = 0$).

Możemy teraz sformułować równoważny warunek na legalność słowa:

Lemat 1. Podsłowo $a_i a_{i+1} \ldots a_j$ słowa w jest legalne wtedy i tylko wtedy, gdy dla każdego $i-1 \leqslant k \leqslant j$ zachodzi

$$pre[i-1] \leqslant pre[k] \leqslant pre[j].$$

Pomyślmy teraz o tablicy *pre* jako o funkcji i przyjrzyjmy się jej wykresowi. Ta funkcja jest zdefiniowana wprawdzie tylko dla $0, 1, \ldots, n$, ale żeby nasza wizualizacja była ładniejsza, możemy uzupełnić ją funkcjami liniowymi na każdym przedziale $[i-1, i]$, dla $1 \leqslant i \leqslant n$. Załóżmy, że $a_i a_{i+1} \ldots a_j$ jest legalnym podsłowem. Wtedy wykres *pre* na przedziale $[i-1, j]$ jest w całości zawarty w prostokącie o lewym dolnym rogu $(i-1, pre[i-1])$ oraz prawym górnym $(j, pre[j])$.

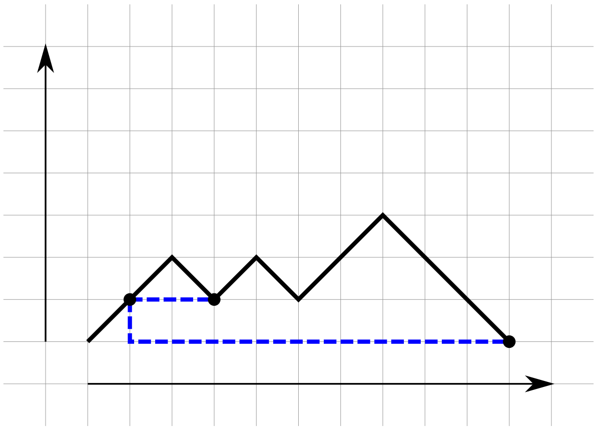


Rys. 1: Legalne podsłowo `pjpjppp` słowa `jjpjpjpppj` i odpowiadający mu prostokąt.

Rozwiązanie wzorcowe $O(n)$

Dla każdego $0 \leqslant i \leqslant n$ chcemy obliczyć i stablicować największe takie $j = end[i]$, że podsłowo $a_{i+1}a_{i+2}\ldots a_j$ jest legalne. Jeśli $a_{i+1} = \mathtt{j}$, to $end[i] = i$, bo żadne słowo zaczynające się od $\mathtt{j}$ nie może być legalne. Przyjmijmy, że $end[n] = n$.

Tablica *end* od razu da nam wynik, ale do jej obliczenia będziemy potrzebowali kilku innych tablic. Dla $0 \leqslant i \leqslant n$ niech $equal[i]$ będzie najmniejszym takim $j > i$, że $pre[i] = pre[j]$, a jeśli dla danego i takie j nie istnieje, to powiedzmy, że $equal[i] = n+1$. Podobnie definiujemy tablicę *below*, tym razem szukając najmniejszego $j > i$, aby $pre[j] < pre[i]$. Przykładowe wartości obu tablic dla pewnego i zostały przedstawione na rysunku.



Rys. 2: Zaznaczono punkty wykresu dla argumentów i, $equal[i]$ oraz $below[i]$.

Tablicę *equal* wyznaczamy w czasie liniowym, przeglądając kolejne pozycje od prawej do lewej i zapamiętując numer ostatniej pozycji o danej wartości *pre*. Mając tablicę *equal*, można łatwo wyznaczyć tablicę *below*; zresztą w tym rozwiązaniu wykorzystamy ją tylko na potrzeby dowodu.

Chcemy zdefiniować teraz pewien porządek na zbiorze $0, 1, \ldots, n$. Powiemy, że $i \prec j$, jeżeli $(pre[i], i) < (pre[j], j)$ w porządku leksykograficznym, tj. $pre[i] < pre[j]$ lub ($pre[i] = pre[j]$ oraz $i < j$). Okazuje się, że porządek ten ma bezpośredni związek z wartościami tablicy *end*:

Lemat 2. $end[i]$ jest równy tej liczbie spośród $i, i+1, \ldots, below[i]-1$, która jest największa w porządku „$\prec$”.

Jego wyprowadzenie z lematu 1 pozostawiamy jako ćwiczenie.

Możemy teraz skonstruować tablicę *end*. Ponownie przeglądamy pozycje od prawej do lewej. Ustalmy $0 \leqslant i \leqslant n-1$. Jeżeli $a_{i+1} = \mathtt{j}$, to ustawiamy po prostu $end[i] = i$. Odtąd zakładamy, że $a_{i+1} = \mathtt{p}$. Rozważmy kilka przypadków.

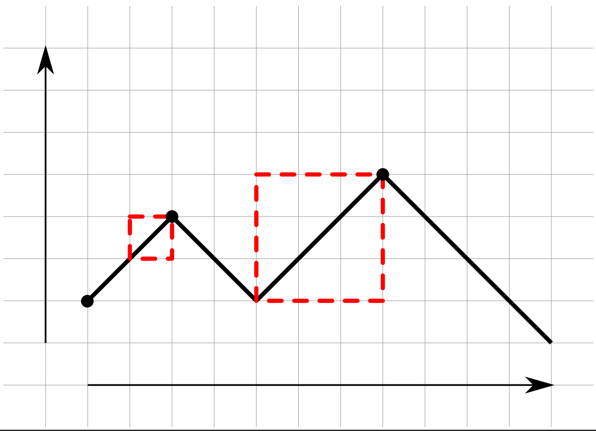
1. Jeśli $i = n-1$, to ustawiamy $end[i] = i+1$.

2. Jeśli zaś $i < n-1$ oraz $equal[i] = n+1$, to powinniśmy ustawić $end[i] = end[i+1]$ (ćwiczenie).

3. Pozostaje przypadek $i < n-1$ oraz $equal[i] \neq n+1$ (nadal zakładamy, że $a_{i+1} = \mathtt{p}$).

   Niech $a = end[i+1]$, $b = end[equal[i]]$. Jeśli $a \prec b$, to powinniśmy ustawić $end[i] = b$, a w przeciwnym przypadku $end[i] = a$.

Udowodnimy teraz poprawność obliczenia wyniku w punkcie 3. Wcześniej warto spojrzeć na rysunek przedstawiający przykładową konfigurację, w której $end[i+1] \prec end[equal[i]]$.



Rys. 3: Zaznaczono punkty wykresu dla argumentów i, $end[i+1]$ oraz $end[equal[i]]$.

Dzięki lematowi 2 wiemy, że szukamy liczby spośród $i, i+1, \ldots, below[i]-1$, która jest największa według porządku „$\prec$" – nazwijmy ją j. Nie jest to na pewno i, bo $pre[i+1] > pre[i]$ (założyliśmy, że $a_{i+1} = \mathtt{p}$).

Może być zatem tak, że

- $i+1 \leqslant j \leqslant equal[i]-1$, wtedy $j = end[i+1]$.

  Uzasadnienie: Zauważmy, że $equal[i] = below[i+1]$. Skoro j jest największe w porządku „$\prec$" na przedziale $i, \ldots, below[i]-1$, to tym bardziej jest największe na przedziale $i+1, \ldots, below[i+1]-1$. Czyli rzeczywiście $j = end[i+1]$.

- $equal[i] \leqslant j \leqslant below[i]-1$, wtedy $j = end[equal[i]]$.

  Uzasadnienie: Tym razem wystarczy zauważyć, że $below[i] = below[equal[i]]$. Ponownie, skoro j jest największe według „$\prec$" na przedziale $i, \ldots, below[i]-1$, to jest największe także na $equal[i], \ldots, below[equal[i]]-1$.

To już koniec opisu rozwiązania wzorcowego. Zostało ono zaimplementowane w plikach `bar.cpp` i `bar1.pas`.

Rozwiązanie alternatywne $O(n \log n)$

Tablica *below* w poprzednim rozwiązaniu przydawała się tylko w dowodzie. Tym razem wykorzystamy ją także w samym programie. Oprócz niej, będziemy potrzebowali drzewa przedziałowego podającego maksimum (według porządku „$\prec$”) na zadanym przedziale.

Dla każdego i znajdujemy $end[i]$, odpytując nasze drzewo o największą liczbę w porządku „$\prec$” na przedziale $[i, below[i]-1]$.

Takie rozwiązanie zostało zaimplementowane w pliku `bar3.pas`. Rozwiązanie `bar2.cpp` ma tę samą złożoność, ale opiera się na trochę innym pomyśle i korzysta ze struktury danych `set` z STL-a. Oba rozwiązania otrzymywały 100 punktów.

Rozwiązanie wolniejsze $O(n\sqrt{n})$

To eleganckie rozwiązanie zostało zaproponowane przez zawodników. Dosyć łatwo udowodnić jego poprawność, trochę trudniej jest z określeniem złożoności czasowej. Będziemy wielokrotnie przeglądać nasze słowo z lewa na prawo i z prawa na lewo, dzieląc je na coraz krótsze fragmenty.

Dla uproszczenia zapisu podsłowo w zaczynające się od i-tej litery, a kończące na j-tej oznaczymy przez $w[i..j]$. Niech $v = w[i..j]$. Powiemy, że v jest *lewostronnie legalne*, jeżeli dla każdego $i \leqslant k \leqslant j$ zachodzi $pre[i-1] \leqslant pre[k]$. Jeżeli v nie jest lewostronnie legalne, to najmniejsze takie $k \geqslant i$, że $pre[k] < pre[i-1]$, nazwiemy *lewostronnym uskokiem* v.

Analogicznie definiujemy słowo *prawostronnie legalne* (dla każdego $i \leqslant k \leqslant j$ $pre[k-1] \leqslant pre[j]$) oraz *prawostronny uskok* słowa, które takie nie jest (największe takie $k \leqslant j$, że $pre[k-1] > pre[j]$). Zgodnie z naszą poprzednią definicją, słowo lewostronnie i prawostronnie legalne jest po prostu legalne.

Skorzystamy z prostej obserwacji:

Lemat 3. Jeżeli k jest lewostronnym lub prawostronnym uskokiem słowa $v = w[i..j]$, zaś $u = w[i'..j']$ jest legalnym podsłowem v, to $j' < k$ lub $i' > k$.

Możemy teraz przejść do opisu algorytmu. Chcemy napisać funkcję $longest(i, j)$, która wyznaczy długość najdłuższego legalnego podsłowa słowa $w[i+1..j]$. Wtedy ostatecznym wynikiem będzie $longest(0, n)$.

Niech $v = w[i+1..j]$. Aby obliczyć $longest(i, j)$, wykonamy co następuje:

1. Jeśli v nie jest lewostronnie legalne, to znajdujemy najdłuższy taki podciąg $p_0, p_1, \ldots, p_r$, że $i-1 = p_0 < p_1 < \ldots < p_r \leqslant j$ oraz p_{x+1} jest lewostronnym uskokiem słowa $w[p_x+1..j]$ dla $x = 0, \ldots, r-1$. Na mocy lematu 3:

   $$longest(i, j) = \max(longest(i, p_1-1), longest(p_1, p_2-1), \ldots, longest(p_r, j)).$$

   Warto przy tym zauważyć, że każde ze słów $w[(p_x+1)..(p_{x+1}-1)]$ dla $x = 0, \ldots, r$ jest lewostronnie legalne (przyjmujemy $p_{r+1} = j+1$).

2. Jeśli v jest lewostronnie, ale nie prawostronnie legalne, to robimy to samo co powyżej, tylko „w drugą stronę”, wywołując się rekurencyjnie dla ciągu prawostronnie legalnych podsłów.

3. Jeśli v jest zarówno lewostronnie, jak i prawostronnie legalne, to wynik to $j - i$.

Poprawność algorytmu wynika bezpośrednio z lematu 3. Zastanówmy się, jaki jest jego czas działania. Widać, że na każdym poziomie rekurencji tracimy $O(n)$ czasu. Trudniej jest uzasadnić, że maksymalna liczba poziomów to $O(\sqrt{n})$. Przytoczony poniżej dowód jest dosyć złożony. Czytelnik, który nie jest szczególnie zainteresowany szacowaniem złożoności czasowej naszego algorytmu, może go śmiało pominąć.

Ustalmy pewien ciąg podsłów

$$w_0 = w[l_0 + 1, r_0], w_1 = w[l_1 + 1, r_1], \ldots, w_k = w[l_k + 1, r_k]$$

o następujących własnościach:

- $w_0 = w$,
- w_{x+1} jest podsłowem w_x i, co więcej, chcąc obliczyć wartość $longest(l_x, r_x)$, musieliśmy wywołać rekurencyjnie $longest(l_{x+1}, r_{x+1})$,
- żadne ze słów w_i nie jest legalne.

Możemy założyć, że słowo w nie było lewostronnie legalne (dowód dla przeciwnego przypadku różni się tylko szczegółami). Wtedy wykonaliśmy dla niego operację typu (1), czyli słowo w_1 było już lewostronnie legalne. Dla niego musieliśmy wykonać operację typu (2) i tak dalej.

Kontynuując to rozumowanie, dochodzimy do wniosku, że słowa w_{2l-1} są lewostronnie legalne, a w_{2l} – prawostronnie legalne dla $l \geqslant 1$.

Załóżmy też bez straty ogólności, że $k = 2m$. Skoncentrujmy uwagę na ciągu $l_1, l_2, \ldots, l_k$. Chcielibyśmy udowodnić kilka nierówności:

1. $pre[l_{2x-1}] < pre[l_{2x+1}]$ dla $1 \leqslant x < m$,
2. $pre[l_{2x}] > pre[l_{2x+2}]$ dla $1 \leqslant x < m$,
3. $pre[l_{2m-1}] < pre[l_{2m}]$.

Zanim przejdziemy do ich dowodu, zobaczmy, w jaki sposób wynika z nich ograniczenie $k = O(\sqrt{n})$. Niech $1 \leqslant x \leqslant m$. Mamy wtedy

$$pre[l_{2x-1}] < pre[l_{2x+1}] < \ldots < pre[l_{2m-1}] < pre[l_{2m}] < pre[l_{2m-2}] < \ldots < pre[l_{2x}].$$

Zatem $pre[l_{2x}] - pre[l_{2x-1}] \geqslant 2(m - x) + 1$. Zauważmy jeszcze, że dla każdych $0 \leqslant a, b \leqslant n$ mamy $|a - b| \geqslant |pre[a] - pre[b]|$, czyli $l_{2x} - l_{2x-1} \geqslant 2(m - x) + 1$.

Wiadomo, że $l_1 \leqslant l_2 \leqslant \ldots \leqslant l_k = l_{2m}$. Wobec tego

$$l_{2m} - l_1 \geqslant (l_{2m} - l_{2m-1}) + (l_{2m-1} - l_{2m-2}) + \ldots + (l_2 - l_1) \geqslant$$

$$\geqslant (l_{2m} - l_{2m-1}) + (l_{2m-2} - l_{2m-3}) + \ldots + (l_2 - l_1) \geqslant$$

$$\geqslant (2 \cdot 0 + 1) + (2 \cdot 1 + 1) + \ldots + (2 \cdot (m - 1) + 1) = m^2.$$

Wiemy jednak, że $l_k - l_1 \leqslant n$, stąd otrzymujemy szukane ograniczenie $m^2 \leqslant n$. Mamy więc nie więcej niż $O(\sqrt{n})$ poziomów.

Trzeba jeszcze tylko udowodnić nierówności (1), (2) i (3). Pozostawimy to jako ćwiczenie dla Czytelnika, podając tylko szkic dowodu w pierwszym przypadku.

Przypuśćmy, że $pre[l_{2x-1}] \geqslant pre[l_{2x+1}]$ dla pewnego $1 \leqslant x < m$. Słowo w_{2x-1} było lewostronnie legalne. Stąd $pre[l_{2x+1}] \geqslant pre[l_{2x-1}]$, czyli musi zachodzić $pre[l_{2x+1}] = pre[l_{2x-1}]$. Pokażemy, że wtedy słowo w_{2x+1} jest legalne, wbrew naszemu założeniu.

Skoro w_{2x-1} jest lewostronnie legalne, to dla każdego $l_{2x-1} \leqslant y \leqslant r_{2x-1}$ zachodzi $pre[y] \geqslant pre[l_{2x-1}] = pre[l_{2x+1}]$. Z kolei w_{2x} jest prawostronnie legalne, więc dla $l_{2x} \leqslant y \leqslant r_{2x}$ mamy $pre[y] \leqslant pre[r_{2x}]$. Wreszcie, wiemy, że $l_{2x} \leqslant l_{2x+1} \leqslant r_{2x}$. Po chwili zastanowienia dochodzimy do wniosku, że musi zachodzić $r_{2x+1} = r_{2x}$, więc słowo w_{2x+1} jest legalne. Otrzymana sprzeczność kończy dowód nierówności (1).

Na każdy z co najwyżej $O(\sqrt{n})$ poziomów poświęcamy $O(n)$ czasu, czyli ostateczna złożoność to $O(n\sqrt{n})$. To ograniczenie jest też optymalne – wartościowym ćwiczeniem może być samodzielne skonstruowanie wejścia, dla którego program musi poświęcić czas rzędu $\Omega(n\sqrt{n})$. Przykład takiego wejścia znajduje się w sekcji poświęconej testom.

Implementacja powyższego algorytmu znajduje się w pliku `bars10.cpp`. Taki program mógł zdobywać nawet do 100 punktów.

Rozwiązania niepoprawne

Zawodnicy zaproponowali dwa ciekawe typy rozwiązań niepoprawnych. Każde z nich opiera się na jakiejś „obserwacji”, która w ogólności jest niepoprawna, ale potrzeba danych o dosyć specyficznej strukturze, aby ją obalić.

1. „Jeżeli i jest lewym końcem najdłuższego legalnego podsłowa, to $pre[i-1]$ jest najmniejsze spośród liczb $pre[0], pre[1], \ldots, pre[i-1]$”.

   To rozwiązanie zbyt pochopnie odrzuca kandydatów na legalne podsłowo i może wypisać zbyt mały wynik. Zostało zaimplementowane w pliku `barb8.cpp`.

2. „Wystarczy znaleźć najdłuższe takie podsłowo $a_i a_{i+1} \ldots a_j$, aby dla każdego k ($i \leqslant k \leqslant j$) istniały takie x, y ($x < k \leqslant y$), że $pre[x] \leqslant pre[k]$ oraz $pre[k-1] \leqslant pre[y]$”.

   W oryginalnej wersji zadania mamy podobny warunek, ale wymagamy przy tym, aby $x = i - 1$ oraz $y = j$. Jeżeli pozbędziemy się tego założenia, to możemy znaleźć fałszywego kandydata na legalne podsłowo i wypisać zbyt dużą liczbę. Takie rozwiązanie zostało zaimplementowane w pliku `barb9.cpp`.

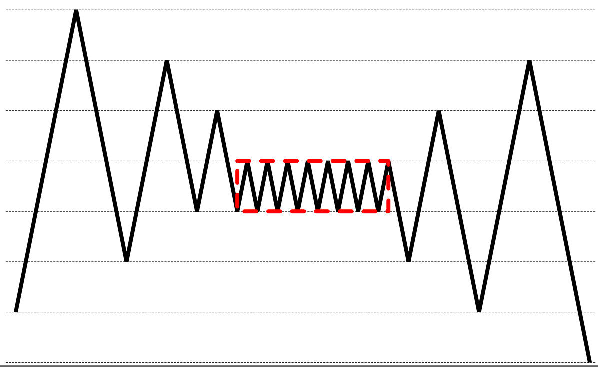
Testy

Przygotowano 10 grup testów. Pierwsza grupa zawiera małe testy poprawnościowe. Pozostałe grupy składają się z 3 lub 4 testów. Testy *a* i *b* są losowe z różnym prawdopodobieństwem występowania liter `p`. Test *c* jest dobrany spomiędzy następujących możliwości:

1. słowo Fibonacciego,

2. słowo Thuego-Morse'a,
3. słowo Bauma-Sweeta,
4. konkatenacja trzech słów, z których każde składa się odpowiednio z:
   - samych liter `j`,
   - samych liter `p`,
   - naprzemiennie ułożonych liter `p` i `j`.

Test *1g* oraz testy *d* występujące w grupach 3, 5, 7, 9 wszystkie mają strukturę podobną do przedstawionej na rysunku. Powodują błędne działanie rozwiązań `barb8.cpp`, `barb9.cpp` oraz maksymalny czas działania rozwiązania `bars10.cpp`.



Rys. 4: Wykres funkcji *pre* dla testu typu *d*; zaznaczono fragment odpowiadający najdłuższemu legalnemu podsłowu.