Jacek Tomasiewicz — Treść zadania, Opracowanie
Zbigniew Wojna — Program

Dostępna pamięć: 32 MB. *OI, Etap II, dzień pierwszy, 09.02.2011*

Różnica

Mamy dane słowo złożone z n małych liter alfabetu angielskiego `a` – `z`. *Chcielibyśmy wybrać pewien niepusty, spójny (tj. jednokawałkowy) fragment tego słowa, w taki sposób, aby różnica pomiędzy liczbą wystąpień najczęściej i najrzadziej występującej w tym fragmencie litery była jak największa. Zakładamy przy tym, że najrzadziej występująca litera w wynikowym fragmencie słowa musi mieć w tym fragmencie co najmniej jedno wystąpienie. W szczególności, jeżeli fragment składa się tylko z jednego rodzaju liter, to najczęstsza i najrzadsza litera są w nim takie same.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n *(*$1 \leqslant n \leqslant 1\,000\,000$*), oznaczającą długość słowa. Drugi wiersz zawiera słowo składające się z* n *małych liter alfabetu angielskiego.*

W testach wartych przynajmniej 30% punktów zachodzi dodatkowy warunek $n \leqslant 100$*.*

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą maksymalnej wartości różnicy między liczbą wystąpień najczęściej i najrzadziej występującej litery, jaką możemy znaleźć w pewnym spójnym fragmencie danego słowa.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`10` `aabbaaabab`	`3`

Wyjaśnienie do przykładu: *Fragment słowa, dla którego różnica między liczbą liter* `a` *i* `b` *wynosi 3, to* `aaaba`.

Rozwiązanie

Wprowadzenie

W tym zadaniu problem wyjątkowo nie jest ukryty w ramach historyjki, więc nie ma potrzeby formułowania go ponownie w bardziej formalny sposób. Ustalmy tylko, że przy szacowaniu złożoności rozwiązań przez A będziemy oznaczać rozmiar alfabetu (w przypadku naszego zadania mamy $A = 26$).

Każdy Czytelnik z pewnością zauważy, że skonstruowanie w pełni poprawnego, aczkolwiek siłowego rozwiązania tego zadania nie stanowi większej trudności.

Rozwiązanie siłowe $O(n^3 + n^2 \cdot A)$

Najprostszym rozwiązaniem jest rozpatrzenie wszystkich możliwych fragmentów, a dla każdego z nich znalezienie najczęściej i najrzadziej występującej litery. Oto pseudokod takiego siłowego algorytmu:

```
1:  wynik := 0;
2:  { rozpatrujemy każdy fragment }
3:  for i := 1 to n − 1 do
4:    for j := i + 1 to n do begin
5:      for k := 'a' to 'z' do
6:        licznik[k] := 0;
7:      for k := i to j do
8:        licznik[slowo[k]] := licznik[slowo[k]] + 1;
9:      { znajdujemy maksymalną i minimalną liczbę wystąpień }
10:     maks := 0;   mini := ∞;
11:     for k := 'a' to 'z' do
12:     begin
13:       maks := max(maks, licznik[k]);
14:       if licznik[k] > 0 then
15:         mini := min(mini, licznik[k]);
16:     end
17:     wynik := max(wynik, maks − mini);
18:   end
19: return wynik;
```

Powyższy algorytm działa w czasie $O(n^3+n^2 \cdot A)$, ponieważ wszystkich fragmentów jest $O(n^2)$, a dla każdego z nich znajdujemy w czasie $O(n + A)$ minimalne i maksymalne wystąpienie liter. Za takie rozwiązanie można było uzyskać około 30 punktów. Jego przykładowa implementacja znajduje się w plikach `rozs1.cpp` oraz `rozs4.pas`.

Rozwiązanie wolne $O(n^2 \cdot A)$

Jak widać, większość czasu w poprzednim rozwiązaniu spędzamy, zliczając wystąpienia liter w danym fragmencie. Ten element możemy łatwo poprawić — gdybyśmy wcześniej policzyli dla każdego prefiksu słowa (czyli początkowego jego fragmentu), ile razy każda litera w nim występuje, wystarczyłoby odjąć odpowiednie wartości.

Wobec tego na początku w następujący sposób wypełnimy tablicę *pref*:

```
1: for k := 'a' to 'z' do pref[0][k] := 0;
2: for i := 1 to n do begin
3:   for k := 'a' to 'z' do pref[i][k] := pref[i − 1][k];
4:   pref[i][slowo[i]] := pref[i][slowo[i]] + 1;
5: end
```

Teraz nie musimy przeglądać wszystkich liter każdego fragmentu. Linie 5–8 poprzedniego rozwiązania możemy zastąpić następującym pseudokodem:

```
1: for k := 'a' to 'z' do
2:     licznik[k] := pref[j][k] − pref[i − 1][k];
```

Nowy algorytm działa w czasie $O(n^2 \cdot A)$, gdyż w każdym fragmencie rozważamy każdą literę z alfabetu. Niestety takie rozwiązanie jest wciąż za wolne i dostaje około 40 punktów. Jego implementacja znajduje się w plikach `rozs2.cpp` oraz `rozs5.pas`.

Szybsze rozwiązanie $O(n \cdot A^2)$

Zastanówmy się, jak lepiej rozwiązać zadanie. Złożoność kwadratowa w zależności od n nie jest satysfakcjonująca, więc nie możemy sobie pozwolić na przeglądanie wszystkich fragmentów. Spróbujmy podejść do problemu od innej strony. Rozważmy wszystkie pary różnych liter z założeniem, że docelowo jedna będzie najczęściej występującą, a druga najrzadziej. Teraz dla danej pary, przykładowo (`a`, `b`), będziemy szukać fragmentu, w którym różnica liczb wystąpień `a` i `b` jest możliwie największa. Wymagamy przy tym, żeby w wynikowym fragmencie litera `b` występowała przynajmniej raz.

Gdyby pominąć ostatni warunek, można by prosto sprowadzić problem do klasycznego zadania: znajdowania w ciągu liczb spójnego fragmentu o maksymalnej sumie. Wystarczy zastąpić wystąpienia `a` przez 1, `b` przez -1, a pozostałych liter przez `0`.

Znane są algorytmy wyznaczające maksymalną sumę spójnego fragmentu ciągu w czasie liniowym. Możemy wyliczać dynamicznie aktualną sumę prefiksu ciągu. Aby wyznaczyć podciąg spójny o maksymalnej sumie kończący się na danej pozycji, wybieramy krótszy prefiks o dotychczas najmniejszej sumie. Wynikiem jest różnica między sumami prefiksów: bieżącego i najmniejszego krótszego. Oto pseudokod takiego klasycznego algorytmu:

```
1: wynik, suma, mini := 0;
2: for i := 1 to n do begin
3:     suma := suma + a[i];
4:     mini := min(mini, suma);
5:     wynik := max(wynik, suma − mini);
6: end
7: return wynik;
```

W naszym przypadku musimy jednak pamiętać, że znaleziony fragment musi zawierać przynajmniej jedną -1, a powyższe rozwiązanie tego nie zakłada. Szukając prefiksu o najmniejszej sumie, musimy ograniczyć się tylko do tych, które nie zawierają ostatnio odwiedzonej -1. Zauważmy jeszcze, że każdy interesujący nas prefiks jest albo pusty, albo kończy się liczbą -1. W przeciwnym razie moglibyśmy ten prefiks skrócić, uzyskując prefiks o mniejszej sumie.

Wobec tego, gdy napotkaliśmy już k razy wartość -1, interesują nas jedynie: prefiks pusty oraz prefiksy kończące się w jednej z $k-1$ pierwszych wartości -1. Jeśli jeszcze na żadną -1 nie natrafiliśmy, czyli $k = 0$, to nie możemy rozważać żadnego prefiksu. Poprzedni pseudokod poprawiamy następująco:

```
1: wynik, suma, suma_pop := 0;
2: mini := ∞;
3: for i := 1 to n do begin
```

```
4:    suma := suma + a[i];
5:    if a[i] = −1 then begin
6:       mini := min(mini, suma_pop);
7:       suma_pop := suma;
8:    end
9:    wynik := max(wynik, suma − mini);
10: end
11: return wynik;
```

Tutaj *sum_pop* to suma wartości prefiksu kończącego się ostatnią wczytaną -1; na początku — prefiksu pustego. Wartość *mini* to natomiast minimalna suma prefiksu, który możemy rozważać; na początku nie ma takiego prefiksu.

Otrzymujemy w ten sposób algorytm, który działa w czasie $O(n \cdot A^2)$, gdyż wszystkich par liter z alfabetu jest $O(A^2)$, a znalezienie podciągu o maksymalnej sumie zajmuje nam czas $O(n)$. Rozwiązania tej złożoności otrzymywały co najmniej 60 punktów. Zaprezentowane wyżej zostało zaimplementowane w pliku `rozs9.cpp`.

Rozwiązanie wzorcowe $O(n \cdot A)$

Zastanówmy się, jak poprawić poprzednie rozwiązanie. Zauważmy, że gdy w ciągu rozpatrujemy $a[i] = 0$, to nie dzieje się nic ciekawego. Dlaczego więc nie pominąć wszystkich zer?

Okazuje się, że jest to dobry pomysł. Trzeba oczywiście pamiętać, żeby wycinając zera, nie przeglądać całego ciągu, gdyż wtedy złożoność pozostanie bez zmian. Należy dla każdej litery stworzyć listę zawierającą pozycje jej wystąpień w słowie, a następnie dla danej pary liter scalać listy odpowiadające tym literom.

Oszacujmy czas działania tak poprawionego algorytmu. Generowanie list wystąpień zajmie nam czas $O(n + A)$. Czas działania głównej fazy to suma czasów dla wszystkich par liter z alfabetu. Niech $occ(\ell)$ oznacza liczbę wystąpień litery ℓ w początkowym słowie. Wtedy czas dla pojedynczej pary liter (ℓ_1, ℓ_2) wynosi $O(occ(\ell_1)+occ(\ell_2))$. Ostatecznie sumaryczna złożoność to z dokładnością do czynnika stałego:

$$\sum_{\ell_1}\sum_{\ell_2}\left(occ\left(\ell_1\right)+occ\left(\ell_2\right)\right)=2\sum_{\ell_1}\sum_{\ell_2}occ\left(\ell_2\right)=2A\cdot\sum_{\ell_2}occ\left(\ell_2\right)=O(n\cdot A).$$

Implementacja opisanego rozwiązania wzorcowego znajduje się w pliku `roz8.cpp`. Takie rozwiązanie otrzymuje maksymalną liczbę punktów, podobnie jak inne rozwiązania o tej złożoności.

Alternatywne warianty rozwiązania $O(n \cdot A)$

Przedstawiony wyżej klasyczny algorytm wyznaczania maksymalnej sumy spójnego fragmentu ciągu można zapisać nieco inaczej. Zmienne *suma* i *mini* możemy zastąpić jedną, równą *suma* − *mini*. Wówczas na algorytm pierwotnie skonstruowany jako dynamiczny można spojrzeć jak na rozwiązanie zachłanne.

```
1: wynik, suma := 0;
2: for i := 1 to n do begin
3:    suma := suma + a[i]; { przedłużamy aktualny ciąg }
4:    if suma < 0 then suma := 0; { zaczynamy konstrukcję od nowa }
5:    wynik := max(wynik, suma);
6: end
7: return wynik;
```

Taki algorytm również można w naturalny sposób dopasować do potrzeb naszego zadania, w którym wymagamy wystąpienia choć raz wartości -1. Jest to jednak bardziej skomplikowane. Szczegóły pozostawiamy zainteresowanemu Czytelnikowi jako ćwiczenie. Przykładowa implementacja znajduje się w pliku `roz9.cpp`.

Redukcję złożoności $O(n \cdot A^2)$ o czynnik A można także wykonać zupełnie inaczej. Można równocześnie prowadzić obliczenia dla wszystkich par liter lub dla par o wspólnym pierwszym elemencie. Wówczas używane zmienne zastępujemy odpowiednio dwu- lub jednowymiarowymi tablicami. To podejście daje w praktyce wyraźnie szybsze programy, gdyż pomijamy fazy generowania i scalania list wystąpień. Przykład takiej implementacji można znaleźć w pliku `roz10.cpp`.

Testy

W zestawie były trzy typy testów: małe testy poprawnościowe wygenerowane ręcznie (typ **R**), większe losowe testy wydajnościowe (typ **L**) oraz testy poprawnościowo-wydajnościowe (typ **RR**), w których litery występowały równomiernie.

Nazwa	n	Opis
roz1a.in	10	test typu **R**
roz1b.in	12	test typu **R**
roz1c.in	5	test typu **R**
roz1d.in	3	test typu **R**
roz2a.in	10	test typu **R**
roz2b.in	50	test typu **L**
roz2c.in	17	test typu **RR**
roz3a.in	1	test typu **R**
roz3b.in	100	test typu **L**
roz3c.in	91	test typu **RR**
roz4a.in	2 500	test typu **L**
roz4b.in	10	test typu **R**
roz5a.in	10 000	test typu **L**
roz5b.in	100	test typu **R**

Nazwa	n	Opis
roz6a.in	100 000	test typu **L**
roz6b.in	10	test typu **R**
roz6c.in	19 800	test typu **RR**
roz7a.in	1 000 000	test typu **L**
roz7b.in	5	test typu **R**
roz7c.in	817 824	test typu **RR**
roz8a.in	1 000 000	test typu **L**
roz8b.in	803 088	test typu **RR**
roz8c.in	856 836	test typu **RR**
roz9a.in	1 000 000	test typu **L**
roz9b.in	999 984	test typu **RR**
roz9c.in	999 027	test typu **RR**
roz10a.in	1 000 000	test typu **L**
roz10b.in	1 000 000	test typu **RR**
roz10c.in	999 999	test typu **RR**