Jakub Radoszewski, Wojciech Rytter — Treść zadania, Opracowanie
Adam Gawarkiewicz — Program

Dostępna pamięć: 32 MB. *OI, Etap I, 20.10–17.11.2008*

Przyspieszenie algorytmu

Bajtazar musi za karę obliczyć pewną paskudną i tajemniczą funkcję logiczną $\mathbf{F}(x,y)$*, która dla dwóch ciągów liczb naturalnych* $x = (x_1,\ldots,x_n)$*,* $y = (y_1,\ldots,y_m)$ *jest zdefiniowana w następujący sposób:*

boolean $\mathbf{F}(x,y)$
  if $\mathbf{W}(x) \neq \mathbf{W}(y)$ **then return** 0
  else if $|\mathbf{W}(x)| = |\mathbf{W}(y)| = 1$ **then return** 1
  else return $\mathbf{F}(\mathbf{p}(x),\mathbf{p}(y)) \wedge \mathbf{F}(\mathbf{s}(x),\mathbf{s}(y))$.

W powyższym zapisie:

- $\mathbf{W}(x)$ *oznacza zbiór złożony ze wszystkich liczb ciągu* x *(ignorujemy powtórzenia i kolejność liczb),*
- $\mathbf{p}(x)$ *jest najdłuższym prefiksem (początkowym fragmentem) ciągu* x*, dla którego* $\mathbf{W}(x) \neq \mathbf{W}(\mathbf{p}(x))$*,*
- $\mathbf{s}(x)$ *jest najdłuższym sufiksem (końcowym fragmentem) ciągu* x*, dla którego* $\mathbf{W}(x) \neq \mathbf{W}(\mathbf{s}(x))$*,*
- $\wedge$ *oznacza koniunkcję logiczną, 1 — prawdę, 0 — fałsz, a* $|Z|$ *— liczność zbioru* Z*.*

Na przykład dla ciągu $x = (2,3,7,2,7,4,7,2,4)$ *mamy:*

$$\mathbf{W}(x) = \{2,3,4,7\}, \quad \mathbf{p}(x) = (2,3,7,2,7), \quad \mathbf{s}(x) = (7,2,7,4,7,2,4).$$

Dla bardzo dużych danych program obliczający funkcję bezpośrednio z definicji jest zdecydowanie zbyt wolny. Twoim zadaniem jest jak największe przyspieszenie obliczania tej funkcji.

Napisz program, który wczyta ze standardowego wejścia kilka par ciągów (x,y) *i wypisze na standardowe wyjście wartości* $\mathbf{F}(x,y)$ *dla każdej pary wczytanych ciągów.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą k $(1 \leqslant k \leqslant 13)$*, oznaczającą liczbę par ciągów do przeanalizowania. Kolejne* $3k$ *wierszy zawiera opisy przypadków testowych. Pierwszy wiersz każdego opisu zawiera dwie liczby całkowite* n *oraz* m $(1 \leqslant n,m \leqslant 100\,000)$*, oddzielone pojedynczym odstępem i oznaczające długości pierwszego i drugiego ciągu. Drugi wiersz zawiera* n *liczb całkowitych* x_i $(1 \leqslant x_i \leqslant 100)$*, pooddzielanych pojedynczymi odstępami i opisujących ciąg* x*. Trzeci wiersz zawiera* m *liczb całkowitych* y_i $(1 \leqslant y_i \leqslant 100)$*, pooddzielanych pojedynczymi odstępami i opisujących ciąg* y*.*

Wyjście

Wyjście powinno składać się z k wierszy; i-ty wiersz (dla $1 \leqslant i \leqslant k$) powinien zawierać jedną liczbę całkowitą — 0 lub 1 — oznaczającą wartość wyrażenia $\mathbf{F}(x,y)$ dla i-tego przypadku testowego.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`2`	`0`
`4 5`	`1`
`3 1 2 1`	
`1 3 1 2 1`	
`7 7`	
`1 1 2 1 2 1 3`	
`1 1 2 1 3 1 3`	

Rozwiązanie

Wstęp

Zadanie o *Przyspieszeniu algorytmu* jest dosyć abstrakcyjne i jego treść nie zawiera żadnej „bajtockiej" fabuły, co sprawiło zawodnikom pewną dodatkową trudność. Było to również pierwsze zadanie w zawodach Olimpiady Informatycznej, w którego treści podany był algorytm i należało napisać program obliczający wynik tego algorytmu. Główną trudnością zadania było zrozumienie struktury działania podanego algorytmu, nie wiedząc nic o tym, jaki sens ma jego wynik (co było do rozwiązania zupełnie niepotrzebne).

Efektywne rozwiązanie wymagało strukturalnej zmiany nieefektywnego algorytmu na algorytm dający takie same wyniki, ale efektywniejszy. Po dokładnym zrozumieniu problem sprowadza się nieoczekiwanie do wielokrotnego sortowania trójek niedużych liczb całkowitych.

Ponieważ w zadaniu występują prefiksy, sufiksy i (w sposób niejawny) podsłowa, wygodniej jest omawiać rozwiązanie, interpretując ciągi jako słowa składające się z symboli (z elementów ciągu).

Rozwiązania brutalne

Najprostszym rozwiązaniem zadania jest implementacja funkcji **F** bezpośrednio z definicji rekurencyjnej, ale daje to algorytm o wykładniczej złożoności czasowej, o czym można się przekonać, analizując działanie takiego algorytmu dla danych $x = y = (1,2,\ldots,n)$. Najprostszą taką implementację można znaleźć w pliku `przs0.cpp`, a nieco ulepszone wersje w plikach `przs1.cpp` i `przs2.pas`.

Algorytm wielomianowy możemy otrzymać, stosując metodę *tablicowania* (czyli spamiętywania lub programowania dynamicznego) — obliczone wartości funkcji **F** zapamiętu-

jemy w tablicy, dzięki czemu nie musimy wywoływać funkcji wielokrotnie dla tych samych par argumentów. Algorytm taki już nie jest wykładniczy, ale jego złożoność czasowa jest wciąż istotnie zbyt duża — rzędu $O(n^2m^2(n+m))$. Faktycznie, na pierwszym z argumentów **F** zawsze występuje jakieś *podsłowo* (czyli spójny fragment) słowa x, a na drugim jakieś podsłowo słowa y, co daje $O(n^2m^2)$ możliwych par argumentów, a obliczenie dla każdej pary można wykonać w czasie $O(n+m)$ (przy założeniu, że liczba liter słów x i y jest ograniczona przez stałą). Implementacje takiego podejścia można znaleźć w plikach `przs3.cpp` i `przs4.pas`.

Rozwiązanie wzorcowe

Niech A będzie alfabetem — zbiorem symboli występujących w obu słowach. Istotnym parametrem w zadaniu jest liczba $K = |A|$ (będziemy zakładać dla uproszczenia, że elementami A są liczby od 1 do K) — z ograniczeń z treści zadania wynika, że $K \leqslant 100$. Słowo puste będziemy oznaczać jako ε.

Zauważmy, że funkcję **F** możemy traktować jako relację. Powiemy, że dwa słowa x i y są w tej relacji (co zapisujemy jako $x \approx y$), jeśli $\mathbf{F}(x,y) = 1$. Okazuje się, że ta relacja jest relacją równoważności[1], co można uzasadnić, korzystając z faktu, że jeżeli $\mathbf{F}(x,y) = 1$, to ciąg argumentów na pierwszej współrzędnej **F** zależy tylko od słowa x (i symetrycznie dla y). Relacja ta wyznacza podział zbioru wszystkich słów nad alfabetem A na klasy równoważności[2] — wiedząc, do jakich klas należą słowa x i y, możemy od razu obliczyć $\mathbf{F}(x,y)$.

Aby istotnie przyspieszyć nasz algorytm, zastosujemy zatem podejście natury strukturalnej: zastąpimy wejściową funkcję **dwuargumentową** funkcją (faktycznie tablicą) **jednoargumentową** obliczającą numer pewnej klasy równoważności.

Oznaczmy przez L_k (*k-ta warstwa*) zbiór podsłów u początkowych słów x i y, takich że $|\mathbf{W}(u)| = k$. Zauważmy, że dla u, v należących do różnych warstw zachodzi $\mathbf{F}(u,v) = 0$, więc zasadniczo interesuje nas sprawdzanie, czy $u \approx v$ dla u i v należących do tej samej warstwy. Zamiast zajmować się funkcją dwuargumentową **F**, będziemy obliczać tablicę jednoargumentową $NUM[u]$, określającą numer klasy równoważności, do której należy u w swojej warstwie. Innymi słowy, dla u, v należących do tej samej warstwy będzie zachodziło:

$$\mathbf{F}(u,v) = 1 \quad \Longleftrightarrow \quad NUM[u] = NUM[v].$$

Dla słowa u zdefiniujmy jego δ-*reprezentację* $\delta(u)$ jako uporządkowaną trójkę obiektów:

$$\delta(u) \;=\; (p,\, a,\, q),$$

przy czym:

- $p = \mathbf{p}(u)$; $q = \mathbf{s}(u)$;

[1] *Relacja równoważności* to taka relacja pomiędzy elementami pewnego zbioru, która jest *zwrotna* (czyli każdy element jest sam ze sobą w relacji), *symetryczna* (jeżeli a jest w relacji z b, to b jest w relacji z a) oraz *przechodnia* (jeżeli a jest w relacji z b oraz b jest w relacji z c, to również a jest w relacji z c).

[2] *Klasa równoważności* (inaczej klasa abstrakcji) to maksymalny ze względu na zawieranie podzbiór elementów, którego każde dwa elementy są ze sobą w danej relacji równoważności.

- $a \in A$ i pa (słowo p z dopisanym symbolem a na końcu) jest prefiksem u.

Przykład.

$$\delta(ababbbcbcbc) \;=\; (ababbb,\; c,\; bbbcbcbc).$$

Zauważmy, że dla $\delta(u) = (p,a,q)$, jeżeli słowo u należy do k-tej warstwy, to słowa p i q należą do warstwy o numerze $k-1$ oraz $\mathbf{W}(u) = \mathbf{W}(p) \cup \{a\}$.

Przypomnijmy, że chcemy sprowadzić problem do obliczania tablicy NUM, która dla słów u,v z tej samej warstwy spełnia $(u \approx v) \iff (NUM[u] = NUM[v])$ oraz zbiór wartości w tej tablicy składa się z małych liczb całkowitych (konkretniej będzie on postaci $\{1,2,\ldots,w\}$). W tym celu, zamiast δ-reprezentacji $\delta(u) = (p,a,q)$ będziemy używać jej skompresowanej wersji (oznaczanej Δ), będącej trójką liczb naturalnych:

$$\Delta(u) \;=\; (NUM[p],\; a,\; NUM[q]).$$

Okazuje się teraz, że skompresowana postać δ-reprezentacji posiada następującą, istotną dla nas własność. Niech x i y należą do tej samej warstwy oraz $\delta(x) = (p,a,q)$, $\delta(y) = (p',a',q')$. Wtedy:

$$\begin{aligned}
\mathbf{F}(x,y)=1 \quad &\iff \quad (\mathbf{F}(p,p')=1 \;\wedge\; \mathbf{F}(q,q')=1 \;\wedge\; \mathbf{W}(x)=\mathbf{W}(y)) && (1)\\
&\iff \quad (\mathbf{F}(p,p')=1 \;\wedge\; \mathbf{F}(q,q')=1 \;\wedge\; \mathbf{W}(p)\cup\{a\}=\mathbf{W}(p')\cup\{a'\}) && (2)\\
&\iff \quad (p\approx p' \;\wedge\; q\approx q' \;\wedge\; a=a') && (3)\\
&\iff \quad (NUM[p]=NUM[p'] \;\wedge\; NUM[q]=NUM[q'] \;\wedge\; a=a') && (4)\\
&\iff \quad \Delta(x)=\Delta(y). && (5)
\end{aligned}$$

Jedyne przejście niewynikające bezpośrednio z definicji to (2)–(3). Korzystamy w nim z faktu, że jeśli $\mathbf{F}(p,p') = 1$, to $\mathbf{W}(p) = \mathbf{W}(p')$ i ponadto $\mathbf{W}(p) \cap \{a\} = \varnothing$ oraz $\mathbf{W}(p') \cap \{a'\} = \varnothing$.

Powyższą własność możemy wykorzystać do obliczania wartości NUM warstwa po warstwie. Aby wyznaczyć numerację NUM dla L_k, możemy posortować elementy k-tej warstwy względem ich wartości Δ. Wówczas elementy z tą samą wartością Δ będą stanowiły spójny fragment posortowanej listy trójek. Przeglądając ją, możemy łatwo przydzielić im kolejne numery w tablicy NUM.

Do sortowania będziemy używać sortowania pozycyjnego (Radix Sort), którego opis Czytelnik może znaleźć np. w książce [20]. Na potrzeby tego zadania wystarczy wiedzieć, że działa ono w czasie $O(n+m+K)$.

Nasz algorytm możemy więc opisać nieformalnie w następujący sposób:

```
1: { numeracja warstwy zerowej: }
2: NUM[ε] := 1;
3:
4: for k := 1 to K do
5:     { numeracja k-tej warstwy: }
6:     wygeneruj elementy L_k;
7:     oblicz wartości Δ dla elementów L_k;
```

posortuj L_k względem wartości Δ;
kolejnym spójnym fragmentom L_k o jednakowej wartości Δ
przypisz kolejne liczby naturalne w tablicy NUM;

Implementacja w czasie $O(n^2)$

Skonstruowaliśmy już pewien abstrakcyjny algorytm, lecz brakuje nam sensownej implementacji podziału podsłów na warstwy i obliczania wartości Δ. Odtąd będziemy zakładać, że rozważamy tylko jedno słowo i jego podsłowa — możemy na przykład dopisać do początkowego słowa x słowo y, oddzielając je jakimś separatorem # (symbolem niewystępującym w x i y). Tak więc w dalszym tekście opiszemy, jak zaimplementować algorytm obliczający tablicę NUM dla podsłów pojedynczego słowa x o długości n.

Podsłowa możemy utożsamiać z przedziałami postaci $[i, j]$ dla $1 \leqslant i \leqslant j \leqslant n$. Każdy taki przedział reprezentuje podsłowo $x[i..j] = x_i x_{i+1} \ldots x_j$.

Wprowadźmy tablicę

$$ILE[i, j] = |\mathbf{W}(x[i..j])| \text{ dla } 1 \leqslant i \leqslant j \leqslant n.$$

Inaczej mówiąc, $ILE[i, j]$ jest liczbą różnych symboli w podsłowie $x[i..j]$. Na podstawie wartości $ILE[i, j]$ możemy stablicować wartości funkcji **p** oraz **s** (w tablicach dwuwymiarowych, podobnie jak w przypadku $ILE[i, j]$). Mając wyznaczone te wartości, możemy już zrealizować omówiony wcześniej ogólny schemat rozwiązania.

Wszystkie te obliczenia można wykonać w czasie $O(n^2)$, co zostawiamy jako proste ćwiczenie, tym bardziej że w następnej części dokładniej opiszemy, jak to wszystko zrobić w czasie $O(nK)$. Dodajmy tylko, że implementacje stosownego rozwiązania można znaleźć w plikach `przs7.cpp` i `przs8.pas`.

Implementacja w czasie $O(nK)$

Implementacja taka ma sens, gdy K jest znacznie mniejsze niż n, a taka sytuacja ma miejsce w naszym zadaniu. W tym rozwiązaniu ograniczymy zbiór podsłów, z którymi mamy do czynienia. Motywacją do tego jest definicja funkcji **p** oraz **s** — wartościami tych funkcji są szczególne słowa odpowiadające szczególnym przedziałom, które nazwiemy *k-przedziałami*.

Przedział $[i, j]$ nazwiemy k-przedziałem, gdy $x[i..j]$ jest najdłuższym prefiksem słowa $x[i..n]$ zawierającym dokładnie k różnych symboli (co zapisujemy jako $j = PREF_k[i]$) lub $x[i..j]$ jest najdłuższym sufiksem słowa $x[1..j]$ zawierającym dokładnie k różnych symboli (co zapisujemy jako $i = SUF_k[j]$). Przyjmijmy, że $SUF_k[j] = PREF_k[i] = 0$, jeśli wartości te nie są zdefiniowane przez k-przedziały, czyli $|\mathbf{W}(x[1..j])| < k$ lub $|\mathbf{W}(x[i..n])| < k$.

W szczególności, jeśli rozpatrujemy słowo $x\#y$, gdzie # jest separatorem, to przedział odpowiadający x jest k-przedziałem, gdyż x jest najdłuższym prefiksem całego słowa o $|\mathbf{W}(x)|$ różnych literach (formalnie $PREF_{|\mathbf{W}(x)|}[1] = |x|$). Z analogicznych powodów przedział odpowiadający y też jest k-przedziałem.

Okazuje się, że wszystkie przedziały z k-tej warstwy osiągalne za pomocą operacji **p** oraz **s** to dokładnie k-przedziały reprezentowane przez tablice $PREF_k$ i SUF_k. Pokażemy teraz, jak obliczyć te tablice dla ustalonego k w czasie liniowym względem n.

W algorytmie użyjemy struktury danych Z reprezentującej *multizbiór* (czyli zbiór z powtórzeniami) symboli alfabetu A. Z implementujemy jako tablicę $Z[1..K]$ rozmiaru K, zliczającą krotności poszczególnych symboli. Dodatkowo pamiętamy liczbę niezerowych komórek tej tablicy — $|Z|$, czyli rozmiar *zbioru* odpowiadającego Z.

Wstawienie elementu do Z implementujemy jako zwiększenie o jeden odpowiedniego licznika w tablicy; powoduje to zwiększenie $|Z|$, jeżeli licznik ten był w tym momencie równy zeru. Podobnie, usunięcie elementu polega na zmniejszeniu odpowiedniego licznika o jeden i zmniejszeniu $|Z|$ w przypadku wyzerowania tego licznika. W ten sposób każda z tych operacji, podobnie jak wyznaczenie wartości $|Z|$, może zostać wykonana w czasie stałym.

Poniższy pseudokod pokazuje, jak za pomocą multizbioru Z obliczyć $PREF_k$ w czasie $O(n)$; obliczenie SUF_k może zostać wykonane analogicznie.

```
1: Z := ∅;
2: j := 1;
3: for i := 1 to n do
4:    if i > 1 then Z := Z \ {w[i−1]};
5:    while j < n and |Z ∪ {w[j+1]}| ⩽ k do
6:       j := j + 1;
7:       Z := Z ∪ {w[j]};
8:    if |Z| = k then PREF_k[i] := j;
```

Oszacowanie $O(n+K) = O(n)$ na złożoność czasową powyższego algorytmu wynika z tego, że łączna liczba obrotów pętli **while** jest mniejsza niż n, gdyż w każdym obrocie wartość zmiennej j wzrasta o 1.

Jak zapowiadaliśmy, mając wyznaczone tablice $PREF_k$ i SUF_k, można już łatwo obliczać wartości funkcji **p** i **s**. Faktycznie, jeżeli $|\mathbf{W}(x[i..j])| = k$, to

$$\mathbf{p}(x[i..j]) = x[i..PREF_{k-1}[i]] \quad \text{oraz} \quad \mathbf{s}(x[i..j]) = x[SUF_{k-1}[j]..j].$$

Używając tych wartości, możemy z kolei obliczać wartości NUM dla wszystkich podsłów odpowiadających k-przedziałom.

Nasz algorytm działa teraz w czasie $O(nK)$ i pamięci również $O(nK)$. Możemy jednak bardzo łatwo zmniejszyć zużycie pamięci do $O(n)$. Wystarczy jedynie trzymać w algorytmie dane związane z bieżącą warstwą L_k i poprzednią warstwą L_{k-1}. Pozostałe dane można usuwać na bieżąco.

W ten sposób uzyskujemy rozwiązanie wzorcowe, zaimplementowane w plikach `prz.cpp`, `prz0.pas` i `prz1.java`.

Testy

Zadanie było sprawdzane na 10 zestawach danych, z których każdy składał się z kilku bądź kilkunastu pojedynczych testów zawartych w jednym pliku. Z tego względu zadanie to miało rekordowe w historii Olimpiady Informatycznej limity czasowe, rzędu kilku minut, co stanowiło kolejną jego nietypową właściwość.

Ponieważ zupełnie losowe testy z ogromnym prawdopodobieństwem dają odpowiedź 0, większość testów była generowana dosyć skomplikowanymi metodami rekurencyjnymi, jakby „naśladującymi" kolejne wywołania rekurencyjne podanego w treści zadania algorytmu obliczania funkcji **F**.

Poniższa tabelka podsumowuje parametry użytych testów.

Nazwa	k	n, m	K
prz1.in	11	≈ 15	4–7
prz2.in	11	≈ 30	6–15
prz3.in	13	≈ 50	15–25
prz4.in	13	≈ 60	10–30
prz5.in	11	$\approx 50\,000$	80–100
prz6.in	9	$\approx 60\,000$	100
prz7.in	8	$\approx 70\,000$	100
prz8.in	8	$\approx 80\,000$	100
prz9.in	8	$\approx 90\,000$	100
prz10.in	8	$\approx 100\,000$	100

Czy funkcja **F** wzięła się z sufitu?

Co prawda do rozwiązania niniejszego zadania zrozumienie istoty funkcji **F** faktycznie okazało się zupełnie niepotrzebne, jednakże na koniec pragniemy wyjaśnić, że funkcja ta nie wzięła się znikąd. Otóż można pokazać, że $\mathbf{F}(x,y) = 1$ wtedy i tylko wtedy, gdy słowo x można przekształcić w y za pomocą pewnej liczby operacji, z których każda polega na:

1. zastąpieniu podsłowa słowa x będącego kwadratem u^2 (sklejenie u z u) pojedynczym słowem u albo
2. zastąpieniu dowolnego podsłowa u słowa x jego kwadratem u^2.

Ciąg takich przekształceń dla drugiego przykładu z treści zadania wygląda następująco (nad strzałkami zapisano numery stosowanych operacji):

$$1\ \underbrace{1\ 2\ 1\ 2}\ 1\ 3 \xrightarrow{1} 1\ 1\ 2\ \underbrace{1\ 3} \xrightarrow{2} 1\ 1\ 2\ 1\ 3\ 1\ 3.$$

Ktoś mógłby zapytać, dlaczego w treści zadania zamiast występującej tam abstrakcyjnej i technicznej definicji nie pojawiła się powyższa, wszak dużo przyjemniejsza i bardziej zrozumiała? Odpowiedź tkwi w dowodzie równoważności tych definicji, który jest istotnie trudniejszy do wymyślenia niż całe rozwiązanie zadania o *Przyspieszeniu algorytmu* w obecnym jego sformułowaniu. A zatem, wbrew pozorom obecna treść tego zadania *ułatwia* jego rozwiązanie!

Czytelników zainteresowanych wspomnianym dowodem zachęcamy do zajrzenia do książki [38] albo poszperania po Internecie pod hasłem *free idempotent monoid*. (Ostrzegamy jednak, że w ten sposób można się „doklikać" do całkiem trudnych i skomplikowanych materiałów).