Jacek Tomasiewicz — Treść zadania
Jakub Radoszewski — Opracowanie
Dawid Dąbrowski, Jan Kanty Milczek — Program

Dostępna pamięć: 64 MB. *OI, etap III, dzień drugi, 29.03.2012*

Prefiksufiks

W tym zadaniu będą nas interesować napisy złożone z małych liter alfabetu angielskiego. **Prefiksem** *danego napisu nazwiemy dowolny jego początkowy fragment.* **Sufiksem** *danego napisu nazwiemy dowolny jego końcowy fragment. W szczególności, pusty napis jest zarówno prefiksem, jak i sufiksem dowolnego napisu. Dwa napisy nazywamy* **cyklicznie równoważnymi***, jeżeli jeden z nich można uzyskać z drugiego, przestawiając pewien jego sufiks z końca napisu na początek. Dla przykładu, napisy* `ababba` *i* `abbaab` *są równoważne cyklicznie, a napisy* `ababba` *i* `ababab` *nie są. W szczególności, każdy napis jest sam sobie cyklicznie równoważny.*

Dany jest napis t złożony z n liter. Szukamy jego prefiksu p i sufiksu s, obu tej samej długości, takich że:

- *p i s są sobie równoważne cyklicznie,*
- *długość p i s nie przekracza $\frac{n}{2}$ (czyli prefiks p i sufiks s nie zachodzą na siebie w t), oraz*
- *długość p i s jest jak największa.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczającą długość danego napisu t. Drugi wiersz wejścia zawiera napis t składający się z n małych liter alfabetu angielskiego.

W testach wartych 30% punktów zachodzi dodatkowy warunek $n \leqslant 500$.

W testach wartych 50% punktów zachodzi dodatkowy warunek $n \leqslant 5\,000$.

Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu standardowego wyjścia jedną liczbę całkowitą, równą długości szukanego prefiksu p i sufiksu s.

Przykład

Dla danych wejściowych:

```
15
ababbabababbaab
```

poprawnym wynikiem jest:

```
6
```

Rozwiązanie

Problem postawiony w zadaniu jest uogólnieniem problemu znajdowania *prefikso-sufiksów* słowa t, czyli prefiksów słowa t, które są zarazem jego sufiksami. Znany jest liniowy algorytm wyznaczający wszystkie prefikso-sufiksy słowa — mowa tu o funkcji prefiksowej z algorytmu wyszukiwania wzorca w tekście metodą Knutha-Morrisa-Pratta (patrz np. książki [21, 23]). Przedstawiony w tym zadaniu problem *prefiksufiksów* jest bardziej złożony: szukany sufiks s może być dowolnym słowem równoważnym cyklicznie prefiksowi p. Nasz opis rozpocznijmy od analizy rozwiązań nieoptymalnych.

Rozwiązania wolniejsze

Załóżmy, że litery słowa t są ponumerowane od 1 do n. Najprostsze rozwiązanie zadania polega na sprawdzeniu wszystkich możliwości: dla każdej wartości parametru $m = 0, 1, \ldots, \lfloor \frac{n}{2} \rfloor$ chcemy stwierdzić, czy prefiks p i sufiks s o długości m są równoważne cyklicznie.

Jak sprawdzić, czy dwa słowa są równoważne cyklicznie? Najłatwiej zrobić to tak: m razy przenosimy pierwszą literę pierwszego słowa na jego koniec, za każdym razem sprawdzając, czy otrzymaliśmy w ten sposób drugie słowo. Ta metoda wymaga wykonania $O(m^2)$ operacji, gdyż sprawdzenie równości słów długości m zajmuje czas $O(m)$. Można jednak lepiej. Zauważmy, że p i s są równoważne cyklicznie wtedy i tylko wtedy, gdy wzorzec s występuje jako spójny fragment tekstu pp (p sklejone z p). Problem wyszukiwania wzorca w tekście można rozwiązać w czasie liniowym np. za pomocą wspomnianego już algorytmu Knutha-Morrisa-Pratta. W ten sposób otrzymujemy algorytm sprawdzania równoważności cyklicznej słów o złożoności czasowej $O(m)$. Znane są też bardziej wysublimowane algorytmy działające w tej samej złożoności czasowej, patrz opis rozwiązania zadania *Naszyjniki* z VIII Olimpiady Informatycznej [8] czy książka [21].

Stosując naiwną metodę sprawdzania równoważności cyklicznej słów, otrzymujemy rozwiązanie zadania o złożoności czasowej $O(n^3)$ (zaimplementowane w plikach `pres1.cpp` oraz `pres2.pas`), które — zgodnie z treścią zadania — uzyskiwało ok. 30 punktów. Z kolei użycie jednej z efektywnych metod o koszcie czasowym $O(m)$ daje rozwiązanie o złożoności $O(n^2)$ (patrz `pres3.cpp`, `pres4.cpp`, `pres5.pas` i `pres6.pas`), uzyskujące ok. 50 punktów.

Rozwiązanie wzorcowe

Kluczowa własność słów

Niech s^R oznacza odwrócenie słowa s, czyli słowo złożone z tych samych liter co s, tylko zapisanych od prawej do lewej. W dalszym opisie będziemy korzystać z następującej, prostej własności słów, której uzasadnienie pozostawiamy Czytelnikowi.

Fakt 1. *Dla dowolnych słów s, u, v, jeśli $s = uv$, to $s^R = v^R u^R$.*

Podzielmy dane słowo t na równej długości połówki, t_1 i t_2. Jeśli t ma nieparzystą długość, możemy usunąć środkową literę słowa, gdyż i tak nie przyda się ona w rozwiązaniu. Kluczowy pomysł polega na rozważeniu słowa będącego *przeplotem* słów t_1 i t_2^R, które otrzymujemy, biorąc po kolei: jedną literę z t_1, jedną z t_2^R, jedną z t_1, jedną z t_2^R itd. Taki przeplot oznaczymy jako $Q(t_1, t_2^R)$.

Przykład 1. Rozważmy słowo $t =$ `ababbabababbaab` z przykładu z treści zadania. W tym przypadku mamy $t_1 =$ `ababbab`, $t_2 =$ `babbaab`, $t_2^R =$ `baabbab`, więc przeplot słów t_1 i t_2^R ma postać:

$$Q(t_1, t_2^R) = \texttt{abbaaabbbbaabb}.$$

Przyjrzyjmy się teraz, jaki związek z naszym zadaniem ma tak zdefiniowany przeplot. Rozważmy najpierw wspomniany na początku opisu prostszy wariant zadania, w którym szukamy najdłuższego prefikso-sufiksu słowa, czyli najdłuższego prefiksu słowa będącego jednocześnie jego sufiksem (przy czym prefiks i sufiks nie mogą na siebie nachodzić). Przypomnijmy, że *palindrom* to słowo czytane tak samo normalnie oraz wspak, np. `anna`.

Obserwacja 1. Słowo t o długości n ma prefikso-sufiks długości m, $m \leqslant n/2$, wtedy i tylko wtedy, gdy prefiks długości $2m$ przeplotu $Q(t_1, t_2^R)$ jest palindromem.

Dowód: Zacznijmy od uzasadnienia implikacji „w prawo". Załóżmy, że słowo t ma prefiks p o długości m, który jest zarazem sufiksem słowa t. Połówki słowa t mają wtedy postać $t_1 = pt_1'$ i $t_2 = t_2'p$, gdzie t_1' i t_2' są słowami o tej samej długości (być może pustymi). Na mocy faktu 1 mamy $t_2^R = p^R(t_2')^R$. Nasz przeplot wygląda zatem tak:

$$Q(t_1, t_2^R) = Q(pt_1', p^R(t_2')^R) = Q(p, p^R)Q(t_1', (t_2')^R).$$

Wystarczy teraz zauważyć, że przeplot $Q(p, p^R)$ o długości $2m$ jest palindromem, gdyż: zaczyna się i kończy pierwszą literą słowa p, drugą i przedostatnią literą w tym przeplocie jest druga litera słowa p itd.

Uzasadnienie implikacji „w lewo" jest bardzo podobne (jeśli się dobrze przyjrzeć, właściwie już je wykonaliśmy). ■

Naszym celem jest teraz uogólnić obserwację 1 na przypadek prefiksufiksu, czyli sytuację, w której szukany prefiks p i sufiks s są cyklicznie równoważne. W takim przypadku zachodzi $p = uv$ i $s = vu$ dla pewnych słów u i v (być może pustych). Mamy zatem $t_1 = uvt_1'$, $t_2 = t_2'vu$ dla pewnych równej długości słów t_1' i t_2', czyli $t_2^R = u^Rv^R(t_2')^R$ i:

$$Q(t_1, t_2^R) = Q(u, u^R)Q(v, v^R)Q(t_1', (t_2')^R).$$

To pozwala nam już sformułować odpowiednią obserwację:

Obserwacja 2. Słowo t o długości n ma prefiksufiks długości m, $m \leqslant n/2$, wtedy i tylko wtedy, gdy prefiks długości $2m$ przeplotu $Q(t_1, t_2^R)$ jest sklejeniem dwóch palindromów parzystych.

Przykład 2. Przyjrzyjmy się przeplotowi z przykładu 1. Prefiks tego przeplotu o długości 12:

$$\texttt{abba} \cdot \texttt{aabbbbaa}$$

jest sklejeniem dwóch palindromów, więc odpowiada prefiksowi `ababba` i sufiksowi `abbaab` słowa t, które są cyklicznie równoważne.

Rozwiązanie wzorcowe opiera się na (jakże pomysłowej) charakteryzacji prefiksufiksów zawartej w obserwacji 2. Dokończenie rozwiązania jest już tylko kwestią sprawności technicznej, pod warunkiem, że mamy do dyspozycji algorytm Manachera.

Wyszukiwanie palindromów

Algorytm Manachera, opisany np. w książce [21] lub w opracowaniu zadania *Antysymetria* z XVII Olimpiady Informatycznej [17], wyznacza w czasie liniowym dla każdej pozycji danego słowa promień palindromu parzystego o środku na tej pozycji. Dokładniej, dla danego słowa $s[1..n]$ wyznaczamy tablicę $R[1..n-1]$, w której $R[i]$ to maksymalne takie $j > 0$, że słowo $s[i-j+1..i+j]$ jest palindromem; jeśli żadne takie j nie istnieje, to przyjmujemy $R[i] = 0$. Promienie $R[i]$ zawierają w sobie strukturę wszystkich palindromów parzystych w słowie, gdyż każdy taki palindrom znajduje się w środku jakiegoś palindromu o maksymalnym promieniu.

Przykład 3. Tablica promieni palindromów parzystych dla przeplotu z przykładu 1 wygląda następująco:

i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
słowo:	a	b	b	a	a	a	b	b	b	b	a	a	b	b
$R[i]$:	0	2	0	1	1	0	1	4	1	0	3	0	1	

Na mocy obserwacji 2, wystarczy teraz pokazać, jak w danym słowie s znaleźć najdłuższy prefiks będący sklejeniem dwóch palindromów parzystych. Przede wszystkim warto zacząć od znalezienia prefiksów słowa będących palindromami parzystymi. Łatwo zauważyć, że prefiks długości $2k$ ma tę własność wtedy i tylko wtedy, gdy $R[k] = k$. Dalsza część algorytmu będzie polegała na rozpatrzeniu każdego kandydata na „drugi palindrom" w prefiksie i dobraniu do niego najlepszego „pierwszego palindromu".

Spróbujmy sformalizować podaną intuicję. Niech $P = \{2k : R[k] = k\}$. Załóżmy, że w poszukiwanym najdłuższym prefiksie drugi palindrom pochodzi z grupy palindromów odpowiadającej promieniowi $R[i]$. Oczywiście, najlepiej byłoby wybrać najdłuższy palindrom z tej grupy, czyli słowo $s[i-j+1..i+j]$ dla $j = R[i]$. Możemy tak zrobić jedynie wtedy, gdy o jeden wcześniejsza pozycja, $i-j$, należy do zbioru P. Jeśli tak nie jest, zamiast tego weźmiemy pierwszą pozycję ze zbioru P następującą po pozycji $i-j$ i przytniemy palindrom $s[i-j+1..i+j]$ tak, aby zaczynał się tuż po tej wybranej pozycji.

Jesteśmy już gotowi na to, by przedstawić nasz algorytm w postaci pseudokodu. Najpierw procedura wyznaczająca tablicę najbliższych pozycji na prawo należących do zbioru P:

```
procedure wyznacz_najbliższe
begin
  najbliższy[0] := 0;
  for k := 1 to n do najbliższy[k] := ∞;
  for k := 1 to n div 2 do
    if R[k] = k then  { 2k ∈ P }
      najbliższy[2k] := 2k;
  for k := n − 1 downto 1 do
    najbliższy[k] := min(najbliższy[k], najbliższy[k + 1]);
end
```

Następnie funkcja znajdująca optymalny prefiks:

```
function oblicz_wynik
begin
  wynik := 0;
  for i := 1 to n − 1 do begin
    pierwszy := najbliższy[i − R[i]];
    if pierwszy ⩽ i then
      wynik := max(wynik, 2i − pierwszy);
  end
  return wynik;
end
```

Rozwiązanie wzorcowe ma liniową złożoność czasową. Jego implementacje można znaleźć w plikach `pre.cpp` i `pre1.pas`.

Rozwiązanie alternatywne

Poniższe rozwiązanie ma taką samą złożoność czasową, a do tego opiera się na nieco bardziej intuicyjnych spostrzeżeniach. Ceną, jaką za to płacimy, jest wykorzystanie haszowania na słowach, która to metoda przy złym (albo pechowym) doborze parametrów może działać niepoprawnie. Haszowanie pozwala w czasie stałym odpowiadać na pytania o to, czy dane dwa podsłowa $s[a..a+k]$ i $s[b..b+k]$ słowa s są równe, i wymaga jedynie $O(n)$ wstępnych obliczeń. Więcej na temat tej metody można przeczytać w drugiej części tej sekcji.

Przypomnijmy, że jeśli prefiks p i sufiks s stanowią szukany prefiksufiks, to $p = uv$ i $s = vu$ dla pewnych słów u i v. To oznacza, że słowo t możemy zapisać jako $t = uvxvu$, przy czym x jest pewnym (być może pustym) słowem. Widzimy, że słowo u jest jednym z prefikso-sufiksów słowa t (nie dłuższym niż połowa słowa t). Dalej, słowo v jest prefikso-sufiksem słowa $t' = vxv$, czyli słowa t z usuniętym prefiksem i sufiksem u. Co więcej, słowo v jest najdłuższym prefikso-sufiksem słowa t' nie dłuższym niż połowa tego słowa.

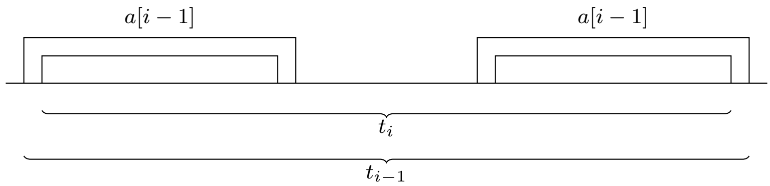
Jak już zauważyliśmy wcześniej, wszystkie prefikso-sufiksy słowa można wyznaczyć w czasie liniowym. W tym celu możemy posłużyć się wspomnianą na wstępie funkcją prefiksową; możemy także wykorzystać haszowanie, które pozwala sprawdzać równość prefiksu i sufiksu słowa w czasie stałym. Aby dokończyć rozwiązanie, wystarczy teraz znaleźć najdłuższy prefikso-sufiks każdego ze słów

$$t_1 = t[1..n], \quad t_2 = t[2..n-1], \quad t_3 = t[3..n-2], \ \ldots$$

o długości nieprzekraczającej połowy długości słowa. Oznaczmy długość tak określonego prefikso-sufiksu słowa t_i jako $a[i]$. Wszystkie komórki tablicy $a[]$ możemy wyznaczyć, dla $i = \left\lfloor \frac{n}{2} \right\rfloor, \ldots, 1$, w czasie $O(n)$, korzystając z następującej obserwacji.

Obserwacja 3. $a[i-1] \leqslant a[i] + 2$.

Dowód: Prefikso-sufiks długości $a[i-1]$ słowa t_{i-1} wyznacza w słowie t_i prefikso-sufiks długości $a[i-1] - 2$, patrz też rys. 1. ■



Rys. 1: $a[i] \geqslant a[i-1] - 2$.

W naszym algorytmie zastosujemy podejście (pozornie) siłowe. Jako kandydatów na $a[i]$ będziemy rozpatrywać kolejne wartości $j = a[i+1] + 2, a[i+1] + 1, \ldots, 0$, a sprawdzanie przerwiemy w chwili, gdy stwierdzimy istnienie prefikso-sufiksu słowa t_i o długości j. Aby przypadkiem nie wygenerować prefikso-sufiksu dłuższego niż połowa rozważanego słowa, najlepiej przed wykonaniem algorytmu wstawić w środek słowa t jakiś symbol niebędący literą. Oto pseudokod ilustrujący opisany algorytm:

```
1: j := n div 2;
2: for i := n div 2 downto 1 do begin
3:   j := j + 2;
4:   while j > 0 and t[i..i + j − 1] ≠ t[n + 1 − (i + j − 1)..n + 1 − i] do
5:     { sprawdzenie warunku pętli używa haszowania }
6:     j := j − 1;
7:   a[i] := j;
8: end
```

Łatwo widać, że powyższa pętla **while** wykonuje łącznie tylko $O(n)$ kroków. Faktycznie, każdy obrót tej pętli powoduje zmniejszenie zmiennej j o 1; zmienna ta ma początkowo wartość $\left\lfloor \frac{n}{2} \right\rfloor$, nie przyjmuje nigdy wartości ujemnych, a jest zwiększana tylko w instrukcji w 3. wierszu pseudokodu, łącznie o nie więcej niż n. Zakładając, że porównywanie podsłów możemy wykonywać w czasie stałym, powyższy algorytm wyznacza tablicę $a[]$ w czasie liniowym.

Długość szukanego prefiksufiksu obliczamy teraz jako maksimum z wartości postaci $i + a[i+1]$ dla i będących długościami prefikso-sufiksów słowa t, $i \geqslant 0$. Cały algorytm działa więc w czasie $O(n)$. Jego implementacje można znaleźć w plikach `preb1.cpp` i `preb2.pas`.

O wykorzystaniu haszowania

Najpopularniejszą metodą haszowania na słowach jest tak zwane haszowanie wielomianowe, wykorzystywane np. w algorytmie wyszukiwania wzorca w tekście metodą Karpa-Rabina. Wartość funkcji haszującej (tzw. *hasz*) dla słowa obliczamy, traktując poszczególne litery słowa jako współczynniki wielomianu z jedną zmienną i wyznaczając resztę z dzielenia przez M wartości tego wielomianu w wybranym punkcie p:

$$H(t) = (t[1] + t[2]p + t[3]p^2 + \ldots + t[n]p^{n-1}) \bmod M.$$

Parametr M należy dobrać tak, aby wartości haszy nie powodowały przepełnienia wbudowanych typów całkowitych.

Aby móc obliczać hasze dla podsłów słowa t, w ramach obliczeń wstępnych wyznaczamy hasze wszystkich sufiksów słowa t za pomocą schematu Hornera:

$$h[n+1] = 0, \quad h[i] = (t[i] + p \cdot h[i+1]) \bmod M \quad \text{dla} \;\; i = n, n-1, \ldots, 1.$$

Będziemy także potrzebować tablicy pierwszych n potęg liczby p modulo M. Teraz hasze dla podsłów słowa t wyznaczamy już w czasie stałym ze wzoru:

$$H(t[i\,..\,j]) = (h[i] - p^{j-i+1}h[j+1]) \bmod M.$$

Jeśli dwa podsłowa słowa t są równe, to na pewno wartości ich haszy są równe. W drugą stronę nie musi to być prawdą: ponieważ nawet bardzo długie słowa reprezentujemy za pomocą hasza będącego liczbą całkowitą z zakresu od 0 do $M-1$, możemy natrafić na *kolizję*, czyli na sytuację, gdy dwa różne słowa mają takie same hasze. Gdy hasze dwóch słów są równe, mamy zatem do wyboru dwie opcje: albo zakładamy, że mamy szczęście i rzeczywiście porównywane słowa są równe, albo dla większej pewności wykonujemy dodatkowe sprawdzenia, na przykład porównujemy litery na kilku losowo wybranych pozycjach słów (względnie na wszystkich pozycjach, jeśli jesteśmy gotowi poświęcić czas działania na rzecz stuprocentowej pewności) lub też sprawdzamy równość haszy przy kilku innych doborach parametrów p oraz M. Dobrą praktyką jest wybieranie jako p oraz M możliwie dużych liczb pierwszych. Ponadto, liczba p powinna być większa niż rozmiar alfabetu (w przeciwnym razie bardzo łatwo o kolizję już dla słów dwuliterowych).

Czasem próbuje się stosować haszowanie z pominięciem parametru M: po prostu wszystkie obliczenia wykonuje się w wybranym typie całkowitym, zazwyczaj 32- albo 64-bitowym. Parametr M jest wówczas tak naprawdę potęgą dwójki o odpowiednim wykładniku, a liczba p powinna być nieparzysta, gdyż w przeciwnym razie wpływ na wartość hasza miałoby jedynie kilkadziesiąt pierwszych liter słowa. Podana metoda pozwala pisać bardzo szybkie programy (unikamy wykonywania kosztownych operacji modulo), okazuje się jednak podatna na kolizje nawet wśród dość typowych rodzin słów.

Przykładem takiej rodziny są słowa Thuego-Morse'a, które pojawiły się np. w zadaniu *Ciągi bez zająknięć* z X Olimpiady Informatycznej [10]. Dla wygody zamiast liter `a` i `b` będziemy tu korzystać z cyfr 0 i 1. Niech $\bar{s}$ oznacza słowo powstałe poprzez negację wszystkich bitów w słowie s. Wówczas ciąg słów Thuego-Morse'a można zdefiniować rekurencyjnie:

$$s_0 \;=\; 0; \qquad s_i \;=\; s_{i-1}\bar{s}_{i-1} \;\; \text{dla} \;\; i > 0.$$

Oto kilka pierwszych wyrazów tego ciągu; zauważmy, że słowo s_i ma długość 2^i:

$$s_0 = 0,\ s_1 = 01,\ s_2 = 0110,\ s_3 = 01101001,\ s_4 = 0110100110010110,\ \ldots$$

Przyjrzyjmy się teraz, dlaczego słowa Thuego-Morse'a są podatne na występowanie kolizji przy haszowaniu z parametrem $M = 2^k$. Oznaczmy przez $W(s)$ hasz słowa s z pominięciem parametru M:

$$W(s) = s[1] + s[2]p + s[3]p^2 + \ldots + s[n]p^{n-1}.$$

Wówczas prawdziwy jest następujący fakt.

Fakt 2. *Dla dowolnego $n \geqslant 0$ oraz $2 \nmid p$ zachodzi $2^{n(n+1)/2} \mid W(\bar{s}_n) - W(s_n)$.*

Dowód: Na mocy definicji rekurencyjnej ($s_n = s_{n-1}\bar{s}_{n-1}$ i $\bar{s}_n = \bar{s}_{n-1}s_{n-1}$) dla $n \geqslant 1$ mamy:

$$\begin{aligned} W(\bar{s}_n) - W(s_n) &= W(\bar{s}_{n-1}) + p^{2^{n-1}}W(s_{n-1}) - W(s_{n-1}) - p^{2^{n-1}}W(\bar{s}_{n-1}) \\ &= W(\bar{s}_{n-1})(1 - p^{2^{n-1}}) - W(s_{n-1})(1 - p^{2^{n-1}}) \\ &= (1 - p^{2^{n-1}})(W(\bar{s}_{n-1}) - W(s_{n-1})) \end{aligned}$$

Stąd łatwo wykazać przez indukcję, że

$$W(\bar{s}_n) - W(s_n) = (1 - p^{2^{n-1}})(1 - p^{2^{n-2}}) \ldots (1 - p).$$

Aby zakończyć dowód, wystarczy udowodnić, że

$$2^i \mid 1 - p^{2^{i-1}} \quad \text{dla każdego } i \geqslant 1. \tag{1}$$

Dowód tej zależności przeprowadzimy przez indukcję. Dla $i = 1$ korzystamy z faktu, że p jest liczbą nieparzystą. Krok indukcyjny (dla $i > 1$) wynika ze wzoru skróconego mnożenia:

$$1 - p^{2^{i-1}} = (1 - p^{2^{i-2}})(1 + p^{2^{i-2}}).$$

Faktycznie, pierwszy z powyższych czynników dzieli się, na mocy założenia indukcyjnego, przez 2^{i-1}, natomiast drugi jest parzysty. To kończy dowód zależności (1). ■

Fakt 2 pozwala nam stwierdzić, że słowa s_n i $\bar{s}_n$ z pewnością spowodują wystąpienie kolizji, jeśli tylko wartość wyrażenia $n(n+1)/2$ będzie co najmniej taka, jak liczba bitów wykorzystywanego typu całkowitego. Tak więc nawet jeśli użyjemy typu 64-bitowego, już słowa s_{11} i $\bar{s}_{11}$ o długości 2048 będą miały tę samą wartość funkcji haszującej, i to niezależnie od wyboru parametru p!

Podsumowując: stosowanie metody haszowania na słowach często prowadzi do efektywnych i prostych algorytmów, należy jednak pamiętać, że otrzymane w ten sposób rozwiązania mogą czasem dawać niepoprawne wyniki, i zachować pewną ostrożność w doborze parametrów tej metody.

Rozwiązania błędne

Spostrzeżenie, że naszym celem jest znalezienie najdłuższego słowa uv spełniającego $t = uvxvu$, prowadzi do kilku możliwych rozwiązań błędnych opartych na podejściu zachłannym. Przykładowo, możemy wybrać u jako najdłuższy prefikso-sufiks słowa t nie dłuższy niż połowa słowa, a następnie wybrać v jako najdłuższy prefikso-sufiks słowa t obustronnie skróconego o u (implementacja tego podejścia: `preb4.cpp`). Takie rozwiązania nie uzyskiwały na zawodach żadnych punktów. Można też jako kandydatów na słowo u rozpatrzyć pewną liczbę najdłuższych prefikso-sufiksów słowa t (implementacje: `preb5.cpp` i `preb6.cpp`). W zależności od szczegółów implementacyjnych takie rozwiązania uzyskiwały co najwyżej 70 punktów.

Aby zobaczyć, że tego typu rozwiązania nie są poprawne, rozważmy słowo t postaci:

`aaaaaaaabaaaaaabaaaaaa`.

Wówczas szukany najdłuższy prefiksufiks pokrywa dokładnie całe słowo:

`aaaaaaaabaa` · `aaaabaaaaaa`.

Pierwsze z podanych rozwiązań zachłannych w pierwszym kroku odetnie prefikso-sufiks `aaaaaa`, otrzymując obustronnie skrócone słowo:

`aabaaaaaab`.

W drugim kroku to rozwiązanie zdoła już tylko odciąć prefikso-sufiks `aab`, a zatem wynik przez nie uzyskany nie będzie optymalny. Zauważmy, że wybór w pierwszym kroku prefikso-sufiksu `aaaaa` (czyli o jeden krótszego) również nie prowadzi do znalezienia optymalnego prefiksufiksu, a odpowiedni jest dopiero wybór prefikso-sufiksu `aaaa`. Ogólnie, odpowiednio zwiększając długości podsłów złożonych z samych liter `a` (przy czym drugie podsłowo musi mieć taką samą długość jak trzecie), można uzyskać sytuację, w której rozpatrzenie nawet k najdłuższych prefikso-sufiksów nie prowadzi do optymalnego rozwiązania; tu $k > 0$ może być dowolnie dużą stałą.

Dodajmy jeszcze, że na podstawie ostatniego rozwiązania zachłannego można skonstruować rozwiązanie poprawne, ale o złożoności $O(n^2)$, w którym jako u próbujemy wybierać kolejno wszystkie prefikso-sufiksy słowa t (patrz plik `pres7.cpp`). Takie rozwiązanie uzyskiwało na zawodach ok. 70 punktów.

Testy

Rozwiązania zawodników były sprawdzane za pomocą 10 zestawów danych testowych, z których każdy zawierał od 5 do 6 testów. W teście *1f* występuje słowo jednoliterowe, a w teście *10f* — słowo zawierające milion takich samych liter. Oto charakterystyka pozostałych grup testów:

- grupa a: słowa Thuego-Morse'a;
- grupa b: słowa Fibonacciego (patrz np. zadanie *Słowa* z XVI Olimpiady Informatycznej [16]);

- grupa c: słowa o krótkim powtarzającym się okresie (rzędu $O(\sqrt{n})$);
- grupa d: słowa o dużej liczbie prefikso-sufiksów, sprawiające trudność zachłannym rozwiązaniom błędnym;
- grupa e: słowa składające się z samych liter `a` i strategicznie położonych liter `b`, sprawiające trudność rozwiązaniom o złożoności $\Theta(n^3)$.

W poniższej tabeli n oznacza długość słowa t, a m — wynik, czyli długość optymalnego prefiksufiksu.

Nazwa	n	m
pre1a.in	100	16
pre1b.in	1	0
pre1c.in	100	50
pre1d.in	100	45
pre1e.in	100	8
pre1f.in	1	0
pre2a.in	301	128
pre2b.in	2	0
pre2c.in	301	148
pre2d.in	301	149
pre2e.in	301	8
pre3a.in	500	160
pre3b.in	3	1
pre3c.in	500	242
pre3d.in	500	243
pre3e.in	500	8
pre4a.in	4 000	1 280
pre4b.in	8	3
pre4c.in	4 000	1 984
pre4d.in	4 000	1 368
pre4e.in	4 000	8
pre5a.in	5 000	2 048
pre5b.in	89	42
pre5c.in	5 000	2 480
pre5d.in	5 000	2 082
pre5e.in	5 000	8

Nazwa	n	m
pre6a.in	100 000	16 384
pre6b.in	144	68
pre6c.in	100 000	49 928
pre6d.in	100 000	10 002
pre6e.in	100 000	8
pre7a.in	200 000	32 768
pre7b.in	233	110
pre7c.in	200 000	99 872
pre7d.in	200 000	76 600
pre7e.in	200 000	8
pre8a.in	500 000	163 840
pre8b.in	1 597	754
pre8c.in	500 000	249 722
pre8d.in	500 000	232 103
pre8e.in	500 000	8
pre9a.in	1 000 000	327 680
pre9b.in	46 368	21 892
pre9c.in	1 000 000	500 000
pre9d.in	1 000 000	441 598
pre9e.in	1 000 000	8
pre10a.in	1 000 000	327 680
pre10b.in	317 811	150 050
pre10c.in	1 000 000	500 000
pre10d.in	1 000 000	458 390
pre10e.in	1 000 000	8
pre10f.in	1 000 000	500 000

XXIV Międzynarodowa Olimpiada Informatyczna,

Sirmione – Montichiari, Włochy 2012