Jakub Radoszewski, Wojciech Rytter — Treść zadania, Opracowanie
Mirosław Michalski — Program

Dostępna pamięć: 32 MB. *OI, Etap II, dzień próbny, 9.02.2010*

Antysymetria

Bajtazar studiuje różne napisy złożone z zer i jedynek. Niech x będzie takim napisem, przez x^R będziemy oznaczać odwrócony (czyli „czytany wspak”) napis x, a przez $\overline{x}$ będziemy oznaczać napis powstały z x przez zamianę wszystkich zer na jedynki, a jedynek na zera.

Bajtazara interesuje **antysymetria**, *natomiast niezbyt lubi wszystko co symetryczne. Antysymetria nie jest tylko prostym zaprzeczeniem symetrii. Powiemy, że (niepusty) napis x jest* **antysymetryczny**, *jeżeli dla każdej pozycji i w x, i-ty znak od końca jest różny od i-tego znaku, licząc od początku. W szczególności, niepusty napis x złożony z zer i jedynek jest antysymetryczny wtedy i tylko wtedy, gdy $x = \overline{x}^R$. Na przykład, napisy* `00001111` *i* `010101` *są antysymetryczne, natomiast* `1001` *nie jest.*

W zadanym napisie złożonym z zer i jedynek chcielibyśmy wyznaczyć liczbę jego spójnych (tj. jednokawałkowych) niepustych fragmentów, które są antysymetryczne. Jeżeli różne fragmenty odpowiadają takim samym słowom, to i tak należy je policzyć wielokrotnie.

Wejście

Pierwszy wiersz standardowego wejścia zawiera liczbę n ($1 \leqslant n \leqslant 500\,000$), oznaczającą długość napisu. Drugi wiersz zawiera napis złożony z liter 0 i/lub 1 o długości n. Napis ten nie zawiera żadnych odstępów.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą liczbę spójnych fragmentów wczytanego napisu, które są antysymetryczne.

Przykład

Dla danych wejściowych:

```
8
11001011
```

poprawnym wynikiem jest:

```
7
```

Antysymetryczne fragmenty to: `01` *(pojawia się dwukrotnie),* `10` *(także dwukrotnie),* `0101`, `1100` *oraz* `001011`.

Rozwiązanie

Analiza problemu

Zastanówmy się, czym tak naprawdę jest napis antysymetryczny. No dobrze, a czym jest napis *symetryczny*? Napis symetryczny spełnia warunek $x = x^R$, czyli jest niczym innym jak *palindromem* — napisem czytanym tak samo normalnie i wspak. Na palindrom parzystej długości (tzw. palindrom parzysty) możemy także spojrzeć jak na napis postaci $u \cdot u^R = uu^R$, czyli sklejenie pierwszej połówki z drugą połówką będącą jej odwróceniem. Palindromy nieparzyste różnią się od parzystych tylko tym, że mają w środku jeszcze jedną, dodatkową cyfrę c, czyli są postaci ucu^R.

Przez analogię, napisy antysymetryczne będziemy nazywali *antypalindromami*. Binarny antypalindrom parzysty również możemy podzielić na połówki, z których druga jest tym razem odwróceniem i negacją (zamiana zer na jedynki i odwrotnie) pierwszej, czyli jest to napis postaci $u\overline{u}^R$. Faktycznie, wynika to wprost z warunku, że i-ta cyfra od początku musi być inna od i-tej cyfry od końca, co dla napisów złożonych wyłącznie z cyfr $\{0, 1\}$ oznacza, że cyfry te są swoimi negacjami. Przykładowo, $\mathtt{11010100} = \mathtt{1101}\,\overline{\mathtt{1011}}$. Z kolei antypalindrom nieparzystej długości $2k + 1$ powinien być postaci $uc\overline{u}^R$ dla pewnej cyfry c. Powinien, ale tak być nie może: z definicji wynika, że $(k + 1)$-sza cyfra od początku tego napisu, czyli c, powinna być różna od $(k + 1)$-szej cyfry od końca, czyli także c — a wszakże c od c różne być nie może. To pokazuje, że antypalindromy nieparzystej długości nie istnieją!

Podsłowem napisu (słowa) zerojedynkowego s nazwiemy dowolny jego spójny fragment. Jeżeli $s = s_1 s_2 \ldots s_n$, to przez $s[i\,..\,j]$ oznaczymy podsłowo $s_i s_{i+1} \ldots s_j$. Widzimy, że aby rozwiązać zadanie, musimy wyznaczyć liczbę podsłów wyjściowego słowa s będących antypalindromami parzystymi. A jak rozwiązywalibyśmy podobne zadanie, w którym mielibyśmy zliczyć podsłowa będące *palindromami* parzystymi? Bardzo krótka odpowiedź na to pytanie brzmi: obliczamy promienie palindromów parzystych za pomocą algorytmu Manachera i wypisujemy ich sumę, co można wykonać w czasie liniowym względem długości słowa s. W przypadku antypalindromów zrobimy generalnie to samo, tylko użyjemy odpowiednio zmodyfikowanego algorytmu Manachera... Ale po kolei.

Rozwiązanie wzorcowe

O algorytmie Manachera można poczytać przede wszystkim w książce [19], a także na różnych, łatwych do wyszukania stronach internetowych. Pojawił się on także w rozwiązaniu zadania *Osie symetrii* z XIV Olimpiady Informatycznej [14]. W tej sekcji opiszemy ten algorytm zmodyfikowany do wyznaczania antypalindromów parzystych (działający całkiem analogicznie jak oryginalna wersja z palindromami).

Promienie antypalindromów

Dane jest n-literowe słowo zerojedynkowe $s = s_1 s_2 \ldots s_n$. Powiemy, że parzystej długości słowo v jest podsłowem słowa s o *środku* na pozycji i, jeżeli

$$v \;=\; s[i-j+1\,..\,i+j] \;=\; s[i-j+1\,..\,i] \cdot s[i+1\,..\,i+j].$$

Jak wygląda zbiór wszystkich antypalindromów parzystych o środku na ustalonej pozycji i? Zauważmy, że jeżeli $s[i-j+1\,..\,i+j]$ jest takim antypalindromem, to wszystkie inne podsłowa o środku na i-tej pozycji i długości mniejszej niż $2j$ są antypalindromami parzystymi. Wynika to z faktu, że po usunięciu pierwszej i ostatniej cyfry antypalindrom zmienia się także w antypalindrom, tyle że krótszy (proste uzasadnienie tej obserwacji pozostawiamy Czytelnikowi).

Możemy teraz zdefiniować *promień* antypalindromu na i-tej pozycji słowa (oznaczenie: $R[i]$) jako największe takie $j \geqslant 0$, że podsłowo $s[i-j+1\,..\,i+j]$ jest antypalindromem, patrz także tabelka na rys. 1. Dzięki wcześniej poczynionej obserwacji opisującej zbiór antypalindromów o środku na danej pozycji wiemy, że tak zdefiniowany promień ma następującą, intuicyjną własność: wszystkie podsłowa o środku na pozycji i i długości nieprzekraczającej $2R[i]$ są antypalindromami, a wszystkie dłuższe — już nie. To oznacza, że mamy łącznie $R[i]$ antypalindromów o środku na danej pozycji i, a zatem poszukiwana liczba wszystkich podsłów słowa s będących antypalindromami jest równa sumie wszystkich promieni antypalindromów, czyli

$$wynik = R[1] + R[2] + \ldots + R[n-1]. \tag{1}$$

Wniosek z tego prosty: aby rozwiązać zadanie, wystarczy obliczyć promienie antypalindromów na wszystkich pozycjach słowa.

i	1	2	3	4	5	6	7	8	9	10
s_i	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**0**
$R[i]$	1	0	1	2	4	2	1	0	0	

Rys. 1: Promienie antypalindromów na poszczególnych pozycjach słowa 0110101000. Przykładowo, promień na pozycji 5 jest równy 4, ponieważ najdłuższym antypalindromem o środku na pozycji 5 jest 11010100.

Obliczanie promieni antypalindromów

Promienie antypalindromów będziemy obliczać kolejno od lewej do prawej. Załóżmy zatem, że obliczyliśmy już wartości $R[1], R[2], \ldots, R[i]$. W tej sekcji zastanowimy się, czy korzystając z nich, możemy efektywnie obliczać promienie na kolejnych pozycjach słowa. W tym celu pracowicie wyprowadzimy pewien techniczny lemat, który stanowi podstawę działania algorytmu Manachera (Czytelników pragnących od razu poznać sam algorytm odsyłamy do kolejnej sekcji).

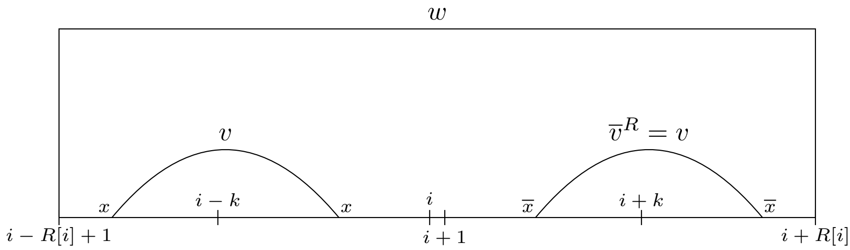
Rozważmy pozycję $i+k$ dla $k < R[i]$, czyli mieszczącą się wewnątrz największego antypalindromu o środku na pozycji i. Okazuje się, że przy obliczaniu $R[i+k]$ możemy skorzystać z wartości $R[i-k]$. Intuicyjnie, dowolny antypalindrom v o środku na pozycji $i-k$, mieszczący się wewnątrz antypalindromu

$$w = s[i - R[i] + 1\,..\,i + R[i]],$$

możemy odbić (anty)symetrycznie względem środka antypalindromu w, otrzymując $\overline{v}^R$, czyli dokładnie antypalindrom v, o środku na pozycji $i+k$.

Spróbujmy sformalizować podaną intuicję — ilustracja poniższego rozumowania znajduje się na rys. 2. Załóżmy na początek, że $R[i-k] < R[i]-k$, czyli że największy antypalindrom

$$v = s[(i-k) - R[i-k] + 1\,..\,(i-k) + R[i-k]]$$



Rys. 2: Obliczanie promienia $R[i+k]$ w przypadku $R[i-k] < R[i]-k$. Wówczas $R[i+k] = R[i-k]$. Na tym rysunku wystąpienia antypalindromu v akurat nie nachodzą na siebie, ale równie dobrze mogłyby.

o środku na pozycji $i - k$ mieści się ściśle wewnątrz antypalindromu w, a zatem, w szczególności, v nie dotyka brzegu w. Ze względu na to, iż w jest antypalindromem, mamy, że dla dowolnego $a = 1, 2, \ldots, R[i]$:

$$s_{i+a} = \overline{s_{i-a+1}}.$$

Stąd w szczególności:

$$\begin{aligned} s_{i+k+R[i-k]} &= \overline{s_{i-k-R[i-k]+1}}, \\ s_{i+k+R[i-k]-1} &= \overline{s_{i-k-R[i-k]+2}}, \\ &\vdots \\ s_{i+k-R[i-k]+1} &= \overline{s_{i-k+R[i-k]}}, \end{aligned}$$

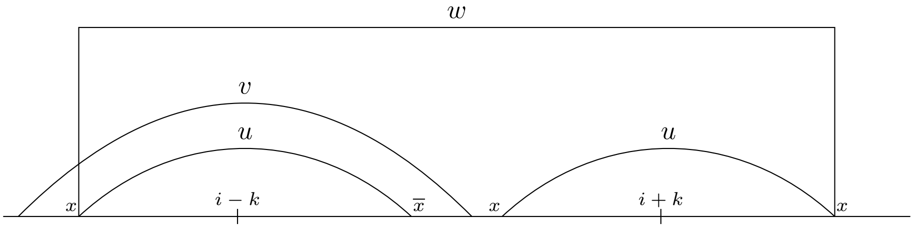
co pokazuje, że

$$s[(i+k) - R[i-k] + 1 \mathinner{\ldotp\ldotp} (i+k) + R[i-k]] \;=\; \overline{v}^R \;=\; v.$$

Stąd $R[i+k] \geqslant R[i-k]$.

Pokażemy, że nie może zachodzić $R[i+k] > R[i-k]$, czyli że w rzeczy samej $R[i+k] = R[i-k]$. W tym celu rozważmy cyfry sąsiadujące z lewej i z prawej z wystąpieniem antypalindromu v o środku na pozycji $i-k$, to znaczy $s_{i-k-R[i-k]}$ oraz $s_{i-k+R[i-k]+1}$. Na mocy definicji $R[i-k]$ wiemy, że te cyfry muszą być takie same (na rys. 2 obie oznaczone są przez x). Zauważmy, że obie te cyfry mieszczą się wewnątrz antypalindromu w. Nie powinno to już stanowić dla Czytelnika niespodzianki, że ich odbicia względem środka antypalindromu w, to znaczy odpowiednio $s_{i+k+R[i-k]+1}$ oraz $s_{i+k-R[i-k]}$, są dokładnie cyframi sąsiadującymi z wystąpieniem antypalindromu v na pozycji $i+k$. Te cyfry są negacjami dwóch wcześniej rozważanych, a zatem także są równe ($\overline{x}$ na rysunku), co pokazuje, że tego wystąpienia antypalindromu v także nie można rozszerzyć. Ostatecznie mamy, że jeśli $R[i-k] < R[i]-k$, to po prostu $R[i+k] = R[i-k]$, co przy wyznaczaniu promieni antypalindromów może być nadzwyczaj pomocne.

Dotychczas poszło nam całkiem nieźle, możemy spróbować przeanalizować kolejne przypadki. Przypadek, w którym $R[i-k] = R[i] - k$ (v stanowi *prefiks*, czyli początkowy fragment, słowa w), zostawimy sobie na deser, wcześniej rozpatrzymy przypadek,



Rys. 3: Obliczanie promienia $R[i+k]$ w przypadku $R[i-k] > R[i]-k$. Wówczas $R[i+k] = R[i] - k$.

w którym $R[i-k] > R[i]-k$, czyli antypalindrom v wystaje poza antypalindrom w — patrz rys. 3. Nie musimy już chyba formalnie dowodzić, że wówczas środkowy fragment u antypalindromu v mieszczący się wewnątrz w, będący antypalindromem o długości $R[i]-k$, możemy odbić antysymetrycznie wewnątrz w, otrzymując antypalindrom o środku na pozycji $i+k$, co oznacza, że $R[i+k] \geqslant R[i]-k$. Pokażemy, że tę nierówność możemy zamienić w istocie w równość.

W tym celu musimy ponownie przyjrzeć się cyfrom sąsiadującym bezpośrednio z wystąpieniem u na pozycji $i-k$: są to $s_{i-R[i]}$ i jeszcze jedna cyfra s_j mieszcząca się w ramach w (dokładny wzór na indeks j celowo pomijamy). Przypomnijmy, że antypalindrom u możemy rozszerzyć do dłuższego antypalindromu v, co pokazuje, że rozważane cyfry muszą być różne: $s_{i-R[i]} = \overline{s_j}$ (odpowiednio x i $\overline{x}$ na rysunku). A jakie są cyfry sąsiadujące bezpośrednio z wystąpieniem u o środku na pozycji $i+k$? Jedna z nich to na pewno negacja cyfry s_j. Druga to cyfra $s_{i+R[i]+1}$, następująca bezpośrednio za wystąpieniem w w ramach s. Skoro tak, to czy możemy cokolwiek o niej powiedzieć? Okazuje się, że tak: ponieważ antypalindromu w nie można rozszerzyć, więc $s_{i+R[i]+1} = s_{i-R[i]}$. Ostatecznie, cyfry sąsiadujące z wystąpieniem u o środku na pozycji $i+k$ muszą być równe:

$$s_{i+R[i]+1} \;=\; s_{i-R[i]} \;=\; \overline{s_j},$$

co pokazuje, że antypalindromu u o środku na pozycji $i+k$ nie można już rozszerzyć. Uzasadniliśmy zatem, że jeżeli $R[i-k] > R[i]-k$, to $R[i+k] = R[i]-k$.

Uważny Czytelnik mógł dostrzec w ostatnim rozumowaniu pewną lukę: zastanawiamy się nad cyframi $s_{i-R[i]}$ oraz $s_{i+R[i]+1}$, choć właściwie nie jest nigdzie powiedziane, że te cyfry w ogóle istnieją! Nie jest to jednak większy problem — $s_{i-R[i]}$ na pewno istnieje, gdyż mieści się w v. Jeśli natomiast $s_{i+R[i]+1}$ jest już poza naszym słowem, to tym bardziej nierówność $R[i+k] \geqslant R[i]-k$ nie może być ostra — wszak antypalindrom musi mieścić się w słowie.

Po tym uzupełnieniu możemy przejść do ostatniego nierozważonego przypadku, w którym $R[i-k] = R[i]-k$ (v jest prefiksem w). Wiemy już, że wówczas antypalindrom v występuje także na pozycji $i+k$, co pokazuje, że $R[i+k] \geqslant R[i-k]$. Łatwo przekonać się, że tym razem argument dotyczący sąsiadujących cyfr nie zadziała, czyli nie jesteśmy w stanie powiedzieć, czy ta nierówność jest równością, czy też nie. Trudno, pozostańmy z tym, co mamy.

Algorytm Manachera

W wyniku powyższego, całkiem obszernego rozumowania uzyskaliśmy pewne przydatne zależności, które możemy podsumować w skondensowany sposób za pomocą następującego lematu (w naszym rozumowaniu nie rozważaliśmy przypadku $k = R[i]$, lecz łatwo sprawdzić, że wówczas także teza lematu zachodzi).

Lemat 1. Niech $i \in \{1, 2, \ldots, n-1\}$ oraz niech $k \in \{1, 2, \ldots, R[i]\}$. Wówczas:

a) jeżeli $R[i-k] \neq R[i] - k$, to $R[i+k] = \min(R[i-k],\ R[i]-k)$,

b) a jeżeli $R[i-k] = R[i] - k$, to $R[i+k] \geqslant R[i-k]$.

Co prawda wyprowadzenie Lematu 1 kosztowało nas trochę wysiłku, jednak na jego podstawie możemy od razu zaproponować algorytm wyznaczania promieni antypalindromów. W algorytmie będziemy próbowali korzystać z części a) Lematu, a jeżeli kiedykolwiek będziemy zmuszeni do użycia części b) lub wyjdziemy poza obręb największego antypalindromu o środku na danej pozycji, to dla tej kłopotliwej pozycji wyznaczymy promień antypalindromu nachalnie (cyferka po cyferce). Po tym znów będziemy próbowali używać Lematu 1, korzystając z promienia obliczonego dla nowej pozycji itd.

W implementacji dodamy na początku i końcu słowa s tzw. strażników: symbole `$` i `#`, niewystępujące poza tym w słowie s, których negacje $\overline{\texttt{\$}}$ oraz $\overline{\texttt{\#}}$ będą jeszcze innymi symbolami. W ten sposób poradzimy sobie z przypadkami brzegowymi — nigdy nie będziemy przedłużać antypalindromów poza słowo.

Algorytm Manachera
$s[0] :=$ '`$`'; $s[n+1] :=$ '`#`';
$i := 1$; $j := 0$;
while $i \leqslant n-1$ **do begin**
  while $s[i-j] = \overline{s[i+j+1]}$ **do** $j := j+1$;
  $R[i] := j$;
  $k := 1$;
  while $(k \leqslant R[i])$ **and** $(R[i-k] \neq R[i]-k)$ **do begin**
    $R[i+k] := \min(R[i-k],\ R[i]-k)$;
    $k := k+1$;
  end
  $j := \max(j-k, 0)$; $i := i+k$;
end

Dokładniejsze sprawdzenie, że powyższa implementacja działa zgodnie z wcześniejszym opisem słownym, pozostawiamy Czytelnikowi. Zastanówmy się natomiast, dlaczego ten algorytm jest efektywny czasowo — pokażemy, że wbrew pewnym pozorom, ma on złożoność czasową $O(n)$.

Owe pozory to zagnieżdżenie pętli **while**. Chcemy pokazać, że łącznie pętle te wykonają co najwyżej $O(n)$ obrotów. Standardowe podejście w takim przypadku polega na znalezieniu jednego parametru (potencjalnie pewnej funkcji od zmiennych występujących w algorytmie), który wzrasta w każdym obrocie wewnętrznych pętli i ma przedział wartości rozmiaru $O(n)$. Tutaj też tak można, ale wygodniej będzie dokonać interpretacji algorytmu, biorąc pod uwagę to, co on tak naprawdę oblicza.

Otóż cały czas utrzymujemy przedział $[i-j+1\,..\,i+j]$ oznaczający (właśnie obliczany bądź już obliczony) największy antypalindrom o środku na aktualnej pozycji i. Pierwsza wewnętrzna pętla **while** (wiersz 5 pseudokodu) przesuwa prawy koniec tego przedziału o jeden, za to środka przedziału (zmienna i) nie rusza. Z kolei druga z rozważanych, wewnętrznych pętli w każdym obrocie przesuwa środek przedziału o jeden w prawo ($i+1, i+2, \ldots$), nie ruszając prawego końca. To oznacza, że każda z tych pętli z osobna wykonuje co najwyżej n obrotów, gdyż zarówno środek, jak i prawy koniec aktualnego przedziału przyjmują wartości ze zbioru $\{1, 2, \ldots, n+1\}$.

Wszystkich pozostałych operacji jest w algorytmie wykonywanych $O(n)$. To oznacza, że rzeczywiście jego złożoność czasowa jest liniowa względem n.

Podsumowanie

Przypomnijmy: w rozwiązaniu wzorcowym najpierw wyznaczamy promienie antypalindromów parzystych za pomocą zmodyfikowanego algorytmu Manachera, a następnie obliczamy wynik ze wzoru (1). Każdy z tych kroków ma złożoność czasową $O(n)$.

Rozwiązanie wzorcowe zostało zaimplementowane w plikach `ant.cpp`, `ant1.pas` i `ant2.cpp`.

Inne rozwiązania

Rozwiązanie wzorcowe jest bardzo eleganckie i zarazem efektywne, jednakże bez znajomości algorytmu Manachera wydaje się dosyć trudne do wymyślenia. Okazuje się, że istnieje cała gama innych rozwiązań tego zadania, a wśród nich i takie, które pozwalały zdobyć na zawodach maksymalną punktację. Poza najmniej efektywnymi, które pozostawiamy do rozważenia Czytelnikowi — najprostsze rozwiązanie o złożoności czasowej $O(n^3)$ oraz trochę lepsze rozwiązanie o złożoności czasowej i pamięciowej $O(n^2)$, oparte na metodzie programowania dynamicznego (porównaj opis rozwiązania zadania *Palindromy* z II Olimpiady Informatycznej [2]) — wszystkie one były oparte na wyznaczaniu promieni antypalindromów i obliczały wynik ze wzoru (1).

W najprostszym z tych rozwiązań obliczamy żądane promienie antypalindromów, każdorazowo poszerzając podsłowo o środku na danej pozycji i o kolejne cyfry, dopóki jest ono antypalindromem. W ten sposób otrzymujemy rozwiązanie o złożoności czasowej $O(n^2)$ i pamięciowej $O(n)$, którego pseudokod zamieszczamy poniżej. Dodajmy jeszcze, że koszt czasowy tego rozwiązania zależy tak naprawdę od sumy promieni antypalindromów, czyli lepszym oszacowaniem na złożoność czasową jest $O(n+m)$, przy czym m to maksymalny wynik. Niestety, pesymistycznie $m = \Theta(n^2)$, o czym łatwo przekonać się, rozważając słowo $010101\ldots01$.

1: **Algorytm** $O(n+m)$
2: $s[0] :=$ '\$'; $s[n+1] :=$ '#';
3: **for** $i := 1$ **to** $n-1$ **do begin**
4:     $R[i] := 0$;
5:     **while** $s_{i-R[i]} = \overline{s_{i+R[i]+1}}$ **do** $R[i] := R[i] + 1$;

```
end
```

Implementację rozwiązania korzystającego z tej metody można znaleźć w pliku `ants3.cpp`. Na zawodach takie rozwiązania uzyskiwały około 50 punktów.

Do wyznaczania promieni $R[i]$ możemy także użyć wyszukiwania binarnego. W tym celu potrzebna nam będzie efektywna funkcja sprawdzająca, czy dane podsłowo $s[i\,..\,j]$ słowa s jest antypalindromem:

```
Algorytm z wyszukiwaniem binarnym
for i := 1 to n − 1 do begin
  a := 0;  b := min(i, n − i);
  while a < b do begin
    c := (a + b) div 2;
    if czy_antypalindrom(s[i − c + 1 .. i + c]) then a := c
    else b := c − 1;
  end
  R[i] := a;
end
```

Na pierwszy rzut oka takie sprawdzenie nie wydaje się proste do wykonania w efektywny sposób, tj. szybciej niż liniowo względem długości podsłowa. Okazuje się jednak, że można tak zmodyfikować wyjściowe słowo s, aby sprawdzanie, czy dane podsłowo s jest antypalindromem, sprowadzało się do sprawdzenia *równości* pewnych dwóch podsłów zmodyfikowanego słowa s. Tym zmodyfikowanym słowem będzie $t = s\overline{s}^R$.

Obserwacja 1. Podsłowo $s[i\,..\,j]$ jest antypalindromem wtedy i tylko wtedy, gdy

$$t[i\,..\,j] \;=\; t[(2n+1-j)\,..\,(2n+1-i)].$$

Dowód: Oznaczmy $w = s[i\,..\,j]$. Wystarczy zauważyć, że:

$$t[i\,..\,j] \;=\; w, \quad t[(2n+1-j)\,..\,(2n+1-i)] \;=\; \overline{w}^R.$$

Kto nie wierzy, niech sprawdzi (co najmniej na przykładzie). ■

Sprowadziliśmy zatem wyjściowy problem do problemu sprawdzania równości zadanych podsłów słowa t. A na to jest już wiele gotowych algorytmów.

Jednym z nich jest *słownik podsłów bazowych*, który przypisuje całkowite dodatnie identyfikatory podsłowom słowa t o długościach będących potęgami dwójki, co następnie pozwala wyznaczać jednoznaczne identyfikatory dowolnych podsłów t. Więcej o tej metodzie można przeczytać w książce [19] (algorytm Karpa-Millera-Rosenberga, KMR) lub w opisie rozwiązania zadania *Powtórzenia* z VII Olimpiady Informatycznej [7]. Złożoność czasowa wstępnych obliczeń to w tej metodzie $O(n\log^2 n)$ lub $O(n\log n)$, w zależności od jakości implementacji, przy złożoności pamięciowej rzędu $O(n\log n)$. Po tych obliczeniach sprawdzanie równości podsłów jest już wykonywane w czasie stałym.

Rozwiązania używające tej metody mają łączną złożoność czasową $O(n\log^2 n)$ lub $O(n\log n)$, a pamięciową $O(n\log n)$. Różne implementacje można znaleźć w plikach `ant[s4-s7].[cpp|pas]`. Ich głównym mankamentem jest duża złożoność pamięciowa (jak na limit pamięciowy 32 MB), przez co zdobywały na zawodach około 50 punktów.

Lubiący odrobinę ryzyka mogą do sprawdzania równości podsłów słowa t użyć metody *haszowania*. W tej metodzie cyfry występujące w danym podsłowie traktujemy jako kolejne współczynniki wielomianu i przydzielamy temu podsłowu identyfikator będący wartością tego wielomianu w jakimś punkcie całkowitym, $x = q$, wziętą modulo pewna liczba całkowita dodatnia (pierwsza) p. Niestety, tak przydzielone identyfikatory nie muszą być różne dla różnych podsłów (w przypadku zajścia takiej równości mówimy o tzw. *kolizji*). Dobierając odpowiednio dużą wartość p i losując wartość q, możemy jednak mieć nadzieję, że dla żadnych podsłów, które musimy akurat porównać, nie napotkamy na kolizję. I rzeczywiście, prawdopodobieństwo kolizji jest małe, ale mimo wszystko nie da się jej wykluczyć.

Więcej o tej metodzie można przeczytać chociażby w książkach [19] i [21] (algorytm Karpa-Rabina, KR), a także — w praktycznym ujęciu — w artykule *Drobne oszustwo* w numerze 11/2009 czasopisma *Delta*[1]. Można ją zaimplementować bardzo efektywnie: złożoność czasowa i pamięciowa wstępnych obliczeń jest rzędu $O(n)$, a na zapytania o równość podsłów możemy odpowiadać w czasie stałym.

Rozwiązania używające tej metody mają — ze względu na wyszukiwanie binarne — złożoność czasową $O(n \log n)$, jednak złożoność pamięciowa $O(n)$ jest lepsza niż w przypadku słownika podsłów bazowych. Różne implementacje (przykłady: `antb1.cpp`, `antb2.cpp`) uzyskiwały na zawodach punktację zbliżoną do maksymalnej bądź maksymalną, w zależności od szczęścia — a, jak mówi znane przysłowie, szczęście sprzyja lepszym, czyli w tym przypadku tym, którzy rozsądnie dobierają parametry używane w haszowaniu.

Rozwiązania błędne

Wśród rozwiązań błędnych możemy wyróżnić głównie niepoprawne implementacje opisanych powyżej rozwiązań. Przede wszystkim należało zwrócić uwagę na to, że wynik mógł nie mieścić się w typie całkowitym 32-bitowym, ale wymagać użycia zmiennych 64-bitowych — rozwiązanie podobne do wzorcowego, lecz nieuwzględniające tego faktu, uzyskiwało na zawodach około 70 punktów.

Testy

Do oceny rozwiązań tego zadania użyto 14 grup testów. Poniżej zamieszczona jest tabela zawierająca podstawowe statystyki poszczególnych testów (n to długość wyjściowego słowa, a m to wynik, czyli liczba podsłów słowa będących antypalindromami).

Nazwa	n	m
ant1a.in	1	0
ant1b.in	1	0
ant1c.in	2	1
ant1d.in	10	11

Nazwa	n	m
ant1e.in	2	0
ant2a.in	45	0
ant2b.in	50	8
ant2c.in	50	50

[1] Artykuł dostępny także na stronie internetowej czasopisma: `http://www.mimuw.edu.pl/delta/`

Nazwa	n	m
ant2d.in	100	146
ant3a.in	61	900
ant3b.in	422	657
ant3c.in	600	832
ant3d.in	800	152 122
ant4a.in	1 002	1 555
ant4b.in	1 030	995
ant4c.in	1 198	4 303
ant4d.in	1 601	361 141
ant5a.in	6 656	235 444
ant5b.in	6 231	247 829
ant5c.in	7 053	3 177 100
ant5d.in	7 668	13 319
ant6a.in	20 480	2 755 601
ant6b.in	18 334	1 073 433
ant6c.in	20 000	100 000 000
ant6d.in	38 100	39 782
ant7a.in	37 008	17 179 122
ant7b.in	100 000	149 818
ant7c.in	106 822	53 411
ant7d.in	499 993	0
ant8a.in	158 304	2 088 399 136
ant8b.in	200 000	687 393
ant8c.in	200 000	940 530
ant8d.in	221 260	332 150

Nazwa	n	m
ant9a.in	400 004	602 414
ant9b.in	407 701	1 783 070 527
ant9c.in	409 600	839 278 590
ant9d.in	400 000	597 956
ant10a.in	500 000	750 062
ant10b.in	500 000	499 417
ant10c.in	500 000	2 084 116 658
ant10d.in	500 000	1 746 350
ant11a.in	500 000	499 915
ant11b.in	500 000	5 208 749 996
ant11c.in	500 000	752 483
ant11d.in	500 000	600 178
ant12a.in	500 000	8 929 285 709
ant12b.in	500 000	10 416 916 665
ant12c.in	500 000	15 625 125 000
ant12d.in	500 000	52 900 520 440
ant13a.in	500 000	20 000 251 084
ant13b.in	500 000	746 684
ant13c.in	500 000	32 034 926 466
ant13d.in	500 000	1 606 456
ant14a.in	499 998	20 833 333 333
ant14b.in	491 520	304 094 082
ant14c.in	500 000	500 006
ant14d.in	500 000	62 500 000 000