Funkcja π Knutha

Bartosz Chomiński

1 Wstęp

Rozwiążemy w tej notatce pewne rudymentarne zadanie wokół algorytmów tekstowych:

Dane jest słowo S oraz wzorzec T (czyli też słowo). Wyznacz wszystkie wystąpienia wzorca T jako spójne podsłowo w słowie S.

2 Pierwszy algorytm

Rozpoczniemy od dość narzucającego się podejścia, które ma szanse zadziałać w mało złośliwych przypadkach (np. gdy słowo zostało wylosowane). Ten sposób sprawdza każdą potencjalną pozycję słowa T w słowie S i wprost zaczyna porównywać kolejne litery wzorca T oraz słowa S. W pseudokodzie to podejście może wyglądać tak:

```
vector< int > findOccurences(string s, string t)
{
  vector< int > ans;
  for (int i = 0; i < s.length(); i++){
    bool match = true;
    for (int j = 0; j < t.length(); j++){
      if (i+j >= s.length() || s[i+j] != t[j]){
        match = false;
        break;
      }
    }
    if (match){
      ans.push_back(i);
    }
  }
  return ans;
}
```

To podejście działa, ale jest za wolne – jego pesymistyczna złożoność czasowa to $\mathcal{O}(|S| \cdot |T|)$ i można ją uzyskać chociażby w przypadku $S = \underbrace{\texttt{aa...a}}_{n}$, $T = \underbrace{\texttt{aa...ab}}_{m}$, $n \geqslant m$. Wtedy algorytm opisany powyższym kodem wykona dokładnie $nm - \frac{(m-1)m}{2}$ porównań par liter i zwróci pustą listę wystąpień T w S.

Przeanalizujmy czemu ten algorytm jest tak wolny. Zauważmy, że tak naprawdę ten algorytm składa się z wielu niezależnych faz – każda pozycja, od której mogłoby się zaczynać wystąpienie słowa T w słowie S, jest sprawdzana osobno bez przepływu jakichkolwiek informacji między kolejnymi sprawdzeniami. W przykładzie z poprzedniego akapitu ($S = \texttt{aa...a}$, $T = \texttt{aa...ab}$) to oznacza, że nawet jeśli w pierwszym sprawdzeniu napracujemy się i pokażemy, że S i T zgadzają się na pierwszych $m-1$ pozycjach, to i tak w drugiej fazie znowu całą tę pracę wykonamy od nowa (zamiast od razu porównywać litery z końca słowa T z dalszymi literami S).

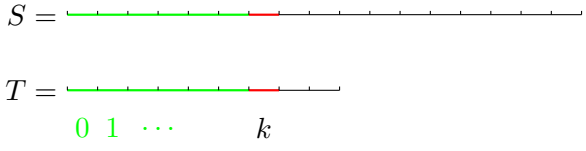
$i = 0$	$i = 1$
$S =$ aaaaaaaaaaaaaaa	$S =$ aaaaaaaaaaaaaaa
$T =$ aaaaaaab	$T =$ aaaaaaab
Po pierwszej fazie wiemy, że zielone podsłowa są równe.	Niebieskie podsłowa są równe i moglibyśmy to wywnioskować z pierwszej fazy, ale naiwny algorytm sprawdza to „na piechotę”.

Ten problem daje nam duże pole do optymalizacji, nad którą teraz się zastanowimy.

3 Optymalizacja

Niech S i T będą dowolnymi słowami, dla których chcemy rozwiązać zadanie ze wstępu.

Załóżmy, że S i T zgadzają się na pierwszych k pozycjach w pierwszej fazie, jak na obrazku poniżej.



Skoro wiemy, że porównując od pierwszej litery słowa S, nie dopasujemy słowa T w pełni (bo na pozycji k te słowa się różnią), to powinniśmy szukać kolejnych potencjalnych pozycji startowych słowa T w słowie S.

Najlepszą kolejną pozycją, od której opłaca nam się sprawdzać wystąpienie wzorca T jest pozycja, od której zaczyna się najdłuższy niepełny sufiks zielonej części słowa S (por. schemat powyżej), który jest jednocześnie prefiksem słowa T.

Uzasadnijmy dlaczego inne pozycje nie są lepsze. Jeśli zaczęlibyśmy z pozycji, od której sufiks zielonej części słowa S nie byłby prefiksem słowa T, to na pewno nie byłoby wystąpienia całego wzorca T – skoro już prefiks T się nie zgadza z dalszym ciągiem słowa S, to tym bardziej całe słowo T się nie zgodzi. Jeśli zaczęlibyśmy z pozycji, od której zaczyna się jakiś inny krótszy sufiks zielonej części słowa S, który jest jednocześnie prefiksem T, to potencjalnie ominęlibyśmy pełne wystąpienie T w S, które zaczynało się wcześniej.

Rozjaśnijmy to podejście przykładem: niech $S =$ `rowerowerowy`, $T =$ `rowerowy`. Najpierw, w pierwszej fazie, porównujemy kolejne litery obu słów naiwnie:

$$S = \texttt{rowerowerowy}$$
$$T = \texttt{rowerowy}$$

Najdłuższy niepełny sufiks zielonej części słowa S, który jest jednocześnie prefiksem słowa T to `row`.

$$S = \texttt{rowerowerowy}$$
$$T = \texttt{rowerowy}$$

Wobec tego następną pozycją, od której będziemy się starali dopasować słowo T, będzie pozycja zaczynająca ten właśnie sufiks zielonej części słowa S.

$$S = \texttt{rowerowerowy}$$
$$T = \texttt{\ \ \ \ \ rowerowy}$$

Dalej porównujemy kolejne litery naiwnie i wobec braku niezgodności idziemy tak aż do końca słowa S.

$$S = \texttt{rowerowerowy}$$
$$T = \texttt{\ \ \ \ \ rowerowy}$$

4 Prefikso-sufiksy

4.1 Wstęp

Zajmijmy się teraz sposobem na szybkie wyznaczanie długości najdłuższego niepełnego sufiksu zielonej części słowa S, który jest jednocześnie prefiksem słowa T. Zacznijmy od definicji:

Definicja. *Prefikso-sufiks* słowa W to słowo P, które: jest prefiksem W, jest sufiksem W oraz $P \neq W$.

Przykładowo, w słowie `abbabbab` wszystkie prefikso-sufiksy to słowo puste, `ab` oraz `abbab`, natomiast słowo `klops` ma wyłącznie pusty prefikso-sufiks.

Udowodnijmy kilka prostych lematów związanych z prefikso-sufiksami:

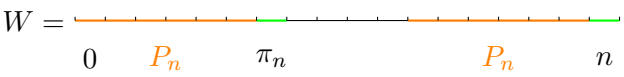
Lemat 1. Prefikso-sufiks prefikso-sufiksu słowa W jest prefikso-sufiksem słowa W.

Dowód. Proste ćwiczenie. Może przydać się poniższy rysunek:

$W =$

Lemat 2. Dane jest słowo W. Niech P_i oznacza najdłuższy prefikso-sufiks słowa $W[0]W[1]\dots W[i-1]$ i niech π_i oznacza długość słowa P_i. Wtedy jeśli $W[\pi_n] = W[n]$, to $P_{n+1} = P_n W[n]$.

Dowód. Słowo $P_n W[n]$ jest prefikso-sufiksem słowa $W[0]W[1]\dots W[n]$, co można łatwo zauważyć korzystając z poniższego rysunku.



Słowo $P_n W[n]$ jest wśród prefikso-sufiksów słowa $W[0]W[1]\dots W[n]$ najdłuższe, ponieważ ma długość $\pi_n + 1$. Gdyby istniał prefikso-sufiks długości co najmniej $\pi_n + 2$, to przez ucięcie ostatniej litery uzyskalibyśmy prefikso-sufiks słowa $W[0]W[1]\dots W[n-1]$ dłuższy niż π_n, co jest sprzeczne z definicją π_n.

Lemat 3. Najdłuższy prefikso-sufiks najdłuższego prefikso-sufiksu słowa W jest drugim najdłuższym prefikso-sufiksem słowa W.

Dowód. Proste ćwiczenie. Najpierw pokazujemy, że drugi najdłuższy prefikso-sufiks W jest prefikso-sufiksem najdłuższego prefikso-sufiksu W,

a potem pokazujemy, że musi to być najdłuższy prefikso-sufiks tegoż najdłuższego prefikso-sufiksu W.

4.2 Algorytm

Zauważmy, że do efektywnego wyznaczania potrzebnych nam niebieskich słów (por. przykład w poprzedniej sekcji), potrzebujemy tylko długości prefikso-sufiksów początkowych fragmentów słowa T.

Będziemy wyznaczać długości najdłuższych prefikso-sufiksów kolejnych początkowych fragmentów słowa T iteracyjnie. Dla jednoliterowego fragmentu odpowiedzią zawsze będzie 0, bo żadne słowo długości 1 nie ma prefikso-sufiksu. Dalsze długości będziemy wyznaczać na podstawie poprzednich wyników:

Załóżmy, że mamy już długości najdłuższych prefikso-sufiksów dla n początkowych fragmentów słowa T i są one oznaczone jako $\pi_1, \pi_2, \ldots, \pi_n$. Chcemy obliczyć π_{n+1}. Jeśli $W[\pi_n] = W[n]$, to zgodnie z lematem 2 mamy $\pi_{n+1} = \pi_n + 1$. W przeciwnym razie próbujemy przedłużyć drugi najdłuższy prefikso-sufiks – jego długość to π_{π_n}. Jeśli i to się nie uda, to próbujemy przedłużyć trzeci najdłuższy prefikso-sufiks o długości $\pi_{\pi_{\pi_n}}$. Kontynuujemy do momentu osiągnięcia pustego najdłuższego prefikso-sufiksu – wtedy wpisujemy jako odpowiedź 0 lub 1, zależnie od prawdziwości równości $W[0] = W[n]$.

4.3 Przykład

Niech $T =$ `abacababaca`. Wtedy kolejne wartości π wyniosą $0, 0, 1, 0, 1, 2, 3, 2, 3, 4, 5$.

4.4 Implementacja

```
vector< int > pi(string t)
{
  vector< int > pi(t.length() + 1, 0);
  pi[0] = -1;
  for (int i = 1; i < t.length(); i++){
    int j = pi[i];
    while (j >= 0 && t[i] != t[j]){
      j = pi[j];
    }
    if (t[i] == t[j]){
      j++;
    }
    pi[i+1] = j;
  }
  return pi;
}
```

4.5 Złożoność czasowa

Oszacujemy złożoność czasową powyższej implementacji, zwracając uwagę na zmiany wartości zmiennej j. Po pierwsze zauważmy, że zmienna j na końcu i-tego przebiegu pętli ma tę samą wartość co na początku $(i+1)$-szego przebiegu pętli, zatem możemy myśleć o tej zmiennej jako o zmiennej globalnej. W każdym przebiegu pętli `for` zmienna j albo się zmniejsza, albo zwiększa o dokładnie 1. Stąd wynika, że liczba wszystkich zmniejszeń j w całym wywołaniu algorytmu nie może być większa, niż liczba zwiększeń j o 1. Oznacza to, że łączna liczba wszystkich przebiegów pętli `while` w całym algorytmie nie przekroczy długości słowa T.

5 Zobacz też

- `https://eduinf.waw.pl/inf/alg/001_search/0049.php`
- `https://zpe.gov.pl/pdf/PAWg1Xbyk`
- `https://cp-algorithms.com/string/prefix-function.html`