Algorytmy tekstowe na przykładzie KMP

Postawowe pojęcia

Niech Σ będzie dowolnym skończonym niepustym zbiorem symboli. Zbiór Σ nazywamy *alfabetem*. Dowolny ciąg symboli ze zbioru Σ nazywamy *słowem (nad alfabetem Σ)*.

Przykład. $abba$ jest słowem nad alfabetem $\Sigma = \{a, b\}$.

Przez ϵ oznaczamy *słowo puste*, przez Σ^* oznaczamy (nieskończony) zbiór wszystkich słów nad alfabetem Σ. Przez $|w|$ będziemy oznaczać *długość słowa* w, np. $|abba| = 4$, $|\epsilon| = 0$. *Konkatenacja słów u i v* jest to słowo uv powstałe po dopisaniu słowa v za słowem u. Słowo w jest *prefiksem* słowa u, jeśli $u = wv$ dla pewnego słowa v. Analogicznie, słowo w jest *sufiksem* słowa u, jeśli $u = vw$ dla pewnego słowa v.

Przykłady.

- te jest prefiksem słowa $teksty$; $ksty$ jest jego sufiksem.
- ϵ jest prefiksem i sufiksem każdego słowa.
- Każde słowo jest swoim własnym prefiksem i sufiksem.

Jeśli w jest prefiksem u, ale $w \neq u$ to mówimy, że w jest *właściwym prefiksem* u. Analogiczne definiujemy *właściwy sufiks*.

Problem wyszukiwania wzorca

Dane: słowa p („wzorzec") i t („tekst") nad pewnym alfabetem Σ, $|p| \leq |t|$.

Problem: znaleźć wszystkie wystąpienia wzorca p w tekście t.

Przykład: wzorzec xy występuje w tekście $xxyyxyxy$ na pozycjach 2, 5 i 7.

Przyjmiemy oznaczenia $|p| = m$ i $|t| = n$. Alg. 1 przedstawia algorytm „naiwny" rozwiązujący problem wyszukiwania wzorca. Algorytm dopasowuje wzorzec we wszystkich możliwych miejscach.

Algorithm 1 „naiwny"

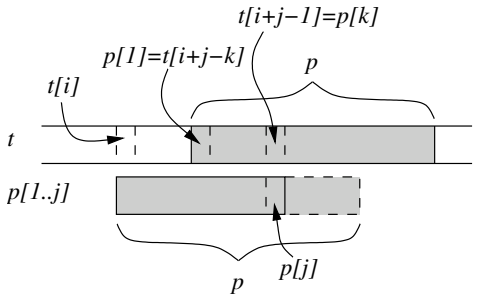
```
1: i ← 1
2: while i ≤ n − m + 1 do
3:     j ← 0
4:     while p[j + 1] = t[i + j] do
5:         j ← j + 1
6:     if j = m then
7:         Write(i)
8:     i ← i + 1
```

Uwaga! Do alfabetu Σ dodajemy dwa *nowe*, specjalne symbole, powiedzmy \$ i @. Przed uruchomieniem algorytmu 1 \$ dopisujemy na końcu t, natomiast @ na końcu p. Pełnią one rolę „strażników" w wewnętrznej pętli **while**.

Złożonośc czasowa: oczywiście $O((n-m+1)m) = O(nm)$.

Poprawianie alg. „naiwnego"

Załóżmy, że dla $j+1$ dostaliśmy niezgodność. Wiemy, że $p[1..j]$ pasuje do $t[i..i+j-1]$. Czy warto sprawdzać od nowa, czy wzorzec występuje na następnej pozycji $(i+1)$? A może możemy „przeskoczyć" kilka symboli? Zobaczmy co się dzieje, gdy wzorzec występuje na jednej z pozycji $i+1, \ldots i+j-1$ (Rys. 1).



Rysunek 1: Wzorzec p występuje na jednej z pozycji $i+1, \ldots i+j-1$.

Widzimy, że w takiej sytuacji prefiks $p[1..j]$ jest równocześnie jego (właściwym) sufiksem! tzn. $p[1..k] = p[j-k+1..j]$. Jeśli to jest pierwsze miejsce, w którym występuje taka sytuacja, możemy bezpiecznie przesunąć i do $i+j-k$. Wystarczy znać k. Innymi słowy, dla $j > 0$ chcemy więc znać liczbę:

$$\Pi[j] = \text{długość największego właściwego sufiksu } p[1..j], \text{ który jest równocześnie prefiksem } p$$

albo inaczej:

$$\Pi[j] = \max\{0 \le k < j : p[1..k] \text{ jest sufiksem } p[1..j]\}$$

Wtedy możemy bezpiecznie dopasowywać wzorzec w miejscu $i+j-\Pi[j]$. Zauważmy, że ponieważ $\Pi[j] < j$ więc $i+j-\Pi[j] > i$. Dla $j = 0$ przyjmijmy $\Pi[j] = 0$. Wtedy nie ma sensu dopasowywać wzorca w miejscu $i+j-\Pi[j] = i$, bo miejsce dopasowania nie zmieni się. W takiej sytuacji przesuwamy się tylko o jedną pozycję ($i \leftarrow i+1$), ale niewiele tracimy, bo skoro $j = 0$ to dla poprzedniej wartości i wykonano tylko jedno porównanie. A więc ostatecznie przesuwamy i na pozycję $i + \max(1, j - \Pi[j])$. Zauważmy też, że nie musimy już sprawdzać pierwszych $\Pi[j]$ liter wzorca!

Przykład. Tablica Π dla wzorca $p = abacabacaa$.

i	1	2	3	4	5	6	7	8	9	10
$p[i]$	a	b	a	c	a	b	a	c	a	a
$\Pi[i]$	0	0	1	0	1	2	3	4	5	1

Załóżmy, że tablicę Π mamy już obliczoną. Wykorzystując powyższe rozważania możemy poprawić algorytm „naiwny" tak, żeby spożytkować informację zawartą w tablicy Π. Poniższy algorytm wymyślili panowie Knuth i Pratt oraz (niezależnie) Moris w 1977 roku.

Algorithm 2 Algorytm KMP (Knutha-Morisa-Pratta)

```
1: i ← 1
2: j ← 0
3: while i ≤ n − m + 1 do
4:    {p[1..Π[j]] = p[j − Π[j] + 1..j] = t[i..i + Π[j] − 1]}
5:    j ← Π[j]
6:    while p[j + 1] = t[i + j] do
7:       j ← j + 1
8:    if j = m then
9:       Write(i)
10:   i ← i + max(1, j − Π[j])
```

Twierdzenie. *Niech p i t będą dowolnymi słowami, $|p| \leq t$, $|p| = m$, $|t| = n$. Algorytm KMP, który na wejściu dostaje słowa p i t oraz tablicę Π działa w czasie $O(n)$.*

Uzasadnienie. Przyjmujemy, że porównanie dowolnych dwóch liter wykonuje się w czasie stałym. Wtedy porównanie dwóch liter możemy traktować jako operację dominującą.

Porównania liter wzorca z literami tekstu możemy podzielić na dwie kategorie: porówania pozytywne (równość zachodzi) i negatywne (litery są różne). Ponieważ na końcu każego obrotu zewnętrznej pętli **while** wartość zmiennej i rośnie, a w każdym obrocie tej pętli tylko raz występuje porównanie negatywne, podczas wykonania całego algorytmu całkowita liczba porównań negatywnych nie przekracza $n - m + 1$.

Równie łatwo policzyć pozytywne porównania. Przyjrzyjmy się jednemu obrotowi zewnętrznej pętli **while**. Załóżmy, że po wykonaniu wewnętrznej pętli **while** $j > 0$. Niech i' oznacza nową wartość i (po wykonaniu $i \leftarrow i + \max(1, j - \Pi[j])$), natomiast j' nową wartość j (po wykonaniu $j \leftarrow \Pi[j]$ na początku kolejnego obrotu pętli). Wtedy $i' = i + j - \Pi[j]$ oraz $j' = \Pi[j]$ czyli $i' + j' = i + j$. Jeśli natomiast po wykonaniu wewnętrznej pętli **while** $j = 0$, to $i' = i + 1$, $j' = \Pi[j] = 0$ czyli $i' + j' = i + 1 > i + j$. Widzimy, że podczas działania algorytmu wartość wyrażenia $i + j$ nie maleje. Tymczasem po każdym pozytywnym porównaniu wartość $i + j$ rośnie. A więc żadna litera tekstu nie będzie więcej niż raz pozytywnie porównana. Stąd wszystkich pozytywnych porównań jest nie więcej niż liter tekstu czyli n. To kończy uzasadnienie. □

Jak obliczyć tablicę Π?

Przypomnijmy:

$$\Pi[j] = \text{długość największego właściwego sufiksu } p[1..j], \text{ który jest równocześnie prefiksem } p$$

A gdybyśmy chcieli znać długości *wszystkich* właściwych sufiksów $p[1..j]$, które są prefiksami p? Niech $p = abababab$ i $j = 8$. Najdłuższym takim sufiksem jest $r_1 = ababab$. Jego długość to $\Pi[8] = 6$. Zastanówmy się nad następnym pod względem długości takim sufiksem (oznaczmy go przez r_2). Ponieważ r_1 i r_2 są sufiksami p i r_2 jest krótszy niż r_1, a więc r_2 jest właściwym sufiksem r_1. Co więcej, ponieważ r_2 był następny pod względem długości, jest on najdłuższym właściwym sufiksem r_1, który jest równocześnie prefiksem p. A więc $|r_2| = \Pi[\Pi[8]]$. Analogicznie postępujemy przy obliczaniu długości kolejnych sufiksów (patrz Rys 2).

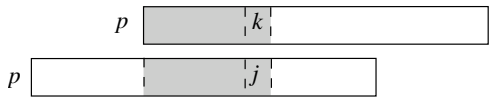
$p[1..8]$	$ab\|ab\ ab\ ab$	$\Pi[8] = 6$
$p[1..6]$	$ab\|ab\ ab$	$\Pi[6] = 4$
$p[1..4]$	$ab\|ab$	$\Pi[4] = 2$
$p[1..2]$	$ab\|$	$\Pi[2] = 0$
$p[0..0]$	ϵ	$\Pi[0] = 0$

Rysunek 2: Długości wszystkich właściwych sufiksów $p[1..8]$, które są równocześnie prefiksami p można znaleźć w tablicy Π.

Wniosek 1. *Z tablicy* Π *można odczytać długości* wszystkich *właściwych sufiksów* $p[1..j]$*, które są równocześnie prefiksami* p*: są to liczby*

$$\Pi[j], \Pi[\Pi[j]], \Pi^3[j], \ldots.$$

Jak obliczyć tablicę Π? Skorzystamy z programowania dynamicznego.



Rysunek 3: $p[1..k]$ jest właściwym sufiksem i równocześnie prefiksem $p[1..j]$.

Jeśli $p[1..k]$ jest właściwym sufiksem i równocześnie prefiksem $p[1..j]$ (patrz Rys. 3) to:

- $p[1..k-1]$ jest właściwym sufiksem i prefiksem $p[1..j-1]$,
- $p[k] = p[j]$.

A więc żeby obliczyć $\Pi[j]$ trzeba znaleźć najdłuższy sufiks $p[1..j-1]$, będący róznocześnie prefiksem $p[1..j-1]$, który *da się przedłużyć*, czyli *najmniejsze* s takie że:

$$p[\Pi^s[j-1]+1] = p[j].$$

Jeśli nie ma takiego s (tzn. żadnego, nawet pustego prefiksu nie da się przedłużyć), to

$$\Pi[j] = 0.$$

Algorithm 3 Obliczanie tablicy Π

```
1:  Π[0] ← 0
2:  Π[1] ← 0
3:  for j ← 2 to m do
4:      {obliczamy Π[j]}
5:      x ← Π[j − 1]
6:      while (x > 0)and (p[x + 1] ≠ p[j]) do
7:          x ← Π[x] {próbujemy któtszy sufiks}
8:      if p[x + 1] = p[j] then
9:          Π[j] ← x + 1
10:     else
11:         Π[j] ← 0
```

Twierdzenie. *Złożoność czasowa algorytmu 3 dla słowa p długości m wynosi $\Theta(m)$.*

Algorithm 4 Obliczanie tablicy Π – wersja II

```
1:  Π[0] ← 0
2:  Π[1] ← 0
3:  x ← 0
4:  for j ← 2 to m do
5:      {obliczamy Π[j]}
6:      while (x > 0)and (p[x + 1] ≠ p[j]) do
7:          x ← Π[x] {próbujemy któtszy sufiks}
8:      if p[x + 1] = p[j] then
9:          x ← x + 1
10:         Π[j] ← x
11:     else
12:         Π[j] ← 0
```

Uzasadnienie. Spójrzmy na algorytm 4. Powstał on na podstawie algorytmu 4 po dodaniu linii 3, 9 i usunięciu linii 5. Zauważmy, że działa on dokładnie tak samo jak pierwotny algorytm,

tylko wartości zmiennej x zmieniają się w innych momentach. Łatwiej nam będzie analizować algorytm 4. Znowu przyjmujemy, że porównanie dowolnych dwóch liter wykonuje się w czasie stałym. Żeby oszacować złożoność algorytmu wystarczy obliczyć liczbę wykonań instrukcji $x \leftarrow \Pi[x]$.

Zauważmy, że wartości zmiennej x zmieniają się tylko w liniach 7 i 9. W linii 7 zawsze maleją, natomiast w linii 9 zawsze rosną. Podczas całego algorytmu x rośnie o $m-1$ jednostek, a więc maleć może co najwyżej $m-1$ razy (maleje co najmniej o 1). To kończy uzasadnienie. □

Wniosek 2. *Złożoność czasowa algorytmu KMP jest rzędu* $\Theta(n+m)$*, złożoność pamięciowa algorytmu KMP jest rzędu* $\Theta(m)$.