# Algorytmy tekstowe: pojęcia

$\Sigma$ - skończony niepusty zbiór symboli zwany *alfabetem*. Dowolny ciąg symboli ze zbioru $\Sigma$ nazywamy *słowem (nad alfabetem $\Sigma$)*.

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

Przez $\epsilon$ oznaczamy *słowo puste*.

Przez $\Sigma^*$ oznaczamy (nieskończony) zbiór wszystkich słów nad alfabetem $\Sigma$.

Przez $|w|$ będziemy oznaczać *długość słowa $w$*, np. $|abba| = 4$, $|\epsilon| = 0$.

# Algorytmy tekstowe: pojęcia cd.

*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 $v$.

Słowo $w$ jest *sufiksem* słowa $u$, jeśli $u = vw$ dla pewnego słowa $v$.

**Przykłady**.

- $te$ jest prefiksem słowa $teksty$; $ksty$ jest jego sufiksem.
- $\epsilon$ 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 $\Sigma$.

$|p| = m$, $|t| = n$, $|p| \leq |t|$.

**Problem:** znaleźć wszystkie wystąpienia wzorca $p$ w tekście $t$.

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

# Algorytm „naiwny”

Próbujemy dopasować wzorzec we wszystkich miejscach:

```
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
```

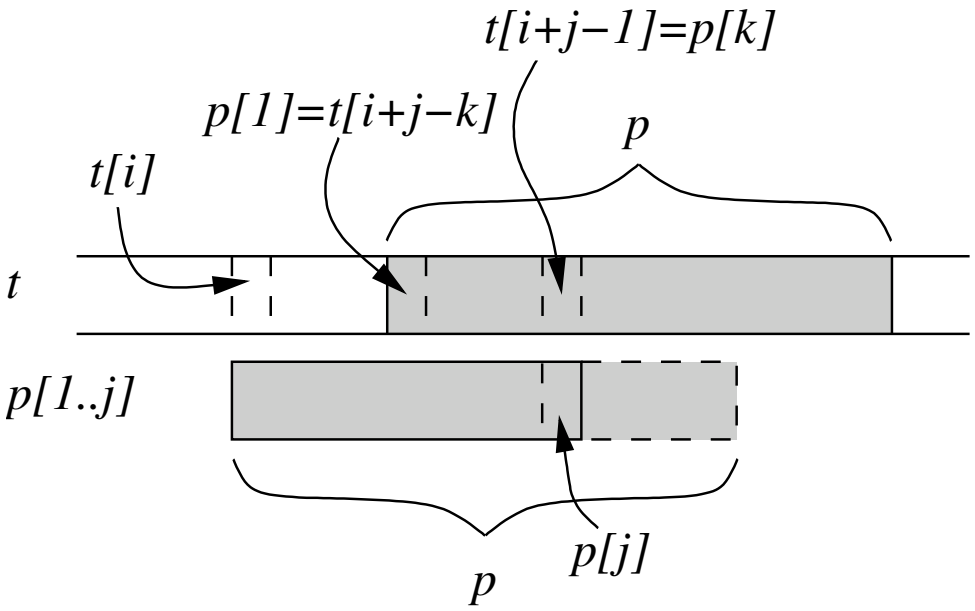
Uwaga! Do alfabetu $\Sigma$ dodajemy dwa *nowe*, specjalne symbole, powiedzmy \$ i @. Przed uruchomieniem algorytmu \$ dopisujemy na końcu $t$, natomiast @ na końcu $p$. Pełnią one rolę „strażników” w wewnętrznej pętli **while**.

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

# Poprawianie alg. „naiwnego”

Załóżmy, że dla $j+1$ dostaliśmy niezgodność. Wiemy, że $p[1..j]$ pasuje do $t[i..i+j-1]$. Ile symboli możemy „przeskoczyć”?

Co się dzieje, gdy wzorzec występuje na jednej z pozycji $i+1, \ldots i+j-1$?




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 $k$.

# Poprawianie alg. „naiwnego”, cd.

Dla $j > 0$ chcemy więc znać liczbę:

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

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

Wtedy możemy bezpiecznie dopasowywać wzorzec w miejscu $i + j - \Pi[j]$.

Dla $j = 0$ przyjmijmy $\Pi[j] = 0$. Ojej! Ale wtedy $i + j - \Pi[j] = i$ i stoimy w miejscu. 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 $\Pi$ 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 |

# Algorytm KMP

1: $i \leftarrow 1$
2: $j \leftarrow 0$
3: **while** $i \leq n - m + 1$ **do**
4: $\quad \{p[1..\Pi[j]] = p[j - \Pi[j] + 1..j] = t[i..i + \Pi[j] - 1]\}$
5: $\quad j \leftarrow \Pi[j]$
6: $\quad$ **while** $p[j + 1] = t[i + j]$ **do**
7: $\quad\quad j \leftarrow j + 1$
8: $\quad$ **if** $j = m$ **then**
9: $\quad\quad$ Write($i$)
10: $\quad i \leftarrow i + \max(1, j - \Pi[j])$

Powyższy algorytm wymyślili panowie Knuth i Pratt oraz (niezależnie) Moris w 1977 roku.

# Własności tablicy $\Pi$

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

A gdybyśmy chcieli znać długości *wszystkich* właściwych sufiksów $p[1..j]$, które są prefiksami $p$? Niech $p = abababab$.

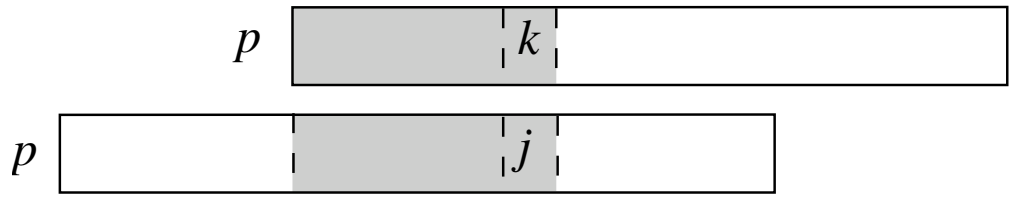
| | | |
|---|---|---|
| $p[1..8]$ | $ab\|ab\ ab\ ab$ | $\Pi[8] = 6$ |
| $p[1..6]$ | $ab\|ab\ ab$ | $\Pi[6] = 4$ |
| $p[1..4]$ | $ab\|ab$ | $\Pi[4] = 2$ |
| $p[1..2]$ | $ab\|$ | $\Pi[2] = 0$ |
| $p[0..0]$ | $\epsilon$ | $\Pi[0] = 0$ |

**Wniosek.** Z tablicy $\Pi$ 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 $\Pi$?

Skorzystamy z programowania dynamicznego.




Jeśli $p[1..k]$ jest właściwym sufiksem i równocześnie prefiksem $p[1..j]$ to:

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

Czyli 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

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

# Obliczanie tablicy $\Pi$

```
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
```

**Złożoność czasowa:** $\Theta(m)$

**Złożoność czasowa KMP:** $\Theta(n+m)$ **Złożoność pamięciowa KMP:** $\Theta(m)$