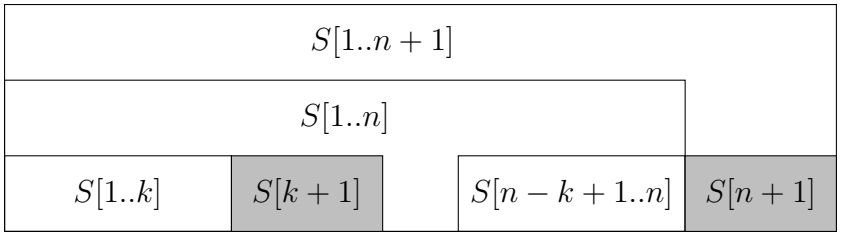
Słowa, słowa, słowa.

William Szekspir

Zdefiniujmy najpierw prefikso-sufiks.

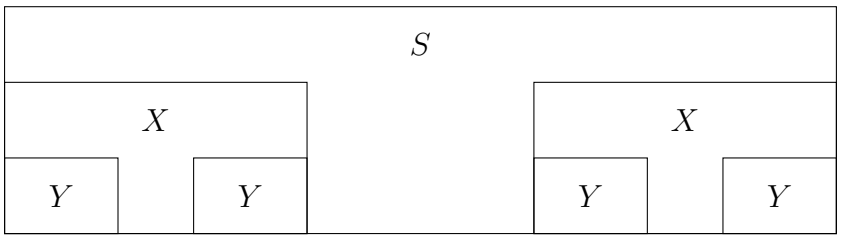
Definicja 1. Powiemy, że $k < n$ jest *prefikso-sufiksem* słowa $S[1..n]$ iff $S[1..k] = S[n-k+1..n]$.

Definicja 2. Powiemy, że prefikso-sufiks k słowa $S[1..n]$ jest *rozszerzalny* na słowo $S[1..n+1]$ iff $S[k+1] = S[n+1]$, tj. możemy go wydłużyć o nową literę.



Podamy teraz pewną przydatną ich własność.

Twierdzenie 1. *Jeśli k jest prefikso-sufiksem S oraz ℓ jest prefikso-sufiksem $S[1..k]$, to ℓ jest prefikso-sufiksem S.*



Będziemy chcieli policzyć prefikso-sufiksy prefiksów słowa.

Rozważmy napis $S[1..n]$. Niech $\pi[i]$ oznacza maksymalny prefikso-sufiks $S[1..i]$. Załóżmy, że znamy $\pi[1], \pi[2], \ldots, \pi[i-1]$. Możemy łatwo obliczyć $\pi[i]$ w następujący sposób.

Jeśli można rozszerzyć $\pi[i-1]$, robimy to. W.p.p. patrzymy na maksymalny prefikso-sufiks $S[1..\pi[i-1]]$, który z tw. 1. jest też prefikso-sufiksem $S[1..i]$. Jest to $\pi[\pi[i-1]]$, który próbujemy rozszerzyć. Potem próbujemy rozszerzyć $\pi[\pi[\pi[i-1]]]$ itd.

Przykład 1. Rozważmy $S =$ `abacabab`. Obliczymy maksymalne prefikso-sufiksy dla każdego prefiksu.

Dla $S[1]$ mamy $\pi[1] = 0$, bo słowo to nie ma prefiksów właściwych.

Dla $S[1..2]$ mamy $\pi[2] = 0$, bo `a` $\neq$ `b`.

Dla $S[1..3]$ mamy $\pi[3] = 1$, bo `a` $=$ `a`.

Dla $S[1..4]$ nie możemy rozszerzyć poprzedniego prefikso-sufiksu równego $\pi[3] = 1$. Mamy więc $\pi[4] = 0$.

Dla $S[1..5]$ mamy znów $\pi[5] = 1$, bo `a` $=$ `a`

Dla $S[1..6]$ możemy rozszerzyć poprzedni prefikso-sufiks i mamy $\pi[6] = 2$.

Dla $S[1..7]$ znów możemy rozszerzyć poprzedni prefikso-sufiks i dostać $\pi[7] = 3$.

Dla $S[1..8] = S$ nie możemy rozszerzyć poprzedniego prefikso-sufiksu $\pi[7] = 3$. Możemy jednak rozszerzyć prefikso-sufiks podsłowa $S[1..3]$ równy $\pi[\pi[7]] = \pi[3] = 1$. Dostajemy $\pi[8] = 2$. $\Diamond$

Algorytm możemy zapisać w C++.

```
pi[0] = b = -1;
for (int i = 1; i <= n; i++)
{
    while (b > -1 && S[b+1] != S[i])
        b = pi[b];
    b++;
    pi[i] = b;
}
```

Warto odnotować, że w powyższej implementacji ustawiamy *wartownika* w `pi[0]`.

Mimo tego, że powyższy algorytm wydaje się być kwadratowy, to w istocie się amortyzuje.

Twierdzenie 2. *Powyższy algorytm działa w czasie* $O(n)$.

Dowód: Przy każdym zwiększeniu `b++` włożymy monetę do banku. Natomiast przy przypisaniu `b = pi[b]` zmienna `b` ściśle maleje, zatem możemy wyjąć pewną liczbę monet z banku, tak by pozostało ich `b`. Zauważmy, że włożymy n monet – dla każdej iteracji jedną monetę. Zatem wyjmiemy też co najwyżej n monet. Ostatecznie więc złożoność to $O(n)$. □

Dzięki powyższemu algorytmowi możemy rozwiązać problem znajdowania wzorca w tekście i opisać algorytm Knuth'a-Morrsi'a-Pratt'a, w skrócie KMP.

Niech dany będzie wzorzec W o długości m i słowo S. Aby rozwiązać problem znajdowania wzorca W w słowie S wystarczy zdefiniować słowo $T := W\$S$ i uruchomić na nim powyższą procedurę. Istotnie, na pozycji i w słowie S występuje wzorzec W iff maksymalny prefikso-sufiks słowa $T[1..2m+i]$ to m.