# Algorytmy i struktury danych

wykład 8

# Plan wykładu:

- Kodowanie.
- Algorytmy przetwarzające napisy:
  - wyszukiwanie wzorca w tekście,
  - odległość edycyjna.

# Kodowanie

**Kodowanie** – jest to proces przekształcania informacji wybranego typu w informację innego typu.

Kod:

- jest sposobem przekształcania informacji,
- wyróżnia się kody otwarte i zamknięte.

**Kodowanie** – jest to proces przekształcania informacji wybranego typu w informację innego typu.

Kod:

- jest sposobem przekształcania informacji,
- wyróżnia się kody otwarte i zamknięte.

Kod jest funkcją, która do ciągów wejściowych (pierwotnych) przyporządkowuje inne ciągi, nazywane ciągami kodowymi.

**Kodowanie** – jest to proces przekształcania informacji wybranego typu w informację innego typu.

Kod:

- jest sposobem przekształcania informacji,
- wyróżnia się kody otwarte i zamknięte.

W kodzie otwartym istnieje możliwość dodawania nowych ciągów pierwotnych, W kodach zamkniętych takiej możliwości nie ma (np. w alfabecie).

Kod jest funkcją, która do ciągów wejściowych (pierwotnych) przyporządkowuje inne ciągi, nazywane ciągami kodowymi.

**Kodowanie** – jest to proces przekształcania informacji wybranego typu w informację innego typu.

Kod:
- jest sposobem przekształcania informacji,
- wyróżnia się kody otwarte i zamknięte.

W kodzie otwartym istnieje możliwość dodawania nowych ciągów pierwotnych, W kodach zamkniętych takiej możliwości nie ma (np. w alfabecie).

Kod jest funkcją, która do ciągów wejściowych (pierwotnych) przyporządkowuje inne ciągi, nazywane ciągami kodowymi.

Przykłady kodów:
- Binarny, BCD – stosowane w komputerach w zapisie liczb,
- ASCII, Unicode – stosowane w komputerach do zapisywania znaków i napisów,
- Gray'a – stosowany w zapisie liczb,
- Morse'a – stosowany w komunikacji,
- Hamminga – stosowany w systemach komputerowych,
- Huffmana – stosowany w systemach komputerowych,
- pocztowy – stosowany do określania lokalizacji docelowej przesyłek pocztowych,
- ...

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Algorytm:
Dla informacji zawierającej *n* znaków ze zbioru *S*:
1. Obliczenie częstości występowania znaków w informacji dla każdego elementu zbioru *S*.
2. Utworzenie listy drzew binarnych, które w węzłach będą zawierały pary: symbol–częstość występowania (prawdopodobieństwo). Na początku drzewa będą składały się wyłącznie z korzenia.
3. Dopóki w liście jest więcej niż jedno drzewo:
   - usunięcie z listy drzewa o najmniejszej liczności zapisanej w korzeniu,
   - wstawienie nowego drzewa, w którego korzeniu jest suma liczności (prawdopodobieństwa) usuniętych drzew – usunięte drzewa stają się jego lewym i prawym poddrzewem (korzeń drzewa nie przechowuje symbolu).
4. Końcowe drzewo nazywane jest drzewem Huffmana i stanowi wynik działania algorytmu.

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Algorytm:
Dla informacji zawierającej *n* znaków ze zbioru *S*:
1. Obliczenie częstości występowania znaków w informacji dla każdego elementu zbioru *S*.
2. Utworzenie listy drzew binarnych, które w węzłach będą zawierały pary: symbol–częstość występowania (prawdopodobieństwo). Na początku drzewa będą składały się wyłącznie z korzenia.
3. Dopóki w liście jest więcej niż jedno drzewo:
   - usunięcie z listy drzewa o najmniejszej liczności zapisanej w korzeniu,
   - wstawienie nowego drzewa, w którego korzeniu jest suma liczności (prawdopodobieństwa) usuniętych drzew – usunięte drzewa stają się jego lewym i prawym poddrzewem (korzeń drzewa nie przechowuje symbolu).
4. Końcowe drzewo nazywane jest drzewem Huffmana i stanowi wynik działania algorytmu.

Wyznaczanie kodów symboli:
1. Każdej lewej krawędzi drzewa przypisuje się wartość 0, dla prawej przypisuje się wartość 1.
2. Podczas przechodzenia w głąb drzewa od korzenia do każdego liścia (symbolu):
   a. Jeśli wybierana jest droga w prawo, dopisanie do kodu bit o wartości 1.
   b. Jeśli wybierana jest droga w lewo, dopisanie do kodu wartości 0.

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Algorytm:
Dla informacji zawierającej *n* znaków ze zbioru *S*:
1. Obliczenie częstości występowania znaków w informacji dla każdego elementu zbioru *S*.
2. Utworzenie listy drzew binarnych, które w węzłach będą zawierały pary: symbol–częstość występowania (prawdopodobieństwo). Na początku drzewa będą składały się wyłącznie z korzenia.
3. Dopóki w liście jest więcej niż jedno drzewo:
   - usunięcie z listy drzewa o najmniejszej liczności zapisanej w korzeniu,
   - wstawienie nowego drzewa, w którego korzeniu jest suma liczności (prawdopodobieństwa) usuniętych drzew – usunięte drzewa stają się jego lewym i prawym poddrzewem (korzeń drzewa nie przechowuje symbolu).
4. Końcowe drzewo nazywane jest drzewem Huffmana i stanowi wynik działania algorytmu.

Wyznaczanie kodów symboli:
1. Każdej lewej krawędzi drzewa przypisuje się wartość 0, dla prawej przypisuje się wartość 1.
2. Podczas przechodzenia w głąb drzewa od korzenia do każdego liścia (symbolu):
   a. Jeśli wybierana jest droga w prawo, dopisanie do kodu bit o wartości 1.
   b. Jeśli wybierana jest droga w lewo, dopisanie do kodu wartości 0.

**Długość słowa kodowego jest równa głębokości symbolu w drzewie.**

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

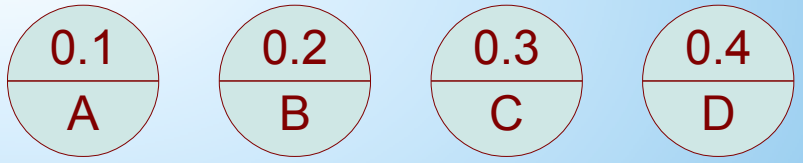
Przykład:
Wykonać kodowanie Huffmana dla informacji zawierającej 100 symboli ze zbioru S={A, B, C, D}, których liczebność jest następująca: A – 10, B – 20, C – 30, D – 40.

Rozwiązanie:
1. Obliczenie prawdopodobieństwa wystąpienia symboli w informacji:
   A – 0.1, B – 0.2, C – 0.3, D – 0.4.

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Przykład:
Wykonać kodowanie Huffmana dla informacji zawierającej 100 symboli ze zbioru S={A, B, C, D}, których liczebność jest następująca: A – 10, B – 20, C – 30, D – 40.

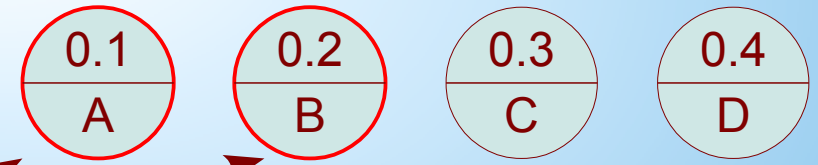
Rozwiązanie:
2. Utworzenie drzew:



**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Przykład:
Wykonać kodowanie Huffmana dla informacji zawierającej 100 symboli ze zbioru S={A, B, C, D}, których liczebność jest następująca: A – 10, B – 20, C – 30, D – 40.

Rozwiązanie:
3. Łączenie drzew:




Wybranie drzew z najmniejszymi prawdopodobieństwami.

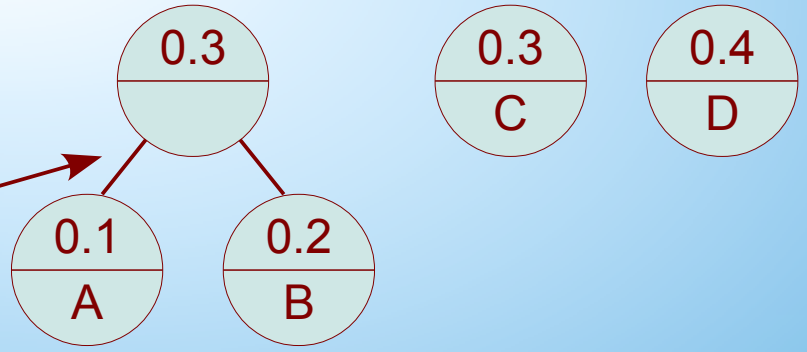
Krok 1 – łączenie A i B

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Przykład:
Wykonać kodowanie Huffmana dla informacji zawierającej 100 symboli ze zbioru S={A, B, C, D}, których liczebność jest następująca: A – 10, B – 20, C – 30, D – 40.

Rozwiązanie:
3. Łączenie drzew:

Utworzenie nowego drzewa – w korzeniu wpisuje się sumę prawdopodobieństw z liści.

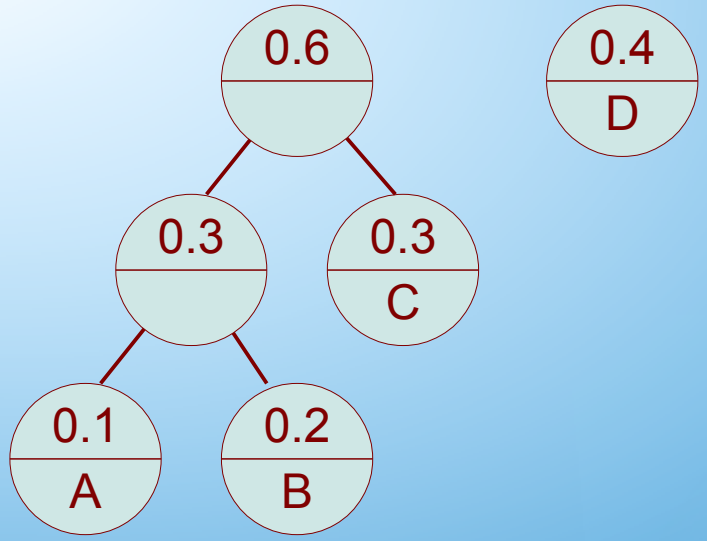



Krok 1 – łączenie A i B

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Przykład:
Wykonać kodowanie Huffmana dla informacji zawierającej 100 symboli ze zbioru S={A, B, C, D}, których liczebność jest następująca: A – 10, B – 20, C – 30, D – 40.

Rozwiązanie:
3. Łączenie drzew:

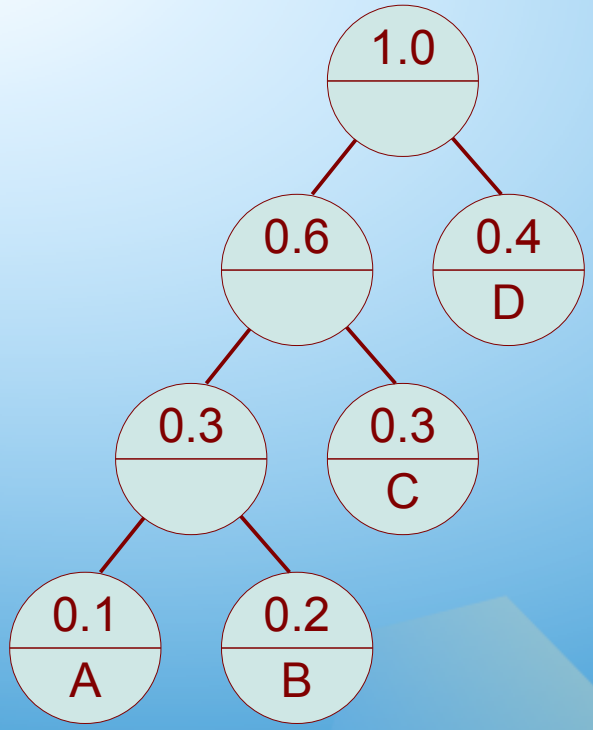



Krok 2 – łączenie (A, B) i C

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Przykład:
Wykonać kodowanie Huffmana dla informacji zawierającej 100 symboli ze zbioru S={A, B, C, D}, których liczebność jest następująca: A – 10, B – 20, C – 30, D – 40.

Rozwiązanie:
3. Łączenie drzew:

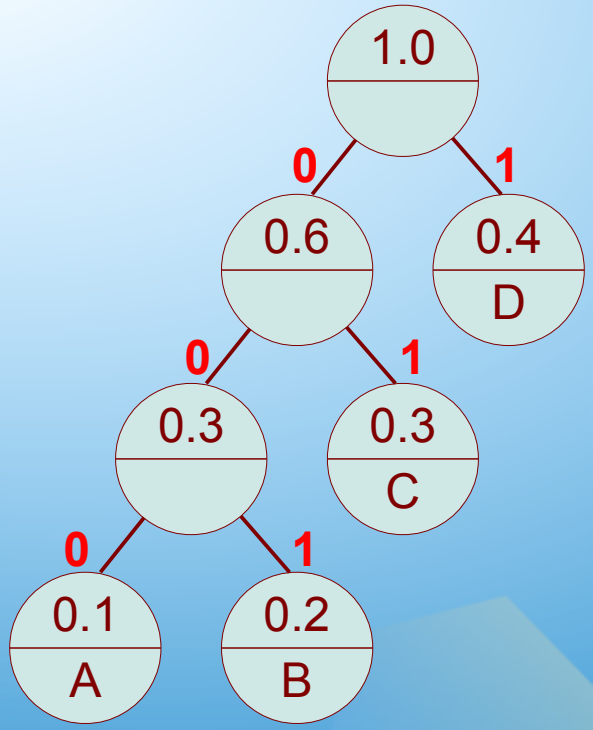



Krok 3 – łączenie ((A, B), C) i D

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Przykład:
Wykonać kodowanie Huffmana dla informacji zawierającej 100 symboli ze zbioru S={A, B, C, D}, których liczebność jest następująca: A – 10, B – 20, C – 30, D – 40.

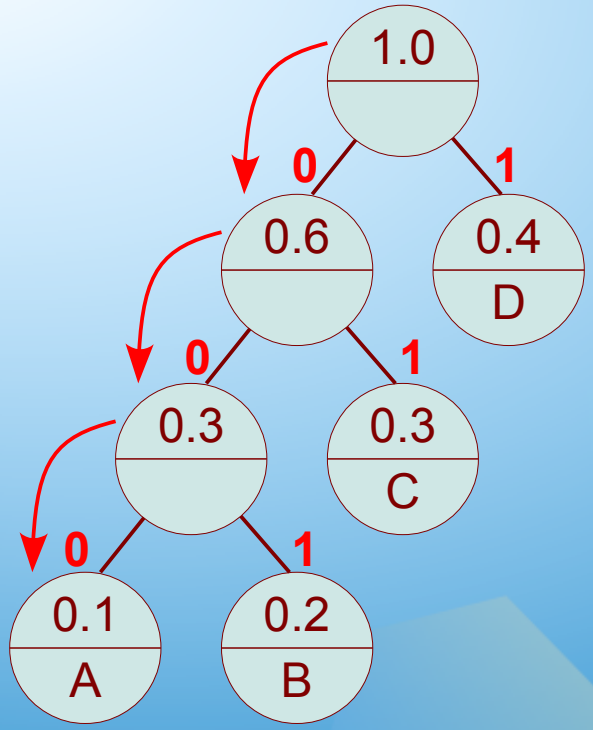
Rozwiązanie:
4. Wyznaczenie kodów:



**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Przykład:
Wykonać kodowanie Huffmana dla informacji zawierającej 100 symboli ze zbioru S={A, B, C, D}, których liczebność jest następująca: A – 10, B – 20, C – 30, D – 40.

Rozwiązanie:
4. Wyznaczenie kodów:

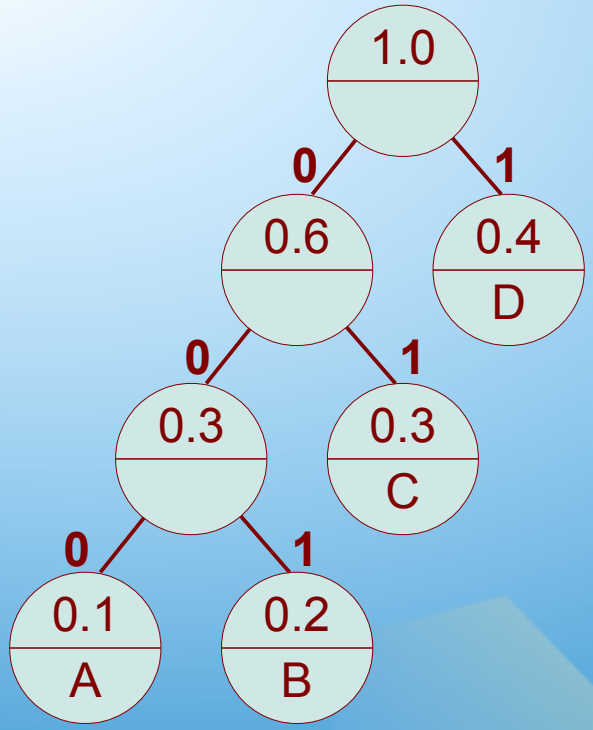



Przejście od korzenia do symbolu A wyznaczy ciąg:
000 – kod tego symbolu

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Przykład:
Wykonać kodowanie Huffmana dla informacji zawierającej 100 symboli ze zbioru S={A, B, C, D}, których liczebność jest następująca: A – 10, B – 20, C – 30, D – 40.

Rozwiązanie:
4. Wyznaczenie kodów:



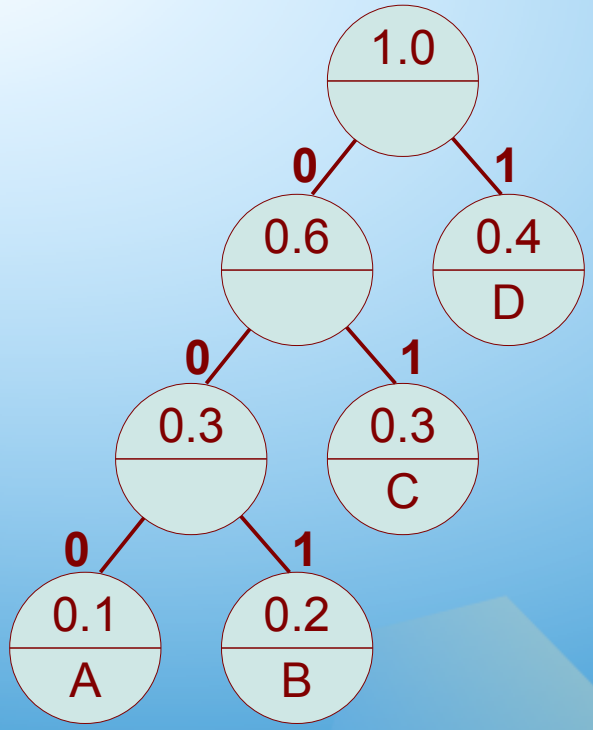

Wyznaczone kody:
A – 000, B – 001,
C – 01, D – 1.

Średnia długość kodu wynosi:
$L = p(A) \cdot 3 + p(B) \cdot 3 + p(C) \cdot 2 + p(A) \cdot 1 = 1.9$ bita

**Kod Huffmana** – jest to sposób organizacji danych, pozwalających na zmniejszenie ilości pamięci potrzebnej do ich przechowywania.

Przykład:
Wykonać kodowanie Huffmana dla informacji zawierającej 100 symboli ze zbioru S={A, B, C, D}, których liczebność jest następująca: A – 10, B – 20, C – 30, D – 40.

Rozwiązanie:
4. Wyznaczenie kodów:




Wyznaczone kody:
A – 000, B – 001,
C – 01, D – 1.

Średnia długość kodu wynosi:
$L = p(A) \cdot 3 + p(B) \cdot 3 + p(C) \cdot 2 + p(A) \cdot 1 = 1.9$ bita

Po zwiększeniu liczby znaków w kodzie algorytm stosuje się jako metodę kompresji danych.

**Kod Hamminga** – to liniowy sposób kodowania informacji posiadający zdolność autokorekcji.

Cechy:

- pozwala wykryć i skorygować błędy pojedyncze,
- pozwala wykryć błędy podwójne (bez możliwości korekcji).

**Kod Hamminga** – to liniowy sposób kodowania informacji posiadający zdolność autokorekcji.

Cechy:
- pozwala wykryć i skorygować błędy pojedyncze,
- pozwala wykryć błędy podwójne (bez możliwości korekcji).

Model kodowania:
- Na pozycjach bitów w kodzie będących potęgami liczby 2 znajdują się bity parzystości informacji – pozycje: 1, 2, 4, 8, 16, 32, ... .
- Na pozostałych pozycjach znajdują się bity informacji.
- Każdy bit parzystości wskazuje parzystość pewnej grupy bitów w słowie, zgodnie z jego pozycją w słowie kodowym. Dla bitów na pozycji:
  1. nie sprawdza 0 bitów, sprawdza 1 bit, nie sprawdza 1 bit, sprawdza 1 bit, ... (sprawdza pozycje 1, 3, 5, 7, 9, ...),
  2. nie sprawdza 1 bit, sprawdza 2 bity, nie sprawdza 2 bity, sprawdza 2 bity, ... (sprawdza pozycje 2, 3, 6, 7, 10, ...),
  4. nie sprawdza 3 bity, sprawdza 4 bity, nie sprawdza 4 bity, sprawdza 4 bity, ... (sprawdza pozycje 4, 5, 6, 7, 12, 13, 14, 15, ...),
  8. nie sprawdza 7 bitów, sprawdza 8 bitów, nie sprawdza 8 bitów, sprawdza 8 bitów, ...
  ...

**Kod Hamminga** – to liniowy sposób kodowania informacji posiadający zdolność autokorekcji.

Cechy:
- pozwala wykryć i skorygować błędy pojedyncze,
- pozwala wykryć błędy podwójne (bez możliwości korekcji).

Model kodowania:
- Na pozycjach bitów w kodzie będących potęgami liczby 2 znajdują się bity parzystości informacji – pozycje: 1, 2, 4, 8, 16, 32, ... .
- Na pozostałych pozycjach znajdują się bity informacji.
- Każdy bit parzystości wskazuje parzystość pewnej grupy bitów w słowie, zgodnie z jego pozycją w słowie kodowym. Dla bitów na pozycji:
  1. nie sprawdza 0 bitów, sprawdza 1 bit, nie sprawdza 1 bit, sprawdza 1 bit, ... (sprawdza pozycje 1, 3, 5, 7, 9, ...),
  2. nie sprawdza 1 bit, sprawdza 2 bity, nie sprawdza 2 bity, sprawdza 2 bity, ... (sprawdza pozycje 2, 3, 6, 7, 10, ...),
  4. nie sprawdza 3 bity, sprawdza 4 bity, nie sprawdza 4 bity, sprawdza 4 bity, ... (sprawdza pozycje 4, 5, 6, 7, 12, 13, 14, 15, ...),
  8. nie sprawdza 7 bitów, sprawdza 8 bitów, nie sprawdza 8 bitów, sprawdza 8 bitów, ...
  ...

Każdy bit kodu Hamminga posiada unikalną kombinację bitów sprawdzających – dotyczy to części informacyjnej oraz części kontrolnej kodu. Ta własność umożliwia korekcję pojedynczych błędów.

**Kod Hamminga** – to liniowy sposób kodowania informacji posiadający zdolność autokorekcji.

## Tabela kodowania słowa:

| Pozycja bitu | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit parzyst.(p), danych (d) | p1 | p2 | d1 | p3 | d2 | d3 | d4 | p4 | d5 | d6 | d7 | d8 | d9 | d10 | d11 | p5 | d12 | d13 | d14 | d15 | |
| Sekwencje sprawdzanych bitów | p1 | X | | X | | X | | X | | X | | X | | X | | X | | X | | X | |
| | p2 | | X | X | | | X | X | | | X | X | | | X | X | | | X | X | ... |
| | p3 | | | | X | X | X | X | | | | | X | X | X | X | | | | | |
| | p4 | | | | | | | | X | X | X | X | X | X | X | X | | | | | |
| | p5 | | | | | | | | | | | | | | | | X | X | X | X | |

## Zastosowanie:

- w pamięciach RAM,
- do korekcji danych w trakcie transmisji,
- wszędzie tam, gdzie może zaistnieć potrzeba korygowania małej liczby błędów.

# Algorytmy przetwarzające napisy

**Wyszukiwanie wzorca** – ang. pattern matching – to jeden z podstawowych problemów w algorytmach przetwarzających tekst.

Cechy:

- rozwiązaniem problemu wzorca jest wskazanie w ciągu $S$ wszystkich pozycji $n$ podciągu $p$, aby zachodziła równość: $S[n .. n + |p|] = p$ (co oznacza, że podciąg $p$ jest fragmentem ciągu $S$).
- operacją dominującą w wyszukiwaniu wzorca jest operacja porównania znaków.

**Wyszukiwanie wzorca** – ang. pattern matching – to jeden z podstawowych problemów w algorytmach przetwarzających tekst.

Cechy:

- rozwiązaniem problemu wzorca jest wskazanie w ciągu $S$ wszystkich pozycji $n$ podciągu $p$, aby zachodziła równość: $S[n .. n + |p|] = p$ (co oznacza, że podciąg $p$ jest fragmentem ciągu $S$).
- operacją dominującą w wyszukiwaniu wzorca jest operacja porównania znaków.

Pojęcia związanie z wzorcami:

- prefiks,
- sufiks,
- maksymalny prefiks właściwy,
- maksymalny sufiks właściwy,
- prefikso-sufiks.

**Wyszukiwanie wzorca** – ang. pattern matching – to jeden z podstawowych problemów w algorytmach przetwarzających tekst.

Cechy:

- rozwiązaniem problemu wzorca jest wskazanie w ciągu $S$ wszystkich pozycji $n$ podciągu $p$, aby zachodziła równość: $S[n .. n + |p|] = p$ (co oznacza, że podciąg $p$ jest fragmentem ciągu $S$).
- operacją dominującą w wyszukiwaniu wzorca jest operacja porównania znaków.

Pojęcia związanie z wzorcami:

- prefiks, → Dla ciągu $S$ są to $k$ początkowe jego znaki.
- sufiks,
- maksymalny prefiks właściwy,
- maksymalny sufiks właściwy,
- prefikso-sufiks.

**Wyszukiwanie wzorca** – ang. pattern matching – to jeden z podstawowych problemów w algorytmach przetwarzających tekst.

Cechy:

- rozwiązaniem problemu wzorca jest wskazanie w ciągu $S$ wszystkich pozycji $n$ podciągu $p$, aby zachodziła równość: $S[n .. n + |p|] = p$ (co oznacza, że podciąg $p$ jest fragmentem ciągu $S$).
- operacją dominującą w wyszukiwaniu wzorca jest operacja porównania znaków.

Pojęcia związanie z wzorcami:

- prefiks, → Dla ciągu $S$ są to $k$ początkowe jego znaki.
- sufiks, → Dla ciągu $S$ są to $k$ końcowe jego znaki.
- maksymalny prefiks właściwy,
- maksymalny sufiks właściwy,
- prefikso-sufiks.

**Wyszukiwanie wzorca** – ang. pattern matching – to jeden z podstawowych problemów w algorytmach przetwarzających tekst.

Cechy:

- rozwiązaniem problemu wzorca jest wskazanie w ciągu *S* wszystkich pozycji *n* podciągu *p*, aby zachodziła równość: $S[n .. n + |p|] = p$ (co oznacza, że podciąg *p* jest fragmentem ciągu *S*).
- operacją dominującą w wyszukiwaniu wzorca jest operacja porównania znaków.

Pojęcia związanie z wzorcami:

- prefiks, → Dla ciągu *S* są to *k* początkowe jego znaki.
- sufiks, → Dla ciągu *S* są to *k* końcowe jego znaki.
- maksymalny prefiks właściwy, → Ang. maximal proper prefix – dla ciągu *S* są to (*k - 1*) początkowe jego znaki.
- maksymalny sufiks właściwy,
- prefikso-sufiks.

**Wyszukiwanie wzorca** – ang. pattern matching – to jeden z podstawowych problemów w algorytmach przetwarzających tekst.

Cechy:

- rozwiązaniem problemu wzorca jest wskazanie w ciągu *S* wszystkich pozycji *n* podciągu *p*, aby zachodziła równość: $S[n .. n + |p|] = p$ (co oznacza, że podciąg *p* jest fragmentem ciągu *S*).
- operacją dominującą w wyszukiwaniu wzorca jest operacja porównania znaków.

Pojęcia związanie z wzorcami:

- prefiks, → Dla ciągu *S* są to *k* początkowe jego znaki.
- sufiks, → Dla ciągu *S* są to *k* końcowe jego znaki.
- maksymalny prefiks właściwy, → Ang. maximal proper prefix – dla ciągu *S* są to (*k - 1*) początkowe jego znaki.
- maksymalny sufiks właściwy, → Ang. maximal proper sufix – dla ciągu *S* są wszystkie znaki za wyjątkiem pierwszego.
- prefikso-sufiks.

**Wyszukiwanie wzorca** – ang. pattern matching – to jeden z podstawowych problemów w algorytmach przetwarzających tekst.

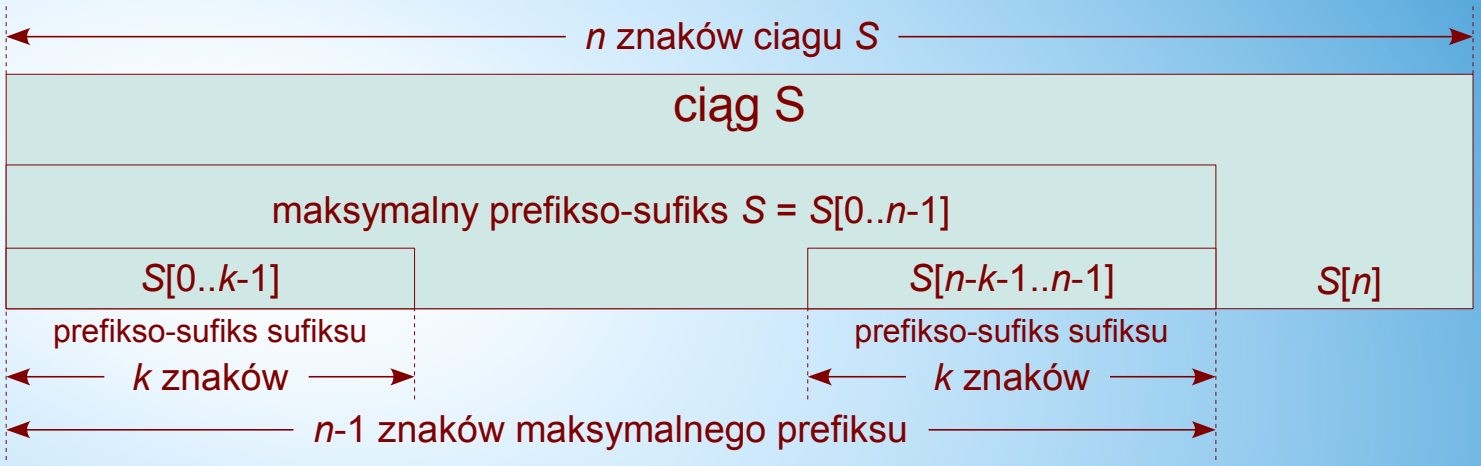
Cechy:

- rozwiązaniem problemu wzorca jest wskazanie w ciągu *S* wszystkich pozycji *n* podciągu *p*, aby zachodziła równość: $S[n .. n + |p|] = p$ (co oznacza, że podciąg *p* jest fragmentem ciągu *S*).
- operacją dominującą w wyszukiwaniu wzorca jest operacja porównania znaków.

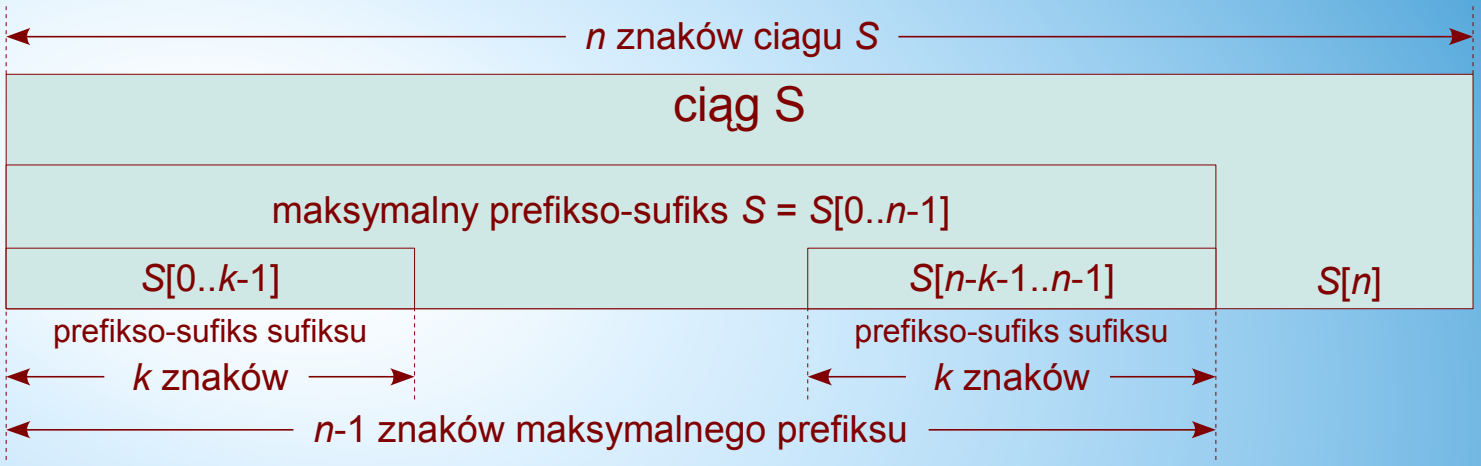
Pojęcia związanie z wzorcami:

- prefiks, → Dla ciągu *S* są to *k* początkowe jego znaki.
- sufiks, → Dla ciągu *S* są to *k* końcowe jego znaki.
- maksymalny prefiks właściwy, → Ang. maximal proper prefix – dla ciągu *S* są to (*k - 1*) początkowe jego znaki.
- maksymalny sufiks właściwy, → Ang. maximal proper sufix – dla ciągu *S* są wszystkie znaki za wyjątkiem pierwszego.
- prefikso-sufiks. → W ogólnym przypadku prefiks może pokrywać się z sufiksem, wtedy mówi się o prefikso-sufiksie (ang. border).

## Rozszerzenie prefikso-sufiksu:

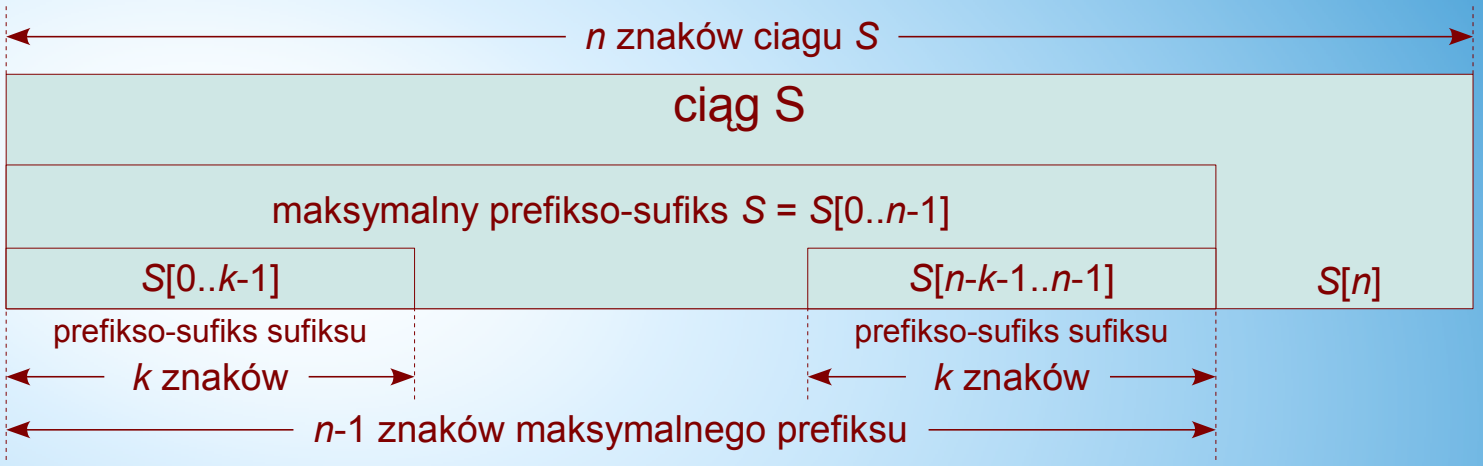


## Rozszerzenie prefikso-sufiksu:

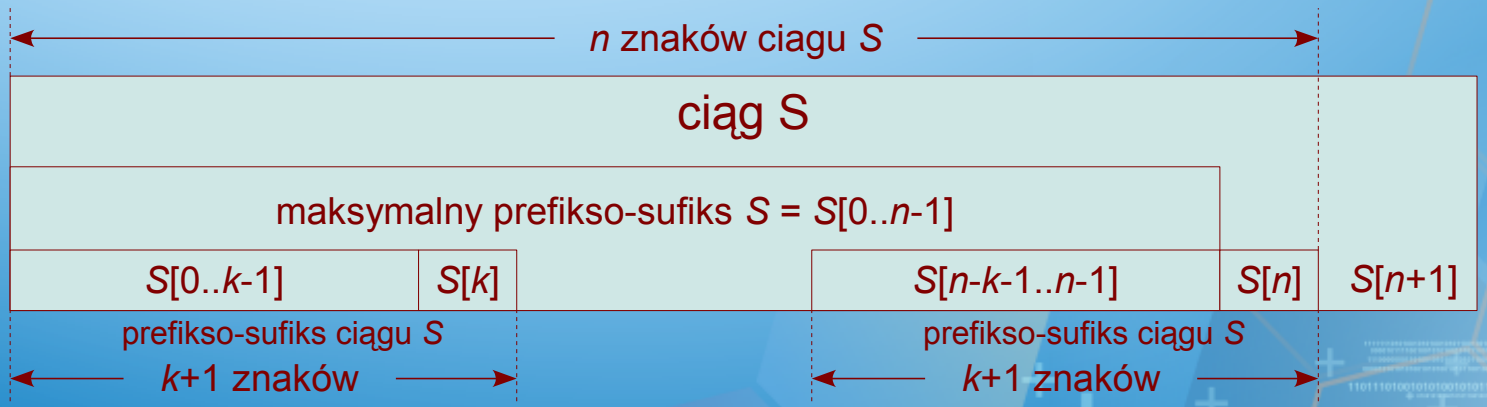



Jeśli maksymalny prefiks właściwy ciągu $S$ posiada prefikso-sufiks o długości $k$ znaków to prefikso-sufiks prefiksu jest rozszerzalny do prefikso-sufiksu całego ciągu $S$, gdy znak $S[k]$ jest równy znakowi $S[n]$. Wtedy długość prefikso-sufiksu zwiększa się do $k+1$ i staje się on prefikso-sufiksem ciągu $S$.

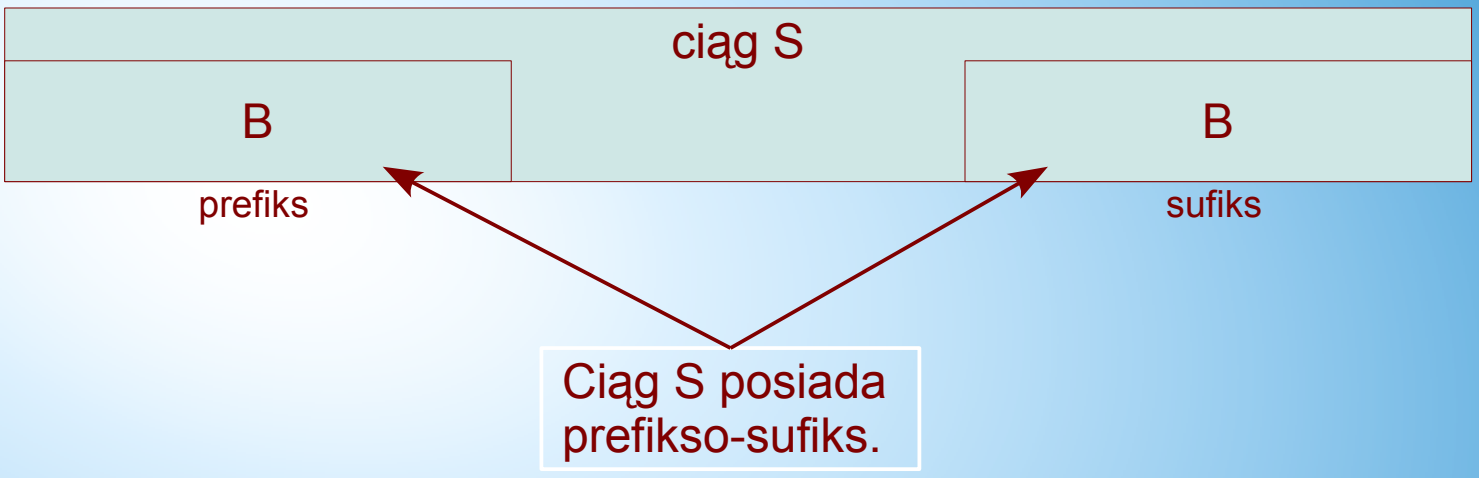
## Rozszerzenie prefikso-sufiksu:



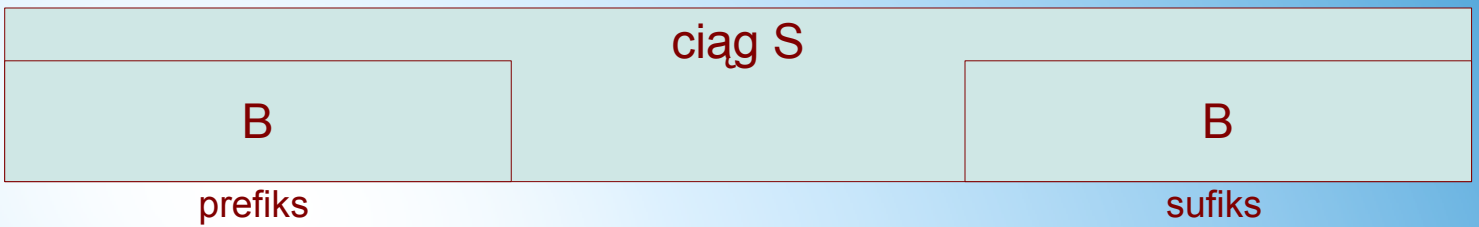

Jeśli maksymalny prefiks właściwy ciągu $S$ posiada prefikso-sufiks o długości $k$ znaków to prefikso-sufiks prefiksu jest rozszerzalny do prefikso-sufiksu całego ciągu $S$, gdy znak $S[k]$ jest równy znakowi $S[n]$. Wtedy długość prefikso-sufiksu zwiększa się do $k+1$ i staje się on prefikso-sufiksem ciągu $S$.



## Redukcja prefikso-sufiksu:



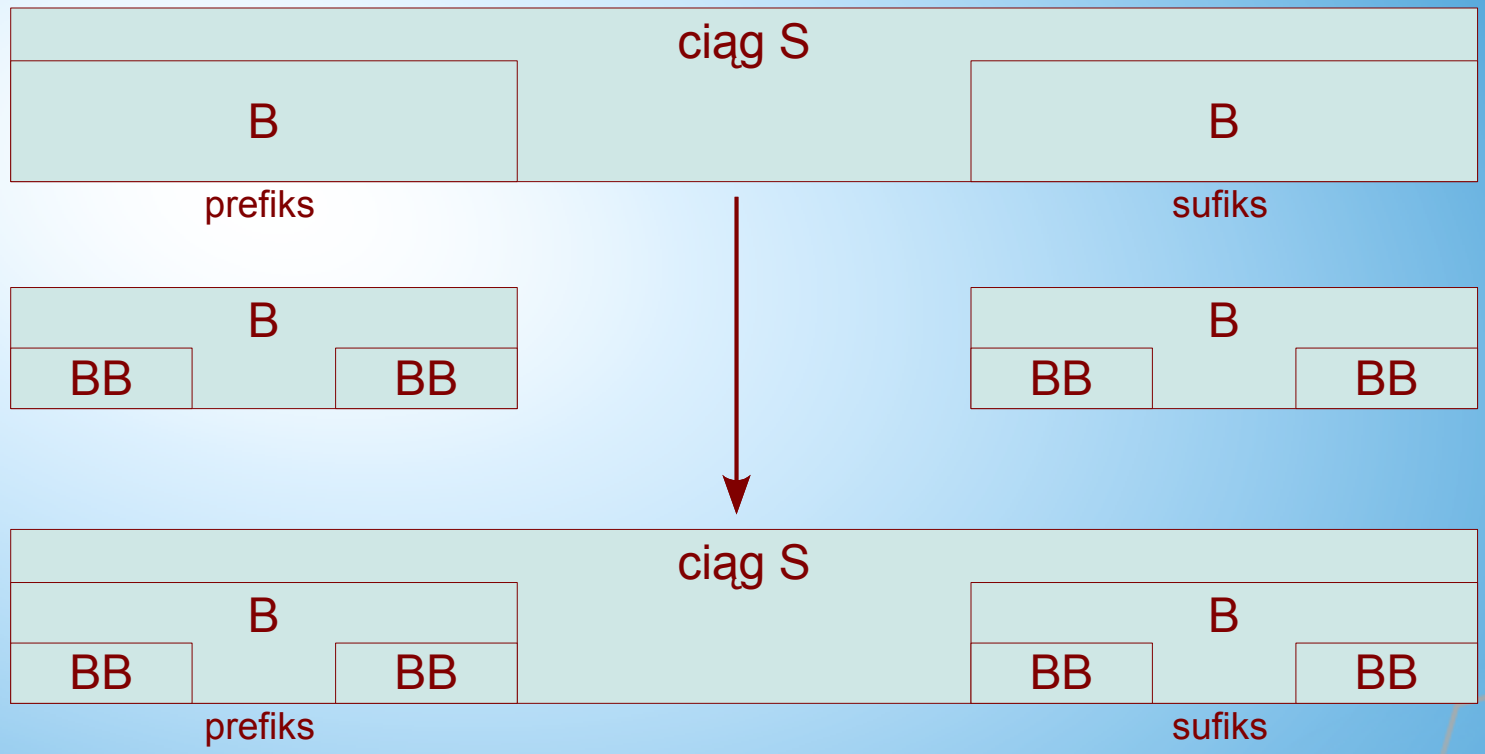
## Redukcja prefikso-sufiksu:



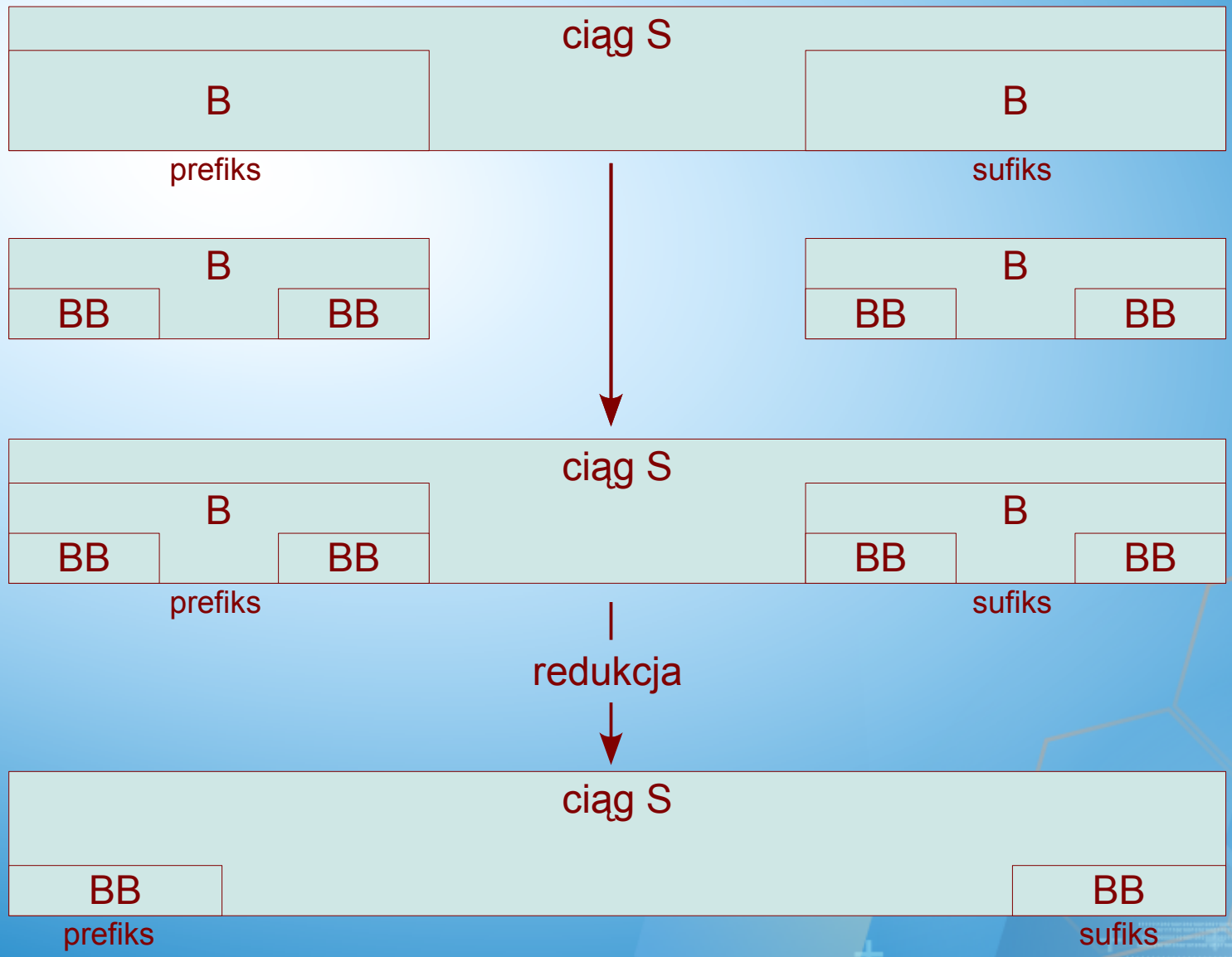


Prefikso-sufiks ciągu S posiada własny prefikso-sufiks.

## Redukcja prefikso-sufiksu:



## Redukcja prefikso-sufiksu:



Tablica maksymalnych prefikso-sufiksów Π[]:

- Tablica przechowuje informacje o prefiksach i sufiksach ciągu znaków,
- Do wyznaczenia tablicy stosuje się własności:
  - rozszerzania prefikso-sufiksu,
  - redukcji prefikso-sufiksu.
- Indeks w tablicy oznacza długość prefikso-sufiksu ciągu, a wartość wskazana przez ten indeks jest równa długości maksymalnego prefikso-sufiksu dla danego prefiksu.
- Tablica ma tyle elementów ile znajduje się znaków w ciągu, dla którego została wyznaczona.

Tablica maksymalnych prefikso-sufiksów Π[]:

- Tablica przechowuje informacje o prefiksach i sufiksach ciągu znaków,
- Do wyznaczenia tablicy stosuje się własności:
  - rozszerzania prefikso-sufiksu,
  - redukcji prefikso-sufiksu.
- Indeks w tablicy oznacza długość prefikso-sufiksu ciągu, a wartość wskazana przez ten indeks jest równa długości maksymalnego prefikso-sufiksu dla danego prefiksu.
- Tablica ma tyle elementów ile znajduje się znaków w ciągu, dla którego została wyznaczona.

Własność umożliwia rozszerzenie prefikso-sufiksu dla kolejnego prefiksu, jeśli znany jest prefikso-sufiks poprzedniego prefiksu.

Tablica maksymalnych prefikso-sufiksów Π[]:

- Tablica przechowuje informacje o prefiksach i sufiksach ciągu znaków,
- Do wyznaczenia tablicy stosuje się własności:
  - rozszerzania prefikso-sufiksu,
  - redukcji prefikso-sufiksu.
- Indeks w tablicy oznacza długość prefikso-sufiksu ciągu, a wartość wskazana przez ten indeks jest równa długości maksymalnego prefikso-sufiksu dla danego prefiksu.
- Tablica ma tyle elementów ile znajduje się znaków w ciągu, dla którego została wyznaczona.

Własność umożliwia rozszerzenie prefikso-sufiksu dla kolejnego prefiksu, jeśli znany jest prefikso-sufiks poprzedniego prefiksu.

Własność pozwala wyszukać prefikso-sufiksy, które mogą być rozszerzane, jeśli nie można tego zrobić z bieżącym.

Tablica maksymalnych prefikso-sufiksów Π[]:

- Tablica przechowuje informacje o prefiksach i sufiksach ciągu znaków,
- Do wyznaczenia tablicy stosuje się własności:
  - rozszerzania prefikso-sufiksu,
  - redukcji prefikso-sufiksu.
- Indeks w tablicy oznacza długość prefikso-sufiksu ciągu, a wartość wskazana przez ten indeks jest równa długości maksymalnego prefikso-sufiksu dla danego prefiksu.
- Tablica ma tyle elementów ile znajduje się znaków w ciągu, dla którego została wyznaczona.

Własność umożliwia rozszerzenie prefikso-sufiksu dla kolejnego prefiksu, jeśli znany jest prefikso-sufiks poprzedniego prefiksu.

Własność pozwala wyszukać prefikso-sufiksy, które mogą być rozszerzane, jeśli nie można tego zrobić z bieżącym.

Np. jeśli Π[6] = 2, to prefiks 6-cio znakowy ciągu posiada prefikso-sufiks maksymalny o długości dwóch znaków.

## Tablica maksymalnych prefikso-sufiksów Π[]:

- Tablica przechowuje informacje o prefiksach i sufiksach ciągu znaków,
- Do wyznaczenia tablicy stosuje się własności:
  - rozszerzania prefikso-sufiksu,
  - redukcji prefikso-sufiksu.
- Indeks w tablicy oznacza długość prefikso-sufiksu ciągu, a wartość wskazana przez ten indeks jest równa długości maksymalnego prefikso-sufiksu dla danego prefiksu.
- Tablica ma tyle elementów ile znajduje się znaków w ciągu, dla którego została wyznaczona.

Własność umożliwia rozszerzenie prefikso-sufiksu dla kolejnego prefiksu, jeśli znany jest prefikso-sufiks poprzedniego prefiksu.

Własność pozwala wyszukać prefikso-sufiksy, które mogą być rozszerzane, jeśli nie można tego zrobić z bieżącym.

Indeksy tablicy Π[ ] mają wartości od 0 do n, Zerowy element tablicy ma zawsze wartość równą -1 i pełni funkcję tzw. wartownika.

Np. jeśli Π[6] = 2, to prefiks 6-cio znakowy ciągu posiada prefikso-sufiks maksymalny o długości dwóch znaków.

## Tablica maksymalnych prefikso-sufiksów Π[]:

- Tablica przechowuje informacje o prefiksach i sufiksach ciągu znaków,
- Do wyznaczenia tablicy stosuje się własności:
  - rozszerzania prefikso-sufiksu,
  - redukcji prefikso-sufiksu.
- Indeks w tablicy oznacza długość prefikso-sufiksu ciągu, a wartość wskazana przez ten indeks jest równa długości maksymalnego prefikso-sufiksu dla danego prefiksu.
- Tablica ma tyle elementów ile znajduje się znaków w ciągu, dla którego została wyznaczona.

Własność umożliwia rozszerzenie prefikso-sufiksu dla kolejnego prefiksu, jeśli znany jest prefikso-sufiks poprzedniego prefiksu.

Własność pozwala wyszukać prefikso-sufiksy, które mogą być rozszerzane, jeśli nie można tego zrobić z bieżącym.

Indeksy tablicy Π[ ] mają wartości od 0 do n, Zerowy element tablicy ma zawsze wartość równą -1 i pełni funkcję tzw. wartownika.

Np. jeśli Π[6] = 2, to prefiks 6-cio znakowy ciągu posiada prefikso-sufiks maksymalny o długości dwóch znaków.

## Pseudokod algorytmu:

```
Π[0] := -1;
b := -1;
for i := 1 to |p| do begin
    while (b > -1) and (p[b] ≠ p[i – 1]) do b := Π[b];
    b := b + 1;
    Π[i] := b;
end;
```

## Tablica maksymalnych prefikso-sufiksów Π[]:

- Tablica przechowuje informacje o prefiksach i sufiksach ciągu znaków,
- Do wyznaczenia tablicy stosuje się własności:
  - rozszerzania prefikso-sufiksu,
  - redukcji prefikso-sufiksu.
- Indeks w tablicy oznacza długość prefikso-sufiksu ciągu, a wartość wskazana przez ten indeks jest równa długości maksymalnego prefikso-sufiksu dla danego prefiksu.
- Tablica ma tyle elementów ile znajduje się znaków w ciągu, dla którego została wyznaczona.

Własność umożliwia rozszerzenie prefikso-sufiksu dla kolejnego prefiksu, jeśli znany jest prefikso-sufiks poprzedniego prefiksu.

Własność pozwala wyszukać prefikso-sufiksy, które mogą być rozszerzane, jeśli nie można tego zrobić z bieżącym.

Indeksy tablicy Π[ ] mają wartości od 0 do n, Zerowy element tablicy ma zawsze wartość równą -1 i pełni funkcję tzw. wartownika.

Np. jeśli Π[6] = 2, to prefiks 6-cio znakowy ciągu posiada prefikso-sufiks maksymalny o długości dwóch znaków.

## Pseudokod algorytmu:

p – jest to wzorzec,
|p| – liczba znaków wzorca.

```
Π[0] := -1;
b := -1;
for i := 1 to |p| do begin
    while (b > -1) and (p[b] ≠ p[i – 1]) do b := Π[b];
    b := b + 1;
    Π[i] := b;
end;
```

Początkowo długość prefikso-sufiksu ustawia się na -1.

**Wyszukiwanie naiwne** – polega na porównaniu wzorca *p* z kolejnymi podciągami ciągu *S*, utworzonymi poprzez usunięcie prefiksu niepasującego do wzorca *p*.

Cechy:

- złożoność obliczeniowa pesymistyczna wynosi O($n \cdot m$), gdzie *n* oznacza liczbę znaków ciągu *S*, a *m* liczbę znaków wzorca *p*,
- zazwyczaj wystarczy porównać kilka pierwszych znaków ciągu i wzorca, dlatego złożoność średnia wynosi O(*n*).

**Wyszukiwanie naiwne** – polega na porównaniu wzorca *p* z kolejnymi podciągami ciągu *S*, utworzonymi poprzez usunięcie prefiksu niepasującego do wzorca *p*.

Cechy:

- złożoność obliczeniowa pesymistyczna wynosi O(*n*·*m*), gdzie *n* oznacza liczbę znaków ciągu *S*, a *m* liczbę znaków wzorca *p*,
- zazwyczaj wystarczy porównać kilka pierwszych znaków ciągu i wzorca, dlatego złożoność średnia wynosi O(*n*).

Pseudokod algorytmu:

```
for i := 0 to |S| do begin                    // S – przeszukiwany ciąg
    znaleziono_wzorzec_w_ciagu := true
    for j := 0 to |p| do                      // p – poszukiwany wzorzec
        if S[i + j] ≠ p[j] then begin
            znaleziono_wzorzec_w_ciagu := false
            break
        end
    if znaleziono_wzorzec_w_ciagu then drukuj_pozycje_wzorca
end
```

**Algorytm Morrisa-Pratta –** służy do wyszukiwania wszystkich wystąpień wzorca *p* w ciągu *S*.

Cechy:

- algorytm wyszukuje wszystkie wystąpienia wzorca *p* w ciągu *S*,
- w konstrukcji algorytmu wykorzystano fakt, że w przypadku wystąpienia niezgodności ze wzorcem, sam wzorzec zawiera w sobie informację pozwalającą określić gdzie powinna się zacząć kolejna próba dopasowania, pomijając ponowne porównywanie już dopasowanych znaków,
- złożoności obliczeniowa wynosi O(*n*+*m*), co ma znaczenie dla dużych wzorców.

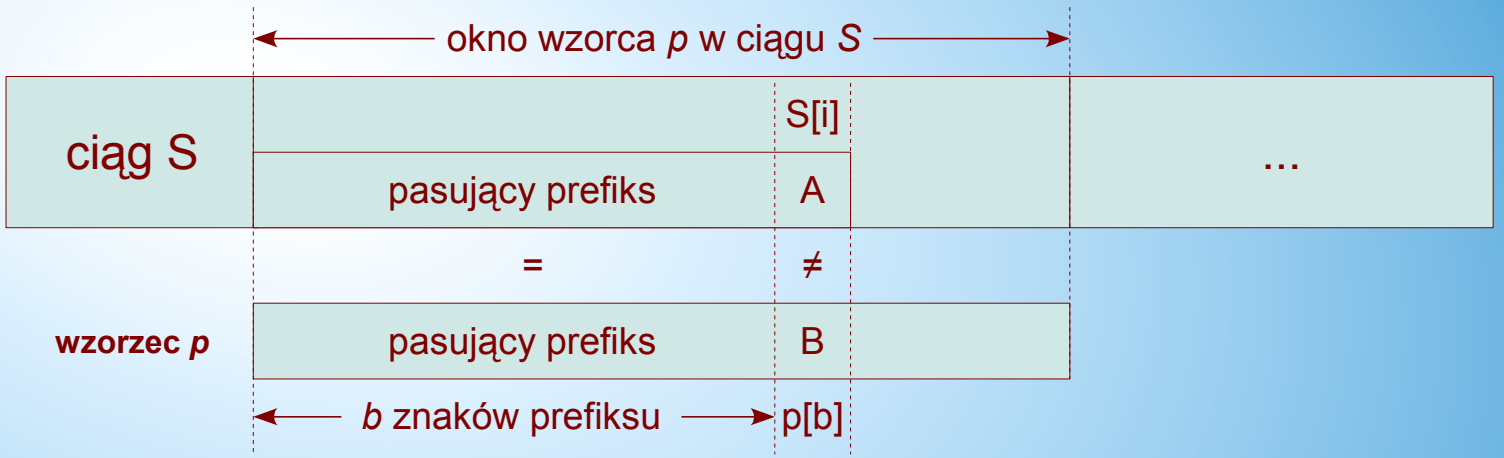
**Algorytm Morrisa-Pratta –** służy do wyszukiwania wszystkich wystąpień wzorca *p* w ciągu *S*.

Cechy:
- algorytm wyszukuje wszystkie wystąpienia wzorca *p* w ciągu *S*,
- w konstrukcji algorytmu wykorzystano fakt, że w przypadku wystąpienia niezgodności ze wzorcem, sam wzorzec zawiera w sobie informację pozwalającą określić gdzie powinna się zacząć kolejna próba dopasowania, pomijając ponowne porównywanie już dopasowanych znaków,
- złożoności obliczeniowa wynosi O(*n*+*m*), co ma znaczenie dla dużych wzorców.

Algorytm:
1. Wyznaczenie tablicy maksymalnych prefikso-sufiksów Π[ ].
2. Porównywanie kolejnych znaków, aż do napotkania znaków zgodnych ze wzorcem.
3. Odczytanie z tablicy Π[ ] maksymalnej szerokości prefikso-sufiksu dla *b* znaków prefiksu wzorca *p*.
4. Porównanie znaków wzorca *p* na pozycji odczytanej na etapie 3 ze znakiem ciągu *S*[i].
5. Jeśli na etapie 4 nie ma zgodności znaków, to wykonanie algorytmu od etapu 3 (aż do wyczerpania się prefikso-sufiksów) – w takim przypadku okno wzorca oraz indeks *i* przesuwa się o jedną pozycję w prawo.
6. Jeśli na etapie 4 jest zgodność znaków, to pasujący prefiks zwiększa swoją długość o 1 znak. Przesuwa się również indeks *i* o 1 (znak na tej pozycji został już całkowicie wykorzystany przez algorytm).
7. Jeśli prefiks obejmie cały wzorzec (czyli $b = |p|$), to znaleziona zostanie pozycja wzorca w ciągu *S* i będzie ona równa $(i - b + 1)$. W przeciwnym razie wykonuje się etap 4.

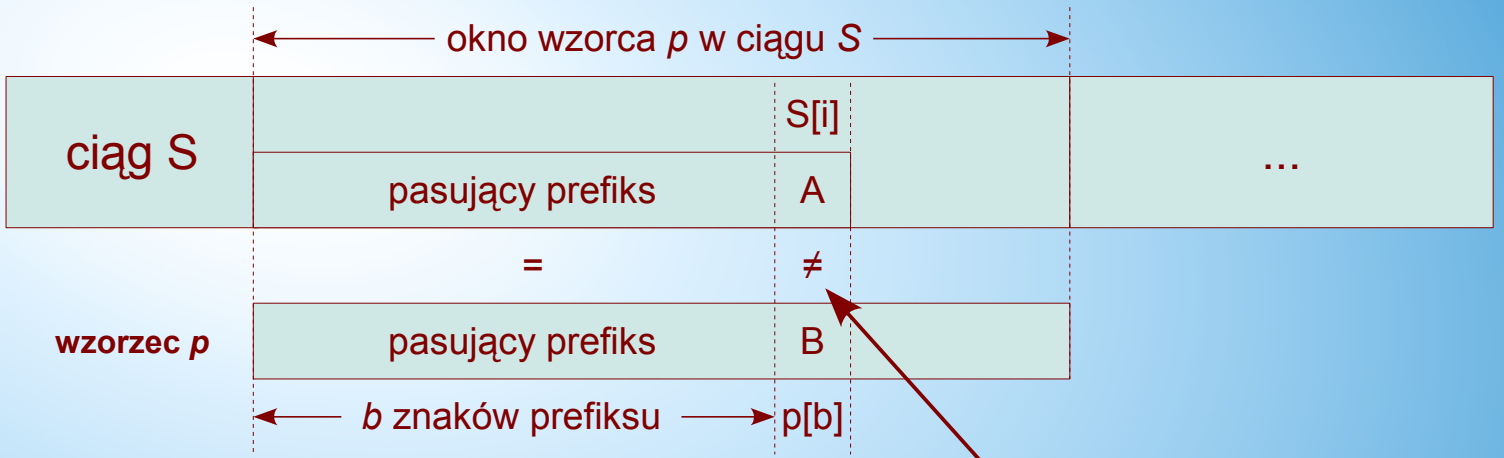
**Algorytm Morrisa-Pratta –** służy do wyszukiwania wszystkich wystąpień wzorca *p* w ciągu *S*.

Proces dopasowania:



**Algorytm Morrisa-Pratta –** służy do wyszukiwania wszystkich wystąpień wzorca *p* w ciągu *S*.

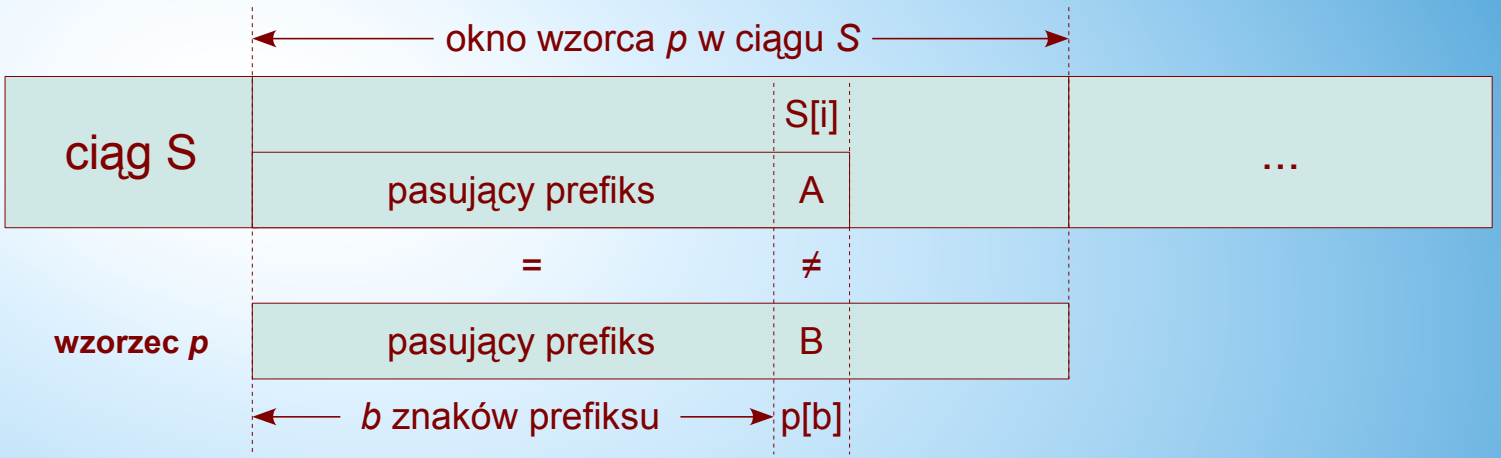
Proces dopasowania:




Prefiks wzorca *p*, o długości *b* znaków, pasuje do prefiksu okna w ciągu *S* przed *i*-tą pozycją, jednakże znak *S*[i] różni się od znaku *p*[*b*], który znajduje się we wzorcu tuż za pasującym prefiksem.

**Algorytm Morrisa-Pratta –** służy do wyszukiwania wszystkich wystąpień wzorca *p* w ciągu *S*.

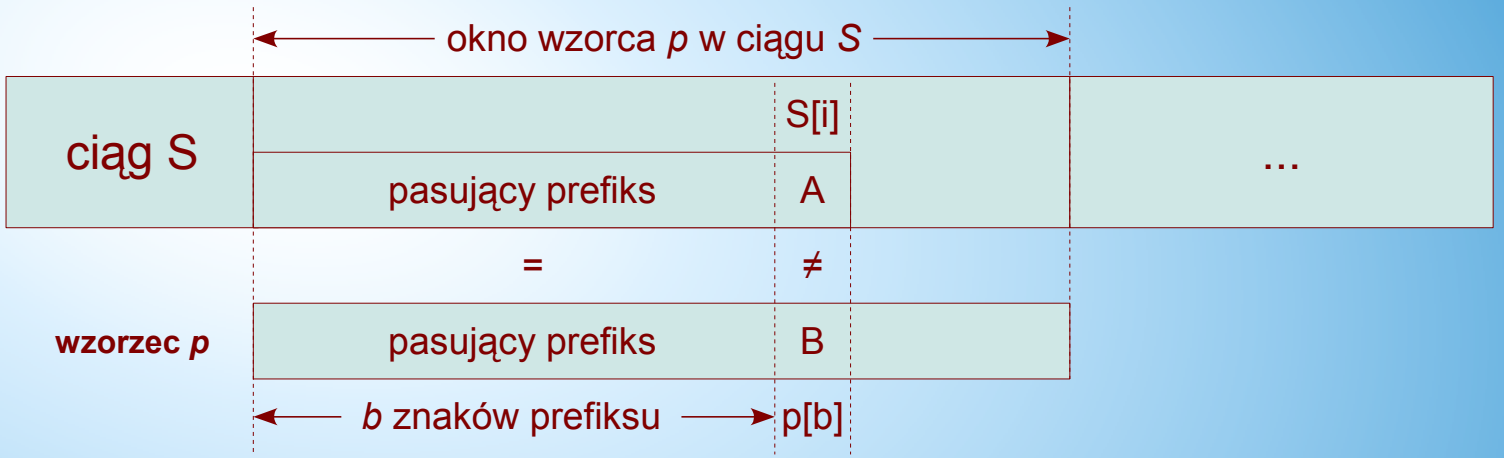
Proces dopasowania:




W algorytmie naiwnym w takiej sytuacji przesuwa się okno wzorca o jedną pozycję w prawo względem przeszukiwanego tekstu i rozpoczyna od początku porównywanie znaków wzorca *p* ze znakami okna. Powoduje to wzrost złożoności obliczeniowej do $O(n^2)$.

**Algorytm Morrisa-Pratta –** służy do wyszukiwania wszystkich wystąpień wzorca *p* w ciągu *S*.

Proces dopasowania:

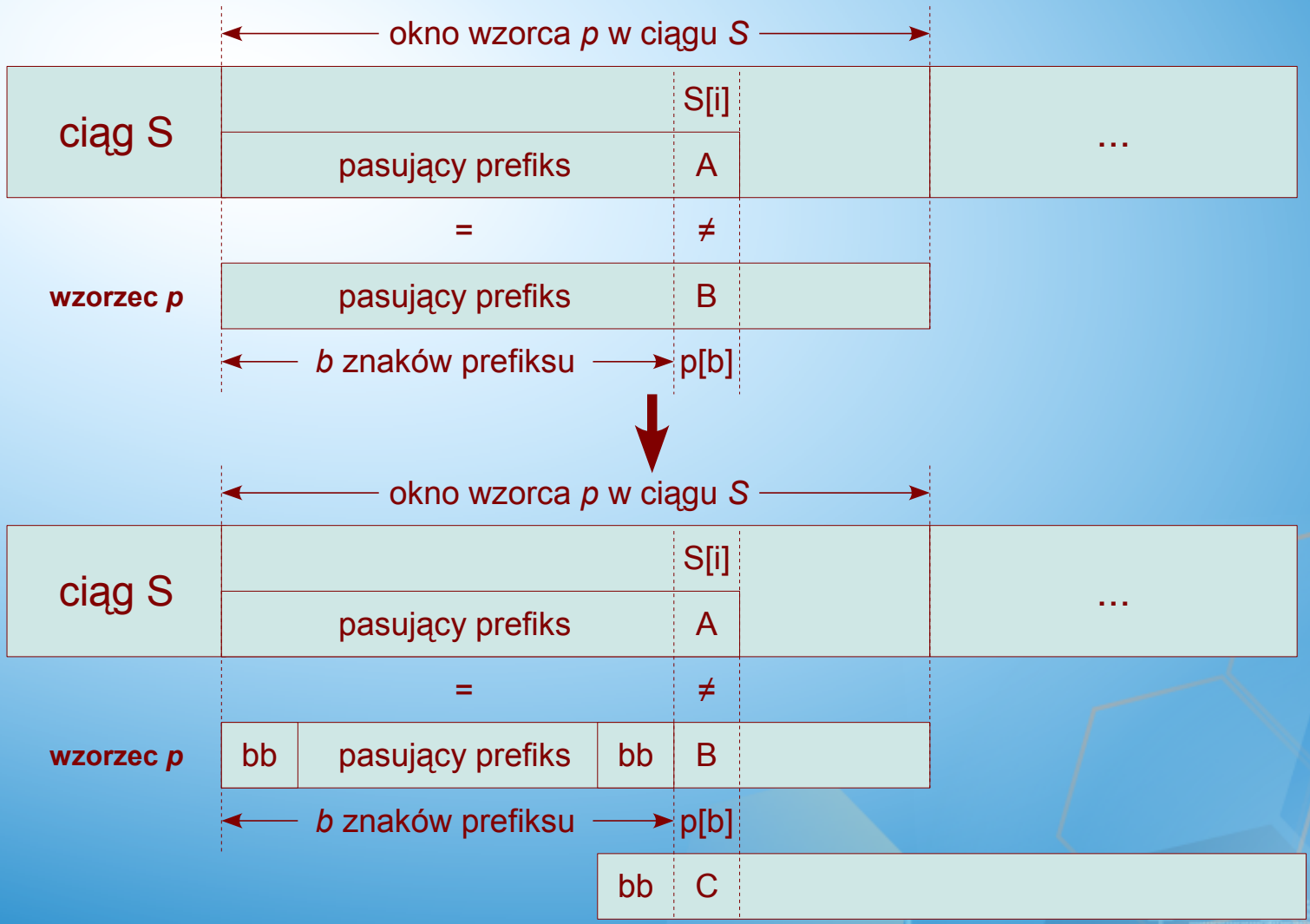



W algorytmie naiwnym w takiej sytuacji przesuwa się okno wzorca o jedną pozycję w prawo względem przeszukiwanego tekstu i rozpoczyna od początku porównywanie znaków wzorca *p* ze znakami okna. Powoduje to wzrost złożoności obliczeniowej do $O(n^2)$.

Można wykorzystać fakt istnienia pasującego prefiksu i pominąć pewne znaki w następnym porównaniu.

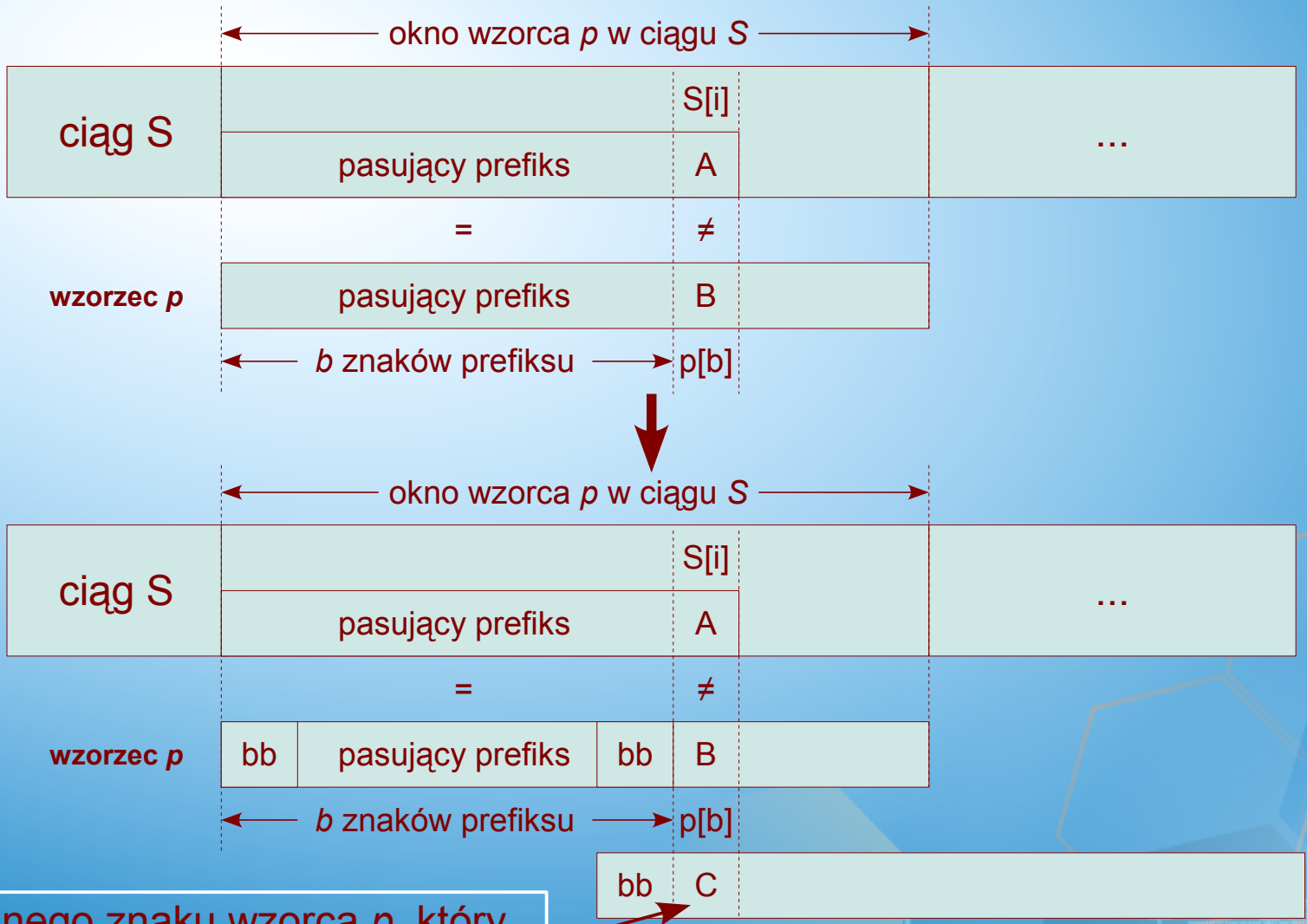
**Algorytm Morrisa-Pratta –** służy do wyszukiwania wszystkich wystąpień wzorca *p* w ciągu *S*.

Proces dopasowania:



**Algorytm Morrisa-Pratta –** służy do wyszukiwania wszystkich wystąpień wzorca *p* w ciągu *S*.

Proces dopasowania:




Pozycja kolejnego znaku wzorca *p*, który należy porównać z ciągiem *S* wynosi Π[*b*].

**Algorytm Morrisa-Pratta –** służy do wyszukiwania wszystkich wystąpień wzorca *p* w ciągu *S*.

Pseudokod algorytmu:

```
wyznaczenie tablicy Π[ ];
pp := -1;                                          // pozycja wzorca w ciągu
b := 0;                                            // długość prefiso-sufiksu
for i := 0 to |S| - 1 do begin                     // S – przeszukiwany ciąg
   while (b > -1) and (p[b] ≠ S[i]) do b := Π[b];  // p – poszukiwany wzorzec
   b := b + 1;
   if b = |p| then begin
      pp := i – b + 1;
      wypisanie pp;
      b := Π[b];
   end;
   if pp = -1 then KONIEC;
end;
```

**Algorytm Morrisa-Pratta –** służy do wyszukiwania wszystkich wystąpień wzorca *p* w ciągu *S*.

Pseudokod algorytmu:

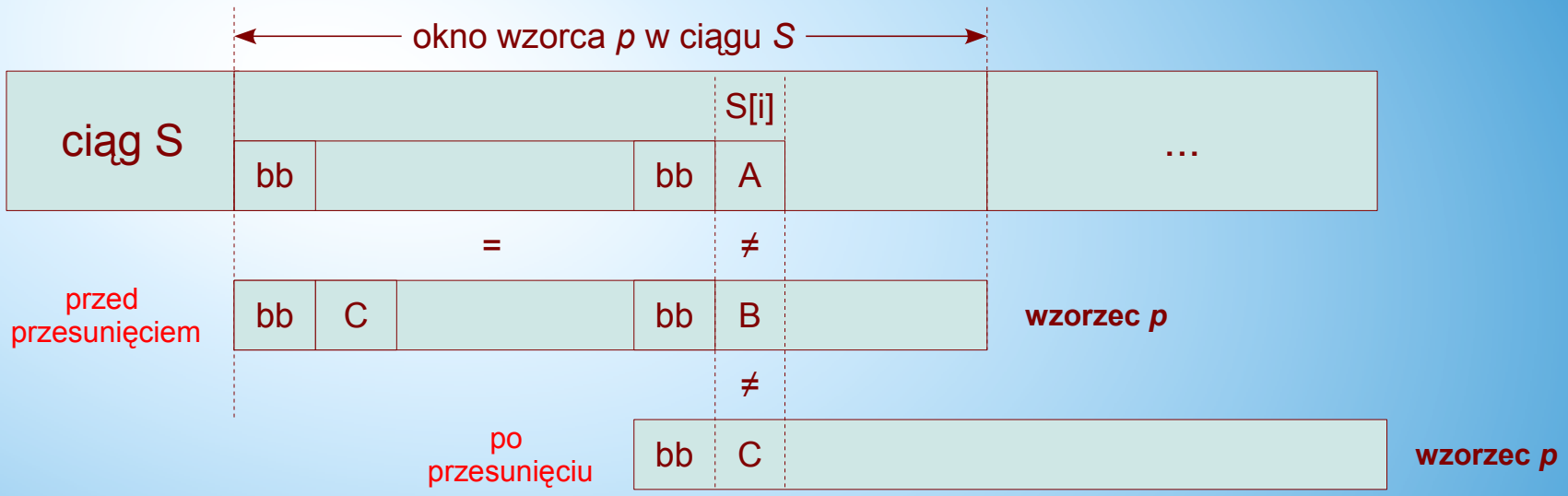
```
wyznaczenie tablicy Π[ ];
pp := -1;                                          // pozycja wzorca w ciągu
b := 0;                                            // długość prefiso-sufiksu
for i := 0 to |S| - 1 do begin                     // S – przeszukiwany ciąg
   while (b > -1) and (p[b] ≠ S[i]) do b := Π[b];  // p – poszukiwany wzorzec
   b := b + 1;
   if b = |p| then begin
      pp := i – b + 1;
      wypisanie pp;
      b := Π[b];
   end;
   if pp = -1 then KONIEC;
end;
```

Uwagi:

- Cechą charakterystyczną algorytmu MP jest brak cofania indeksu *i*. Dzięki tej własności algorytm umożliwia przetwarzanie danych sekwencyjnych – np. wyszukiwanie położenia wzorca w dużym pliku, który może nie mieścić się w pamięci operacyjnej komputera.

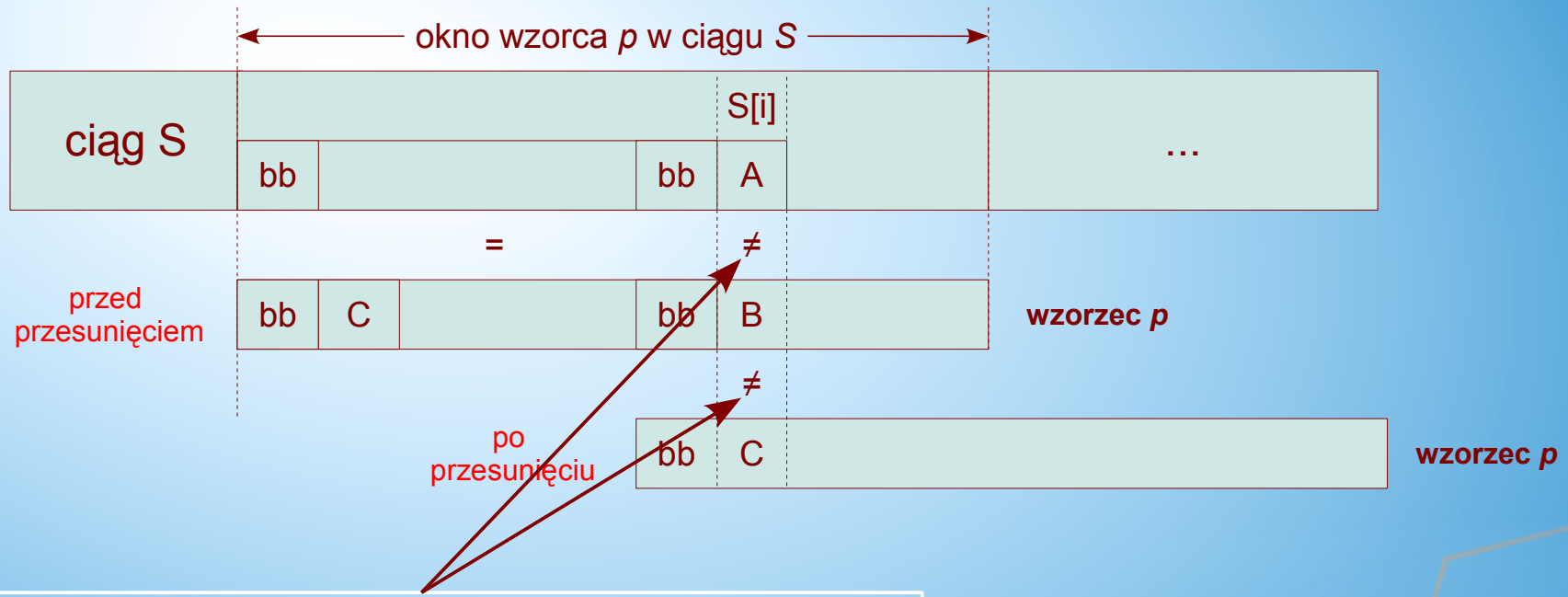
**Algorytm Knutha-Morrisa-Pratta –** jest ulepszeniem algorytmu Morrisa-Pratta.

Proces dopasowania:



**Algorytm Knutha-Morrisa-Pratta –** jest ulepszeniem algorytmu Morrisa-Pratta.

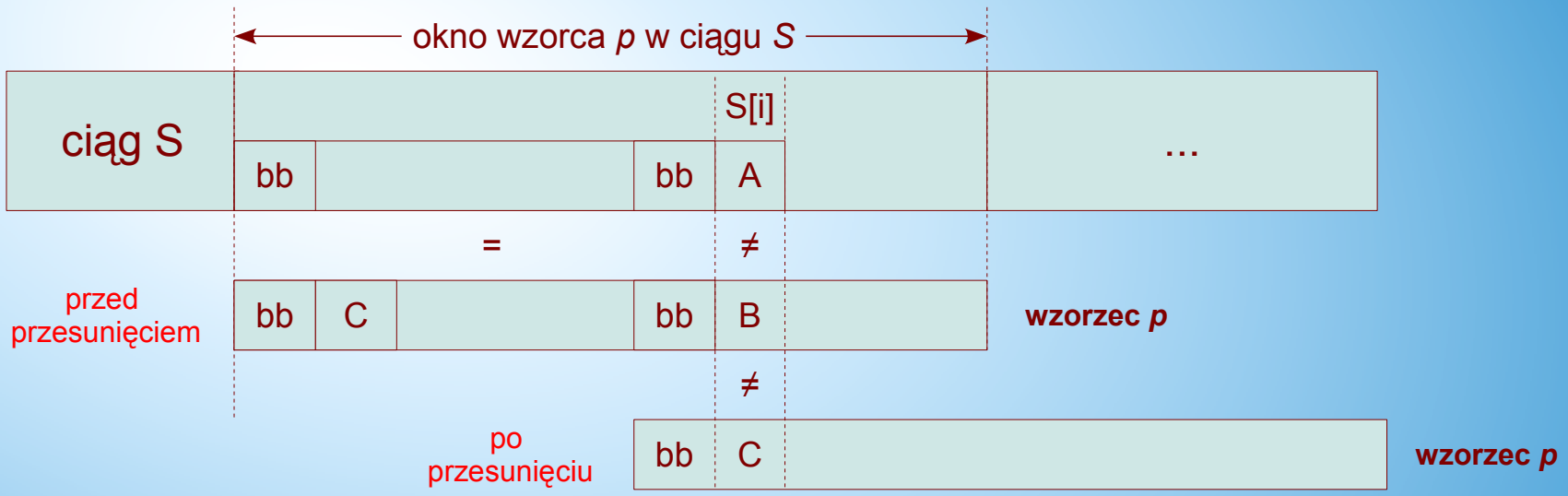
Proces dopasowania:




Po przesunięciu wzorca może wystąpić sytuacja, w której od razu będzie niezgodność znaków.

**Algorytm Knutha-Morrisa-Pratta –** jest ulepszeniem algorytmu Morrisa-Pratta.

Proces dopasowania:




Modyfikacja algorytmu MP:
Polega na zmianie sposobu wyznaczania tablicy Π[ ], w której zostaną umieszczone maksymalne szerokości prefikso-sufiksów prefiksu wzorca *p*. Jeśli dany prefikso-sufiks nie istnieje (nawet prefikso-sufiks pusty), to element tablicy ma wartość -1. Pozostała część algorytmu nie zmienia się.

**Algorytm Knutha-Morrisa-Pratta –** jest ulepszeniem algorytmu Morrisa-Pratta.

Wyznaczanie tablicy KMPNext[ ] (Π[ ]):

```
KMPNext[0] := -1;
b := -1;                          // długość prefikso-sufiksu
for i := 1 to |p| do              // p - wzorzec
begin
   while (b > -1) and (p[b] ≠ p[i-1]) do b := KMPNext[b];
   b := b + 1;
   if (i = |p|) or (p[i] ≠ p[b]) then KMPNext[i] := b
   else KMPNext[i] := KMPNext[b];
end;
```

Uwagi:

- złożoność obliczeniowa wynosi O(*n*+*m*), jednak algorytm KMP jest efektywniejszy niż MP,
- w algorytmie MP w miejsce odwołań do tablicy Π[ ] należy użyć odwołań do tablicy KMPNext[ ].

**Algorytm Karpa-Rabina –** jest algorytmem służącym do wyszukiwania wzorca w tekście.

Cechy:

- algorytm przeszukuje wszystkie możliwe podciągi ciągu *S*, wykorzystując funkcję mieszającą *h*,
- złożoność obliczeniowa algorytmu wynosi O(*n*+*m*) – zależy głównie od złożoności funkcji mieszającej – najczęściej wykorzystuje się funkcje mieszającеje o złożoności O(1).

**Algorytm Karpa-Rabina –** jest algorytmem służącym do wyszukiwania wzorca w tekście.

Cechy:

- algorytm przeszukuje wszystkie możliwe podciągi ciągu *S*, wykorzystując funkcję mieszającą *h*,
- złożoność obliczeniowa algorytmu wynosi O(*n*+*m*) – zależy głównie od złożoności funkcji mieszającej – najczęściej wykorzystuje się funkcje mieszającеje o złożoności O(1).

Pseudokod algorytmu:

```
Hp := h(p);              // obliczenie wartości funkcji mieszającej dla wzorca p
Hs := h(S[0..|p|]);      // obliczenie wartości funkcji mieszającej dla ciągu S
i := 0;
repeat
   if Hs ≠ Hp then i := i + 1
   else
      if p = S[i..i + |p|] then wypisz_pozycje_wzorca;
   Hs := h(S[i..i + |p|]);
until i = |S| – |p|;
```

**Algorytm Karpa-Rabina –** jest algorytmem służącym do wyszukiwania wzorca w tekście.

Cechy:

- algorytm przeszukuje wszystkie możliwe podciągi ciągu *S*, wykorzystując funkcję mieszającą *h*,
- złożoność obliczeniowa algorytmu wynosi O(*n*+*m*) – zależy głównie od złożoności funkcji mieszającej – najczęściej wykorzystuje się funkcje mieszającеje o złożoności O(1).

Pseudokod algorytmu:

```
Hp := h(p);              // obliczenie wartości funkcji mieszającej dla wzorca p
Hs := h(S[0..|p|]);      // obliczenie wartości funkcji mieszającej dla ciągu S
i := 0;
repeat
   if Hs ≠ Hp then i := i + 1
   else
      if p = S[i..i + |p|] then wypisz_pozycje_wzorca;
   Hs := h(S[i..i + |p|]);
until i = |S| – |p|;
```

Uwagi:

- najczęściej funkcja *h* pozwala na wyznaczenie wartości *h*(*y*+1) na podstawie już znanej wartości *h*(*y*) – w praktyce tekst traktuje się jako liczbę o podstawie równej liczbie znaków w alfabecie.

**Odległość Levenshteina –** jest to miara odmienności napisów.

Definicje:

- *działanie proste na napisie* – wstawienie nowego znaku, usunięcie znaku, zamiana znaku na inny znak,
- *odległość pomiędzy dwoma napisami* – jest to najmniejsza liczba działań prostych zmieniająca jeden napis w drugi.

**Odległość Levenshteina –** jest to miara odmienności napisów.

Definicje:

- *działanie proste na napisie* – wstawienie nowego znaku, usunięcie znaku, zamiana znaku na inny znak,
- *odległość pomiędzy dwoma napisami* – jest to najmniejsza liczba działań prostych zmieniająca jeden napis w drugi.

Przykłady:

| | | |
|---|---|---|
| ciąg | ciąg | – odległość równa 0 (brak czynności koniecznych do przekształcenia napisów), |
| śnieg | śniąg | – odległość równa 1 – zamiana znaku na inny, |
| podciąg | pdcięg | – odległość równa 2 – wstawienie znaku, zamiana znaku na inny. |

**Odległość Levenshteina –** jest to miara odmienności napisów.

Definicje:

- *działanie proste na napisie* – wstawienie nowego znaku, usunięcie znaku, zamiana znaku na inny znak,
- *odległość pomiędzy dwoma napisami* – jest to najmniejsza liczba działań prostych zmieniająca jeden napis w drugi.

Przykłady:

| | | |
|---|---|---|
| ciąg | ciąg | – odległość równa 0 (brak czynności koniecznych do przekształcenia napisów), |
| śnieg | śniąg | – odległość równa 1 – zamiana znaku na inny, |
| podciąg | pdcięg | – odległość równa 2 – wstawienie znaku, zamiana znaku na inny. |

Posługując się odległością Levenshteina można określić podobieństwo wyrazów: $P = \frac{1}{O_L + 1}$, gdzie:

$O_L$ – jest wyliczoną odległością edycyjną dwóch wyrazów.

**Odległość Levenshteina –** jest to miara odmienności napisów.

Definicje:

- *działanie proste na napisie* – wstawienie nowego znaku, usunięcie znaku, zamiana znaku na inny znak,
- *odległość pomiędzy dwoma napisami* – jest to najmniejsza liczba działań prostych zmieniająca jeden napis w drugi.

Przykłady:

| | | |
|---|---|---|
| ciąg | ciąg | – odległość równa 0 (brak czynności koniecznych do przekształcenia napisów), |
| śnieg | śniąg | – odległość równa 1 – zamiana znaku na inny, |
| podciąg | pdcięg | – odległość równa 2 – wstawienie znaku, zamiana znaku na inny. |

Posługując się odległością Levenshteina można określić podobieństwo wyrazów: $P = \frac{1}{O_L + 1}$, gdzie:

$O_L$ – jest wyliczoną odległością edycyjną dwóch wyrazów.

Zastosowanie:

- w przetwarzaniu informacji, głównie do określania podobieństw napisów (tzw. odległość edycyjna) – w korekcie pisowni, do generowania podpowiedzi, itp.

**Odległość Levenshteina –** jest to miara odmienności napisów.

Algorytm:

1. Dla dowolnych liter napisu funkcja $c(a,b)$ przyjmuje wartości: 0 – gdy $a = b$, 1 – gdy $a \neq b$.
2. Dla dwóch napisów: $s_1$ o długości $m$ i $s_2$ o długości $n$ tworzy się tablicę $d$ o wymiarach $[m + 1, n + 1]$, w której wpisuje się liczby wyznaczane wg następujących zasad:
   a. $d[i, 0] = i$ – pierwszy wiersz tabeli $d$, $i = 0..m$,
   b. $d[0, j] = j$ – pierwsza kolumna tabeli $d$, $j = 0..n$,
   c. $d[i, j] = min\{d[i - 1, j] + 1,$ // operacja usuwania znaku
      $d[i, j - 1] + 1,$ // operacja wstawiania znaku
      $d[i - 1, j - 1] + c(s_1[i], s_2[j])\}$ // operacja zamiany znaków
3. Po wykonaniu obliczeń odległość edycyjna będzie znajdowała się w $d[m, n]$.

**Odległość Levenshteina –** jest to miara odmienności napisów.

Algorytm:

1. Dla dowolnych liter napisu funkcja $c(a,b)$ przyjmuje wartości: 0 – gdy $a = b$, 1 – gdy $a \neq b$.
2. Dla dwóch napisów: $s_1$ o długości $m$ i $s_2$ o długości $n$ tworzy się tablicę $d$ o wymiarach $[m + 1, n + 1]$, w której wpisuje się liczby wyznaczane wg następujących zasad:
   a. $d[i, 0] = i$ – pierwszy wiersz tabeli $d$, $i = 0..m$,
   b. $d[0, j] = j$ – pierwsza kolumna tabeli $d$, $j = 0..n$,
   c. $d[i, j] = min\{d[i - 1, j] + 1$ // operacja usuwania znaku
      $d[i, j - 1] + 1$ // operacja wstawiania znaku
      $d[i - 1, j - 1] + c(s_1[i], s_2[j])\}$ // operacja zamiany znaków
3. Po wykonaniu obliczeń odległość edycyjna będzie znajdowała się w $d[m, n]$.

Przykład:

$s_1$ = „wykład”
$s_2$ = „wkład”

| | | w | y | k | ł | a | d |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| w | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| k | 2 | 1 | 1 | 1 | 2 | 3 | 4 |
| ł | 3 | 2 | 3 | 4 | 1 | 2 | 3 |
| a | 4 | 3 | 3 | 4 | 2 | 1 | 2 |
| d | 5 | 4 | 5 | 4 | 3 | 2 | 1 |

**Odległość Levenshteina –** jest to miara odmienności napisów.

Algorytm:

1. Dla dowolnych liter napisu funkcja $c(a,b)$ przyjmuje wartości: 0 – gdy $a = b$, 1 – gdy $a \neq b$.
2. Dla dwóch napisów: $s_1$ o długości $m$ i $s_2$ o długości $n$ tworzy się tablicę $d$ o wymiarach $[m + 1, n + 1]$, w której wpisuje się liczby wyznaczane wg następujących zasad:
   a. $d[i, 0] = i$ – pierwszy wiersz tabeli $d$, $i = 0..m$,
   b. $d[0, j] = j$ – pierwsza kolumna tabeli $d$, $j = 0..n$,
   c. $d[i, j] = min\{d[i - 1, j] + 1,$ // operacja usuwania znaku
   $d[i, j - 1] + 1,$ // operacja wstawiania znaku
   $d[i - 1, j - 1] + c(s_1[i], s_2[j])\}$ // operacja zamiany znaków
3. Po wykonaniu obliczeń odległość edycyjna będzie znajdowała się w $d[m, n]$.

Przykład:

$s_1$ = „wykład”
$s_2$ = „wkład”

| | | w | y | k | ł | a | d |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| w | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| k | 2 | 1 | 1 | 1 | 2 | 3 | 4 |
| ł | 3 | 2 | 3 | 4 | 1 | 2 | 3 |
| a | 4 | 3 | 3 | 4 | 2 | 1 | 2 |
| d | 5 | 4 | 5 | 4 | 3 | 2 | 1 |

Odległość $O_L = 1$
Podobieństwo wyrazów $P = 0.5$

# Koniec wykładu