MINUT 2024 (6), s. 70–80
ISSN 2719-3063

Marta KUCHARCZYK[1], Bożena WIECZOREK[2]

[1]Studentka, Wydział Automatyki, Elektroniki i Informatyki, Politechnika Śląska, ul.Akademicka 16, 44-100 Gliwice

[2]Katedra Algorytmiki i Oprogramowania, Wydział Automatyki, Elektroniki i Informatyki, Politechnika Śląska, ul. Akademicka 16, 44-100 Gliwice

# Algorytmy wyszukiwania wzorca w tekście

**Streszczenie**. Algorytmy wyszukiwania wzorca w tekście odgrywają kluczową rolę w wielu dziedzinach informatyki, takich jak przetwarzanie języka naturalnego, wyszukiwanie informacji czy bioinformatyka. W artykule przedstawione zostały dwa zaawansowane algorytmy wyszukiwania wzorca: Knutha-Morrisa-Pratta (KMP) oraz Rabina-Karpa.

**Słowa kluczowe:** wyszukiwanie wzorca, algorytm KMP, algorytm Rabina-Karpa, funkcje skrótu.

## 1. Wstęp

Wyszukiwanie wzorca w tekście to zadanie informatyczne o szerokim spektrum zastosowań, od kompresji danych i bioinformatyki po edytory tekstu, wyszukiwarki internetowe i systemy rozpoznawania mowy. Wraz ze wzrostem objętości danych znaczenia nabiera efektywne wyszukiwanie wzorców, minimalizujące czas i zasoby obliczeniowe potrzebne do wykonania zadania. Dwa z najbardziej znanych, zaawansowanych algorytmów wyszukiwania wzorca to algorytm Knutha-Morrisa-Pratta (KMP) oraz algorytm Rabina-Karpa. Oba podejścia oferują znaczną poprawę wydajności w porównaniu z prostym, naiwnym algorytmem, który polega na porównywaniu każdego znaku wzorca z każdym znakiem tekstu.

Algorytm KMP został opracowany przez Donalda Knutha, Jamesa H. Morrisa oraz Vaughan R. Pratta w 1977 roku [3]. Jego główną zaletą jest efektywność czasowa, ponieważ działa w czasie liniowym względem długości tekstu i wzorca, nawet w przypadku pesymistycznym. KMP osiąga to dzięki zastosowaniu analizy wzorca przed rozpoczęciem wyszukiwania, co pozwala na uniknięcie zbędnych porównań.

Z kolei algorytm Rabina-Karpa, zaproponowany przez Michaela Rabina i Richarda Karpa w 1987 roku, wykorzystuje funkcję skrótu do wyszukiwania wzorca [2]. Przetwarzając ciąg znaków na postać numeryczną (skrót), algorytm bazuje na założeniu, że jeżeli dwa teksty są identyczne, to ich skróty również będą takie same.

W dalszej części artykułu szczegółowo omówione zostaną oba algorytmy. Zaprezentowane zostanie ich działanie i przeanalizowana złożoność czasowa. Wykonane zostaną eksperymenty dla przykładowych

Autor korespondencyjny: B. Wieczorek (Bozena.Wieczorek@polsl.pl).
Data wpłynięcia: 05.07.2024.

implementacji algorytmów, co pozwoli na porównanie czasów działania obu metod podczas rozwiązywania konkretnego zadania. Stosowane będą następujące oznaczenia: $t$ – tekst, $n$ – długość tekstu, $w$ – wzorzec, $m$ – długość wzorca, $i$ – pozycja w tekście, $j$ – pozycja we wzorcu.

# 2. Algorytm Knutha-Morrisa-Pratta

## 2.1. Podstawowe pojęcia

W celu omówienia algorytmu KMP, należy przedstawić kilka definicji kluczowych w tej metodzie [1].
**Prefiks** – podsłowo (fragment ciągu znaków) znajdujące się na początku słowa. Przykładowo, w wyrazie *abcd* prefiksem może być sama litera *a*, ale również mogą być ciągi: *ab*, *abc* lub *abcd*.
**Sufiks** – podsłowo, znajdujące się na końcu danego słowa. Dla przykładu, w słowie *abcd* sufiksem może być sama litera *d*, ale również mogą być ciągi: *cd*, *bcd* lub cały wyraz *abcd*.
**Prefikso-sufiks** – fragment słowa, będący zarówno prefiksem jak i sufiksem danego ciągu znaków. Przykładowo, w wyrazie *abcdab* prefikso-sufiksem będzie ciąg *ab*, gdyż jest zarówno na początku, jak i na końcu słowa.
**Właściwy prefikso-sufiks** – prefikso-sufiks, który nie jest równy całemu wyrazowi.

## 2.2. Szczegółowy opis algorytmu

Algorytm składa się z dwóch etapów: budowy tablicy prefiksów oraz właściwego znajdowania wzorca w tekście.

**Budowa tablicy prefiksów**

Dla wzorca $w[0 \ldots m-1]$ tablica prefiksów $pref$ jest tablicą pomocniczą, w której $pref[j]$ oznacza długość najdłuższego właściwego prefikso-sufiksu dla $w[0 \ldots j]$. Pozwala ona, w drugim etapie algorytmu, na przesunięcie pozycji we wzorcu tak, aby uniknąć porównywania już wcześniej dopasowanych znaków. Wyznaczanie tablicy prefiksów dla wzorca *abababca* zostało przedstawione w tabeli 1.

Tabela 1. Wyznaczanie tablicy prefiksów

| $j$ | $w[0 \ldots j]$ | *Prefikso-sufiks* | $pref[j]$ |
|---|---|---|---|
| 0 | *a* | – | 0 |
| 1 | *ab* | – | 0 |
| 2 | *aba* | *a* | 1 |
| 3 | *abab* | *ab* | 2 |
| 4 | *ababa* | *aba* | 3 |
| 5 | *ababab* | *abab* | 4 |
| 6 | *abababc* | – | 0 |
| 7 | *abababca* | *a* | 1 |

**Znajdowanie wzorca w tekście**

W drugim etapie korzystamy z poprzednio utworzonej tablicy prefiksowej. Rozpoczynamy od ustawienia początkowej pozycji we wzorcu i tekście na wartość 0. Następnie zaczynamy porównywać znaki tekstu ze znakami wzorca rozważając trzy przypadki:

- jeżeli aktualny znak tekstu jest zgodny ze znakiem wzorca ($t[i] == w[j]$), to przesuwamy oba wskaźniki $i$ oraz $j$ o jedną pozycję w prawo,

- jeżeli aktualny znak tekstu nie jest zgodny ze znakiem wzorca i istnieje już częściowe dopasowanie wzorca i tekstu ($j > 0$), to przesuwamy $j$ do pozycji wyznaczonej przez tablicę prefiksową, czyli na wartość najdłuższego poprzedniego prefikso-sufiksu ($j = pref[j-1]$); jeżeli $j$ jest równe 0, to przesuwamy się w tekście o jedną pozycję w prawo,
- jeżeli wskaźnik $j$ osiągnie długość wzorca, to oznacza, że udało się znaleźć wzorzec w tekście; możemy wyznaczyć pozycję dopasowania oraz kontynuować wyszukiwanie dalszych dopasowań przesuwając $j$ do określonej pozycji odczytanej z tablicy prefiksów.

Załóżmy, że dany jest tekst *abababababcaabaaababababca* i wzorzec *abababca*. Kolejne kroki drugiego etapu algorytmu KMP zostały przedstawione poniżej.
**Rozpoczynamy od** $i = 0, j = 0$,

$$t[i] = a, w[j] = a, \text{zgodny} \Rightarrow i = 1, j = 1$$
$$t[i] = b, w[j] = b, \text{zgodny} \Rightarrow i = 2, j = 2$$
$$t[i] = a, w[j] = a, \text{zgodny} \Rightarrow i = 3, j = 3$$
$$t[i] = b, w[j] = b, \text{zgodny} \Rightarrow i = 4, j = 4$$
$$t[i] = a, w[j] = a, \text{zgodny} \Rightarrow i = 5, j = 5$$
$$t[i] = b, w[j] = b, \text{zgodny} \Rightarrow i = 6, j = 6$$
$$t[i] = a, w[j] = c, \text{niezgodny} \Rightarrow j = pref[5] = 4$$

$i = 6, j = 4$ – **znaki wzorca** $w[0 \ldots 3]$ **pasują już do tekstu**,

$$t[i] = a, w[j] = a, \text{zgodny} \Rightarrow i = 7, j = 5$$
$$t[i] = b, w[j] = b, \text{zgodny} \Rightarrow i = 8, j = 6$$
$$t[i] = a, w[j] = c, \text{niezgodny} \Rightarrow j = pref[5] = 4$$

$i = 8, j = 4$ – **znaki wzorca** $w[0 \ldots 3]$ **pasują już do tekstu**.

$$t[i] = a, w[j] = a, \text{zgodny} \Rightarrow i = 9, j = 5$$
$$t[i] = b, w[j] = b, \text{zgodny} \Rightarrow i = 10, j = 6$$
$$t[i] = c, w[j] = c, \text{zgodny} \Rightarrow i = 11, j = 7$$
$$t[i] = a, w[j] = a, \text{zgodny} \Rightarrow i = 12, j = 8$$

**Wzorzec został znaleziony na pozycji** $4$**, zaczynamy od** $i = 12$ **i** $j = pref[7] = 1$.

$$t[i] = a, w[j] = b, \text{ niezgodny} \Rightarrow j = pref[0] = 0$$
$$i = 12, j = 0$$
$$t[i] = a, w[j] = a, \text{zgodny} \Rightarrow i = 13, j = 1$$
$$t[i] = b, w[j] = b, \text{zgodny} \Rightarrow i = 14, j = 2$$
$$t[i] = a, w[j] = a, \text{zgodny} \Rightarrow i = 15, j = 3$$
$$t[i] = a, w[j] = b, \text{niezgodny} \Rightarrow j = pref[2] = 1$$
$$i = 15, j = 1$$

$$t[i] = a, w[j] = b, \text{niezgodny} \Rightarrow j = pref[0] = 0$$
$$i = 15, j = 0$$
$$t[i] = a, w[j] = a, \text{zgodny} \Rightarrow i = 16, j = 1$$
$$t[i] = a, w[j] = b, \text{niezgodny} \Rightarrow j = pref[0] = 0$$
$$i = 16, j = 0$$
$$t[i] = a, w[j] = a, \text{zgodny} \Rightarrow i = 17, j = 1$$

Kolejne znaki wzorca pasują do znaków tekstu i znajdujemy dopasowanie na pozycji 16.

## 2.3. Przykładowa implementacja

Przykładową implementację algorytmu KMP przedstawiono na listingu 1. Tablica prefiksów, wyznaczana w funkcji `tablicaPS`, przechowywana jest w kontenerze `std::vector<int>`.

```
#include <iostream>
#include <string>
#include <vector>

//tablica prefiksowa pref dla wzorca w
void tablicaPS(const std::string& w, std::vector<int>& pref)
{
  int j = 0; // początkowa długość prefikso-sufiksu
  int m = w.length();
  pref.resize(m, 0);
  for (int k = 1; k < m; k++)
  {
    while (j > 0 && w[j] != w[k])
      j = pref[j - 1];

    // jeśli jest zgodność, zwiększ długość prefikso-sufiksu
    if (w[j] == w[k])
      j++;

    pref[k] = j; // aktualizujemy długość prefikso-sufiksu w tablicy
  }
}

//algorytm KMP
void KMP(const std::string& t, const std::string& w, const std::vector<int>& pref)
{
  tablicaPS(w, pref);
  int n = t.length();
  int m = w.length();
  int j = 0;
  for (int i = 0; i < n; i++)
  {
    while (j > 0 && w[j] != t[i])
      j = pref[j-1];

    if (w[j] == t[i])
      j++;

    if (j == m)
    {
```

```
41        std::cout << "Wystapienie wzorca na pozycji: " << i - m  + 1<< std::endl;
42        j = pref[j-1];
43      }
44    }
45  }
46  int main()
47  {
48    std::string tekst, wzorzec;
49    std::cout << "Wprowadz tekst:\n";
50    getline(std::cin, tekst);
51
52    std::cout << "Wprowadz wzorzec:\n";
53    getline(std::cin, wzorzec);
54
55    std::vector<int> pref;
56    KMP(tekst, wzorzec, pref);
57
58    return 0;
59  }
60
```

Listing 1. Algorytm KMP

### 2.4. Złożoność obliczeniowa

Algorytm Knutha-Morrisa-Pratta działa w czasie liniowym względem długości tekstu i wzorca – $O(n+m)$. Budowa tablicy prefiksowej zajmuje czas $O(m)$. Właściwe przeszukiwanie tekstu w celu znalezienia wszystkich wystąpień wzorca zajmuje czas $O(n)$, bowiem w każdym kroku algorytmu, gdy wystąpi niezgodność wzorca z tekstem, algorytm korzysta z tablicy prefiksowej, aby określić nową wartość $j$, nie zmieniając przy tym pozycji w tekście (wartości $i$).

## 3. Algorytm Rabina-Karpa

### 3.1. Ogólny opis algorytmu

W algorytmie Rabina-Karpa proces porównywania wzorca z tekstem jest realizowany przy użyciu funkcji skrótu, która przekształca ciąg znaków w wartość numeryczną zwaną skrótem (ang. hash) [4]. Jeśli dwa teksty są identyczne, ich skróty będą sobie równe. Jednak odwrotność tej zależności nie zawsze jest prawdziwa — możliwe jest, by dla dwóch różnych ciągów znaków funkcja zwróciła te same wartości. W algorytmie najpierw obliczana jest wartość skrótu dla wzorca, a następnie wyznaczane są skróty dla kolejnych podciągów tekstu, których długość wynosi $m$ (tzw. okno tekstu). Jeśli skrót okna jest równy skrótowi wzorca, algorytm dokonuje dokładnego porównania znak po znaku, aby upewnić się, że znaleziono rzeczywiste dopasowanie.

### 3.2. Funkcja skrótu

W celu przedstawienia ogólnej koncepcji algorytmu, zastosowana zostanie prosta funkcja skrótu, która zwraca sumę kodów ASCII wszystkich znaków danego fragmentu tekstu. Funkcja korzysta z wcześniej

obliczonych wartości, co skutkuje wyznaczaniem skrótów dla kolejnych okien tekstu w stałym czasie (algorytm postępującego skrótu). Funkcja *ord* zwraca numer porządkowy znaku w kodzie ASCII. Algorytm kończy działanie po znalezieniu pierwszego wystąpienia wzorca w tekście.
Załóżmy, że dany jest tekst *alamakota* i wzorzec *kot*. Kroki algorytmu zaprezentowane zostały poniżej.
**Obliczenie wartości skrótu dla wzorca:**

$$\begin{aligned} H(\text{kot}) &= ord('k') + ord('o') + ord('t') \\ &= 107 + 111 + 116 \\ &= 334 \end{aligned}$$

**Obliczenie skrótu dla pierwszego okna tekstu:**

$$\begin{aligned} H(\text{ala}) &= ord('a') + ord('l') + ord('a') \\ &= 97 + 108 + 97 \\ &= 302 \end{aligned}$$

**Przesunięcie i aktualizacja skrótu dla kolejnych okien:**

$$\begin{aligned} H(\text{lam}) &= H(\text{ala}) - ord('a') + ord('m') \\ &= 302 - 97 + 109 \\ &= 314 \end{aligned}$$

$$\begin{aligned} H(\text{ama}) &= H(\text{lam}) - ord('l') + ord('a') \\ &= 314 - 108 + 97 \\ &= 303 \end{aligned}$$

$$\begin{aligned} H(\text{mak}) &= H(\text{ama}) - ord('a') + ord('k') \\ &= 303 - 97 + 107 \\ &= 313 \end{aligned}$$

$$\begin{aligned} H(\text{ako}) &= H(\text{mak}) - ord('m') + ord('o') \\ &= 313 - 109 + 111 \\ &= 315 \end{aligned}$$

$$\begin{aligned} H(\text{kot}) &= K(\text{ako}) - ord('a') + ord('t') \\ &= 315 - 97 + 116 \\ &= 334 \end{aligned}$$

**Ponieważ skróty są zgodne, dokonujemy dokładnego porównania znak po znaku:**

$$\text{kot} == \text{kot} \quad \text{(dopasowanie znalezione)}$$

Funkcja skrótu zastosowana w powyższym przykładzie charakteryzuje się niską efektywnością, ponieważ podatna jest na kolizje. Dla różnych ciągów znaków możemy otrzymać tę samą wartość skrótu, przykładowo dla ciągów *abc* i *bbb* otrzymamy:

$$H(\text{abc}) = ord('a') + ord('b') + ord('c') = 97 + 98 + 99 = 294$$

$$H(\text{bbb}) = ord('b') + ord('b') + ord('b') = 98 + 98 + 98 = 294$$

W praktycznych zastosowaniach zaleca się użycie bardziej zaawansowanych funkcji skrótu. Najczęściej traktuje się ciąg znaków jako wielomian o pewnej podstawie i współczynnikach równych kodom poszczególnych znaków ciągu. Podstawą wielomianu jest wartość większa od maksymalnego współczynnika. Zaletą takiego podejścia jest znacznie mniejsze prawdopodobieństwo wystąpienia kolizji i szybkie wyznaczanie wartości skrótu dla kolejnych podciągów na podstawie wcześniejszych obliczeń. Aby obliczyć skrót dla kolejnego okna tekstu, używamy skrótu poprzedniego okna i aktualizujemy go, usuwając wpływ pierwszego znaku i dodając wpływ nowego znaku w oknie. Załóżmy, że przeszliśmy do kolejnego okna zaczynającego się na pozycji $j + 1$. Wartość skrótu obliczymy następująco:

$$H_{j+1} = (H_j - t[j] \cdot p^{m-1}) \cdot p + t[j + m]$$

Gdzie:

- $H_j$ jest skrótem poprzedniego okna,
- $t[j]$ to kod pierwszego znaku poprzedniego okna,
- $p^{m-1}$ to wartość podstawy wielomianu podniesionej do potęgi $m - 1$,
- $t[j + m]$ to kod ostatniego znaku aktualnego okna,

Załóżmy, że dany jest tekst *alamakota* i wzorzec *kot*. Poniżej przedstawione zostaną obliczenia wartości skrótów dla wzorca i okien tekstu. W celu uproszczenia obliczeń, literom przypisano kody o wartościach od 1 (dla litery $'a'$) do 26 (dla $'z'$). Jako podstawę wielomianu $p$ przyjęto wartość 31 (liczba pierwsza większa od 26). Wykonywana operacja wyznaczania reszty z dzielenia całkowitego przez $M = 20011$ (`mod` 20011) pozwala na utrzymanie wartości skrótów w określonym zakresie, co jest kluczowe dla uniknięcia problemu przepełnienia. Wartość $M$ jest odpowiednio dobraną, dużą liczbą pierwszą.

**Obliczanie skrótu dla wzorca:**

$$\begin{aligned}
k &\to 11 \\
o &\to 15 \\
t &\to 20 \\
H(\text{kot}) &= (11 \cdot 31^2 + 15 \cdot 31^1 + 20 \cdot 31^0) \bmod 20011 \\
&= (11 \cdot 961 + 15 \cdot 31 + 20) \bmod 20011 \\
&= (10571 + 465 + 20) \bmod 20011 \\
&= 11056 \bmod 20011 \\
&= 11056
\end{aligned}$$

**Obliczanie skrótu dla pierwszego okna tekstu:**

$$
\begin{aligned}
a &\to 1 \\
l &\to 12 \\
a &\to 1 \\
H(\text{ala}) &= (1 \cdot 31^2 + 12 \cdot 31^1 + 1 \cdot 31^0) \bmod 20011 \\
&= (31^2 + 12 \cdot 31 + 1) \bmod 20011 \\
&= (961 + 372 + 1) \bmod 20011 \\
&= 1334 \bmod 20011 \\
&= 1334
\end{aligned}
$$

**Przesuwanie okna i obliczanie kolejnych skrótów na podstawie obliczeń w poprzednim kroku:**

$$
\begin{aligned}
m &\to 13 \\
H(\text{lam}) &= ((H(\text{ala}) - 1 \cdot 31^2) \cdot 31 + 13) \bmod 20011 \\
&= ((1334 - 961) \cdot 31 + 13) \bmod 20011 \\
&= 11576 \bmod 20011 \\
&= 11576
\end{aligned}
$$

$$
\begin{aligned}
H(\text{ama}) &= ((H(\text{lam}) - 12 \cdot 31^2) \cdot 31 + 1) \bmod 20011 \\
&= ((11576 - 11532) \cdot 31 + 1) \bmod 20011 \\
&= 1365 \bmod 20011 \\
&= 1365
\end{aligned}
$$

$$
\begin{aligned}
H(\text{mak}) &= ((H(\text{ama}) - 1 \cdot 31^2) \cdot 31 + 11) \bmod 20011 \\
&= ((1365 - 961) \cdot 31 + 11) \bmod 20011 \\
&= 12535 \bmod 20011 \\
&= 12535
\end{aligned}
$$

$$
\begin{aligned}
H(\text{ako}) &= ((H(\text{mak}) - 13 \cdot 31^2) \cdot 31 + 15) \bmod 20011 \\
&= ((12535 - 12493) \cdot 31 + 15) \bmod 20011 \\
&= 1317 \bmod 20011 \\
&= 1317
\end{aligned}
$$

$$
\begin{aligned}
H(\text{kot}) &= ((H(\text{ako}) - 1 \cdot 31^2) \cdot 31 + 20) \bmod 20011 \\
&= ((1317 - 961) \cdot 31 + 20) \bmod 20011 \\
&= 11056 \bmod 20011 \\
&= 11056
\end{aligned}
$$

Wartości skrótów są równe, więc porównujemy okno tekstu ze wzorcem znak po znaku. Wzorzec jest zgodny z fragmentem tekstu, więc znaleźliśmy dopasowanie na pozycji 5.

### 3.3. Przykładowa implementacja

Listing 2 przedstawia przykładową implementację algorytmu Rabina-Karpa. Jako podstawę wielomianu $p$ przyjęto wartość 127 zakładając, że znaki tekstu i wzorca należą do podstawowego kodu ASCII. Wartość $M = 1000000007$ jest jedną z możliwych liczb pierwszych, które pozwalają utrzymać wartości skrótów w zakresie liczby całkowitej zapisanej przy użyciu czterech bajtów. Podczas wyznaczania skrótów kolejnych okien tekstu używana jest zmienna `przesuniecie`, która w wyniku operacji odejmowania może przyjmować wartości ujemne. Aby uniknąć obliczania reszty z dzielenia całkowitego liczby ujemnej, wartość zmiennej `przesuniecie` jest iteracyjnie powiększana o $M$ aż do uzyskania wartości nieujemnej.

```
#include <iostream>
#include<string>
void Rabin_Karp(const std::string& t, const std::string& w)
{
  long long H_t = 0LL, H_w= 0LL;          //skróty dla okna tekstu i wzorca
  long long p = 127LL, M = 1e9+7LL;
  long long p_n = 1LL;
  int n = t.length();
  int m = w.length();
  //wyznaczenie wartości skrótu dla wzorca
  for (int i = 0; i < m; i++)
    H_w = (H_w * p + w[i]) % M;

  //wyznaczenie p^(m-1) zgodnie z arytmetyką modularną
  for (int i = 1; i < m; i++)
    p_n = (p_n * p) % M;

  //Szukanie wzorca w tekście
  //Wyznaczenie skrótu pierwszego okna tekstu
  for (int i = 0; i < m;  i++)
    H_t = (H_t * p + t[i]) % M;

  int i = 0;
  do
  {
    if (H_w == H_t)
    {
      //porównujemy znak po znaku
      int k = i;
      while (k - i < m && t[k] == w[k - i])
        k++;
      if(k-i == m)
        std::cout << "Wystapienie wzorca na pozycji: " << i << std::endl;
    }

    //wyznaczenie skrótu następnego okna
    long long przesuniecie = H_t - t[i] * p_n;

    while (przesuniecie < 0)
      przesuniecie += M;

```

```
42      H_t = (przesuniecie * p + t[i + m]) % M;
43      i++;
44    } while (i <= n - m);
45  }
46  int main()
47  {
48    std::string tekst, wzorzec;
49    std::cout << "Wprowadz tekst:\n";
50    getline(std::cin, tekst);
51    std::cout << "Wprowadz wzorzec:\n";
52    getline(std::cin, wzorzec);
53    Rabin_Karp(tekst, wzorzec);
54  }
```

Listing 2. Algorytm Rabina-Karpa

### 3.4. Złożoność obliczeniowa

Obliczenie wartości skrótu dla wzorca i początkowego fragmentu tekstu ma złożoność $O(m)$. Wyznaczanie skrótów dla kolejnych okien zajmuje stały czas $O(1)$, więc dla tekstu o długości $n$ wykonujemy $O(n)$ aktualizacji. W najgorszym przypadku wartości skrótów dla wszystkich przesunięć będą się równały, co wymaga dokładnego porównania $m$ znaków dla każdego przesunięcia. W takim przypadku, złożoność czasowa wynosi $O(mn)$. W rzeczywistości, jeśli funkcja skrótu jest odpowiednio dobrana, prawdopodobieństwo kolizji jest niskie i dokładne porównanie znak po znaku wykonuje się rzadko. Średnia złożoność czasowa algorytmu Rabina-Karpa wynosi więc $O(n+m)$.

## 4. Przeprowadzone eksperymenty

W celu porównania czasów wykonania algorytmów wyszukiwania wzorca w tekście zaimplementowano algorytm naiwny, którego działanie polega na porównywaniu wzorca ze wszystkimi możliwymi podciągami tekstu, znak po znaku. Przeprowadzono eksperymenty dla tekstu o długości $n = 100000$ znaków oraz dwóch wzorców: krótkiego o długości $m = 40$ oraz długiego o długości $m = 400$ znaków. Wzorzec umieszczono jednokrotnie na końcu tekstu.

Najpierw sprawdzono przypadek pesymistyczny dla algorytmu naiwnego, czyli wygenerowano tekst i wzorce, które składają się z samych liter $a$ i zakończone są literą $b$. W tabeli 2 zamieszone zostały wyniki eksperymentów. Podane czasy są minimalnymi wartościami uzyskanymi w 5 próbach.

Tabela 2. Czasy działania algorytmów dla przypadku pesymistycznego

| $m$ | Algorytm KMP [ms] | Algorytm Rabina-Karpa [ms] | Algorytm naiwny [ms] |
|---|---|---|---|
| 40 | 37 | 17 | 487 |
| 400 | 37 | 15 | 4751 |

W kolejnym eksperymencie tekst i wzorce składały się z różnych znaków. Minimalne czasy z 5 prób pokazano w tabeli 3.

Wyniki eksperymentów potwierdziły większą efektywność zaprezentowanych algorytmów. Szczególnie w przypadku pesymistycznym czas działania algorytmu naiwnego jest znacząco dłuższy - ponad stokrotnie

Tabela 3. Czasy działania algorytmów dla przypadku średniego

| $m$ | Algorytm KMP [ms] | Algorytm Rabina-Karpa [ms] | Algorytm naiwny [ms] |
|---|---|---|---|
| 40 | 26 | 13 | 56 |
| 400 | 24 | 14 | 492 |

dla $m = 400$. Algorytm Rabina-Karpa okazał się najlepszy dla zastosowanych danych testowych uzyskując najkrótsze czasy we wszystkich próbach.

## 5. Podsumowanie

W niniejszym artykule szczegółowo omówione zostały dwa algorytmy wyszukiwania wzorca w tekście: Knutha-Morrisa-Pratta (KMP) oraz Rabina-Karpa. Algorytm KMP, dzięki swojej złożoności czasowej $O(n + m)$, zapewnia stabilną i przewidywalną wydajność. Jego mechanizm przetwarzania pozwala na skuteczne pomijanie niepotrzebnych porównań, co czyni go wysoce efektywnym w wielu praktycznych zastosowaniach. Algorytm Rabina-Karpa wyróżnia się zastosowaniem funkcji skrótu, dzięki której możliwe jest szybkie porównanie podciągów tekstu. Algorytm ten uzyskał najlepsze czasy działania we wszystkich przeprowadzonych eksperymentach. Oba algorytmy oferują znaczące usprawnienia w porównaniu do bardziej podstawowych metod, takich jak algorytm naiwny, co potwierdzają wyniki eksperymentów.

Oprócz omówionych algorytmów, istnieje wiele innych interesujących metod wyszukiwania wzorca. Algorytm Boyera-Moore'a wykorzystuje heurystyki, aby przesuwać wzorzec o większą liczbę pozycji. Drzewa sufiksowe pozwalają na efektywne wyszukiwanie wszystkich wystąpień wzorca w tekście. Automaty Aho-Corasicka służą do wyszukiwania wielu wzorców jednocześnie, co jest przydatne w wielu praktycznych zastosowaniach. Algorytm Shift-Or wykorzystuje operacje bitowe do szybkiego przetwarzania danych i działa szczególnie efektywnie, gdy wzorzec jest stosunkowo krótki. Każdy z tych algorytmów ma swoje unikalne zalety i zastosowania, co czyni dziedzinę wyszukiwania wzorca w tekście niezwykle interesującą.

## Literatura

1. T. H. Cormen, C. E. Leiserson i in., *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012.
2. R. M. Karp, M. O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development, vol. 31(2), 1987, p. 249 - 260.
3. D. Knuth, J. H. Morris, V. Pratt, *Fast pattern matching in strings*, SIAM Journal on Computing, vol. 6(2), 1977, p. 323 - 350.
4. A. Rasool, A. Tiwari at al., *String matching methodologies:a comparative analysis*, International Journal of Computer Science and Information Technologies, vol. 3(2), 2012, p. 3394 - 3397.