# 11. Wyszukiwanie wzorca w tekście

Problem wyszukiwania wzorca w tekście jest niezwykle istotny w zastosowaniach informatyki. Praktycznie wszystkie edytory tekstu są wyposażone w narzędzie, które pozwala znaleźć dany ciąg znaków w dokumencie. Szukanie informacji w internecie też często sprowadza się do wpisania w wyszukiwarce fragmentu tekstu. Oczekujemy wtedy, że wyniki nie tylko będą poprawne, lecz także pojawią się szybko. W tym temacie poznasz algorytmy związane z wyszukiwaniem wzorca w tekście.

## Cele lekcji

- Wyszukasz wzorzec w tekście z wykorzystaniem algorytmu naiwnego.
- Dowiesz się, na czym polega metoda haszowania.
- Poznasz podstawy arytmetyki modularnej.
- Zastosujesz funkcję haszującą do wyszukiwania wzorca w tekście.
- Poznasz algorytm Karpa–Rabina, służący do znajdowania wzorca w tekście.

Problem wyszukiwania wzorca w tekście

**Problem wyszukiwania wzorca w tekście** można sformułować w następujący sposób. Dane są dwa napisy: jeden nazwiemy tekstem, a drugi wzorcem. Celem jest określenie, w którym miejscu w tekście znajduje się wzorzec, o ile w ogóle w nim występuje.

Przyjmiemy, że długość wzorca (rozumiana jako liczba znaków) jest nie większa niż długość tekstu, a w praktyce jest ona wielokrotnie mniejsza. Ograniczymy znaki, które mogą występować w tekście i we wzorcu, do małych liter alfabetu łacińskiego.

Oto specyfikacja problemu:

**Dobra rada**

Biblioteki większości języków programowania oferują funkcję do poszukiwania wzorca w tekście. W języku C++ możesz skorzystać z biblioteki `string` i metody `find` dla klasy `string`.

**Specyfikacja**

**Dane:** `t`, `w` – niepuste napisy złożone z małych liter alfabetu łacińskiego, długość tekstu `t` jest większa lub równa długości wzorca `w`.

**Wynik:** `p` – liczba całkowita określająca pozycję (indeks) pierwszego wystąpienia wzorca `w` w tekście `t` – lub –1, gdy wzorzec nie występuje w tekście.

## 11.1. Algorytm naiwny

Algorytm naiwny wyszukiwania wzorca w tekście

Jednym z najprostszych rozwiązań jest tzw. **algorytm naiwny wyszukiwania wzorca w tekście**, który polega na porównywaniu znak po znaku wzorca z tekstem.



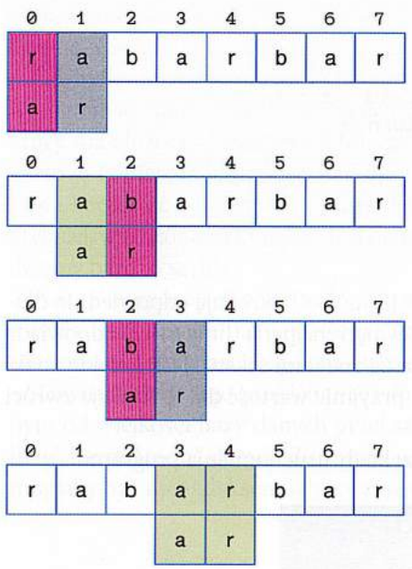


W algorytmie naiwnym ustawiamy wzorzec pod tekstem tak, aby pierwszy znak wzorca znajdował się pod pierwszym znakiem tekstu i porównujemy kolejne pary znaków obu napisów.

Jeśli wszystkie znaki wzorca będą odpowiadały znakom fragmentu tekstu, to znaleźliśmy położenie wzorca w tekście. Jeśli jednak znajdziemy pierwszą parę różniących się znaków, to przesuwamy wzorzec o jeden znak w prawo względem tekstu i ponownie rozpoczynamy porównywanie kolejnych par znaków.

Algorytm kończy działanie po znalezieniu pozycji wzorca w tekście lub wysunięciu wzorca poza tekst, czyli gdy pozycja początkowa wzorca jest większa od długości tekstu pomniejszonej o długość wzorca.

Opisany algorytm pokażemy na przykładzie. Poszukamy wzorca *ar* w tekście *rabarbar*. Przedstawia to rysunek 11.1.




**Rys. 11.1.** Naiwne wyszukiwanie wzorca w tekście

Oto algorytm naiwny w postaci funkcji, zapisany w pseudokodzie:

```
funkcja Znajdz(w,t)
    p ← 0
    dopóki p ≤ długość t - długość w wykonuj
        i ← 0
        dopóki i < długość w oraz w[i] = t[p+i] wykonuj
            i ← i + 1
        jeśli i = długość w to zwróć p i zakończ
        w przeciwnym przypadku p ← p + 1
    zwróć -1 i zakończ
```

**Dobra rada**

Jeśli chcesz znaleźć wszystkie wystąpienia wzorca w tekście, kontynuuj działanie algorytmu, aż pozycja początkowa wzorca będzie większa od długości tekstu pomniejszonej o długość wzorca.

Zmienna pomocnicza i określa indeks porównywanego znaku wzorca. W zależności od przesunięcia wzorca względem tekstu (wartości zmiennej p) porównujemy znaki w[i] z t[p+i] dla i w zakresie od 0 do długości wzorca minus 1, chyba że wcześniej pojawi się para różnych znaków. Wówczas zwiększamy wartość zmiennej p, czyli przesuwamy wzorzec o jeden znak w prawo względem tekstu.

Oto kod źródłowy funkcji Znajdz, realizującej algorytm naiwny wyszukiwania wzorca w tekście:

**Kod źródłowy funkcji Znajdz, realizującej algorytm naiwny wyszukiwania wzorca w tekście**

```
1.  int Znajdz(string w, string t)
2.  {
3.      int dw=w.size();
4.      int dt=t.size();
5.      int i, p=0;
6.      while (p<=dt-dw)
7.      {
8.          i=0;
9.          while (i<dw && w[i]==t[p+i]) i++;
10.         if (i==dw) return p;
11.         else p++;
12.     }
13.     return -1;
14. }
```

Zmienne pomocnicze dw i dt (linie 3–4) określają odpowiednio długość wzorca i długość tekstu. Zewnętrzna pętla (linie 6–12) odpowiada za sprawdzenie znaków wzorca ze znakami tekstu. Jeśli kolejne znaki będą sobie równe, a zmienna i przyjmie wartość dw, to funkcja zwróci pozycję wzorca (linia 10).

Rysunek 11.2 przedstawia przykłady uruchomienia programu.

```
Tekst: rabarbar
Szukany wzorzec: ar
Pozycja wzorca w tekscie: 3
```

```
Tekst: rabarbar
Szukany wzorzec: rak
Wzorzec nie wystepuje w tekscie
```

**Rys. 11.2.** Efekt działania programu poszukującego wzorców *ar* i *rak* w tekście *rabarbar*

### Ćwiczenie 1

Napisz program, który wczyta tekst oraz wzorzec, zapisane małymi literami alfabetu łacińskiego, a następnie wyszuka wzorzec w tekście, stosując algorytm naiwny. W programie wykorzystaj kod źródłowy funkcji Znajdz.





Złożoność obliczeniowa algorytmu naiwnego wynosi $O(n \cdot m)$, gdzie $n$ oznacza długość tekstu, a $m$ – długość poszukiwanego wzorca. Warto się zastanowić, czy nie można znaleźć wzorca w tekście (lub sprawdzić, że on nie występuje), wykonując mniej operacji porównywania znaków.

Jednym ze sposobów jest porównywanie wzorca z fragmentem tekstu tylko czasami – kiedy istnieje duże prawdopodobieństwo zgodności napisów. Wykorzystamy do tego technikę algorytmiczną zwaną haszowaniem.

**Warto wiedzieć**

Istnieją algorytmy wyszukiwania wzorca w tekście, które wykorzystują przesunięcie wzorca względem tekstu o więcej niż jedną pozycję.

## 11.2. Metoda haszowania

Ideę haszowania wytłumaczymy na przykładzie wybierania numeru z listy kontaktów zapisanej w telefonie. Jeśli chcemy do kogoś zadzwonić, podajemy jego nazwisko, które jest powiązane z numerem telefonu. Oprogramowanie telefonu szuka więc właściwego numeru po nazwisku (lub nazwie kontaktu powiązanej z numerem).

Gdyby kontakty w telefonie były nieuporządkowane, oprogramowanie stosowałoby zapewne algorytm **przeszukiwania liniowego**, który ma złożoność czasową $O(n)$, gdzie $n$ to liczba zapisanych kontaktów. Ponieważ jednak są one uporządkowane, oprogramowanie może korzystać z algorytmu **przeszukiwania binarnego**, który działa szybciej, w złożoności $O(\log n)$. Dla niewielkiej wartości $n$ kontakt znajdziemy bardzo szybko.

**Przeszukiwanie liniowe i binarne**, podręcznik *Informatyka na czasie 2. Zakres rozszerzony*, s. 153–154

Powyższy sposób zawodzi jednak w przypadku bardzo dużych zbiorów danych, np. dla bazy operatora telekomunikacyjnego, ponieważ czas odszukania właściwego nazwiska może być nadal zbyt długi.

Rozwiązaniem tego problemu jest funkcja, która w czasie niezależnym od wielkości bazy danych przekształca nazwisko tak, aby umożliwić bezpośredni dostęp do numeru telefonu. Wartość tej funkcji mogłaby być np. indeksem tablicy zawierającej numery telefonów.

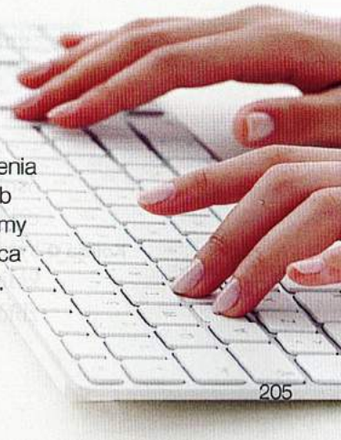
**Dobra rada**

Jeśli dane są uporządkowane, możesz zastosować algorytm przeszukiwania binarnego.

A to ciekawe

## Jak algorytmy pomagają wykrywać plagiaty

Plagiat polega na przypisaniu sobie autorstwa fragmentu lub całości cudzego utworu, czyli wykorzystanie go bez podania źródła i oznaczenia autora. W przypadku utworów tekstowych, np. prac magisterskich lub doktorskich, powszechnie stosuje się systemy antyplagiatowe. Systemy te mają zaimplementowane algorytmy szybkiego wyszukiwania wzorca w tekście, które pozwalają sprawdzić, czy dana praca jest oryginalna. Pozwalają również ustalić, jaka jej część składa się z fragmentów znajdujących się w innych publikacjach z bazy systemu.



Funkcja haszująca (funkcja skrótu)

Taką funkcję nazywamy **funkcją haszującą** (ang. *hash function*) albo **funkcją skrótu**. W ogólnym przypadku przekształca ona napis na pewną wartość, która umożliwia dostęp do danych w czasie niezależnym od ich rozmiaru. Argument funkcji haszującej nazywamy

Klucz, hasz

**kluczem** (ang. *key*), a jej wartość – **haszem** (ang. *hash*).

Dla danego problemu, np. wyszukiwania numeru na liście kontaktów w telefonie, można zdefiniować wiele różnych funkcji haszujących. Przykładem bardzo prostej funkcji haszującej może być funkcja przekształcająca nazwisko na liczbę na podstawie kodu ASCII pierwszej litery nazwiska.

## Wykorzystanie funkcji haszującej

Sposób wykorzystania funkcji haszującej zaprezentujemy na uproszczonym przykładzie wyszukiwania numeru telefonu na liście kontaktów.


| Indeks | Numer telefonu |
|---|---|
| 65 | 123 123 123 |
| **66** | **456 456 456** |
| 67 | 789 789 789 |
| ... | ... |

Zauważ, że na liście kontaktów mogą się pojawić przynajmniej dwie osoby o nazwisku rozpoczynającym się na tę samą literę, zatem funkcja haszująca dla różnych kluczy może zwrócić tę samą wartość. Taką

Kolizja

sytuację nazywamy **kolizją** (ang. *collision*). Jednak nawet w przypadku kolizji poszukiwanie zostaje zawężone do znacznie mniejszego podzbioru danych.





### Ćwiczenie 2

Podaj przykład funkcji haszującej, która zmniejszy liczbę kolizji podczas wyszukiwania kontaktów na liście.

**Warto wiedzieć**

Wartością funkcji haszującej nie musi być liczba, może to być również napis. Wartość ta powinna mieć określony rozmiar, np. liczba powinna należeć do określonego przedziału, a napis mieć określoną maksymalną długość.

Złożoność algorytmów wykorzystujących haszowanie zależy od funkcji haszującej. Funkcja ta nie powinna zwracać tej samej wartości dla dużego zbioru kluczy, a dla poszczególnych kluczy powinna równomiernie przydzielać wartości. Nie powinna też powodować kolizji dla zbliżonych wartości kluczy.

## 11.3. Algorytm Karpa–Rabina

Algorytm Karpa–Rabina

Przedstawimy teraz **algorytm Karpa–Rabina**, rozwiązujący problem znajdowania wzorca w tekście z wykorzystaniem metody haszowania, która znacząco ograniczy liczbę porównań wzorca i fragmentu tekstu.

W algorytmie Karpa–Rabina odpowiednio dobrana **funkcja haszująca** wyznacza wartość haszu dla wzorca oraz dla fragmentu porównywanego tekstu. Jeśli obie wartości haszy są różne, to nie ma zgodności wzorca z fragmentem tekstu. Jeśli wartości haszy są takie same, to musimy dodatkowo potwierdzić, że wzorzec został odnaleziony, ponieważ możemy mieć do czynienia z **kolizją**. Wówczas porównujemy znaki wzorca i fragmentu tekstu analogicznie jak w **algorytmie naiwnym**.

Funkcja haszująca, s. 206

Kolizja, s. 206

Algorytm naiwny wyszukiwania wzorca w tekście, s. 202

Liczba porównań jest znacząco ograniczona, bo porównujemy tylko te fragmenty tekstu, których hasze zgadzają się z haszem wzorca.

Zapiszmy algorytm Karpa–Rabina w pseudokodzie.

```
funkcja Znajdz(w,t)
    hw ← hasz wzorca w
    ht ← hasz początkowego fragmentu tekstu t
    p ← 0
    dopóki p ≤ długość t - długość w wykonuj
        jeśli hw = ht to
            i ← 0
            dopóki i < długość w oraz w[i] = t[p+i] wykonuj
                i ← i + 1
            jeśli i = długość w to zwróć p i zakończ
        p ← p + 1
        jeśli p ≤ długość t - długość w
            ht ← hasz fragmentu tekstu t
                 po przesunięciu wzorca
    zwróć -1 i zakończ
```

Zmienna hw pamięta hasz wzorca w – jest to wartość stała. Wartością początkową zmiennej ht jest hasz początkowego fragmentu tekstu o długości równej długości wzorca. Jeśli hasze hw i ht są sobie równe, to prawdopodobnie znaleźliśmy wystąpienie wzorca w tekście. Potwierdzamy to przez porównanie wzorca z fragmentem.

Jeśli hasze nie były sobie równe lub nastąpiła kolizja, przesuwamy wzorzec względem tekstu o jedną pozycję w prawo (zwiększamy wartość zmiennej p o 1). Jeśli wzorzec nie wychodzi poza tekst, to obliczamy nową wartość zmiennej ht – wartość funkcji haszującej dla nowego fragmentu tekstu.

Funkcja haszująca, s. 206

Istotą algorytmu jest właściwy dobór **funkcji haszującej**, tak aby dawała równomierny rozkład wartości haszu i można było szybko wyznaczyć jego wartość. W takich sytuacjach często stosuje się tzw. arytmetykę modularną.

**Zapamiętaj**

Funkcja haszująca przekształca napis na pewną wartość umożliwiającą dostęp do danych w czasie niezależnym od ich rozmiaru. Argument funkcji nazywamy kluczem, a jej wartość – haszem.

## Arytmetyka modularna

Arytmetyka modularna

**Arytmetyka modularna** (ang. *modular arithmetic*) nazywana jest również arytmetyką reszt lub arytmetyką modulo $n$. Jest to arytmetyka liczb całkowitych, w której wyniki działań arytmetycznych sprowadza się do przedziału $[0; n)$ w taki sposób, aby wynik działania był przedstawiony jako reszta z dzielenia przez $n$.

**Warto wiedzieć**

Arytmetykę modularną stosujemy na co dzień, np. przy odliczaniu czasu. Cztery godziny po godzinie 22 będzie godzina 2, ponieważ (22 + 4) mod 24 = 2.

Pokażemy przykład dodawania, mnożenia i odejmowania liczb 3 i 13 w arytmetyce modulo 11.

$$(3 + 13) \bmod 11 = 16 \bmod 11 = 5$$
$$(3 \cdot 13) \bmod 11 = 39 \bmod 11 = 6$$

Jeśli wynik odejmowania modulo $n$ jest mniejszy od zera, to do wyniku należy dodać wartość $n$ – w naszym przykładzie liczbę 11:

$$(3 - 13) \bmod 11 = (-10) \bmod 11 = -10 + 11 = 1$$

Ponieważ prawdziwe są poniższe zależności:

$$(a + b) \bmod n = (a \bmod n + b \bmod n) \bmod n$$
$$(a \cdot b) \bmod n = (a \bmod n \cdot b \bmod n) \bmod n$$

**Warto wiedzieć**

W arytmetyce modularnej nie ma operacji dzielenia. Występuje natomiast odwrotność liczby. Odwrotnością liczby $a$ w arytmetyce modulo $n$ jest taka liczba $b$, że $(a \cdot b) \bmod n = 1$.

można przyjąć, że argumenty działań są także z przedziału $[0; n)$. Na przykład:

$$(3 + 13) \bmod 11 = (3 + 2) \bmod 11 = 5$$
$$(3 \cdot 13) \bmod 11 = (3 \cdot 2) \bmod 11 = 6$$
$$(3 - 13) \bmod 11 = (3 - 2) \bmod 11 = 1$$

### Ćwiczenie 3

Wykonaj dodawanie, odejmowanie i mnożenie liczb:

**a.** 7 i 15 w arytmetyce modulo 5,

**b.** 5 i 11 w arytmetyce modulo 13.





## Dobór funkcji haszującej w algorytmie Karpa–Rabina

Wyjaśnimy na przykładzie sposób konstruowania funkcji haszującej w algorytmie Karpa–Rabina. Przyjmijmy na moment, że alfabet składa się tylko z trzech liter: *a*, *b* i *c*. Od kodu ASCII danej litery odejmiemy liczbę 97 (kod ASCII litery *a*), więc litera *a* będzie reprezentowana przez liczbę 0, litera *b* –przez liczbę 1, a litera *c* – przez liczbę 2.

Litery alfabetu możemy potraktować jako cyfry systemu pozycyjnego o podstawie równej liczebności alfabetu – w naszym przykładzie podstawa jest równa 3. Klucz (czyli wzorzec lub fragment tekstu) potraktujemy jako liczbę zapisaną w tym systemie. **Haszem** będzie wartość dziesiętna tej liczby w arytmetyce modulo $n$, gdzie $n$ jest liczbą pierwszą.

Hasz, s. 206

Na przykład dla klucza *bbaabc* i $n = 29$ wartość funkcji Hash, haszującej w arytmetyce modulo 29, będzie równa:

$$\text{Hash}(bbaabc) = (1 \cdot 3^5 + 1 \cdot 3^4 + 0 \cdot 3^3 + 0 \cdot 3^2 + 1 \cdot 3^1 + 2 \cdot 3^0) \bmod 29 =$$
$$= (11 + 23 + 3 + 2) \bmod 29 = 10$$

**Dobra rada**

Podczas wykonywania obliczeń w arytmetyce modulo $n$ często korzysta się ze standardowych symboli działań arytmetycznych. Pamiętaj jednak, że wynik tego działania zawsze jest przekształcany na resztę z dzielenia przez $n$, dlatego dla $n = 29$ wynik działania $3^5$ to 11.

Zwróć uwagę, że jest to zapis odpowiadający obliczeniu wartości wielomianu. Zauważ też, że kolejne potęgi liczby 3 w arytmetyce modulo 29 to: 1, 3, 9, 27, 23, 11.

### Ćwiczenie 4

Oblicz wartość funkcji Hash dla:

**a.** klucza *bcba* i $n = 11$,

**b.** klucza *abccbb* i $n = 29$.

Zapiszemy algorytm działania funkcji haszującej, który będzie właściwy dla dowolnego tekstu utworzonego z małych liter alfabetu łacińskiego. Do obliczenia wartości funkcji wykorzystamy **schemat Hornera** – stopień wielomianu będzie o jeden mniejszy od długości klucza.

Schemat Hornera, s. 125

Niech n będzie liczbą pierwszą, a m oznacza długość alfabetu. Wartość funkcji jest obliczana w arytmetyce modulo n i przechowywana w zmiennej w. Liczba pierwsza n powinna być dobrana tak, aby z jednej strony uniknąć zbyt dużych wartości funkcji haszującej, a z drugiej strony ograniczyć liczbę **kolizji**.

Kolizja, s. 206

Oto zapis w pseudokodzie funkcji Hash0:

```
funkcja Hash0(klucz)
    w ← 0
    dla i ← 0, ..., długość klucz - 1 wykonuj
        w ← ((w*m) mod n + kod_ASCII(klucz[i]) - 97) mod n
    zwróć w i zakończ
```

Jeśli wartość kod_ASCII(klucz[i]) – 97 jest mniejsza od n, to nie musimy brać reszty z dzielenia przez n.

Zauważ, że kolejne fragmenty porównywanego tekstu różnią się od siebie tylko nieznacznie – wystarczy pominąć pierwszy znak aktualnego fragmentu tekstu i uwzględnić kolejny znak. Dzięki temu można szybko obliczyć wartość funkcji haszującej dla kolejnego fragmentu na podstawie dotychczasowej wartości funkcji. Zapiszmy w pseudokodzie funkcję Hash1, która to realizuje.

```
funkcja Hash1(h0,jd,ch)
    w ← (h0 - jd) mod n
    jeśli w<0 to w ← w + n
    zwróć ((w*m) mod n + kod_ASCII(ch) - 97) mod n i zakończ
```

Parametry funkcji Hash1 to: h0 – dotychczasowa wartość funkcji haszującej dla fragmentu tekstu, jd – wartość jednomianu odpowiadająca usuwanemu znakowi w arytmetyce modulo n, ch – kolejny znak tekstu. Wartość m określa liczbę znaków alfabetu.

W naszym przykładzie wartość funkcji dla klucza *bbaabc* jest równa 10. Dla tekstu *bbaabccac* kolejnym kluczem będzie *baabcc*. Od dotychczasowej wartości funkcji należy odjąć wartość jednomianu $1 \cdot 3^5$ (odpowiadającą pierwszej literze *b*), czyli wartość 11 w arytmetyce modulo 29. Następnie należy przesunąć pozostałe litery o jedną pozycję w lewo (czyli pomnożyć otrzymaną wartość przez 3) i dodać wartość odpowiadającą nowemu znakowi fragmentu tekstu (litera *c*, $2 \cdot 3^0$):

Hash(*bbaabc*) = 10

Hash(*baabcc*) = ((10 – 11) · 3 + 2) mod 29 = (28 · 3 + 2) mod 29 =
= (26 + 2) mod 29 = 28

Do wyznaczenia wartości odejmowanego jednomianu potrzebna będzie jeszcze funkcja wyznaczająca wartość potęgi w arytmetyce modulo *n*. Będzie to funkcja Potega.

Przedstawimy teraz definicje stałych globalnych określających wartość liczby pierwszej oraz długość alfabetu, a także kody źródłowe funkcji: Potega, Hash0, Hash1 oraz Znajdz.

**Definicje stałych globalnych i kod źródłowy funkcji Potega w programie realizującym algorytm Karpa–Rabina**

```
1.  const int N=997; // liczba pierwsza
2.  const int M=26;  // długość alfabetu
3.
4.  int Potega(int podst, int wykl)
5.  {
6.      int w=1;
7.      while (wykl>0)
8.      {
9.          if (wykl%2==1) w=(w*podst)%N;
10.         wykl=wykl/2;
11.         if (wykl>0) podst=(podst*podst)%N;
12.     }
13.     return w;
14. }
```





W liniach 1–2 zdefiniowane są stałe N i M, określające odpowiednio liczbę pierwszą oraz liczebność alfabetu zgodną ze specyfikacją. W liniach 4–14 zdefiniowana jest funkcja Potega, która służy do **szybkiego obliczania wartości potęgi** w arytmetyce modulo N.

**Algorytm szybkiego podnoszenia do potęgi**, podręcznik *Informatyka na czasie 2. Zakres rozszerzony*, s. 57–58

Oto kod źródłowy funkcji Hash0 oraz Hash1:

**Kod źródłowy funkcji Hash0 oraz Hash1 w programie realizującym algorytm Karpa–Rabina**

```
1.  int Hash0(string s)
2.  {
3.      int w=0;
4.      for (int i=0;i<s.size();i++)
5.          w=((w*M)%N+s[i]-'a')%N;
6.      return w;
7.  }
8.
9.  int Hash1(int h0, int jd, char ch)
10. {
11.     int w=(h0-jd)%N;
12.     if (w<0) w=w+N;
13.     return ((w*M)%N+ch-'a')%N;
14. }
```

W liniach 1–7 zdefiniowana jest funkcja Hash0, którą wykorzystamy do obliczenia haszy wzorca i pierwszego fragmentu tekstu. Zdefiniowana w liniach 9–14 funkcja Hash1 posłuży do obliczania wartości haszy dla kolejnych fragmentów tekstu.

Powyższe funkcje wykorzystuje funkcja Znajdz. Oto jej kod źródłowy:

**Kod źródłowy funkcji Znajdz w programie realizującym algorytm Karpa–Rabina**

```
1.  int Znajdz(string w, string t)
2.  {
3.      int i, p=0, dw=w.size(), dt=t.size();
4.      int hw=Hash0(w);
5.      int ht=Hash0(t.substr(0,dw));
6.      int pot=Potega(M,dw-1);
7.      while (p<=dt-dw)
8.      {
9.          if (hw==ht)
10.         {
11.             i=0;
12.             while (i<dw && w[i]==t[p+i]) i++;
13.             if (i==dw) return p;
14.         }
15.         if (p<dt-dw)
16.             ht=Hash1(ht,(pot*(t[p]-'a'))%N,t[p+dw]);
17.         p++;
18.     }
19.     return -1;
20. }
```

Kod źródłowy funkcji `Znajdz`, realizującej algorytm naiwny wyszukiwania wzorca w tekście, s. 204

Znaczenie zmiennych `i`, `p`, `dw` i `dt` w funkcji `Znajdz` jest takie samo jak w **programie realizującym algorytm naiwny**. W linii 5 zmienna `ht` przyjmuje wartość haszu dla pierwszego fragmentu tekstu. W linii 6 w zmiennej `pot` zapamiętana jest wartość $M^{dw-1}$. Pętla główna w liniach 7–18 działa podobnie do pętli głównej programu realizującego algorytm naiwny, ale porównywanie wzorca z fragmentem tekstu następuje tylko w przypadku takich samych wartości funkcji haszującej (linia 9).

W liniach 15–16 obliczamy wartość funkcji haszującej dla kolejnego fragmentu tekstu i przypisujemy ją zmiennej `ht`. Dzieje się tak pod warunkiem, że wzorzec można jeszcze przesunąć (linia 15).

Funkcję `Hash1` wywołujemy dla parametrów:

- `ht` – dotychczasowa wartość funkcji haszującej,
- `(pot*(t[p]-'a'))%N` – wartość jednomianu reprezentującego pierwszy usuwany znak dotychczasowego fragmentu tekstu,
- `t[p+dw]` – kolejny znak tekstu, który należy dodać do fragmentu tekstu.

**Dobra rada**

Zauważ, że powiększenie wartości zmiennej `p` warto wykonać po ewentualnym wywołaniu funkcji `Hash1`, aby przy określaniu wartości parametrów funkcji nie odejmować liczby 1.

### Ćwiczenie 5

Napisz program, który wczyta tekst oraz wzorzec, składające się z małych liter alfabetu łacińskiego, a następnie wyszuka wzorzec w tekście z wykorzystaniem algorytmu Karpa–Rabina.

### Złożoność obliczeniowa algorytmu Karpa–Rabina

Funkcja `Potega` jest wywoływana w funkcji `Znajdz` tylko raz, a funkcja `Hash0` dwa razy. Czas wykonania tych funkcji nie ma więc wpływu na czas wyszukiwania wzorca w długim tekście.

Funkcja `Hash1` jest wielokrotnie wywoływana w pętli (linia 16 w kodzie źródłowym funkcji `Znajdz`). Działa ona szybko i wykonuje jedynie kilka podstawowych działań arytmetycznych. Jej złożoność czasowa nie zależy ani od długości wzorca, ani od długości tekstu, wynosi więc $O(1)$. Algorytm Karpa–Rabina działa w złożoności czasowej $O(n + m)$, gdzie $n$ oznacza długość tekstu, a $m$ – długość wzorca.

**Warto wiedzieć**

Istnieją także inne algorytmy szybkiego wyszukiwania wzorca w tekście mające złożoność czasową $O(n + m)$, gdzie $n$ oznacza długość tekstu, a $m$ – długość wzorca.

**A to ciekawe**

## Kryptograficzne funkcje haszujące

Funkcje haszujące są wykorzystywane również w kryptografii. Wiele serwisów internetowych przechowuje hasła użytkowników w postaci haszy. W czasie logowania podane przez użytkownika hasło jest haszowane i porównywane z haszem zapisanym w bazie danych. Funkcje haszujące stosuje się również m.in. w podpisach cyfrowych oraz w systemach kontroli wersji.






## Podsumowanie

- Algorytm naiwny wyszukiwania wzorca w tekście polega na porównywaniu wzorca z tekstem znak po znaku, a następnie przesuwaniu położenia wzorca względem tekstu o jedną pozycję w prawo.
- Złożoność obliczeniowa algorytmu naiwnego wynosi $O(n \cdot m)$, gdzie $n$ oznacza długość tekstu, a $m$ – długość wzorca.
- Metoda haszowania polega na przyporządkowaniu napisowi odpowiadającej mu wartości o określonym rozmiarze.
- Złożoność czasowa algorytmów wykorzystujących metodę haszowania zależy od doboru funkcji haszującej. Powinna ona działać w złożoności czasowej niezależnej od rozmiaru danych.
- Algorytm Karpa–Rabina, służący do wyszukiwania wzorca w tekście, wykorzystuje metodę haszowania – zamiast porównywać wzorzec z fragmentem tekstu, porównuje odpowiadające im hasze. Funkcja haszująca wyznacza wartość dla kolejnego fragmentu tekstu na podstawie dotychczasowej wartości w czasie $O(1)$ za pomocą arytmetyki modularnej.
- W arytmetyce modulo $n$ wyniki działań arytmetycznych sprowadza się do przedziału $[0; n)$. Wynikiem jest reszta z dzielenia przez $n$.
- Złożoność obliczeniowa algorytmu Karpa–Rabina wynosi $O(n + m)$, gdzie $n$ oznacza długość tekstu, a $m$ – długość wzorca.

## Zadania

* **1** Przygotuj prezentację wyjaśniającą ideę metody haszowania.

* **2** Dane są dwa napisy określające autora i tytuł książki. Zaproponuj funkcję haszującą, która umożliwi szybkie wyszukiwanie książek.

* **3** Przygotuj prezentację wyjaśniającą zasady arytmetyki modularnej liczb całkowitych.

** **4** Program realizujący algorytm naiwny wyszukiwania wzorca w tekście zmodyfikuj tak, aby znajdował wszystkie wystąpienia wzorca w tekście.

** **5** Program realizujący algorytm Karpa–Rabina zmodyfikuj tak, aby znajdował wszystkie wystąpienia wzorca w tekście.

** **6** Poszukaj w internecie informacji na temat innych algorytmów wyszukiwania wzorca w tekście niż omówione w tym temacie i przygotuj prezentację na ten temat.

*** **7** Poszukaj w internecie informacji na temat uproszczonego algorytmu Boyera–Moore'a, służącego do znajdowania wzorca w tekście, i napisz program realizujący ten algorytm.

*** **8** Poszukaj w internecie informacji na temat algorytmu Morrisa–Pratta, służącego do znajdowania wzorca w tekście, i napisz program realizujący ten algorytm.