Proiectarea algoritmilor: Căutare peste șiruri

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

stefan.ciobaca@info.uaic.ro

PA 2015/2016

Plan

1. Introducere
2. Algoritmul Rabin-Karp
3. Algoritmul Boyer-Moore
4. Algoritmul Z
5. Algoritmul Boyer-Moore revizuit

Domeniul problemei = șiruri de caractere

1. *Alfabet* = mulțime nevidă

Domeniul problemei = șiruri de caractere

1. *Alfabet* = mulțime nevidă
   Exemplu: $A = \{A, a, B, b, C, c, \ldots, Z, z\}$

Domeniul problemei = șiruri de caractere

1. *Alfabet* = mulțime nevidă
   Exemplu: $A = \{A, a, B, b, C, c, \ldots, Z, z\}$, $A' = \{0, 1\}$, $A'' = \mathbb{N}$ (alfabet infinit)

Domeniul problemei = șiruri de caractere

1. *Alfabet* = mulțime nevidă
   Exemplu: $A = \{A, a, B, b, C, c, \ldots, Z, z\}$, $A' = \{0, 1\}$, $A'' = \mathbb{N}$ (alfabet infinit)
2. *Șir de caractere* (engl. *string*) = secvență de caractere (caracter = element al alfabetului)

Domeniul problemei = șiruri de caractere

1. *Alfabet* = mulțime nevidă
   Exemplu: $A = \{A, a, B, b, C, c, \ldots, Z, z\}$, $A' = \{0, 1\}$, $A'' = \mathbb{N}$ (alfabet infinit)
2. *Șir de caractere* (engl. *string*) = secvență de caractere (caracter = element al alfabetului)
   Exemplu: $s = \texttt{Ana are mere}$

Domeniul problemei = șiruri de caractere

1. *Alfabet* = mulțime nevidă
   Exemplu: $A = \{A, a, B, b, C, c, \ldots, Z, z\}$, $A' = \{0, 1\}$, $A'' = \mathbb{N}$ (alfabet infinit)
2. *Șir de caractere* (engl. *string*) = secvență de caractere (caracter = element al alfabetului)
   Exemplu: $s =$ `Ana are mere`
3. Dacă s este un șir de caractere, notăm cu $s[0]$ (sau cu s_0) primul caracter al șirului, cu $s[1]$ (sau cu s_1) al doilea element, ... și cu $S[n-1]$ (sau cu s_{n-1}) ultimul element, dacă S are n caractere.

Domeniul problemei = șiruri de caractere

1. *Alfabet* = mulțime nevidă
   Exemplu: $A = \{A, a, B, b, C, c, \ldots, Z, z\}$, $A' = \{0, 1\}$, $A'' = \mathbb{N}$ (alfabet infinit)
2. *Șir de caractere* (engl. *string*) = secvență de caractere (caracter = element al alfabetului)
   Exemplu: s = `Ana are mere`
3. Dacă s este un șir de caractere, notăm cu $s[0]$ (sau cu s_0) primul caracter al șirului, cu $s[1]$ (sau cu s_1) al doilea element, ... și cu $S[n-1]$ (sau cu s_{n-1}) ultimul element, dacă S are n caractere.
4. Cu $S[i..j]$ notăm *subșirul* lui S care conține în ordine caracterele $S[i], S[i+1], \ldots, S[j]$, în această ordine.

Domeniul problemei = șiruri de caractere

1. *Alfabet* = mulțime nevidă
   Exemplu: $A = \{A, a, B, b, C, c, \ldots, Z, z\}$, $A' = \{0, 1\}$, $A'' = \mathbb{N}$ (alfabet infinit)
2. *Șir de caractere* (engl. *string*) = secvență de caractere (caracter = element al alfabetului)
   Exemplu: $s =$ `Ana are mere`
3. Dacă s este un șir de caractere, notăm cu $s[0]$ (sau cu s_0) primul caracter al șirului, cu $s[1]$ (sau cu s_1) al doilea element, ... și cu $S[n-1]$ (sau cu s_{n-1}) ultimul element, dacă S are n caractere.
4. Cu $S[i..j]$ notăm *subșirul* lui S care conține în ordine caracterele $S[i], S[i+1], \ldots, S[j]$, în această ordine.
5. Dacă S și T sunt șiruri, spunem că S *apare în* T *la poziția* i (sau că i este o apariție a lui S în T) dacă $T[i..i+n-1] = S$, unde n este numărul de caractere al lui S.

Problema

Input Două șiruri: $s = s_0 \cdots s_{n-1}$, numit subiect sau text, și $p = p_0 \cdots p_{m-1}$, numit pattern.

Output Prima apariție a patternului p în textul s, dacă există; -1, altfel.

Problema

Input Două șiruri: $s = s_0 \cdots s_{n-1}$, numit subiect sau text, și $p = p_0 \cdots p_{m-1}$, numit pattern.

Output Prima apariție a patternului p în textul s, dacă există; -1, altfel.

Variantă: găsirea tuturor aparițiilor:

Problema

Input Două șiruri: $s = s_0 \cdots s_{n-1}$, numit subiect sau text, și $p = p_0 \cdots p_{m-1}$, numit pattern.

Output Prima apariție a patternului p în textul s, dacă există; -1, altfel.

Variantă: găsirea tuturor aparițiilor:
Input Două șiruri: $s = s_0 \cdots s_{n-1}$, numit subiect sau text, și $p = p_0 \cdots p_{m-1}$, numit pattern.

Output O mulțime M care conține exact aparițiile lui p în textul s.

Algoritmul naiv (funcție ajutătoare)

```
int apare_la_pozitia(s, n, p, m, i)
{
  // verifica daca patternul p (de lungime m) apare
  // la pozitia i in subiectul s (de lungime n)
  for (j = 0; j < m; ++j) {
    if (i + j >= n || s[i + j] != p[j]) {
      return 0;
    }
  }
  return 1;
}
```

Algoritmul naiv (funcție ajutătoare)

```
int apare_la_pozitia(s, n, p, m, i)
{
  // verifica daca patternul p (de lungime m) apare
  // la pozitia i in subiectul s (de lungime n)
  for (j = 0; j < m; ++j) {
    if (i + j >= n || s[i + j] != p[j]) {
      return 0;
    }
  }
  return 1;
}
```

Complexitate: $O(m)$ în cazul cel mai nefavorabil, $O(1)$ în cazul cel mai favorabil.

Algoritmul naiv

```
int prima_aparitie(s, n, p, m)
{ // intoarce prima aparitie a lui p in s
  // sau -1 daca nu exista aparitii ale lui p in s
  for (i = 0; i < n; ++i) {
    if (apare_la_pozitia(s, n, p, m, i)) {
      return i;
    }
  }
  return -1;
}
```

Algoritmul naiv

```
int prima_aparitie(s, n, p, m)
{ // intoarce prima aparitie a lui p in s
  // sau -1 daca nu exista aparitii ale lui p in s
  for (i = 0; i < n; ++i) {
    if (apare_la_pozitia(s, n, p, m, i)) {
      return i;
    }
  }
  return -1;
}
```

Complexitate: $O(n \cdot m)$ în cazul cel mai nefavorabil, $O(min(n, m))$ în cazul cel mai favorabil.

Algoritmul naiv

```
int prima_aparitie(s, n, p, m)
{ // intoarce prima aparitie a lui p in s
  // sau -1 daca nu exista aparitii ale lui p in s
  for (i = 0; i < n; ++i) {
    if (apare_la_pozitia(s, n, p, m, i)) {
      return i;
    }
  }
  return -1;
}
```

Complexitate: $O(n \cdot m)$ în cazul cel mai nefavorabil, $O(min(n, m))$ în cazul cel mai favorabil.
Exercițiu: Care este cazul cel mai nefavorabil? Care este cazul cel mai favorabil? Algoritmul este corect și dacă înlocuim $i < n$ cu $i < n - m$ în bucla `for`? Complexitatea algoritmului se schimbă?

Puțin istoric

Algoritmii care rezolvă problema de căutării peste șiruri au o istorie interesantă.

Puțin istoric

Algoritmii care rezolvă problema de căutării peste șiruri au o istorie interesantă. În 1970, S.A. Cook a demonstrat un rezultat teoretic despre un anumit tip abstract de mașină, unde se presupunea existența unui algoritm de căutare peste șiruri ce necesită un timp proporțional cu $n + m$ în cazul cel mai nefavorabil.

Puțin istoric

Algoritmii care rezolvă problema de căutării peste șiruri au o istorie interesantă. În 1970, S.A. Cook a demonstrat un rezultat teoretic despre un anumit tip abstract de mașină, unde se presupunea existența unui algoritm de căutare peste șiruri ce necesită un timp proporțional cu $n + m$ în cazul cel mai nefavorabil. D.E. Knuth și V.R. Pratt au utilizat construcția laborioasă din teorema lui Cook și au elaborat un algoritm care, mai apoi, a fost rafinat într-un algoritm practic și simplu.

Puțin istoric

Algoritmii care rezolvă problema de căutării peste șiruri au o istorie interesantă. În 1970, S.A. Cook a demonstrat un rezultat teoretic despre un anumit tip abstract de mașină, unde se presupunea existența unui algoritm de căutare peste șiruri ce necesită un timp proporțional cu $n + m$ în cazul cel mai nefavorabil. D.E. Knuth și V.R. Pratt au utilizat construcția laborioasă din teorema lui Cook și au elaborat un algoritm care, mai apoi, a fost rafinat într-un algoritm practic și simplu. J.H. Morris a descoperit același algoritm în timpul implementării unui editor de texte.

Puțin istoric

Algoritmii care rezolvă problema de căutării peste șiruri au o istorie interesantă. În 1970, S.A. Cook a demonstrat un rezultat teoretic despre un anumit tip abstract de mașină, unde se presupunea existența unui algoritm de căutare peste șiruri ce necesită un timp proporțional cu $n + m$ în cazul cel mai nefavorabil. D.E. Knuth și V.R. Pratt au utilizat construcția laborioasă din teorema lui Cook și au elaborat un algoritm care, mai apoi, a fost rafinat într-un algoritm practic și simplu. J.H. Morris a descoperit același algoritm în timpul implementării unui editor de texte. Este unul dintre numeroasele exemple când un rezultat pur teoretic poate conduce la rezultate cu aplicabilitate imediată.

Puțin istoric

Algoritmii care rezolvă problema de căutării peste șiruri au o istorie interesantă. În 1970, S.A. Cook a demonstrat un rezultat teoretic despre un anumit tip abstract de mașină, unde se presupunea existența unui algoritm de căutare peste șiruri ce necesită un timp proporțional cu $n + m$ în cazul cel mai nefavorabil. D.E. Knuth și V.R. Pratt au utilizat construcția laborioasă din teorema lui Cook și au elaborat un algoritm care, mai apoi, a fost rafinat într-un algoritm practic și simplu. J.H. Morris a descoperit același algoritm în timpul implementării unui editor de texte. Este unul dintre numeroasele exemple când un rezultat pur teoretic poate conduce la rezultate cu aplicabilitate imediată. Algoritmul dat de Knuth, Morris și Pratt a fost publicat de abia în 1976.

Puțin istoric

Algoritmii care rezolvă problema de căutării peste șiruri au o istorie interesantă. În 1970, S.A. Cook a demonstrat un rezultat teoretic despre un anumit tip abstract de mașină, unde se presupunea existența unui algoritm de căutare peste șiruri ce necesită un timp proporțional cu $n + m$ în cazul cel mai nefavorabil. D.E. Knuth și V.R. Pratt au utilizat construcția laborioasă din teorema lui Cook și au elaborat un algoritm care, mai apoi, a fost rafinat într-un algoritm practic și simplu. J.H. Morris a descoperit același algoritm în timpul implementării unui editor de texte. Este unul dintre numeroasele exemple când un rezultat pur teoretic poate conduce la rezultate cu aplicabilitate imediată. Algoritmul dat de Knuth, Morris și Pratt a fost publicat de abia în 1976. Între timp, R.S. Boyer și J.S. Moore (și indepedent W. Gosper) au descoperit un algoritm care este mult mai rapid în multe situații.

Puțin istoric

Algoritmii care rezolvă problema de căutării peste șiruri au o istorie interesantă. În 1970, S.A. Cook a demonstrat un rezultat teoretic despre un anumit tip abstract de mașină, unde se presupunea existența unui algoritm de căutare peste șiruri ce necesită un timp proporțional cu $n + m$ în cazul cel mai nefavorabil. D.E. Knuth și V.R. Pratt au utilizat construcția laborioasă din teorema lui Cook și au elaborat un algoritm care, mai apoi, a fost rafinat într-un algoritm practic și simplu. J.H. Morris a descoperit același algoritm în timpul implementării unui editor de texte. Este unul dintre numeroasele exemple când un rezultat pur teoretic poate conduce la rezultate cu aplicabilitate imediată. Algoritmul dat de Knuth, Morris și Pratt a fost publicat de abia în 1976. Între timp, R.S. Boyer și J.S. Moore (și indepedent W. Gosper) au descoperit un algoritm care este mult mai rapid în multe situații. În 1980, R.M. Karp și M.O. Rabin au proiectat un algoritm cu o descriere foarte simplă și care poate fi extins la texte și "pattern"-uri bidimensionale, deci foarte util la procesarea imaginilor grafice.

Plan

Descriere - noțiunea de simbol

Acest algoritm utilizează tehnica tabelelor de dispersie (hash).

Descriere - noțiunea de simbol

Acest algoritm utilizează tehnica tabelelor de dispersie (hash). Un simbol este o secvență de m caractere.

Descriere - noțiunea de simbol

Acest algoritm utilizează tehnica tabelelor de dispersie (hash). Un simbol este o secvență de m caractere.
Exemplu: dacă $m = 3$, `dar` este un simbol.

Descriere - noțiunea de simbol

Acest algoritm utilizează tehnica tabelelor de dispersie (hash). Un simbol este o secvență de m caractere.
Exemplu: dacă $m = 3$, `dar` este un simbol.

Descriere - așezarea simbolurilor în tabele hash

$h(s)$	s
0	aaa
1	aab
...	
2045	dar
...	
17575	zzz

Descriere - așezarea simbolurilor în tabele hash

$h(s)$	s
0	aaa
1	aab
...	
2045	dar
...	
17575	zzz

Să ne imaginăm că toate simbolurile posibile sunt memorate într-o tabelă de dispersie foarte mare, astfel încât nu există coliziune.

Descriere - potrivirea pe o poziție

Cum testăm dacă patternul p apare la poziția i în textul s?

Descriere - potrivirea pe o poziție

Cum testăm dacă patternul p apare la poziția i în textul s?
Este suficient să verificăm dacă $h(p) = h(s[i..i + m - 1])$.

Descriere - calculul valorii funcției hash

Este suficient să calculăm $h(p)$ o singură dată:

Descriere - calculul valorii funcției hash

Este suficient să calculăm $h(p)$ o singură dată:

```
val_hash_pattern = h(p)
for (i = 0; i < n - m; ++i) {
  if (val_hash_pattern == h(s[i .. i + m - 1])) {
    return i;
  }
}
return -1;
```

Descriere - calculul valorii funcției hash

Este suficient să calculăm $h(p)$ o singură dată:

```
val_hash_pattern = h(p)
for (i = 0; i < n - m; ++i) {
  if (val_hash_pattern == h(s[i .. i + m - 1])) {
    return i;
  }
}
return -1;
```

Totuși, calculul `h(s[i .. i + m - 1])` se efectuează la fiecare iterație a buclei.

Descriere - calculul valorii funcției hash

Este suficient să calculăm $h(p)$ o singură dată:

```
val_hash_pattern = h(p)
for (i = 0; i < n - m; ++i) {
  if (val_hash_pattern == h(s[i .. i + m - 1])) {
    return i;
  }
}
return -1;
```

Totuși, calculul `h(s[i .. i + m - 1])` se efectuează la fiecare iterație a buclei. **Cum putem evita să refacem calculul** `h(s[i .. i + m - 1])` **la fiecare iterație?**

Descriere - calculul valorii funcției hash

Cum putem evita să refacem calculul `h(s[i .. i + m - 1])` **la fiecare iterație?**

Idee:

Descriere - calculul valorii funcției hash

Cum putem evita să refacem calculul `h(s[i .. i + m - 1])` **la fiecare iterație?**

Idee:

Definim funcția *h* astfel încât să putem calcula foarte rapid `h(s[i .. i + m - 1])` în funcție de `h(s[i - 1 .. i + m - 2])`.

Descriere - calculul valorii funcției hash

Cum putem evita să refacem calculul `h(s[i .. i + m - 1])` **la fiecare iterație?**

Idee:

Definim funcția *h* astfel încât să putem calcula foarte rapid `h(s[i .. i + m - 1])` în funcție de `h(s[i - 1 .. i + m - 2])`.

```
val_hash_pattern = h(p)
val_hash_subsir = h(s[0 .. m - 1])
for (i = 0; i < n - m; ++i) {
  if (val_hash_pattern == val_hash_subsir) {
    return i;
  }
  val_hash_subsir = update(val_hash_subsir);
}
return -1;
```

Funcția de dispersie (hash) - idee

Un mod convenabil de a defini funcția de dispersie este următorul:

Funcția de dispersie (hash) - idee

Un mod convenabil de a defini funcția de dispersie este următorul:

Se consideră fiecare șir de m caractere ca fiind reprezentarea unui număr întreg în baza d, unde d este numărul de caractere din alfabet.

Funcția de dispersie (hash) - idee

Un mod convenabil de a defini funcția de dispersie este următorul:

Se consideră fiecare șir de m caractere ca fiind reprezentarea unui număr întreg în baza d, unde d este numărul de caractere din alfabet.

Exemplu: Dacă alfabetul este $\{a, b, \ldots, z\}$, atunci patternul `dar` are asociat numărul $3 * 26^2 + 0 * 26^1 + 17 = 2045$.

Funcția de dispersie (hash) - definiție

În general, numărul corespunzător unui șir t de lungime l este:

$$x = t[0]d^{l-1} + t[1]d^{l-2} + \cdots + t[l-1]d^0$$

Funcția de dispersie (hash) - definiție

În general, numărul corespunzător unui șir t de lungime l este:

$$x = t[0]d^{l-1} + t[1]d^{l-2} + \cdots + t[l-1]d^0$$

Funcția de dispersie h va fi definită prin

$$h(t) = x \bmod q$$

, unde q este un număr prim foarte mare.

Funcția de dispersie (hash) - definiție

În general, numărul corespunzător unui șir t de lungime l este:

$$x = t[0]d^{l-1} + t[1]d^{l-2} + \cdots + t[l-1]d^0$$

Funcția de dispersie h va fi definită prin

$$h(t) = x \bmod q$$

, unde q este un număr prim foarte mare.
În formula de mai sus, am făcut un abuz de notație: prin $t[i]$ înțelegem atât caracterul $t[i]$, cât și indexul acestui caracter în alfabet.

Algoritmul Rabin-Karp - calculul funcției hash

Cum calculăm eficient formula
$h(t, l) = t[0]d^{l-1} + t[1]d^{l-2} + \cdots + t[l-1]d^0$.

Algoritmul Rabin-Karp - calculul funcției hash

Cum calculăm eficient formula
$h(t, l) = t[0]d^{l-1} + t[1]d^{l-2} + \cdots + t[l-1]d^0$.

```
h(t, l)
{
  s = 0;
  p = 1;
  for (i = l - 1; i >= 0; --i) {
    // invariant: p = pow(d, l - 1 - i)
    s = s + t[i] * p;
    p = p * d;
    s = s % q;
  }
  return s;
}
```

Calculul funcției hash pentru fiecare subșir

Cum putem găsi rapid valoarea `y = h(s[i..i + m - 1])` știind valoarea `x = h(s[i - 1..i + m - 2])`?

Calculul funcției hash pentru fiecare subșir

Cum putem găsi rapid valoarea `y = h(s[i..i + m - 1])` știind valoarea `x = h(s[i - 1..i + m - 2])`?

$$y = \quad (s_i d^{m-1} + s_{i+1} d^{m-2} + \cdots + s_{i+m-2} d^1 + s_{i+m-1} d^0) \bmod q$$

Calculul funcției hash pentru fiecare subșir

Cum putem găsi rapid valoarea `y = h(s[i..i + m - 1])` știind valoarea `x = h(s[i - 1..i + m - 2])`?

$$\begin{aligned} y = \quad & (s_i d^{m-1} + s_{i+1} d^{m-2} + \cdots + s_{i+m-2} d^1 + s_{i+m-1} d^0) \bmod q \\ = \quad & (s_{i-1} d^m + s_i d^{m-1} + s_{i+1} d^{m-2} + \cdots + s_{i+m-2} d^1 + s_{i+m-1} d^0 - s_{i-1} d^m) \bmod q \end{aligned}$$

Calculul funcției hash pentru fiecare subșir

Cum putem găsi rapid valoarea `y = h(s[i..i + m - 1])` știind valoarea `x = h(s[i - 1..i + m - 2])`?

$$
\begin{aligned}
y = \; & (s_i d^{m-1} + s_{i+1} d^{m-2} + \cdots + s_{i+m-2} d^1 + s_{i+m-1} d^0) \bmod q \\
= \; & (s_{i-1} d^m + s_i d^{m-1} + s_{i+1} d^{m-2} + \cdots + s_{i+m-2} d^1 + s_{i+m-1} d^0 - s_{i-1} d^m) \bmod q \\
= \; & (d(s_{i-1} d^{m-1} + s_i d^{m-2} + s_{i+1} d^{m-2} + \cdots + s_{i+m-2} d^0) + s_{i+m-1} d^0 - s_{i-1} d^m) \bmod q
\end{aligned}
$$

Calculul funcției hash pentru fiecare subșir

Cum putem găsi rapid valoarea `y = h(s[i..i + m - 1])` știind valoarea `x = h(s[i - 1..i + m - 2])`?

$$
\begin{aligned}
y = \quad & (s_i d^{m-1} + s_{i+1} d^{m-2} + \cdots + s_{i+m-2} d^1 + s_{i+m-1} d^0) \bmod q \\
= \quad & (s_{i-1} d^m + s_i d^{m-1} + s_{i+1} d^{m-2} + \cdots + s_{i+m-2} d^1 + s_{i+m-1} d^0 - s_{i-1} d^m) \bmod q \\
= \quad & (d(s_{i-1} d^{m-1} + s_i d^{m-2} + s_{i+1} d^{m-2} + \cdots + s_{i+m-2} d^0) + s_{i+m-1} d^0 - s_{i-1} d^m) \bmod q \\
= \quad & (dx + s_{i+m-1} d^0 - s_{i-1} d^m) \bmod q
\end{aligned}
$$

Algoritmul Rabin-Karp - update-ul funcției hash

Știind $x = h(s[i-1..i+m-2])$, cum calculăm eficient $y = h(s[i..i+m-1])$?

Algoritmul Rabin-Karp - update-ul funcției hash

Știind $x = h(s[i-1..i+m-2])$, cum calculăm eficient $y = h(s[i..i+m-1])$?
Trebuie să știm, bineînțeles, și valorile $c_{old} = s[i-1], c_{new} = s[i+m-1]$.

Algoritmul Rabin-Karp - update-ul funcției hash

Știind $x = h(s[i-1..i+m-2])$, cum calculăm eficient $y = h(s[i..i+m-1])$?
Trebuie să știm, bineînțeles, și valorile $c_{old} = s[i-1], c_{new} = s[i+m-1]$.

```
update(x, cnew, cold) {
  return (d * x + cnew + q - (cold * pow(d, m)) % q) % q;
}
```

Algoritmul Rabin-Karp - update-ul funcției hash

Știind $x = h(s[i-1..i+m-2])$, cum calculăm eficient
$y = h(s[i..i+m-1])$?
Trebuie să știm, bineînțeles, și valorile $c_{old} = s[i-1], c_{new} = s[i+m-1]$.

```
update(x, cnew, cold) {
  return (d * x + cnew + q - (cold * pow(d, m)) % q) % q;
}
```

Codul corespunde formulei obținute pe slide-ul precedent:
$y = (dx + s[i+m-1]d^0 - s[i-1]d^m) \bmod q$.

Algoritmul Rabin-Karp

```
val_hash_pattern = h(p)
val_hash_subsir = h(s[0 .. m - 1])
for (i = 0; i < n - m; ++i) {
  if (val_hash_pattern == val_hash_subsir) {
    if (apare_la_pozitia(s, n, p, m, i)) {
      return i;
    }
  }
  val_hash_subsir = update(val_hash_subsir, s[i], s[i + m]);
}
return -1;
```

Algoritmul Rabin-Karp

```
val_hash_pattern = h(p)
val_hash_subsir = h(s[0 .. m - 1])
for (i = 0; i < n - m; ++i) {
  if (val_hash_pattern == val_hash_subsir) {
    if (apare_la_pozitia(s, n, p, m, i)) {
      return i;
    }
  }
  val_hash_subsir = update(val_hash_subsir, s[i], s[i + m]);
}
return -1;
```

Observație: din cauza coliziunilor posibile, dacă valorile funcțiilor hash corespund pe două subșiruri, trebuie să ne asigurăm, folosind de exemplu funcția `apare_la_pozitia`, ca cele două șiruri sunt într-adevăr egale.

Observații

- Pentru a evita lucrul cu numere mari, operațiile se execută modulo un număr natural q.

Observații

- Pentru a evita lucrul cu numere mari, operațiile se execută modulo un număr natural q.
- Ne putem imagina ca q este dimensiunea "tabelei" hash care memorează "simbolurile" (= secvențele de m caractere) din s.

Observații

- Pentru a evita lucrul cu numere mari, operațiile se execută modulo un număr natural q.
- Ne putem imagina ca q este dimensiunea "tabelei" hash care memorează "simbolurile" (= secvențele de m caractere) din s.
- Pentru o dispersie bună, q trebuie să fie un număr prim mare.

Observații

- Pentru a evita lucrul cu numere mari, operațiile se execută modulo un număr natural q.
- Ne putem imagina ca q este dimensiunea "tabelei" hash care memorează "simbolurile" (= secvențele de m caractere) din s.
- Pentru o dispersie bună, q trebuie să fie un număr prim mare.
- În practică, dacă q este bine ales, numărul de coliziuni este mic și se poate considera că algoritmul rulează în timp $O(n+m)$.

Exercițiu: incercați să găsiți s, p și q astfel încât timpul de rulare al algoritmului să fie $O(n \cdot m)$.

Algoritmul Rabin-Karp: o posibilă implementare C

```
#define REHASH(a, b, h) ((((h)-(a)*dM) << 1) (b))
int RK(char *p, int m, char *s, int n) {
  long dM, hs, hp, i, j;
  /* Preprocesare */
  for (dM = i = 1; i < m; ++i) dM = (dM << 1); // d = 2
  for (hp = hs = i = 0; i < m; ++i) {
    hp = ((hp << 1) + p[i]);
    hs = ((hs << 1) + s[i]);
  }
  /* Cautare */
  i = 0;
  while (i <= n-m) {
    if (hp == hs && memcmp(p, s + i, m) == 0) return i;
    hs = REHASH(s[i], s[i + m], hs);
    ++i;
  }
  return -1;
}
```

sursa: http://www-igm.univ-mlv.fr/ lecroq/string/node5.html

Analiza implementării C

- S-a presupus că $d = 2$. Dacă $d = 2^k$, atunci se poate utiliza relația $d^{m-1} = (2^k)^{m-1} = (2^{m-1})^k$.

Analiza implementării C

- S-a presupus că $d = 2$. Dacă $d = 2^k$, atunci se poate utiliza relația $d^{m-1} = (2^k)^{m-1} = (2^{m-1})^k$.
- Dacă `long` este reprezentat pe 64 biți, atunci `dM` este 0 pentru $m = 65$. Deci trebuie să avem $m \leq \frac{65}{k}$.

Analiza implementării C

- S-a presupus că $d = 2$. Dacă $d = 2^k$, atunci se poate utiliza relația $d^{m-1} = (2^k)^{m-1} = (2^{m-1})^k$.
- Dacă `long` este reprezentat pe 64 biți, atunci `dM` este 0 pentru $m = 65$. Deci trebuie să avem $m \leq \frac{65}{k}$.
- Nu mai este utilizat numărul prim q deoarece se utilizează aritmetica modulară peste long.

Analiza implementării C

- S-a presupus că $d = 2$. Dacă $d = 2^k$, atunci se poate utiliza relația $d^{m-1} = (2^k)^{m-1} = (2^{m-1})^k$.
- Dacă `long` este reprezentat pe 64 biți, atunci `dM` este 0 pentru $m = 65$. Deci trebuie să avem $m \leq \frac{65}{k}$.
- Nu mai este utilizat numărul prim q deoarece se utilizează aritmetica modulară peste long.

O implementare Java cu q determinat aleatoriu (Robert Sedgewick and Kevin Wayne) poate fi gasita la adresa http://algs4.cs.princeton.edu/53substring/RabinKarp.java.html.

Plan

Exemplu

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
		$\neq$																
I	A	R																

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{1}$

Exemplu

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
					$\neq$													
			I	A	R													

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{2}$

Exemplu

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
								$\neq$										
						I	A	R										

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{3}$

Exemplu

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
											$\neq$							
									I	A	R							

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{4}$

Exemplu

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
														$\neq$				
												I	A	R				

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{5}$

Exemplu

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
																	$\neq$	
															I	A	R	

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{6}$

Exemplu

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
																		I

19	20	21	22	23	24
	I	A	R	N	A
	$\neq$				
A	R				

7

Exemplu

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E

19	20	21	22	23	24
	I	A	R	N	A
			=		
	I	A	R		

8

Exemplu

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E

19	20	21	22	23	24
	I	A	R	N	A
		=			
	I	A	R		

$\boxed{9}$

Exemplu

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E

19	20	21	22	23	24
	I	A	R	N	A
	=				
	I	A	R		

$\boxed{10}$

Exemplu

Prima dată se compară R (ultimul caracter din "pattern") cu S (al treilea caracter din text). Deoarece S nu apare în "pattern", se deplasează "pattern"-ul cu trei poziții (lungimea sa) la dreapta. Apoi se compară R cu caracterul spațiu. Nici caracterul spațiu nu apare în "pattern" așa că acesta se deplasează din nou la dreapta cu trei poziții. Procesul continuă până când R este comparat cu I (al 21-lea caracter din text). Deoarece I apare în "pattern" pe prima poziție se deplasează "pattern"-ul la dreapta cu două poziții. Apoi se compară R cu R, deci există potrivire. Se continuă comparația cu penultimul caracter din "pattern" (de fapt al doilea) și precedentul din text (al 22-lea). Se obține din nou potrivire și se compară următoarele două caractere de la stânga (primul din "pattern" și al 21-lea din text). Deoarece există potrivire și "pattern"-ul a fost parcurs complet, rezultă că s-a determinat prima apariție a "pattern"-ului în text.

Regula caracterului rău 1/3

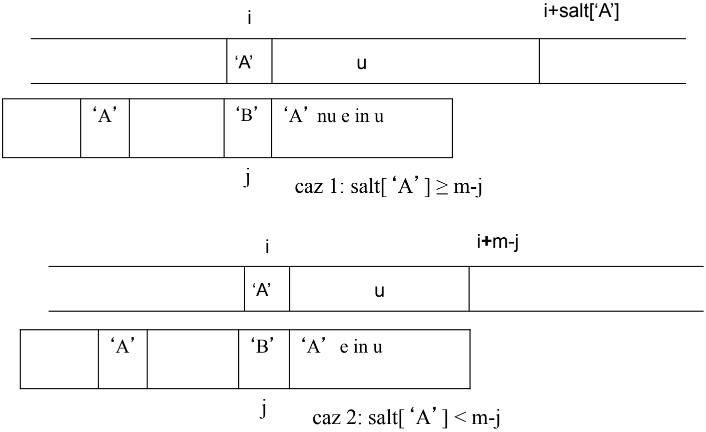
Engleză: "bad character shift rule"

Evită repetarea comparațiilor fără succes în cazul caracterelor care nu apar în pattern sau într-un sufix (maximal) al acestuia.

$$salt[C] = \begin{cases} m - \text{poziţia ultimei apariţii} & \text{, dacă } C \text{ apare în pattern} \\ \text{a lui } C \text{ în pattern} & \\ m & \text{, altfel} \end{cases}$$

(alternativ, $salt(C) = \max(\{0\} \cup \{i < m \mid p[i] = C\})$)

Regula caracterului rău 2/3



Primul caz: `i = i + salt[s[i]];`
Al doilea caz: `i = i + m - j;`

Regula caracterului rău 3/3

Dacă $p[j] \neq s[i] = C$,

1. dacă apariția cea mai din dreapta a lui C în p este $k < j$, $p[k]$ și $s[i]$ sunt aliniate (i = i + salt[s[i]])
2. dacă apariția cea mai din dreapta a lui C în p este $k > j$, p este translatat la dreapta cu o poziție (i = i + m - j)
3. dacă C nu apare în p, patternul p este aliniat cu $s[i+1..i+m]$ (i = i + m). Devine caz particular al primului dacă $salt[s[i]] = m$.

Algoritmul Boyer-Moore (varianta 1)

```
BM(s, n, p, m, salt) {
  i = m-1; j = m-1;
  repeat
    if (s[i] = p[j]) {
      i = i-1;
      j = j-1;
    }
    else if ((m-j) > salt[s[i]]) i = i+m-j;
    else i = i+salt[s[i]];
    j = m-1;
  until (j<0 or i>n-1);
  if (j<0)  return i+1;
  else return -1;
}
```

Analiza

Pentru cazul cel mai nefavorabil are complexitatea $O(m \cdot n)$.

În schimb are o comportare bună în medie.

Vom vedea o altă versiune mai târziu care este liniară.

Plan

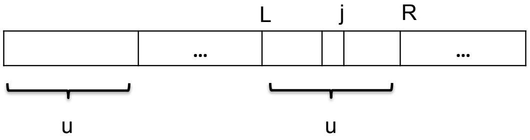
Domeniul problemei: definiția pentru $Z[j]$

Fie $p[0..m-1]$ un șir.

$Z[j] =$ lungimea celui mai lung subșir care pleacă din $p[j]$ și este prefix al lui p.

Exercițiu. Să se dea o definiție formală pentru $Z[j]$.

Algoritmul Z - invariant



Algorimul Z, care calculează valorile $Z[j]$, $j = 1, \ldots, m-1$, menține următorul invariant:
$[L, R]$ este un interval cu R maxim astfel încât
$1 \leq L \leq j \leq R$ și
$p[L..R]$ este prefix al lui p.

Dacă nu există un astfel de interval atunci $L = R = -1$.

Algoritmul Z - iterația curentă 1/3

Presupunem toate valorile până la $Z[j-1]$ calculate.
Pentru calculul lui $Z[j]$ distingem următoarele cazuri:

- $j > R$ (i.e. $j - 1 = R$). Nu există un prefix al lui p care se termină la j sau după j.
  Se resetează L și R pentru $p[0..]$ și $p[j..]$:

```
L = R = j;    // reseteaza L si R
while (R < M && p[R-L] == p[R])
   R = R + 1;
// p[j..R-1] cel mai lung prefix
Z[j] = R - L;  // = |p[j..R-1]|
R = R -1;
```

Algoritmul Z - iterația curentă 2/3

- $j \leq R$. $[L, R]$ se extinde. Fie $k = j - L$. Avem $Z[j] \geq \min\{Z[k], R - j + 1\}$ (explicații pe tablă)
  Distingem subcazurile:
  - $Z[k] < R - j + 1$. Nu există prefix mai lung care pleacă din j:
    ```
    Z[j] = Z[k];
    ```
  - $Z[k] \geq R - j + 1$. Există un prefix mai lung ce pleacă din j. Se resetează L la j si se calculează noul R pornind de la $R + 1$:
    ```
    L = j;
    while (R < m && p[R-L] == p[R])
      R = R + 1;
    // p[L..R-1] cel mai lung prefix
    Z[j] = R - L;  // = |p[L..R-1]|
    R = R - 1;
    ```

Algoritmul Z - iterația curentă 3/3

asamblăm cazurile:

```
if (j > R) {
  L = R = j;
  while (R < M && p[R-L] == p[R])
     R = R + 1;
  Z[j] = R - L;
  R = R -1;
} else {
  k = j - L;
  if (Z[k] < R-j+1)
    Z[j] = Z[k];
  else {
    L = j;
    while (R < m && p[R-L] == p[R])
      R = R + 1;
    Z[j] = R - L;
    R = R - 1;
  }
```

Plan

Motivație

Regula caracterului rău nu este eficientă dacă alfabetul este mic.

În astfel de cazuri se poate câștiga în eficiență dacă se consideră sufixele potrivite deja.

Asta este realizată de regula sufixului bun.

Regula sufixului bun: ilustrare caz 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		17	18	19	20	21	22	23	24
a	c	c	a	c	b	b	c	a	b	b	a	c	b	a	b	a	b	b	a	b	b	c	b	a	b

a	a	b	a	b	b	c	b	a	b
0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		17	18	19	20	21	22	23	24
a	c	c	a	c	b	b	c	a	b	b	a	c	b	a	b	a	b	b	a	b	b	c	b	a	b

a	a	b	a	b	b	c	b	a	b
0	1	2	3	4	5	6	7	8	9

Regula sufixului bun: cazul 1 formal

Cazul 1:
dacă $p[j-1]$ nu se potrivește și p include o copie a lui $p[j..m-1]$ precedată de un caracter $\neq p[j-1]$, se face salt la cea mai apropiată copie din stânga cu această proprietate.

Regula sufixului bun: ilustrare caz 2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		17	18	19	20	21	22	23	24
a	b	c	a	c	b	b	c	a	b	b	a	c	b	a	b	a	b	b	a	b	b	c	b	a	b

a	b	b	a	b	c	b	c	a	b
0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		17	18	19	20	21	22	23	24
a	b	c	a	c	b	b	c	a	b	b	a	c	b	a	b	a	b	b	a	b	b	c	b	a	b

a	b	b	a	b	c	b	c	a	b
0	1	2	3	4	5	6	7	8	9

Regula sufixului bun: cazul 2 formal

Cazul 2:
– dacă nu se aplică regula din cazul 1, se face saltul cel mai mic astfel încât un sufix al lui $s[0..i]$ se potrivește cu un prefix al lui p

– dacă cel mai lung prefix al lui $s[0..i]$ care se potrivește peste un prefix al lui p este șirul vid, atunci se face un salt cu m poziții

$goodSuff(j)$ - definiție (cazul 1)

$goodSuff(j) =$ poziția de sfârșit a apariției lui $p[j..m-1]$ cea mai apropiataă de j și care nu este precedată de $p[j-1]$.

Dacă nu există o astfel de copie, $goodSuff(j) = 0$.

Avem $0 \leq goodSuff(j) < m - 1$;

– dacă $goodSuff(j) > 0$ atunci el reprezintă o copie a unui "sufix bun" care permite un salt de $m - goodSuff(j)$ poziții

– deoarece $p[m..m-1]$ este șirul vid, $goodSuff(m) =$ cea mai din dreapta poziție k cu $p[k] \neq p[m-1]$ (-1 dacă toate caracterele sunt egale).

$goodSuff(j)$ - calcul

$lcs(j, p) =$ lungimea celui mai lung sufix comun lui $p[0..j]$ și p.

Lemă

Valorile $lcs(j, p)$ pot fi calculate în timpul $O(m)$.

Exercițiu Să se demonstreze lema. (se utilizează algoritmul Z adaptat).

Teoremă

Dacă $goodSuff(j) > 0$, atunci $goodSuff(j)$ este cel mai mare $k < m - 1$ cu $lcs(k, p) = m - j$ $(= |p[j..m-1]|)$.

Rezultă că valorile $goodSuff(j)$ pot fi calculate în timpul $O(m)$.

Preprocesarea în cazul 2

$lp(j)$ = lungimea celui mai lung prefix al lui p care este sufix al lui $p[j..m-1]$.

Lemă

$lp(j) = \max\{k \mid 0 \leq k \leq |p[j..m-1]| \wedge lcs(k,p) = k\}$.

Exercițiu Să se demonstreze lema.

Rezultă că valorile $lp(j)$ pot fi calculate în timpul $O(m)$.

Regula sufixului bun

Presupunem că $p[j-1]$ nu se potrivește (după ce s-au potrivit $p[j..m-1]$.

1. dacă $goodSuff(j) > 0$, face un salt egal cu $m - goodSuff(j)$ (cazul 1)
2. dacă $goodSuff(j) = 0$, face un salt egal cu $m - lp(j)$ (cazul 2)

Dacă $p[m-1]$ nu se potrivește, atunci $j = m$ și saltul este corect.

Algoritmul Boyer-Moore (versiunea 2)

```
BM(s, n, p, m, goodSuff, lp) {
  k = m-1;
  while (k < n) {
    i = k;  j = m-1;
    while (j > 0 && p[j] == s[i]) {
      i = i-1;
      j = j-1;
    }
    if (j < 0) return i+1;
    // nepotrivire pe poziţia p[j]
    măreşte k cu maximul dintre salturile date de
    regula caracterului rău şi regula sufixului bun
  }
}
```

Analiza

Teoremă

Algoritmul `BM` *execută totdeauna cel mult* $m + n$ *comparații de caractere și aproximativ* $\frac{n}{m}$ *salturi când alfabetul nu este mic și "pattern"-ul nu este prea lung.*

Observație: Dacă alfabetul are numai două caractere (cazul șirurilor binare), atunci performanțele algoritmului *BM* nu sunt cu mult mai bune decât cele ale căutării naive. În acest caz se recomandă împărțirea șirurilor în grupe cu un număr fixat de biți. Fiecare grupă reprezintă un caracter. Dacă dimensiunea unei grupe este k atunci vor exista 2^k caractere, i.e. dintr-un alfabet mic obținem unul cu multe caractere. Totuși, k va trebui ales suficient de mic pentru ca dimensiunea tabelei de salturi să nu fie prea mare. $\boxed{\text{sfobs}}$