Outline

Cuprins

1 Domeniul problemei

Alfabet, șir

1. Alfabet = mulţime nevidă A de caractere Exemple: $A_1 = \{A, a, B, b, C, c, \ldots, Z, z\}$ (caracterele sunt literele din alfabetul englez),
   $A_2 = \{0, 1\}$(caracterele sunt cifrele binare), $A_3 = \mathbb{N}$ (caracterele sunt numerele naturale, alfabet infinit)
2. șir de caractere (string): o secvență finită de caractere din A
3. șirul vid: ε
4. lungimea uni șir s (sau $length(s)$ sau $s.length()$): numărul de caractere din șir
   $|\varepsilon| = 0$
5. $s[i]$ - caracterul de pe poziția i $(0 \leq i \leq |s|)$;
6. subșir: $s[i..j] = s[i]s[i+1] \ldots s[j]$ $(i \leq j)$

Mulțimea șirurilor A^*

- concatenarea (produsul) a două șiruri s_1 și s_2: este șirul s_1 urmat de s_2
  $s_1 s_2 = s_1[0] \ldots s_1[|s_1|]s_2[0] \ldots s_2[|s_2|]$
  $|s_1 s_2| = |s_1| + |s_2|$
- $s\varepsilon = \varepsilon s = s$
- A^* este mulțimea tuturor șirurilor cu caractere din A
  A^* este monoidul liber generat de A
- un limbaj este o submulțime $L \subseteq A^*$

Factor, subsecvență

- x este un factor al lui s dacă există $u, v.$ a.î. $s = uxv$
  factor și *subșir* sunt echivalente
  x este un factor propriu dacă $x \neq s$ (echivalent $uv \neq \epsilon$)
- x este un prefix al lui s dacă există $v.$ a.î. $s = xv$
  scriem $x \leq_{pref} s$
- x este un sufix al lui s dacă există $u.$ a.î. $s = ux$
  scriem $x \leq_{suff} s$
- x este subsecvență a lui s dacă exista $|x| + 1$ șiruri $w_0, w_1, \ldots, w_{|x|}$ a.î. $s = w_0x[0]w_1x[1]\ldots x[|x| - 1]w_{|x|}$ (i.e., x este obținut prin ștergerea a $|y| - |x|$ caractere din s); notăm $x \leq_{sseq} s$

Ordonare lexicografică

- ordonare lexicografică:
  Presupunem o relație de ordine totală $\leq$ peste A. Extindem $\leq$ la $A^* \times A^*$ astfel;
  $s_1 \leq s_2$ ddacă $s_1 \leq_{pref} s_2$ sau există $u, v, w \in A^*$ și $a, b \in A$ a.î. $s_1 = uav$, $s_2 = ubw$ și $a < b$.

Exercițiu. Să se scrie în Alk funcții de decizie pentru ordonarea lexicografică.

Apariții

- x apare în s dacă x este un factor al lui s
- x are o poziție de start i în s dacă $s[i]\ldots s[i + |x| - 1] = x$
- x are o poziție de sfârșit j în s dacă $s[j - |x| + 1]\ldots s[j] = x$
- prima apariție a lui x în s este cea mai mică poziție de start (dacă există)

String Searching (Matching) Problem

Input Două şiruri: $s = s[0]\ldots s[n - 1]$, numit subiect sau text, şi $p = p[0]\cdots p[m - 1]$, numit pattern.

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

Variantă: găsirea tuturor aparițiilor:

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

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

Algoritmul naiv (funcție ajutătoare)

```
/*
  @input: strings s[0..n-1], p[0..m-1],
          a position i, 0 <= i < n
  @output: true,  if p <=_pref s[i..n-1]
           false, otherwise
*/
occAtPos(s, n, p, m, i)
{
  for (j = 0; j < m; ++j) {
    if (i + j >= n || s[i + j] != p[j]) {
      return false;
    }
  }
  return true;
}
```

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

Algoritmul naiv

```
/*
  @input:  strings s[0..n-1], p[0..m-1]
  @ouput:  the first occurence of p in s, if any
           -1, otherwise
*/
firstOcc(s, n, p, m)
{
  for (i = 0; i < n; ++i) {
    if (occAtPos(s, n, p, m, i)) {
      return i;
    }
  }
  return -1;
}
```

Algoritmul naiv: complexitatea în cazul cel mai nefavorabil

Complexitate: $O(n \cdot m)$ în cazul cel mai nefavorabil, $O(min(n, m))$ în cazul cel mai favorabil. [3ex] *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?

Algoritmul naiv: complexitatea timp medie

Presupunem că alfabetul A are d caractere, $d \geq 2$.

$$X_{ij} = \begin{cases} 1 & \text{, s[i-1] și p[j-1] sunt comparate} \\ 0 & \text{, } altfel \end{cases}$$

$p_{ij} = Prob(X_{ij} = 1) = \frac{1}{d^{j-1}}$ (trebuie ca primele $j-1$ caractere din p să se potrivească)
numărul de comparații = $\sum_{i=1}^{n+1-m} \sum_{j=1}^{m} X_{ij}$
numărul mediu de comparații = $M(\sum_{i=1}^{n+1-m} \sum_{j=1}^{m} X_{ij}) = \sum_{i=1}^{n+1-m} \sum_{j=1}^{m} M(X_{ij}) = \sum_{i=1}^{n+1-m} \sum_{j=1}^{m} p_{ij} = \sum_{i=1}^{n+1-m} \sum_{j=1}^{m} \frac{1}{d^{j-1}}$
Avem $\sum_{i=1}^{n+1-m} \sum_{j=1}^{m} \frac{1}{d^{j-1}} = (n+1-m) \sum_{j=1}^{m} \frac{1}{d^{j-1}} \leq 2(n+1-m)$
Justificare:

$$\sum_{j=1}^{m} \frac{1}{d^{j-1}} =$$

$$\frac{1 - \frac{1}{d^m}}{1 - \frac{1}{d}} \leq$$

$$\frac{1}{1 - \frac{1}{d}} \leq$$

$$2$$

Întrebare: putem obține O(n) în cazul cel mai nefavorabil?

Puţin istoric

Algoritmii care rezolvă problema de căutării peste şiruri au o istorie interesantă. În 1970, S.A. Cook a demonstrat un rezultat teoretic despre un anumit tip abstract de maşină, unde se presupunea existenţa unui algoritm de căutare peste şiruri ce necesită un timp proporţional cu $n+m$ în cazul cel mai nefavorabil. D.E. Knuth şi V.R. Pratt au utilizat construcţia laborioasă din teorema lui Cook şi au elaborat un algoritm care, mai apoi, a fost rafinat într-un algoritm practic şi simplu. J.H. Morris a descoperit acelaşi 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 şi Pratt a fost publicat de abia în 1976. Între timp, R.S. Boyer şi J.S. Moore (şi indepedent W. Gosper) au descoperit un algoritm care este mult mai rapid în multe situaţii. În 1980, R.M. Karp şi M.O. Rabin au proiectat un algoritm cu o descriere foarte simplă şi care poate fi extins la texte şi "pattern"-uri bidimensionale, deci foarte util la procesarea imaginilor grafice.

2 Algoritmul Rabin-Karp

Descriere - noțiunea de simbol

Acest algoritm utilizează tehnica tabelelor de dispersie (hash).

Un simbol este o secvenţă de m caractere.

Exemplu: dacă $m = 3$, `bla` este un simbol.

Descriere - așezarea simbolurilor în tabele hash

$h(s)$	s
0	`aaa`
1	`aab`
...	
962	`bla`
...	
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?

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:

```
hashPatt = h(p)
for (i = 0; i < n - m; ++i) {
  if (hashPatt == 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:

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])`.

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

Funcţia de dispersie (hash) - idee

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

Se consideră fiecare şir 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 `bla` are asociat numărul $1 * 26^2 + 11 * 26^1 + 0 * 26^0 = 962$.

Funcţia de dispersie (hash) - definiţie

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

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

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

```
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])`? Notație: $s_i = s[i]$

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

Algoritmul Rabin-Karp - update-ul funcției hash

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

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

```
hashPatt = h(p)
hashStr = h(s[0 .. m - 1])
for (i = 0; i < n - m; ++i) {
  if (hashPatt == hashStr) {
    if (occAtPos(s, n, p, m, i)) {
      return i;
    }
  }
  hashStr = update(hashStr, 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, operaţiile se execută modulo un număr natural q.
- Ne putem imagina ca q este dimensiunea "tabelei" hash care memorează "simbolurile" (= secvenţele 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 relaţia $d^{m-1} = (2^k)^{m-1} = (2^{m-1})^k$.
- Dacă `long` este reprezentat pe 64 biţi, 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.

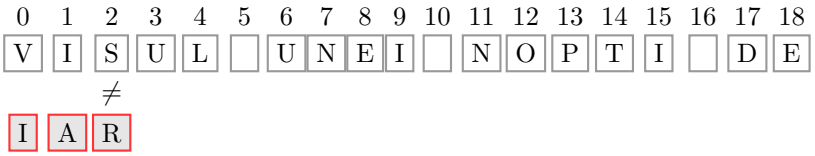
O implementare Java cu q determinat aleatoriu (Robert Sedgewick and Kevin Wayne) poate fi gasita la adresa http://algs4.cs.princeton.edu/53substring/RabinKarp.java.html.

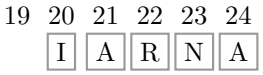
Algoritmul Rabin-Karp: sumar

- ar trebui sa fie ușor de comparat două valori hash
- ar trebui să fie ușor de calculat $h(s[i..i+m-1])$ din $h(s[i-1..i+m-2])$
- numărul coliziunilor este asemănător cu cel al distribuirii aleatorii șirurilor în găleți (buckets) (văzute aici ca elemente în $\mathbb{Z}_q$); valoarea așteptată $\frac{n}{q}$
- complexitatea în cazul cel mai nefavorabil $O(n \cdot m)$, dar foarte puțin probabil să apară în practică
- complexitatea medie $O(m + n)$
- extensibil la cazul bidimensional (imagini)

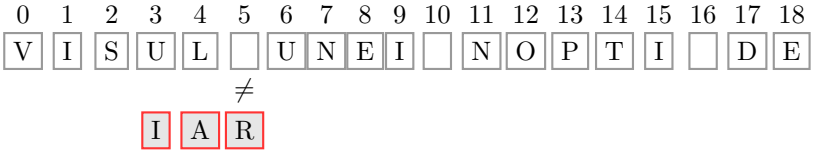
3 Algoritmul Boyer-Moore

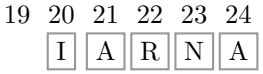
Exemplu


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 A R

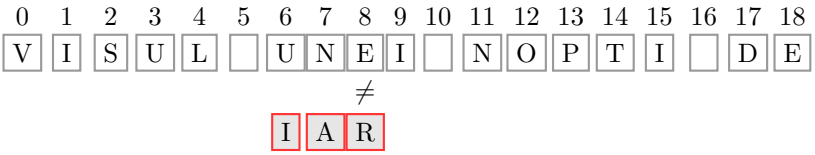

19 20 21 22 23 24
I A R N A

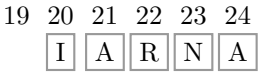
1


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 A R

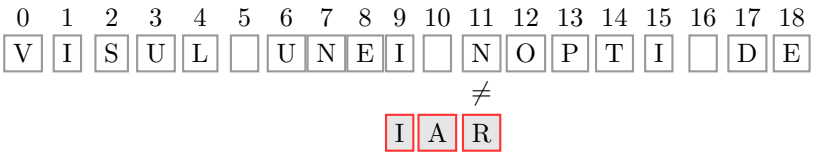

19 20 21 22 23 24
I A R N A

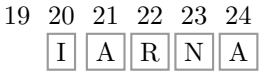
2


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 A R


19 20 21 22 23 24
I A R N A

3


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 A R


19 20 21 22 23 24
I A R N A

4

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	A	R

19	20	21	22	23	24
	I	A	R	N	A

5

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	A	R

19	20	21	22	23	24
	I	A	R	N	A

6

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

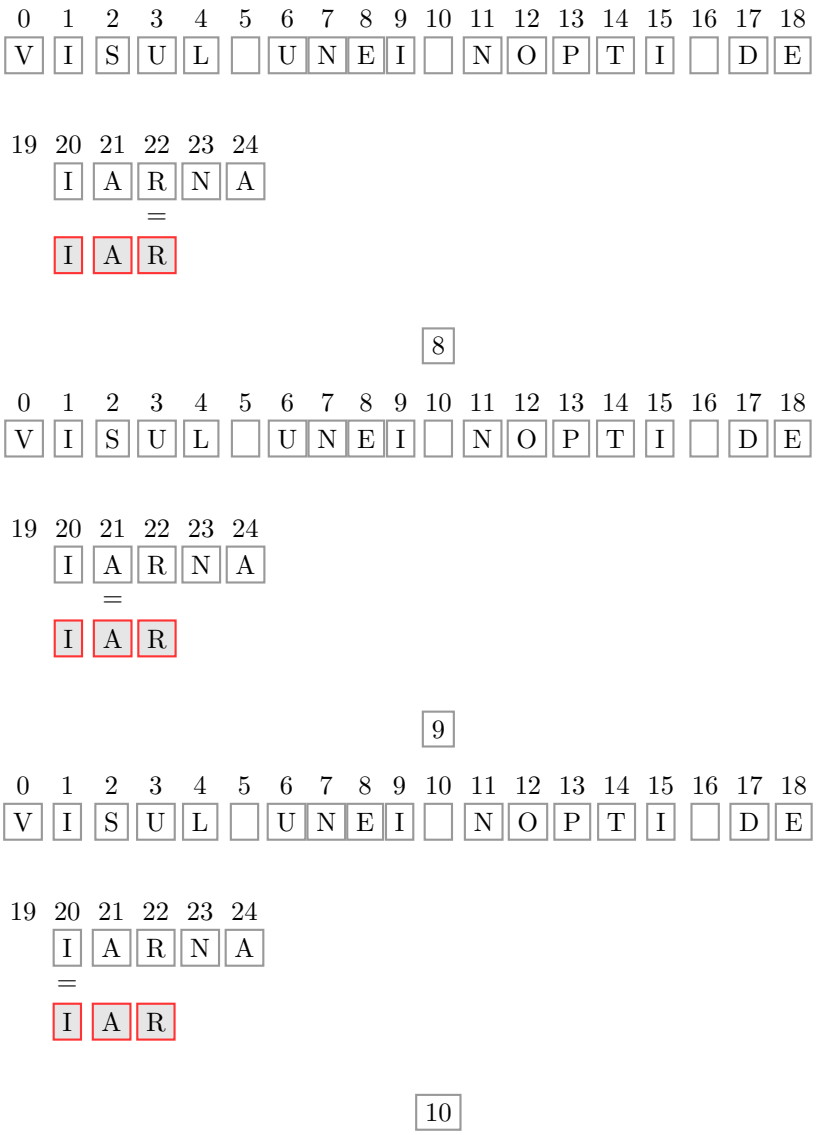
I

19	20	21	22	23	24
	I	A	R	N	A

≠

A	R

7



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 poziţii (lungimea sa) la dreapta. Apoi se compară R cu caracterul spaţiu. Nici caracterul spaţiu nu apare în "pattern" aşa că acesta se deplasează din nou la dreapta cu trei poziţii. Procesul continuă până când R este comparat cu I (al 21-lea caracter din text). Deoarece I apare în "pattern" pe prima poziţie se deplasează "pattern"-ul la dreapta cu două poziţii. Apoi se compară R cu R, deci există potrivire. Se continuă comparaţia cu penultimul caracter din "pattern" (de fapt al doilea) şi precedentul din text (al 22-lea). Se obţine din nou potrivire şi se compară următoarele două caractere de la stânga (primul din "pattern" şi al 21-lea din text). Deoarece există potrivire şi "pattern"-ul a fost parcurs complet, rezultă că s-a determinat prima apariţie a "pattern"-ului în text.

Regula caracterului rău 1/3

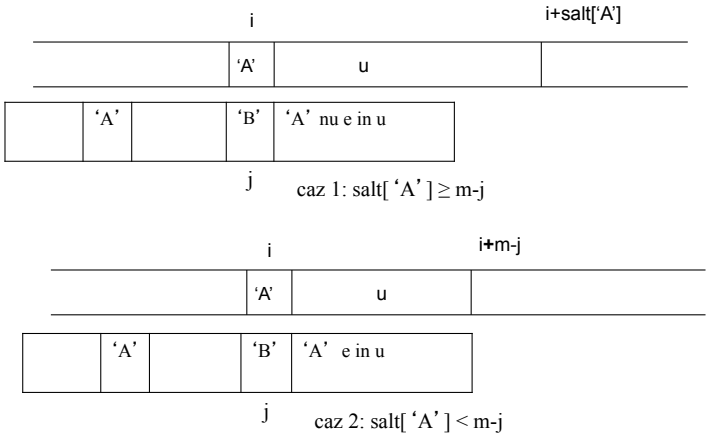
Engleză: "bad character shift rule"

Evită repetarea comparaţiilor 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 apariţia cea mai din dreapta a lui C în p este $k < j$, $p[k]$ şi $s[i]$ sunt aliniate (i = i + salt[s[i]])
2. dacă apariţia cea mai din dreapta a lui C în p este $k > j$, p este translatat la dreapta cu o poziţie (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.

4 Algoritmul Z

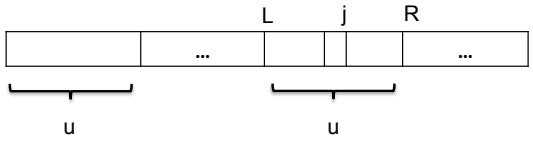
Domeniul problemei: definiţia pentru $Z[j]$

Fie $p[0..m-1]$ un şir.

$Z[j]$ = lungimea celui mai lung subşir care pleacă din poziția de start j şi este prefix al lui p, $j = 0, \ldots, m-1$.

Exerciţiu. Să se dea o definiţie formală pentru $Z[j]$.

Algoritmul Z - invariant



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

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

Algoritmul Z - iteraţia curentă 1/3

Presupunem toate valorile până la $Z[j-1]$ calculate. [1.5ex] Deci are loc invariantul pentru $L, j-1, R$ și trebui să calculăm noile valori pentru L și R ca sa aibă loc invariantul pentru L, j, R. [1.5ex] 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 şi R pentru $p[0..]$ şi $p[j..]$:

```
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 - iteraţia curentă 2/3

- $j \leq R$. $[L, R]$ se extinde. Fie $k = j - L$. Avem $Z[j] \geq \min\{Z[k], R - j + 1\}$ (*explicaţii 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 - iteraţia 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;
  }
```

Algoritmul Z pentru sufix

Exerciţiu. Să se proiecteze o versiune a algoritmului Z pentru cazul în care $Z[j]$ este lungimea celui mai lung subşir care pleacă din poziția de start j şi este sufix al lui p, $j = 0, \ldots, m - 1$.

5 Algoritmul Boyer-Moore revizuit

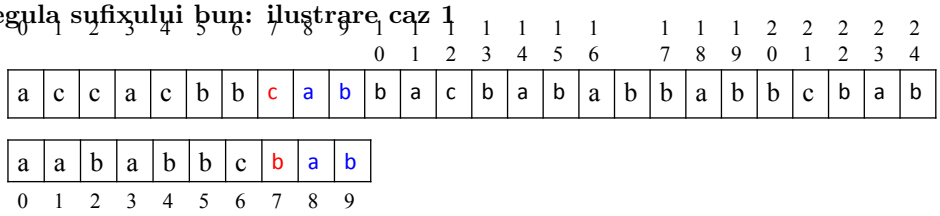
Motivaţie

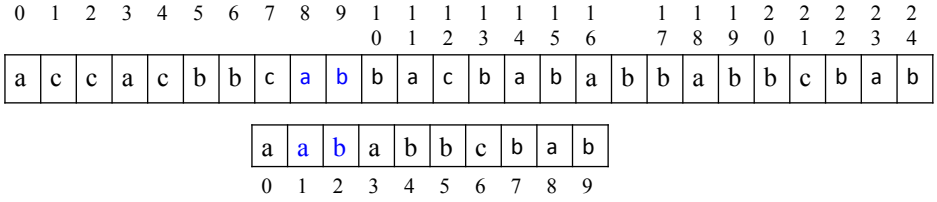
Regula caracterului rău nu este eficientă dacă alfabetul este mic.

În astfel de cazuri se poate câştiga în eficienţă dacă se consideră sufixele potrivite deja.

Asta este realizată de regula sufixului bun.

Regula sufixului bun: ilustrare caz 1



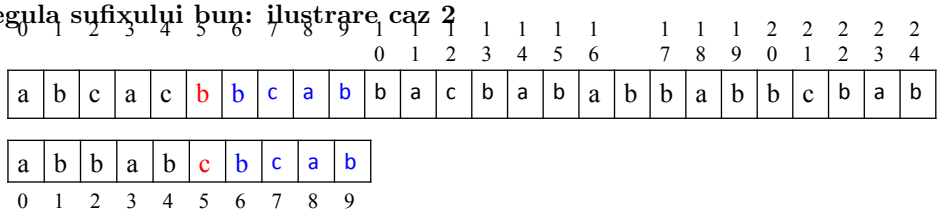


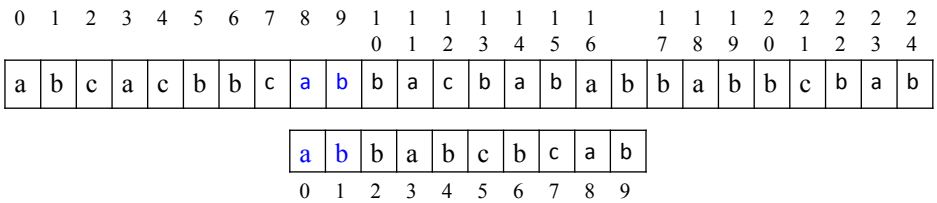
$i-1 = j-1 = 7$, $m = 10$, $s[i-1] \neq p[j-1]$, $p[j..m-1] = s[i..i+m-j-1]$ $p[1..2] = p[8..9]$, $p[0] \neq p[7]$ și $p[1..2]$ este cea mai apropiată de $p[8..9]$ cu această proprietate

Regula sufixului bun: cazul 1 formal

Cazul 1: dacă $p[j-1]$ nu se potriveşte şi 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





$i-1 = j-1 = 5$, $m = 10$, $p[j..m-1] = s[i+m-j-1,\ \ p[0..1] = s[8..9]$ este cel mai lung prefix al lui p care este sufix al lui $s[0..9]$

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 $s[0..i+m-j-1]$ se potriveşte cu un prefix al lui p

– dacă cel mai lung sufix al lui $s[0..i+m-j-1]$ care se potriveşte peste un prefix al lui p este şirul vid, atunci se face un salt cu m poziţii

– dar "sufix al lui $s[0..i+m-j-1]$ se potriveşte cu un prefix al lui p" este echivalent cu "sufix al lui p se potriveşte cu un prefix al lui p

– un factor propriu al uniu șir care este și prefix și sufix al acelui șir se numește frontieră (border)

goodSuff(j) **- definiţie (cazul 1)**

goodSuff(j) = poziţia de sfârşit a apariţiei lui $p[j..m-1]$ cea mai apropiată de j şi 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) poziţii

– deoarece $p[m..m-1]$ este şirul vid, *goodSuff*(m) = cea mai din dreapta poziţie 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 $p[0..j]$ şi p.

Lemă 1. *Valorile $lcs(j,p)$ pot fi calculate în timpul $O(m)$.*

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

Teoremă 1. *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 2. $lp(j) = \max\{k \mid 0 \leq k \leq |p[j..m-1]| \wedge lcs(k,p) = k\}$.

Exerciţiu 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 potriveşte (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 potriveşte, atunci $j = m$ şi 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 */
  }
}
```

Observaţia 1. *Dacă alfabetul are numai două caractere (cazul şirurilor binare), atunci performanţele algoritmului BM nu sunt cu mult mai bune decât cele ale căutării naive. În acest caz se recomandă împărţirea şirurilor în grupe cu un număr fixat de biţi. Fiecare grupă reprezintă un caracter. Dacă dimensiunea unei grupe este k atunci vor exista 2^k caractere, i.e. dintr-un alfabet mic obţinem unul cu multe caractere. Totuşi, k va trebui ales suficient de mic pentru ca dimensiunea tabelei de salturi să nu fie prea mare.*

Algoritmul Boyer-Moore: sumar

- complexitate $O(n+m)$ dacă patternul p nu apare în subiect; altfel rămâne $O(n \cdot n)$
- totuși, cu o simplă modifcare (regula Galil, 1979) se poate obține $O(n+m)$ în toate cazurile
- algoritmul original al lui Boyer-Moore (1977) utilizează o variantă simplificată a regulei sufixului bun
- prima demonstrație pentru $O(n+m)$, când patternul p nu apare în subiect, a fost dată de Knuth, Morris și Pratt (1977); o altă demonstrație a fost dată în mod independent de Guibas și Odlyzko (1980)
- Richard Colen (1991) a stabilit o limită de $4n$ (cu o demonstrație mai ușoară), apoi limita de $3n$ (cu o demonstrație mai dificilă)