

Structuri de Date și Algoritmi

Căutări în siruri de caractere

Mihai Nan

Departamentul de Calculatoare
Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA din București



Anul Universitar 2022–2023

Conținutul cursului

- ① Aplicații
- ② Algoritmul căutării directe
- ③ Algoritmul Rabin-Karp
- ④ Compresie Lempel-Ziv-Welch

Aplicații

Există o gamă largă de aplicații pentru tabelele de dispersie

- **Compilatoare** – tabele de simboluri
- Determinarea rapidă a elementelor cu chei egale în colecții mari de date
- Determinarea rapidă a grupurilor de elemente cu chei similare
- Algoritmul de căutare Rabin-Karp a sirurilor de caractere în siruri lungi de caractere (string pattern match)
- Algoritmul de compresie a sirurilor de caractere LZW

Algoritmi de căutare în siruri

Ideea algoritmilor

Determină prima apariție sau toate aparițiile a unui sir de M caractere, notat cu p , numit şablon într-un sir de N caractere, notat cu a , numit text.

Observație

În practică, N este foarte mare și M poate fi mare.

- În continuare, vom analiza cum diferă căutarea în sir a unui şablon de căutarea unei chei

Algoritmi de căutare în siruri – Scurt istoric

① Algoritmul căutării directe

- Cazul cel mai nefavorabil – timp proporțional cu $N \cdot M$
- În multe cazuri practice, timpul mediu este proporțional cu $N + M$.

- ② **Anul 1970 – D.E. Knuth, J.H. Morris și V.R. Pratt** au construit un algoritm de căutare rapidă (Algoritmul KMP);
- ③ **Anul 1976 – R.S. Boyer și J.S. Moore** au descoperit un algoritm mai rapid decât KMP în multe aplicații, deoarece examinează în multe cazuri numai o parte din caracterele textului.
- ④ **Anul 1980 – R.M. Karp și M.O. Rabin** au observat că problema căutării în siruri nu este mult diferită de o problema standard de căutare și au descoperit un algoritm care are virtual timpul proporțional cu $M + N$.

Algoritmul căutării directe

```
int cautaSablonV1(char *p, char *a) {  
    int i = 0; // i este contorul pentru text  
    int j = 0; // j este contorul pentru sablon  
    int M = strlen(p); // p este sablon  
    int N = strlen(a); // a este text  
    do {  
        while ((i < N) && (j < M) && (a[i] == p[j])) {  
            i++;  
            j++;  
        }  
        if (j == M)  
            return i - M;  
        i -= j - 1;  
        j = 0;  
    } while (i < N);  
    return i;  
}
```

Algoritmul căutării directe

```
int cautaSablonV2(char *p, char *a) {  
    int i, j;  
    int M = strlen(p);  
    int N = strlen(a);  
    for (i = 0, j = 0; j < M && i < N; i++, j++) {  
        while (a[i] != p[j]) {  
            i -= j - 1;  
            j = 0;  
        }  
    }  
    if (j == M)  
        return i - M;  
    else  
        return i;  
}
```

Algoritmul căutării directe

Proprietate

Algoritmul căutării directe necesită, în cazul cel mai nefavorabil, un număr de maxim $N \cdot M$ comparații de caractere.

- Timpul de rulare pentru cazul cel mai nefavorabil este proporțional cu $M \cdot (N - M + 1)$, și cum $M \ll N$ obținem valoarea aproximativă $N \cdot M$.
- Cazul cel mai nefavorabil poate apărea, de exemplu, la prelucrarea imaginilor binare.
- În multe aplicații de prelucrare a textelor, bucla interioară se execută rareori și timpul de rulare este proporțional cu $N + M$.

Algoritmul Rabin-Karp

Ideea algoritmului

Consideră textul ca fiind o memorie mare și tratează fiecare secvență de M caractere a textului ca o cheie într-o tabela de dispersie (hash).

- Trebuie să calculăm funcția de dispersie pentru toate secvențele posibile de M caractere consecutive din text și să verificăm dacă valorile obținute sunt egale cu funcția de dispersie a şablonului.
- Artificiu pentru a nu ține toată tabela de dispersie în memorie.

Calculul funcției hash

- $h(k) = k \bmod q$, unde k este codificarea ca număr a caracterelor din sir, iar q reprezintă dimensiunea tabelei de dispersie și este un număr prim mare;
- q poate fi chiar foarte mare deoarece tabela hash nu se stochează în memorie
- Funcția de dispersie pentru poziția i din text se calculează pe baza valorii acestei funcții pentru poziția $i - 1$
- Se transformă grupuri de M caractere în numere, prin împachetare ca reprezentare într-o anumită bază
- Aceasta corespunde la a scrie caracterele ca numere în baza d , unde d este numărul de caractere posibile din alfabet.

Calculul funcției hash

- O secvență de M caractere consecutive $a[i], a[i + 1], \dots, a[i + M - 1]$ corespunde:

$$y = a[i] \cdot d^{M-1} + a[i + 1] \cdot d^{M-2} + \dots + a[i + M - 1]$$

- Presupunem că știm valoarea lui $h(y) = y \bmod q$.
- Deplasarea în text cu o poziție la dreapta corespunde înlocuirii lui y cu:

$$y_1 = (y - a[i] \cdot d^{M-1}) * d + a[i + M]$$

- Consideram $d = 32$ (sau orice multiplu de 2 corespunzător)
- Cum facem să nu memorăm valorile funcției hash pentru toate grupurile de M caractere din sir?

Calculul funcției hash

Proprietăți modulo

$$(a + b) \bmod q = (a \bmod q + b \bmod q) \bmod q$$

$$(a * b) \bmod q = (a * (b \bmod q)) \bmod q$$

Dorim să arătăm că:

$$(a_0 \cdot x + a_1) \bmod q = ((a_0 \bmod q) \cdot x + a_1) \bmod q$$

Demonstrație:

$$\begin{aligned} (a_0 \cdot x + a_1) \bmod q &= ((a_0 \cdot x) \bmod q + a_1 \bmod q) \bmod q = \\ &= [(x \cdot (a_0 \bmod q)) \bmod q + a_1 \bmod q] \bmod q = \\ &= ((a_0 \bmod q) \cdot x + a_1) \bmod q \end{aligned}$$

Reamintim că $y_1 = (y - a[i] \cdot d^{M-1}) * d + a[i+M]$

$$y_1 \bmod q = ((y - a[i] \cdot d^{M-1}) \cdot d + a[i+M]) \bmod q \Rightarrow$$

$$y_1 \bmod q = ((y - a[i] \cdot d^{M-1}) \bmod q \cdot d + a[i+M]) \bmod q$$

Algoritmul Rabin-Karp

```
int cautareRK(char *p, char *a, int q) {
    int i; long int dM = 1, h1 = 0, h2 = 0;
    int M = strlen(p), N = strlen(a);
    // calculează  $d^{M-1} \text{ mod } q$  și reține în dM
    for (i = 1; i < M; i++)
        dM = (d * dM) % q;
    for (i = 0; i < M; i++) {
        h1 = (h1 * d + index(p[i])) % q; // hash pt şablon
        h2 = (h2 * d + index(a[i])) % q; // hash pt text
    }
    for (i = 0; h1 != h2; i++) {
        // se adaugă  $d * q$  pt a menține expresia >0
        h2 = (h2 + d * q - index(a[i]) * dM) % q;
        h2 = (h2 * d + index(a[i+M])) % q;
        if (i > (N - M)) return N;
    }
    return i;
}
```

Caracteristici Rabin-Karp

- Algoritmul Rabin-Karp are destul de probabil o complexitate timp liniară.
- Algoritmul are timpul proporțional cu $N + M$, dar el găsește două siruri cu valori de dispersie egale.

Important

În aceste condiții, trebuie să comparăm şablonul cu cele M caractere din sir, pentru cazul coliziunilor.

- Din punct de vedere teoretic, acest algoritm ar putea să fie tot $O(N \cdot M)$, în cazul cel mai nefavorabil, în care toate şabloanele găsite ar fi coliziuni. Din fericire, acest lucru este puțin probabil, deoarece q este mare; în practică, algoritmul folosește $N + M$ pași.

Compresie Lempel-Ziv-Welch

- Este un algoritm de compresie a sirurilor de caractere foarte folosit.
- Metoda foloseste un dictionar prin care asociaza unor siruri de caractere de lungimi diferite coduri numerice intregi si inlocuieste secvente de caractere din fisierul initial prin aceste numere.
- Acest dictionar este cercetat la fiecare nou caracter extras din fisierul initial si este extins de fiecare data cand se gaseste o secventa de caractere care nu exista anterior in dictionar.
- Dimensiunea uzuala a dictionarului este 4096, dintre care primele 256 de pozitii contin toate caracterele individuale ce pot apare in fisierele de comprimat.

Compresie Lempel-Ziv-Welch

- Sirul inițial (de comprimat) este analizat și codificat într-o singură trecere, fără revenire.
- La stânga poziției curente sunt subșiruri deja codificate, iar la dreapta cursorului se caută cea mai lungă secvență care există deja în dicționar.
- Odată găsită această secvență, ea este înlocuită prin codul asociat deja și se adaugă la dicționar o secvență cu un caracter mai lungă.
- Dicționarul folosit este implementat ca o **tabelă hash**

Algoritmul de compresie LZW

- 1 Initializează dicționarul cu toate caractere din sir. Pentru fiecare caracter asociem un cod.
- 2 Inițializăm P cu \emptyset
- 3 **Cât timp** nu am ajuns la finalul sirului **repetă**
 - 4 C este caracterul curent din sir
 - 5 Dacă $P + C$ este în dicționar **atunci**
 - 6 $P = P + C$
 - 7 **Altfel**
 - 8 Determină codul lui P din dicționar și depune-l în sirul de ieșire
 - 9 Adaugă $P+C$ în dicționar
 - 10 $P = C$
- 11 Depune în sirul de ieșire codul asociat lui P
- 12 **Returnează** codificarea obținută în sirul de ieșire

Algoritmul de compresie LZW – Exemplu

- Pornim de la sirul wabbawabba

1 În primul pas, inițializăm dicționarul cu toate caracterele din sir.

Algoritmul de compresie LZW – Exemplu

- Pornim de la sirul wabbawabba

1 În primul pas, inițializăm dicționarul cu toate caracterele din sir.

1 2 3 4 5 6 7 8 9 10

a	b	w							
---	---	---	--	--	--	--	--	--	--

Algoritmul de compresie LZW – Exemplu

- Pornim de la sirul wabbawabba

1 În primul pas, initializăm dicționarul cu toate caracterele din sir.

1 2 3 4 5 6 7 8 9 10

a	b	w							
---	---	---	--	--	--	--	--	--	--

2 wabbawabba – $P = \emptyset$ și $C = w \Rightarrow P + C = w$

- $P + C = w$ care există deja în dicționar

Algoritmul de compresie LZW – Exemplu

- Pornim de la sirul wabbawabba

1 În primul pas, initializăm dicționarul cu toate caracterele din sir.

1 2 3 4 5 6 7 8 9 10

a	b	w							
---	---	---	--	--	--	--	--	--	--

2 wabbawabba – $P = \emptyset$ și $C = w \Rightarrow P + C = w$

- $P + C = w$ care există deja în dicționar

3 wabbawabba – $P = w$ și $C = a \Rightarrow P + C = wa$

- $P + C = wa$ care **NU** există în dicționar

• Determină codul lui P din dicționar și depune-l în sirul de ieșire \Rightarrow

Output: 3

1 2 3 4 5 6 7 8 9 10

a	b	w	wa						
---	---	---	----	--	--	--	--	--	--

Algoritmul de compresie LZW – Exemplu

1 2 3 4 5 6 7 8 9 10

a	b	w	wa						
---	---	---	----	--	--	--	--	--	--

4 wabbawabba – $P = a$ și $C = b \Rightarrow P + C = ab$

- $P + C = ab$ care **NU** există în dicționar
 - Determină codul lui P din dicționar și depune-l în sirul de ieșire ⇒
- Output:** 3 1

1 2 3 4 5 6 7 8 9 10

a	b	w	wa	ab					
---	---	---	----	----	--	--	--	--	--

Algoritmul de compresie LZW – Exemplu

1 2 3 4 5 6 7 8 9 10

a	b	w	wa	ab					
---	---	---	----	----	--	--	--	--	--

5 wabbawabba – $P = b$ și $C = b \Rightarrow P + C = bb$

- $P + C = bb$ care **NU** există în dicționar
 - Determină codul lui P din dicționar și depune-l în sirul de ieșire ⇒
- Output:** 3 1 2

1 2 3 4 5 6 7 8 9 10

a	b	w	wa	ab	bb				
---	---	---	----	----	----	--	--	--	--

Algoritmul de compresie LZW – Exemplu

1 2 3 4 5 6 7 8 9 10

a	b	w	wa	ab	bb				
---	---	---	----	----	----	--	--	--	--

6 wabb**a**wabba – $P = b$ și $C = a \Rightarrow P + C = ba$

- $P + C = ba$ care **NU** există în dicționar
 - Determină codul lui P din dicționar și depune-l în sirul de ieșire ⇒
- Output:** 3 1 2 2

1 2 3 4 5 6 7 8 9 10

a	b	w	wa	ab	bb	ba			
---	---	---	----	----	----	----	--	--	--

Algoritmul de compresie LZW – Exemplu

1 2 3 4 5 6 7 8 9 10

a	b	w	wa	ab	bb	ba			
---	---	---	----	----	----	----	--	--	--

7 wabba**w**abba – $P = a$ și $C = w \Rightarrow P + C = aw$

- $P + C = aw$ care **NU** există în dicționar
 - Determină codul lui P din dicționar și depune-l în sirul de ieșire \Rightarrow
- Output:** 3 1 2 2 1

1 2 3 4 5 6 7 8 9 10

a	b	w	wa	ab	bb	ba	aw		
---	---	---	----	----	----	----	----	--	--

Algoritmul de compresie LZW – Exemplu

1	2	3	4	5	6	7	8	9	10
a	b	w	wa	ab	bb	ba			

7 wabbaw**a**bba – $P = w$ și $C = a \Rightarrow P + C = wa$

- $P + C = aw$ care există în dicționar

8 wabbaw**a**bba – $P = wa$ și $C = b \Rightarrow P + C = wab$

- $P + C = wab$ care **NU** există în dicționar
 - Determină codul lui P din dicționar și depune-l în sirul de ieșire \Rightarrow
- Output:** 3 1 2 2 1 4

1	2	3	4	5	6	7	8	9	10
a	b	w	wa	ab	bb	ba	aw	wab	

Algoritmul de compresie LZW – Exemplu

1 2 3 4 5 6 7 8 9 10

a	b	w	wa	ab	bb	ba	aw	wab	
---	---	---	----	----	----	----	----	-----	--

9 wabbawab**ba** – $P = b$ și $C = b \Rightarrow P + C = bb$

- $P + C = bb$ care există în dicționar

8 wabbawab**ba** – $P = bb$ și $C = a \Rightarrow P + C = bba$

- $P + C = bba$ care **NU** există în dicționar

• Determină codul lui P din dicționar și depune-l în sirul de ieșire \Rightarrow

Output: 3 1 2 2 1 4 **6**

- $P = C \Rightarrow P = a$

1 2 3 4 5 6 7 8 9 10

a	b	w	wa	ab	bb	ba	aw	wab	bba
---	---	---	----	----	----	----	----	-----	-----

Algoritmul de compresie LZW – Exemplu

1	2	3	4	5	6	7	8	9	10
a	b	w	wa	ab	bb	ba	aw	wab	bba

10 Am terminat de parcurs șirul

- Depune în șirul de ieșire codul asociat lui $P \Rightarrow$ **Output:** 3 1 2 2 1 4 6 1

Compresia finală pentru șirul wabbawabba este **3 1 2 2 1 4 6 1**

Algoritmul de decompresie LZW

- 1 Inițializează dicționarul cu toate caracterele din sir. Pentru fiecare caracter asociem un cod.
- 2 cw o să fie primul cod din sirul de intrare
- 3 Scriem la ieșire $\text{sir}(cw)$
- 4 $pw = cw$
- 5 **Cât timp** nu am parcurs tot sirul **repetă**
 - 6 $pw = cw$ și cw va lua valoarea următorului cod
 - 7 Dacă cw este în dicționar **atunci**
 - 8 Scrie la ieșire $\text{sir}(cw)$
 - 9 $P = \text{sir}(pw)$
 - 10 C este primul caracter din $\text{sir}(cw)$
 - 11 Adaugă $P + C$ în dicționar.
- 12 **Altfel**
 - 13 $P = \text{sir}(pw)$
 - 14 C este primul caracter din $\text{sir}(pw)$
 - 15 Scrie la ieșire $P + C$ și adaugă-l în dicționar

Algoritmul de decompresie LZW – Exemplu

1 În primul pas, inițializăm dicționarul cu toate caracterele din sir.

1	2	3	4	5	6	7	8	9	10
a	b	w							

Intrare: 3 1 2 2 1 4 1 6 1

2 Intrare: 3 1 2 2 1 4 1 6 1 și $cw = 3$

3 Scriem la ieșire $\text{șir}(cw) \Rightarrow$ Output: w

4 Inițializăm pw cu $cw \Rightarrow pw = 3$

5 Intrare: 3 1 2 2 1 4 1 6 1 și $pw = 3, cw = 1$

6 cw există în dicționar

- Scrie la ieșire $\text{șir}(cw) \Rightarrow$ Output: w a

- P va lua valoarea $\text{șir}(pw) \Rightarrow P = \text{șir}(3) = w$

- Adaugă $P + C$ în dicționar \Rightarrow adaugă wa în dicționar

Algoritmul de decompresie LZW – Exemplu

1 2 3 4 5 6 7 8 9 10

a	b	w	wa						
---	---	---	----	--	--	--	--	--	--

7 **Intrare:** 3 1 2 2 1 4 1 6 1 și $pw = 1, cw = 2$

8 cw există în dicționar

- Scrie la ieșire $\text{șir}(cw) \Rightarrow \text{Output: } w \ a \ b$
- P va lua valoarea $\text{șir}(pw) \Rightarrow P = \text{șir}(1) = a$
- Adaugă $P + C$ în dicționar \Rightarrow adaugă ab în dicționar

1 2 3 4 5 6 7 8 9 10

a	b	w	wa	ab					
---	---	---	----	----	--	--	--	--	--

Și tot aşa...

Algoritmul Knuth-Morris-Pratt

- Descoperit independent de:
 - Donald Ervin Knuth
 - profesor emerit la Stanford University
 - PhD CalTech
 - Autor al The Art of Computer Programming
 - James Hiram Morris
 - profesor la Carnegie Mellon University
 - Absolvent CMU si MIT
 - Vaughan Pratt
 - profesor emerit la Stanford University
 - Absolvent Sydney University



Algoritmul Knuth-Morris-Pratt

Metoda

Vom folosi reprezentări binare pentru a explica metoda.

Startul fals al identificării constă din caractere deja cunoscute.

- a — sir, i — index în sir, a de lungime N
- p — şablon, j – index în şablon, p de lungime M
- **Considerăm întâi un caz particular:** primul caracter din şablon nu mai apare în şablon

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

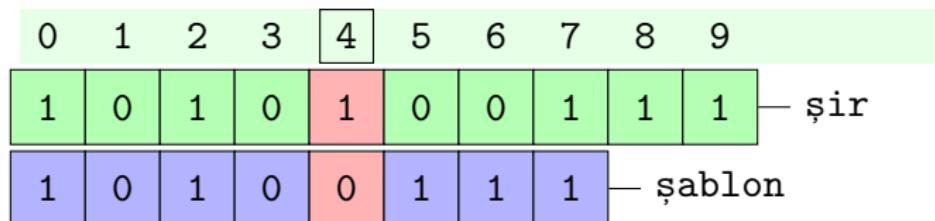
— sir

— şablon

În loc de $i = i - j + 1$ se consideră $i++$ și $j = 0$

Algoritmul Knuth-Morris-Pratt

Cazul General



- $j = 4$ și i trebuie decrementat. **Cu cât îl vom decrementa pe i ?** Se poate cunoaște această valoare — depinde de caracterele din şablon
- Dacă nepotrivirea între sir și şablon apare pe poziția j în şablon și există un $e \in [0, j - 1]$ pentru care $p[0] \dots p[e - 1] = p[j - e] \dots p[j - 1]$ atunci o posibilă identificare între sir și şablon poate apărea începând cu caracterul din sir $a[i - e]$ și caracterul din şablon $p[j - e]$.
- Pentru exemplu, $j = 4$, $e = 2$ ($p[0] \dots p[2-1] = p[4-2] \dots p[4-1]$, adică $p[0] = p[1] = p[2] = p[3]$)

Algoritmul Knuth-Morris-Pratt

Cazul General

Observație

Trebuie căutat e maxim cu această proprietate.

- Dacă nu există e cu această proprietate atunci căutarea se reia de la poziția i și $j = 0$.
- Dacă există un astfel de e atunci $i = i - e, j = 0$.
- Se calculează valorile unui vector $next[j]$ pentru $\forall p_j \in p$ (cu M elemente).
- $next[j]$ – cu cât trebuie decrementat i la apariția unei nepotriviri pe poziția j
- $next[j] = e_{max}$ — pentru $j > 0$, e_{max} este numărul maxim de caractere $e < j$ de la începutul şablonului care sunt identice cu ultimele e caractere din primele j caractere din şablon

Algoritmul Knuth-Morris-Pratt – Vectorul next

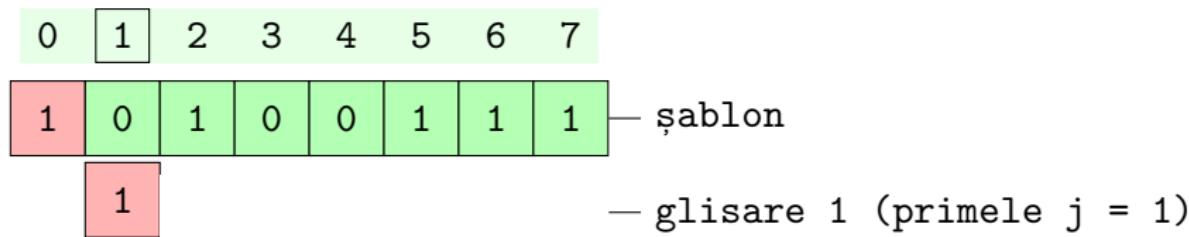
0	1	2	3	4	5	6	7	8	9
1	0	1	0	1	0	0	1	1	1
1	0	1	0	0	1	1	1	1	

- $next[4] = 2 = e_{max} \Rightarrow$ Reluare căutare $i = i - next[j], j = 0$.

Cum determinăm $next[j]$?

- Se glisează o copie a primelor j caractere din şablon peste şablon de la stânga la dreapta, începând cu primul caracter din copie suprapus cu al doilea caracter din şablon.
- Se opreşte glisarea dacă:
 - toate caracterele suprapuse (pană la $j-1$) coincid
 - primul caracter al copiei ajunge pe poziția j .

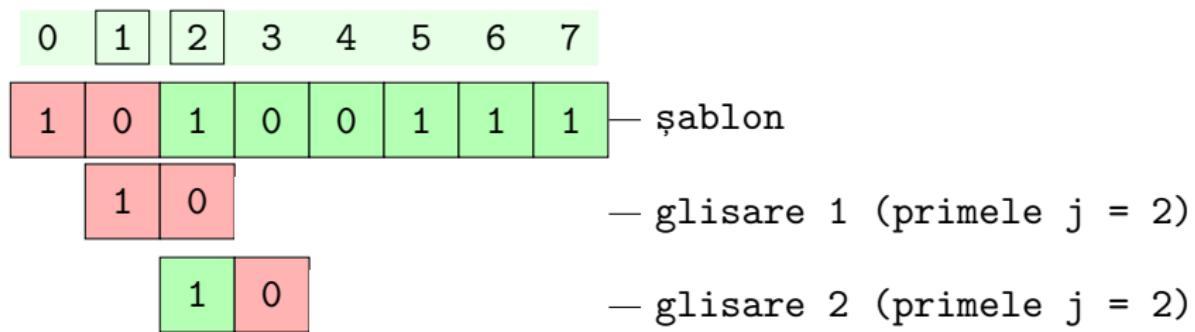
Algoritmul Knuth-Morris-Pratt – Vectorul next ($j = 1$)



$next[j] = e_{max}$ – pentru $j > 0$, e_{max} este numărul maxim de caractere $e < j$ de la începutul şablonului care sunt identice cu ultimele e caractere din primele j caractere din şablon

0 potriviri

Algoritmul Knuth-Morris-Pratt – Vectorul next (j = 2)

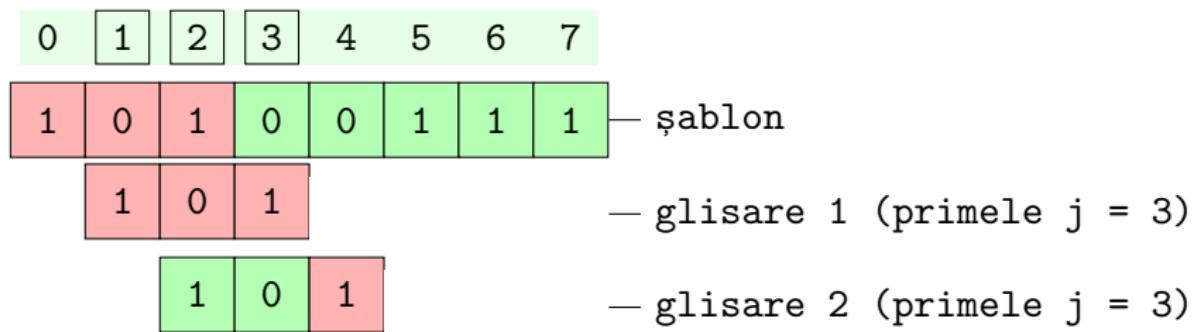


$\text{next}[j] = e_{\max}$ – pentru $j > 0$, e_{\max} este numărul maxim de caractere $e < j$ de la începutul şablonului care sunt identice cu ultimele e caractere din primele j caractere din şablon

Se oprește glisarea dacă: **primul caracter al copiei ajunge pe poziția j (cazul II).**

1 potrivire

Algoritmul Knuth-Morris-Pratt – Vectorul next (j = 3)



$\text{next}[j] = e_{\max}$ – pentru $j > 0$, e_{\max} este numărul maxim de caractere $e < j$ de la începutul şablonului care sunt identice cu ultimele e caractere din primele j caractere din şablon

Se oprește glisarea dacă **toate caracterele suprapuse (până la $j-1$) coincid** (cazul I).

2 potriviri

Algoritmul Knuth-Morris-Pratt – Vectorul next

j	next[j]	sablon si copie sablon	
1	0	<u>1</u> 0100111	0 potriviri
2	0	<u>10</u> 100111	0 potriviri
3	1	<u>101</u> 00111	
4	2	<u>1010</u> 0111	1 potrivire
5	0	<u>10100</u> 111	2 potriviri
6	1	<u>101001</u> 11	
7	1	<u>1010011</u> 1	1 potrivire
		10100111	1 potrivire

Algoritmul Knuth-Morris-Pratt – Vectorul next

- Caracterele identice definesc următorul loc posibil în care şablonul poate identifica cu sirul, după ce se găseşte o nepotrivire pe poziţia j (caracterul $p[j]$).
- Distanţa cu care trebuie să ne întoarcem în sir este exact $\text{next}[j]$, adică numărul de caracter care coincid.
- Este convenabil $\text{next}[0] = -1$.

Cum facem să nu ne întoarcem în sir?

- $a[i] \neq p[j] \Rightarrow$ se reia de la $i - \text{next}[j]$ şi $j = 0$
- Dar primele $\text{next}[j]$ caractere începând de la poziţia $i - \text{next}[j]$ din sir = primele $\text{next}[j]$ caractere din şablon \Rightarrow **i nemodificat şi $j = \text{next}[j]$**

Algoritmul Knuth-Morris-Pratt – Vectorul next

```
void initNext(char *p) {  
    int i, j, M = strlen(p);  
    next[0] = -1;  
    for (i = 0, j = -1; i < M; i++, j++, next[i] = j)  
        while ((j >= 0) && (p[i] != p[j]))  
            j = next[j];  
}
```

Observație

După incrementările $i++$ și $j++$ avem $p[i-j] \dots p[i-1] = p[0] \dots p[j-1]$
Identificarea efectuează maxim $M+N$ comparații de caractere

Algoritmul Knuth-Morris-Pratt – Exemple

Exemplul 1: $i = 4, j = 4, \text{next}[4] = 2$

0	1	2	3	4	5	6	7	8	9
1	0	1	0	1	0	0	1	1	1
1	0	1	0	0	1	1	1		

– sir
– sablon

Cazul I: $a[i] \neq p[j] \Rightarrow$ se reia de la $i - \text{next}[j] = 4 - 2 = 2$ și $j = 0$

0	1	2	3	4	5	6	7	8	9
1	0	1	0	1	0	0	1	1	1
1	0	1	0	0	0	1	1	1	

– sir
– sablon

Algoritmul Knuth-Morris-Pratt – Exemple

Exemplul 2: $i = 4, j = 4, \text{next}[4] = 2$

0	1	2	3	4	5	6	7	8	9
1	0	1	0	1	0	0	1	1	1
1	0	1	0	0	1	1	1	1	1

Cazul II: Primele $\text{next}[j]$ caractere începând de la poziția $i - \text{next}[j]$ din sir sunt egale cu primele $\text{next}[j]$ caractere din şablon \Rightarrow **i nemodificat și $j = \text{next}[j]$**

0	1	2	3	4	5	6	7	8	9
1	0	1	0	1	0	0	1	1	1
1	0	1	0	0	0	1	1	1	1

Se observă același efect pentru acest caz!

Algoritmul Knuth-Morris-Pratt

```
int cautareKMP(char *p, char *a) {  
    int i, j;  
    int M = strlen(p);  
    int N = strlen(a);  
    initNext(p);  
    for (i = 0, j = 0; j < M && i < N; i++, j++)  
        while ((j >= 0) && (a[i] != p[j]))  
            j = next[j];  
    if (j == M)  
        return i - M; // am găsit potrivire  
    else  
        return i;  
}
```

Pt. $j = 0$ si $a[i] \neq p[0]$ este necesar ca $i + j = 0$ deci $\text{next}[0] = -1$

Caracteristici KMP

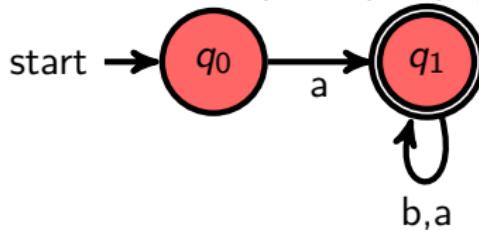
- Algoritmul KMP presupune precompilarea şablonului, ceea ce este justificat dacă $M \ll N$, adică textul este considerabil mai lung decât şablonul;
- Algoritmul KMP nu decrementează indexul din sir, proprietate folositoare mai ales pentru căutările pe suport extern;
- Algoritmul KMP dă rezultate bune dacă o nepotrivire a apărut după o identificare parțială de o anumită lungime. În cazul în care aceste situații sunt o excepție, algoritmul nu aduce îmbunătățiri semnificative;
- Algoritmul KMP îmbunătăște timpul cazului cel mai nefavorabil ($N + M$), dar nu și timpul mediu.

Automat finit determinist

- Se numește automat finit un tuplu $AF = (Q, \Sigma, m, q_0, F)$ unde:
 - ① Q – mulțimea finită a stărilor
 - ② Σ – alfabetul de intrare
 - ③ $m : Q \times (\Sigma \cup \{e\}) \rightarrow \mathcal{P}(Q)$ – funcția parțială a stării următoare (funcția de tranziție).
 $\mathcal{P}(Q)$ – mulțimea părților mulțimii Q (toate submulțimile lui Q)
 - ④ $q_0 \in Q$ – starea de start
 - ⑤ $F \subseteq Q$ – mulțimea stărilor finale

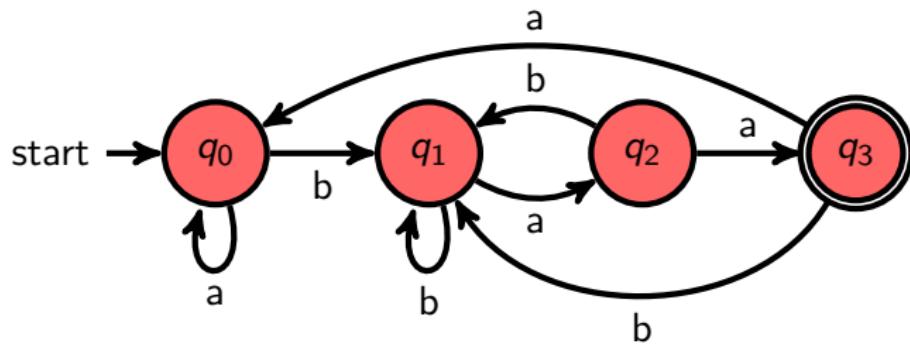
Un automat finit este un graf orientat!

Exemplu: $L = \{w \in \{a, b\}^* | w \text{ începe cu } a\}$



Automat finit determinist

Exemplu: $L = \{w \in \{a, b\}^* | w \text{ se termină cu } baa\}$



Pentru a verifica dacă un sir face sau nu parte din limbaj (*îndeplinește un pattern*), parcurgem graful pornind de la nodul de start și vedem dacă ajungem într-un nod final.

Întrebări?!

