Analiza Algoritmilor
Tema - Etapa 3

Grigore Lucian-Florin

Grupa 324CD
Facultatea de Automatica si Calculatoare
Universitatea Politehnica, Bucuresti

Abstract. Acest document reprezinta forma finala a studiului realizat asupra algoritmilor de pattern-matching Knuth-Morris-Pratt si Rabin-Karp.

Keywords: Rabin-Karp · Knuth-Morris-Pratt · Pattern-Matching.

1 Introducere

1.1 Descrierea problemei rezolvate

Problema abordata presupune gasirea celei mai lungi secvente comune pentru doua siruri de caractere - *pattern-matching*.

1.2 Exemple de aplicatii practice pentru problema aleasa

Conceptul de pattern-matching are multiple aplicatii in domeniul calculatoarelor, unele dintre cele mai interesante fiind:

1. Procesarea limbajului natural
2. Motoarele de cautare web
3. Detectarea si combaterea intruziunilor cibernetice
4. Citirea si interpretarea biosecventelor din ADN
5. Gestionarea pachetelor de internet de catre routere

1.3 Specificarea solutiilor alese

Solutiile alese pentru rezolvarea acestei probleme sunt algoritmii Rabin-Karp (Michael Rabin si Richard Karp) si Knuth-Morris-Pratt (James H. Morris, Donald Knuth si Vaughan Pratt).

1.4 Specificarea criteriilor de evaluare alese pentru validarea solutiilor

- Pentru **Rabin-Karp**, dorim sa gasim probabilitatea unei false potriviri in functie de dimensiunea spatiului cheilor, precum si folosirea mai multor tipuri de functii hash in implementare (eficiente si mai putin eficiente). In schimb, se pot elimina foarte usor secventele care nu se potrivesc cu ajutorul unei functii hash potrivite. Pe de alta parte, se poate slabi performanta algoritmului foarte mult daca este folosite o functie de hashing slaba. Ne intereseaza gasirea unei functii de hashing cat mai *fail-proof* pentru o performanta mai buna.
- Cazurile cand algoritmul **Knuth-Morris-Pratt** functioneaza mai bine decat Rabin-Karp, si invers.
- **Complexitatile** de timp si spatiu ale fiecarui algoritm pentru un set de date de dimensiune cunoscuta *(cea mai lunga secventa fiind de o lungime M, cunoscuta).*
- Dorim sa intocmim un **set de teste** care sa se asigure ca solutiile propuse functioneaza cat mai eficient din punctul de vedere al memoriei si timpului de executare. Testele propuse ar trebui sa acopere situatii precum: potrivirea multipla intre secvente de aceeasi lungime (preferabil lungime mare, pentru a vedea consumul resurselor), secvente care sa para la fel pentru functii hash slabe (de exemplu "aab" si "aba" ar avea aceeasi valoare hash pentru o functie care nu se foloseste de indexul literelor) sau secvente cu grupuri de litere repetitive care sa puna la incercare tabela caracteristica algoritmului KMP.

2 Prezentarea solutiilor

2.1 Descrierea modului in care functioneaza algoritmii alesi

Algoritmul **Rabin-Karp** este la baza *bruteforce* la care se adauga o componenta foarte puternica in programare, si anume *hashingul*. Nu este folosit asa mult la scara larga deoarece are o complexitate de timp slaba. Pe de alta parte, el prezinta avantajul ca isi mentine complexitatea indiferent de numarul de match-uri din text.

Folosirea hashingului are rolul de a creste eficienta acestei solutii prin exploatarea aspectului ca doua siruri egale au aceeasi valoare de hash. Un dezavantaj al acestei abordari este acela ca se pot potrivi ca valoare hash doua siruri care nu sunt identice. Din aceasta cauza acest algoritm presupune si verficarea (caracter cu caracter) a celor doua siruri care au aceeasi valoare hash pentru a asigura potrivirea lor.

Pasii de realizare ale acestei solutii sunt:

- calcularea valorii hash a pattern-ului

- calcularea valorii hash a primelor m caractere din text (unde m este lungimea pattern-ului)
- compararea celor doua valori hash. Daca sunt egale, ori se termina executia functiei, ori se adauga pozitia pe care a fost gasit pattern-ul in text intr-o lista de indici
- in cazul unei nepotriviri, se recalculeaza noua valoare hash a urmatoarelor m litere din text, dupa eliminarea primeia din secventa precedenta
- repetarea pasilor de mai sus pana la finalul textului, mai exact pana la momentul cand pana la finalul textului sunt mai putine caractere decat cate sunt in pattern

Pseudocodul de mai jos descrie succint implementarea acestei solutii:

Pseudo Code…

```
Length of pattern = M;
Hash(p) = hash value of pattern;
Hash(t) = hash value of first M letters in body of text;
do
  if (hash(p) == hash(t))
      brute force comparison of pattern and selected section of
  text
      hash(t) = hash value of next section of text, one character
  over
while (end of text or brute force comparison == true)
```

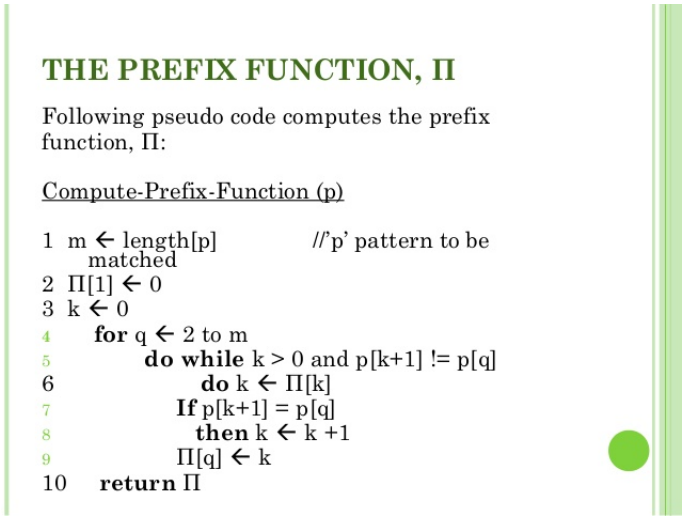
Pentru implementarea acestei solutii am folosit functia Bernstein, care este una consacrata prin performantele si siguranta care i-au fost demonstrate de-a lungul timpului.

Algoritmul **Knuth-Morris-Pratt** cauta potrivirea unui cuvant intr-un text pe baza ideii ca atunci cand se realizeaza o potrivire partiala, cuvantul in sine contine destula informatie astfel incat sa se poata sari peste un anumit numar de caractere despre care se stie ca sunt la fel si in pattern, si in text. Mai precis, functia/tabela specifica acestui algoritm *("failure-function"/"partial match table")* raspunde la intrebarea "care sufix din potrivirea partiala realizata pana intr-un moment de timp poate constitui un prefix pentru o eventuala potrivirire ulterioara?".

Primul pas in implementarea acestei solutii o reprezinta preprocesarea tabelei caracteristice, care presupune:

- verificarea oricarei secvente din pattern care este identica cu inceputul acestuia, astfel fiind un posibil prefix in cautare
- punerea in tabela a valorilor care reprezinta cate caractere pot fi sarite in cazul unei potriviri partiale, in functie de pozitia din cuvant la care s-a ajuns

Urmatorul pseudocod ilustreaza aceasta preprocesare:



THE PREFIX FUNCTION, Π

Following pseudo code computes the prefix function, Π:

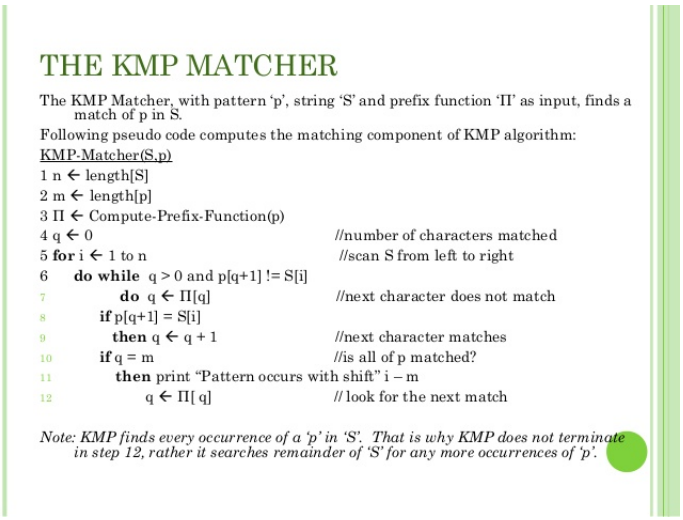
Compute-Prefix-Function (p)

```
1  m ← length[p]            //'p' pattern to be
     matched
2  Π[1] ← 0
3  k ← 0
4     for q ← 2 to m
5           do while k > 0 and p[k+1] != p[q]
6                 do k ← Π[k]
7              If p[k+1] = p[q]
8                then k ← k +1
9              Π[q] ← k
10    return Π
```

Partea de cautare este constituita din urmatorii pasi:

- crearea tabelei caracteristice pentru pattern-ul dat
- parcurgerea efectiva a textului si verificarea caracter cu caracter cu doi indici diferiti a cate unui caracter din text cu unul din pattern
- in cazul unui caracter care se potriveste, se parcurg ambele in continuare. Se verifica ca nu s-a ajung la finalul pattern-ului, caz in care a fost gasita o potrivire in text. Aceasta ori se poate adauga intr-o lista sau se poate intoarce indexul unde a fost gasita potrivirea, daca asta se doreste.
- in cazul unui caracter care nu se potriveste, se revine cu indicele care parcurge pattern-ul pe ultima pozitie indicata de tabela folosita, si se revine la potrivirea caracter cu caracter din acest punct
- repetarea pasilor de mai sus pana la ultima pozitie de la care poate fi gasit pattern-ul in text

Urmatorul pseudocod implementeaza aceasta cautare:



THE KMP MATCHER

The KMP Matcher, with pattern 'p', string 'S' and prefix function 'Π' as input, finds a match of p in S.

Following pseudo code computes the matching component of KMP algorithm:

```
KMP-Matcher(S,p)
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0                                  //number of characters matched
5 for i ← 1 to n                         //scan S from left to right
6     do while  q > 0 and p[q+1] != S[i]
7           do  q ← Π[q]                 //next character does not match
8        if p[q+1] = S[i]
9          then q ← q + 1                //next character matches
10       if q = m                       //is all of p matched?
11         then print "Pattern occurs with shift" i – m
12              q ← Π[ q]                // look for the next match
```

Note: KMP finds every occurrence of a 'p' in 'S'. That is why KMP does not terminate in step 12, rather it searches remainder of 'S' for any more occurrences of 'p'.

2.2 Analiza complexitatii solutiilor

Consideram lungimea textului = n si lungimea patternului = m

Rabin-Karp

timp

- worst-case: O(nm)
- best-case: O(n+m) (la fel si pentru average-case)

memorie - neglijabil

Knuth-Morris-Pratt

timp

- cautare: O(n)
- pre-procesare: O(m)
- overall: O(n+m)

memorie

- O(m)

2.3 Principalele avantaje si dezavantaje pentru fiecare solutie

Rabin-Karp

avantaje

- poate detecta foarte usor plagiarismul pentru ca rezolva foarte bine potrivirile multiple

- nu este mai rapid ca bruteforce in teorie, dar in practica are complexitate de timp mai buna
- cu o functie de hashing buna, poate fi destul de eficient si usor de implementat

dezavantaje

- sunt multiple solutii pentru aceasta problema care sunt mai rapide decat O(m+n) (unde m este lungimea patternului si n este lungimea textului in care se cauta)
- este la baza la fel de incet ca bruteforce si necesita si spatiu in plus

Knuth-Morris-Pratt
avantaje

- are timpul de rulare foarte bun comparativ cu alti algoritmi asemanatori
- nu are nevoie sa se miste inapoi in textul dat, astfel fiind foarte eficient in procesarea fisierelor de dimensiuni mari

dezavantaje

- nu functioneaza asa de bine odata cu cresterea dimensiunii alfabetului folosit deoarece creste posibilitatea unei potriviri gresite

3 Evaluare

3.1 Construirea setului de teste

Am considerat ca un numar prea mic de teste nu este relevant pentru a vedea comportamentul celor doi algoritmi in totalitate. Cele 30 de teste create se impart in mai multe categorii:

- 1 - 5: text de dimensiune scurta cu o singura potrivire
- 6 - 10: text de dimensiune scurta cu nicio potrivire
- 11 - 15: text de dimensiune medie cu multiple potriviri
- 16 - 20: text de dimensiune medie cu nicio potrivire
- 21 - 25: text de dimensiune mare cu multiple potriviri
- 26 - 30: text de dimensiune foarte mare cu multiple potriviri sau niciuna

In crearea testelor am incercat sa fiu cat mai variat pentru a intelege cat mai bine comportamentul algoritmilor in speta, asa ca am inclus de la carti intregi ("Hobbitul" - copiat de 2 ori pentru ca parea cam scurt, multiple carti ale lui Jules Verne in acelasi test, "Povestea lui Harap-Alb" si altele) pana la primul milion de zecimale ale lui pi, sau un sir de zeci de milioane de numere generate random.

3.2 Specificatiile sistemului de calcul

CPU: 2,6 GHz 6-Core Intel Core I7
RAM: 16GB 2400MHz DDR4
Disk: 500GB SSD
GPU: 1,5GB Intel UHD Graphics 630 / 4GB Radeon Pro 560x
OS: MacOS Catalina 10.15.1

3.3 Rezultatele evaluarii solutiilor pe setul de date

In urma rularii celor doi algoritmi pe setul de date construit, in majoritatea cazurilor timpii obtinuti au fost foarte asemanatori, variatiile fiind constante pentru toate testele. Am realizat o medie a timpilor de rulare (cu ajutorul unui script) pentru 10 rulari diferite, valorile aproximate fiind urmatoarele:

KMP VS RB

Dimensiunea testului	RB Ineficient	Rabin-Karp	Knuth-Morris-Pratt
13	**8529**	61766	23422
45	**10517**	50695	31215
110	**23226**	64728	35770
127	**22219**	61516	31365
138	**30517**	57072	73362
149	29451	67074	**28489**
200	43128	83077	**27850**
240	57256	98413	**29458**
252	47200	86461	**28275**
256	53194	107005	**36939**
1099	212371	194037	**65692**
3011	360451	367437	**101903**
3438	304405	353427	**88559**
6990	618929	583461	**151552**
6990	648156	663854	**176294**
9148	643108	597255	**172756**
9299	700300	667230	**162540**
10000	684430	530464	**218475**
10000	663715	526861	**199189**
10000	966109	902077	**218541**
92071	2623669	3501909	**1705369**
96707	4129229	3786922	**1540473**
459035	19564731	7530027	**4238036**
506733	19216692	8480239	**4093131**
967911	13700712	12145181	**6851537**
992065	12870637	13309198	**6137776**
1000000	14817971	11567857	**8758578**
2551732	29258497	20270356	**15442148**
15430688	135129235	109222508	**64102556**
91160918	34449250	26631641	**20184047**

Observatii:

Dimensiunea testului reprezinta numarul de caractere (bytes) pe care il are textul in care se realizeaza cautarea pattern-ului.

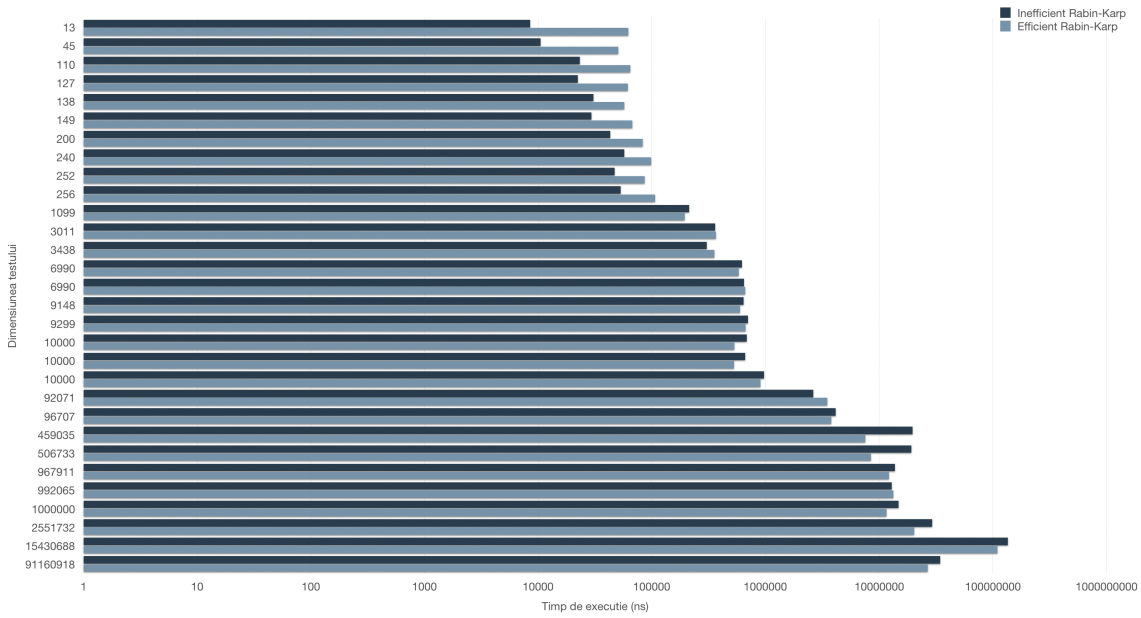
Urmatoarele trei coloane reprezinta timpii de rulare (in nanosecunde) pentru trei puncte de interes ale acestui studiu:

Coloana **RB Ineficient** corespunde rularii algoritmului Rabin-Karp folosind o functie ineficienta din punctul de vedere al performantei.

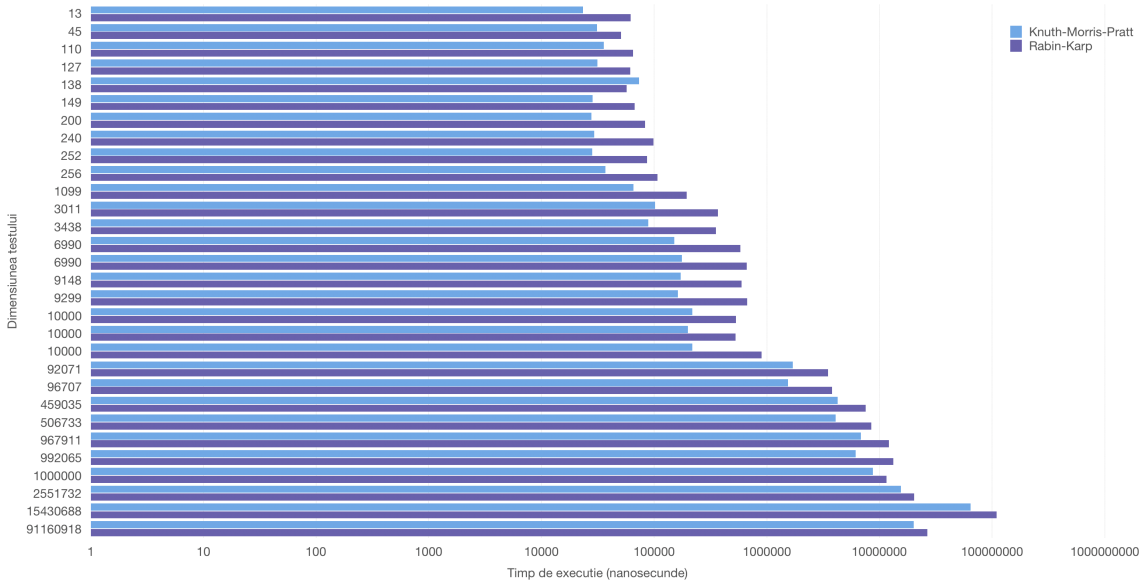
Coloana **Rabin-Karp** corespunde rularii algorimului Rabin-Karp folosind o functie consacrata si eficienta (functia Bernstein) din punctul de vedere al performantei.

Coloana **Knuth-Morris-Pratt** corespunde rularii algoritmului Knuth-Morris-Pratt.

Rularea algoritmului Rabin-Karp cu
o functie ineficienta vs. o functie eficienta de hashing



Rularea algoritmului
Rabin-Karp cu functia Bernstein vs. algoritmul Knuth-Morris-Pratt



Observatie: Timpul de rulare este masurat strict pentru partea caracteristica algoritmului. Citirea datelor si scrierea rezultatelor nu intra in acest timp.

Observatie: Timpul de rulare se doreste a fi cat mai mic. Un timp mai mic de rulare inseamna eficienta sporita.

Observatie: Scara folosita pentru ilustrarea timpilor de rulare in cele doua grafice de mai sus este una logaritmica. Am considerat acest lucru deoarece folosind una liniara nu se putea vedea atat de clar diferenta pe acelasi grafic a testelor de dimensiuni mari si a celor de dimensiuni mici din cauza diferentei timpilor obtinuti. De asemenea, un aspect important de tinut minte in continuare este acela ca orice mica diferenta intre doua valori care este pe grafic este in realitate mult mai decat pare.

3.4 Prezentarea valorilor obtinute pe teste

Rularea algorimului Rabin-Karp cu doua functii diferite de hashing duce la un numar de valori surprinzatoare deoarece in anumite cazuri, rularea algoritmului Rabin-Karp cu functia ineficienta se aseamana foarte mult cu cea eficienta, chiar fiind mai rapida in anumite teste. Acest fapt poate avea mai multe cauze:

– Cauza principala este faptul ca pattern-ul este de o lungime mica. Faptul ca folosind functia ineficienta se parcurg efectiv toate grupurile de lungimea pattern-ului prezente in text nu face nicio diferenta, sau aproape niciuna.

– Functia Bernstein are si o inmultire pe langa adunarea pe care o are si cealalta functie.

De asemenea, acest aspect apare din ce in ce mai putin cu cat dimensiunea textului este mai mare. Asta deoarece creste diferenta dintre a parcurge pattern-ul doar cand valoarea hash Bernstein este aceiasi pentru doua siruri de caractere si atunci cand se parcurge pattern-ul efectiv la fiecare pas din parcurgerea textului.

Nu exista o descriere clara a acestui comportament, dar reprezinta o caracteristica secundara interesanta pentru acest algoritm.

Comparand direct cei doi algoritmi vizati de acest studiu se poate observa ca in majoritatea cazurilor Knuth-Morris-Pratt este mai rapid. Exceptie face un singur test, dar care este de dimensiune prea mica pentru a putea fi luat ca exemplu clar ca Rabin-Karp are cazuri clare cand este mai eficient.

Acest rezultat se datoreaza faptului ca Rabin-Karp trebuie sa parcurga pattern-ul de fiecare data cand crede ca acesta a fost gasit in textul dat. In cazul celeilalte solutii, potrivirea este realizata fara alte apeluri de functii sau parcurgeri suplimentare.

Este adevarat ca algoritmul Knuth-Morris-Pratt are si o parte de preprocesare (producerea tabelei specifice), dar in cronometrarea testelor am inclus aceasta parte in timpul total al algoritmului. Astfel, chiar si cu aceasta aditie fata de cealalta solutie, timpii de rulare sunt clar mai buni.

4 Concluzii

Algoritmul Knuth-Morris-Pratt este mai eficient in majoritatea cazurilor, performanta lui fiind, de asemenea, direct proportionala cu dimensiunea problemei abordate.

4.1 Abordarea problemei in practica

Algoritmul **Rabin-Karp** ar trebui ales in cazurile in care cuvantul sau secventa care trebuie cautata se preconizeaza sa fie gasita de mai multe ori in text (cautarea/verificarea plagiatului, de exemplu).

Algoritmul **Knuth-Morris-Pratt** ar trebui ales in cazurile in care problema presupune cautarea intr-un text de dimensiuni foarte mari sau atunci cand pattern-ul in sine are foarte multe secvente repetitive identice cu una de inceput.

Referinte

1. https://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-notes/CS369-StringAlgs.pdf

2. https://www.slideshare.net/sabiyasabiya/knuth-morris-pratt-string-matching-algo-85219763

3. https://www.javatpoint.com/computer-network-routing

4. https://academic.oup.com/bioinformatics/article-abstract/9/3/299/225240

5. https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=2115&context=cstech

6. https://www.researchgate.net/publication/2432009_Pattern_Matching_Techniques_and_Their_Applications_to_Comp_A_Review

7. https://dzone.com/articles/algorithm-week-rabin-karp

8. https://www.slideshare.net/GajanandSharma1/rabin-karp-string-matching-algorithm

9. https://www.slideshare.net/alokeparnachoudhury/string-matching-algorithm

10. https://brilliant.org/wiki/rabin-karp-algorithm/

11. https://brilliant.org/wiki/knuth-morris-pratt-algorithm/

Toate site-urile care au fost referentiate mai sus au fost ultima data accesate in data de 13 decembrie 2019.