ГИМНАЗИЈА У КРАЉЕВУ

МАТУРСКИ РАД ИЗ ПРОГРАМИРАЊА И ПРОГРАМСКОГ ЈЕЗИКА

ТРАЖЕЊЕ УЗОРКА У ТЕКСТУ

Професор:
Јован Павловић

Ученик:
Славковић Исидора
IV$_7$

Мај 2018. године, Краљево

Садржај

-1-

Увод

Једна од основних операција над стринговима је тражење узорка. Нека је дат стринг *text* дужине *N* и узорак *pattern* дужине *M*. Проблем је установити да ли постоји подстринг у тексту који је једнак узорку. Већина алгоритама везана за овај проблем се могу проширити тако да пронађу сва појављивања узорка или да одреде број појављивања узорка у тексту. Типична ситуација у којој се наилази на овај проблем је при тражењу неке речи у текстуалном фајлу, а данас је јако користан при претраживању информација на интернету. Проблем има примену и у другим областима, на пример у молекуларној биологији, где је корисно пронаћи неке узорке у оквиру великих молекула РНК и ДНК.

ИСТОРИЈА

Алгоритми које ћемо да посматрамо имају интересантну историју. Постоји тривијални алгоритам који се веома често користи. Док он има временску сложеност O(MN) у најгорем случају, у многим стринговима ће имати временску сложеност O(M+N).

С. Кук (S.Cook) је 1970. године доказао да у теорији постоји тип апстрактне машине која се може наћи у алгоритаму који решава проблем тражења узорка и има временску сложеност O(*M+N)* у најгорем случају. Кнут (D. E. Knuth) и Прат (V. R. Pratt) су развили бољи и практичнији алгоритам уз помоћ Куковог доказа, али се испоставило да је Морис (J. H. Morris) открио сличан алгоритам где је избегавао враћање у тексту. С обзиром да су у релативно исто време дошли до сличног решења, Кнут, Морис и Прат су овај алгоритам објавили 1976. године. За то време су Бојер и Мур открили алгоритам који је доста бржи, пошто често користи само део карактера у тексту. Много текст едитора користе овај алгоритам да би убрзали претраживање. Оба алгоритма (КМП и Бојер-Мур) захтевају више функција над узорком и самим тим их чини захтевнијим за схватање. Рабин (M. O. Rabin) и Карп (R. M. Karp) су 1980. године користили хеширање да би развили алгоритам који једноставан као тривијални алгоритам који има временску сложеност O(N+M) са великом вероватноћом.

Алгоритми

ТРИВИЈАЛНИ АЛГОРИТАМ

Очигледан метод за тражење узорка у тексту подразумева проверу сваке позиције у тексту и узорку и проверавати да ли се подударају. У процедури *BruteForce()* се тражи прво појављивање узорка *pat* у тексту *txt*. Узимамо показивач *i* који иде кроз текст и показивач *j* који иде кроз узорак. За свако *i*, *j* се враћа на 0 и повећава се за 1 све док се не нађе неслагање или док се не дође до краја узорка (*j* == *M*). Ако дођемо до краја текста (*i* == *N*-*M*+1) пре краја узорка, онда нема подударања.

У тексту индекс *j* се ретко кад повећава, тако да је најповољнији случај, када нема узорка O(*N),* а кад има O(N+M).

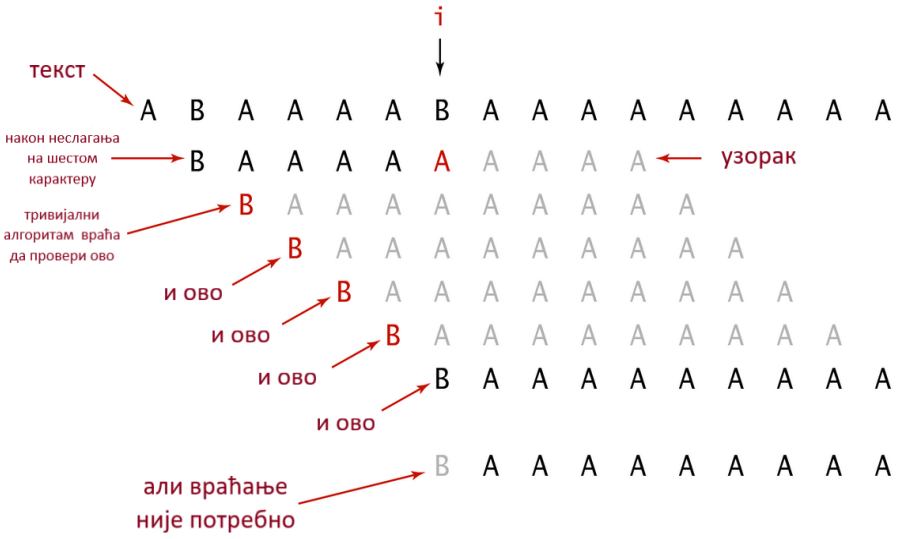
```
int BruteForce (char pat[], char txt[])
{
    int N = strlen(txt);
    int M = strlen(pat);
    for (int i = 0; i <= N - M; i++)
    {
        int j;
        for(j=0; j<M; j++)
            if (txt[j+i]!=pat[j])
break;
        if (j == M) return i;
    }
    return N;
}
```

Овај алгоритам захтева *NM* упоређивања карактера у најгорем случају.

КНУТ-МОРИС-ПРАТ АЛГОРИТАМ

Основна идеја овог алгоритма је следећа: сваки пут када нађемо неслагање, ми смо већ прошли карактере између почетка у тексту и неслагања (пошто су се они подударали са узорком пре неслгања). Ми можемо да искористимо ову информацију да би избегли враћање показивача у тексту преко већ познатих карактера.

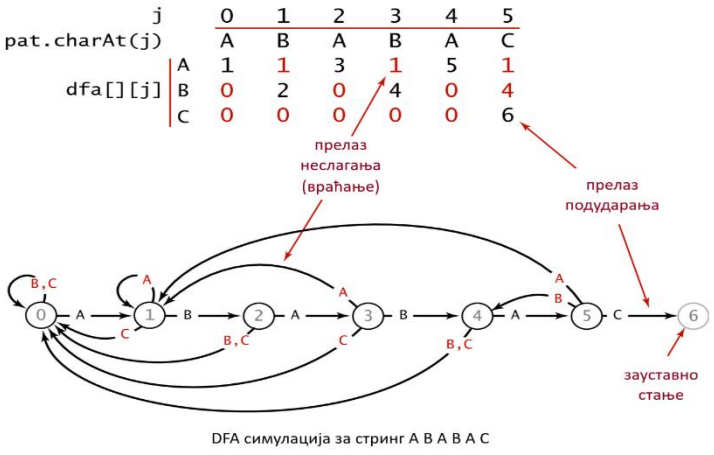
На пример, претпоставимо да имамо алфабет од два карактера и да тражимо узорак B A A A A A A A A A. Сада, претпоставимо да смо нашли пет карактера који се подударају у узорку, са неслагањем на шестом. Када



враћање текстуалног показивача при тражењу узорка

нађемо неслагање ми знамо да претходних шест карактера морају бити B A A A A B. Кључна ствар је да не враћамо показивач *i*, пошто знамо да су претходна четири карактера била A, а они се не подударају са првим карактером у узорку. Карактер на који показује показивач *i* је B, а он се подудара са првим карактером у узорку, тако да можемо да повећамо *i* и да упоредимо следећи карактер у тексту са другим карактером у узорку. Прескакање свих подударених карактера када се пронађе неслагање неће увек радити јер се узорак може подударити са неким делом унутар прескочених карактера и тако се може прескочити решење. На пример, када тражимо узорак A A B A A A у тексту A A B A A B A A A A, прво проналазимо неслагање на позицији 5, а затим уместо да кранемо даље са претраживањем од позиције 6, треба да кренемо од позиције 3 пошто би иначе изгубили решење.

Најбољи начин да објаснимо алгоритам је преко *deterministic finite-state automation (DFA)*. Овај аутомат можемо графички представити помоћу стања (представљени у облику кругова) и прелаза (представљени у облику линија). Постоји једно стање за сваки карактер у узорку и за свако то стање постоји по један прелаз за сваки карактер у алфабету. Један од прелаза је прелаз подударања

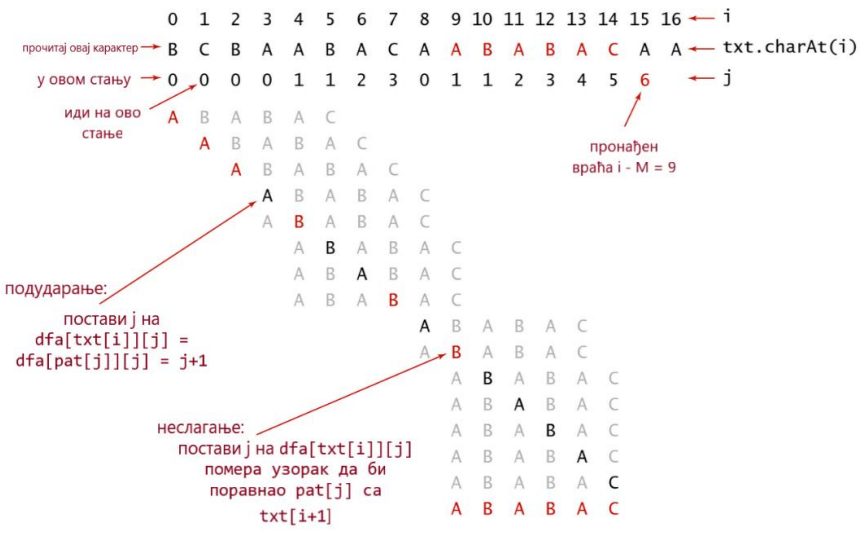


DFA симулација за стринг A B A B A C

```
for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt[i]][j];
if ( j == M ) return i - M;
        else return N;
```

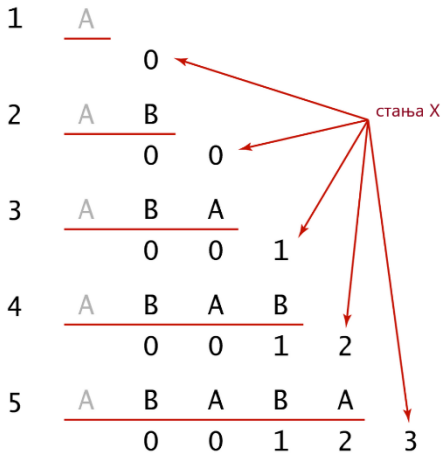
(иде од *j* до *j+1* и обележен је са *pat[j]*), а сви остали су прелази неслагања (иду лево). Стања одговарају на поређења карактера, једно за сваку вредност индекса у узорку. Прелази одговарају на мењање вредности индекса у узорку. Испитивањем карактера *i* у тескту у стању означеним *j*, аутомат чини следеће: узима прелаз ка *dfa[txt[i]][j]* и прелази на следећи карактер (повећавањем *i*). За прелаз подударања, прелазимо на десну позицију јер је *dfa [txt[j]][j]* увек j+1; за прелаз неслагања прелазимо лево. Аутомат чита следеће карактере у тексту идући с лева на десно, прелазећи на следеће стање сваки пут када прочита карактер. Такође укључујемо зауставно стање *M* које нема прелазе. Ми покрећемо аутомат на стању *0*. Ако аутомат достигне стање *M*, онда је пронађено решење, а ако аутомат стигне до краја текста пре стања *M*, онда знамо да се узорак не појављује у тексту.

На почетку процеса, када је *DFA* на стању *0* на почетку текста, он остаје на стању *0*, скенирајући карактере текста, све док не нађе карактер који се поклапа са првим карактером у узорку, тада прелази на следеће стање и наставља даље све док не дође до стања *M* или краја текста. Начин рада можемо видети на слици . Свако подударање помера ДФА на следеће стање (што повећава индекс *j* у узорку), а свако неслагање помера ДФА на неко од претходних стања (што значи смањење индека j на мању вредност). Индекс i се креће с лева на десно, док се индекс j креће у узорку.



Сада када разумемо механизам, можемо прећи на кључно питање КМП алгоритма: Како да конструишемо *dfa[][]* низ који одговара датом узорку? Одговор се налази у самом ДФА користећи идеју Кнута, Мориса и Прата. Када имамо неслагање на позицији *pat[j]*, ми

желимо да знамо на ком стању би ДФА био да смо враћали индекс текста и посматрамо карактере текста које смо већ видели након померања у десно за једну позицију. Ми не желимо стварно да враћамо индекс *i*, већ да рестартујемо ДФА као да је урађено враћање. Ми померамо први и последњи карактер у десно за једно место због неслагања. То су карактери у узорку које знамо, тако да можемо у напред открити, за свако место, стање на ком морамо да рестартујемо ДФА.

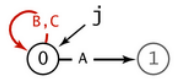


Шта би требало ДФА да ради са следећим карактером? Исто што би радио када би се враћао у назад, осим ако нађе решење са *charAt(j),* када треба да иде у стање *j+1*. На пример, да би схватили шта ДФА треба да ради када имамо неслагање на *j=5* за A B A B A C, ми користимо ДФА да би научили да враћањем би нас оставило на стању 3 за B A B A, тако да можемо да копирамо *dfa[][3]* на *dfa[][5]*, затим да поставимо улаз за *C* на 6 зато што је *charAt(5) C* (решење). Пошто нам је довољно да знамо како ДФА функционише за j-1 карактера када правимо j-то стање, увек можемо да добијемо ту информацију из већ изграђеног ДФА.

Последњи детаљ за изградњу овог алгоритма је да посматрамо позицију *X*. Када посматрамо колону *j* у *dfa[][]* можемо да употребимо већ изграђен ДФА систем : следећа вредност позиције *X* је *dfa[pat[j]][X]*. Даље би ажурирали вредност *X* на *dfa['C'][3]=0*. За свако j он: копира *dfa[][X]* на *dfa[][j]* (у случају неслгања); поставља *dfa[pat[j]][j]* на *j+1* (у случају поклапања); ажурира X.

```
dfa[pat[0]][0] = 1;
for (int X = 0, j = 1; j < M; j++)
{
    for (int c = 0; c < R; c++)
        dfa[c][j] = dfa[c][X];
    dfa[pat[j]][j] = j+1;
    X = dfa[pat[j]][X];
```

j	0
pat.charAt(j)	A
dfa[][j] A	1
dfa[][j] B	0
dfa[][j] C	0

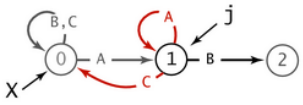


копирај dfa[][X] у dfa[][j];

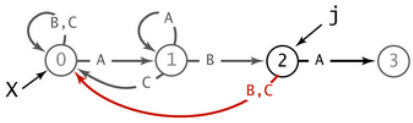
```
dfa[pat[j]][j] = j+1;

X = dfa[pat[j]][X];
```

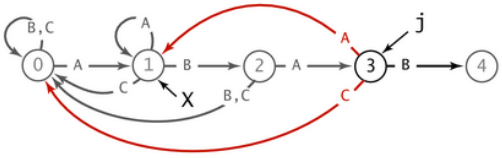
		X ↓	
j	0	1	
pat.charAt(j)	A	B	
dfa[][j] A	1	1	
dfa[][j] B	0	2	
dfa[][j] C	0	0	



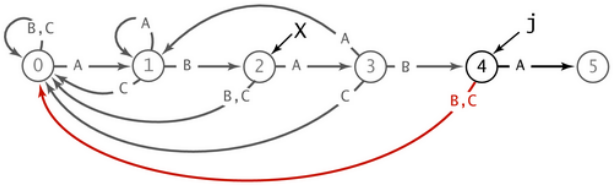
			X ↓	
j	0	1	2	
pat.charAt(j)	A	B	A	
dfa[][j] A	1	1	3	
dfa[][j] B	0	2	0	
dfa[][j] C	0	0	0	



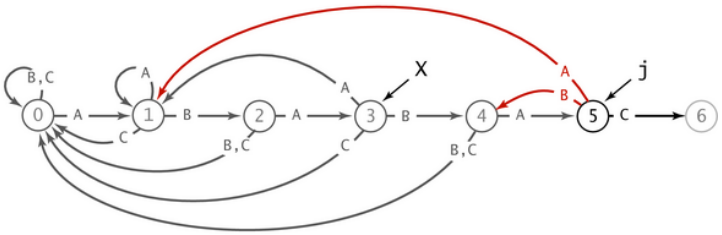
				X ↓
j	0	1	2	3
pat.charAt(j)	A	B	A	B
dfa[][j] A	1	1	3	1
dfa[][j] B	0	2	0	4
dfa[][j] C	0	0	0	0



					X ↓
j	0	1	2	3	4
pat.charAt(j)	A	B	A	B	A
dfa[][j] A	1	1	3	1	5
dfa[][j] B	0	2	0	4	0
dfa[][j] C	0	0	0	0	0



						X ↓
j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][j] A	1	1	3	1	5	1
dfa[][j] B	0	2	0	4	0	4
dfa[][j] C	0	0	0	0	0	6

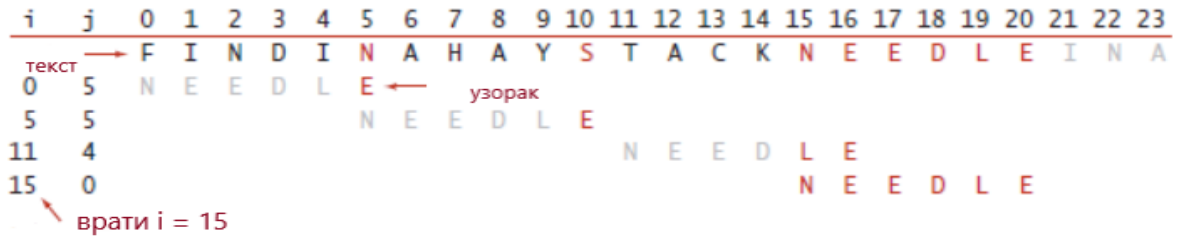


```
int KMP ( char pat[], char txt[])
{
    int M = strlen(pat);
    int R = 256;
    int dfa[R][M];
    dfa[pat[0]][0] = 1;
    int i,j;
    for (int X = 0, j = 1; j < M; j++)
    {
        for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][X];
        dfa[pat[j]][j] = j+1;
        X = dfa[pat[j]][X];
    }

    int N = strlen(txt);
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt[i]][j];
    if ( j == M ) return i - M;
        else return N;
}
```

БОЈЕР-МУР АЛГОРИТАМ

Сада можемо конструисати значајно бржи алгоритам тако што скенирамо узорак с десна на лево. На пример, када тражимо узорак B A A B B A A, ако нађемо подударање на шестом и петом карактеру, али неслагање на четвртом, онда можемо одмах померити узорак за 7 позиција у десно, и даље проверити тринаести карактер у тексту. У суштини, узорак на крају може да се појави још негде, тако да нам је потребан низ за рестартовање позиција као код КМП алгоритма.



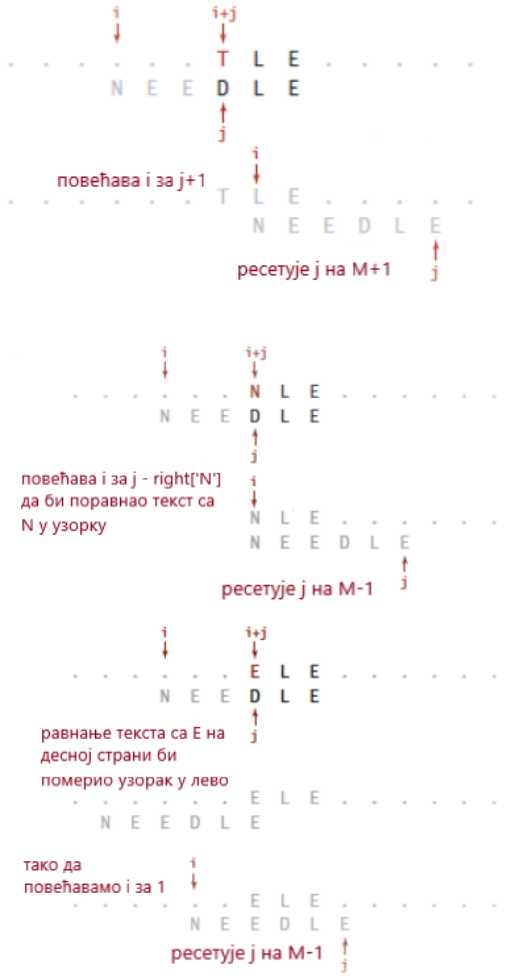
Посматрајмо слику изнад, на којој се приказује тражење узорка N E E D L E у тексту F I N D I N A H A Y S T A C K N E E D L E. Претражујући с десна на лево посматрамо карактер E (који је на десном крају узорка) и упоређујемо га са N (који је на петој позицији у тексту). Пошто се N појављује у узорку, ми померамо узорак у десно за пет позиција да би поклопили N у тексту са N у узорку на десном крају. Затим, поредимо E на десном крају узорку са S (на десетој позицији у тексту). Пошто се они не подударају и S се не појављује у узорку можемо да померимо узорак за шест позиција у десно. Даље, поклапамо E на десном крају у узорку са E на позицији 16 у тексту, и затим проналазимо неслагање и откривамо N на позицији 15 и померамо узорак за 5 позиција у десно. На крају, померајући узорак с лева на десно почевши од позиције 20 увиђамо да је узорак у тексту. Да би имплементирали овај алгоритам биће нам потребан низ *right[],* који даје, за сваки карактер у алфабету, индекс његовог последњег појављивања (најближи десном крају) у тексту или *-1* у колико се не појављује у узорку. Ова вредност нам показује колико тачно позиција у тексту да прескочимо ако се тај карактер појави у тексту и проузрокује неслагање. Да би креирали низ *right[],* постављамо сва места на -1, а затим за

		N	E	E	D	L	E	
c		0	1	2	3	4	5	right[c]
A	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3	3
E	-1	-1	1	2	2	2	5	5
...								-1
L	-1	-1	-1	-1	-1	4	4	4
M	-1	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0	0
...								-1

индекс *j* који иде од *0* до *M-1*, постављамо *right[pat[j]]* на *j*.

Када смо објаснили настанак низа *right[]*, имплементација алгоритма је једноставна. Узимамо индекс *i* који се креће с лева на десно у тексту и индекс *j* који иде с десна на лево у узорку. Унутрашња петља проверава да ли се узорак поклапа са текстом на позицији *i*. Ако је *txt[i+j] = pat[j]* за свако *j* од *M-1* до *0*, онда смо нашли решење. Иначе, настаје неслагање и постоје три даља случаја:

- Ако се карактер који је проузроковао неслагање не налази у узорку, онда можемо да померимо узорак за *j+1* места у десно, односно повећамо *i* за *j+1*.
- Ако се карактер *c*, који је проузроковао неслагање, налази у узорку, онда користимо низ *right[]* који ће поравнати узорак са текстом тако да ће се карактери у тексту поклапати са својим појављивањем у узорку које се налази на десном крају узорка. Да би то учинили ми повећавамо *i* за *j-right[c]*.
- Иначе, ако се индекс *i* не повећа, онда ћемо да га повећамо за 1 да би се узорак увек померао за једно место у десно.



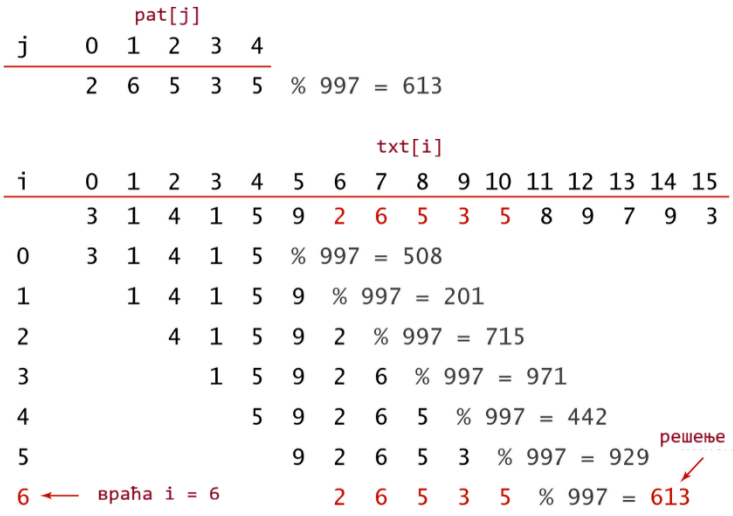
```
int BoyerMoore ( char pat[], char txt[])
{
    int M = strlen(pat);
    int N = strlen(txt);
    int R = 256;
    int right[R];
    int skip;
    for (int c = 0; c < R; c++)
        right[c] = -1;
    for (int j = 0; j < M; j++)
        right[pat[j]] = j;
    for (int i=0; i<= N-M; i+=skip)
    {
        skip = 0;
        for(int j = M-1; j>=0; j--)
            if (pat[j]!=txt[i+j])
            {
                skip = j - right[txt[i+j]];
                if (skip < 1) skip = 1;
                break;
            }
            if (skip == 0) return i;
    }

  return N;
}
```

РАБИН-КАРП АЛГОРИТАМ

Овај метод је потпуно другачији метод од претходна три које смо тумачили и базира се на хеширању. Ми направимо хеш функцију за узорак, а затим тражимо подударања користећи исту хеш функцију за сваки могући подстринг од *M* карактера у тексту. Ако пронађемо подстринг са истом хеш вредности као и узорак, онда можемо проверити да ли постоји подударање. Овај процес је еквивалентан смештању вредости у хеш табелу, затим претраживати за сваки подстринг у тексту, али не морамо да резервишемо меморију за ову табелу јер би она имала само један улаз. Имплементација на основу овог описа би била спорија од тривијалног алгоритма, али Рабин и Карп су показали да је лакше направити хеш функције за подстрингове од *M* карактера у константном времену што доводи до претраживања у линеарном времену.

Стринг од *M* карактера одговара броју од *M* цифара у систему са основом *R*. Да би користили хеш табелу величине *Q*, потребна нам је хеш функција да конвертује број од *M* цифара у систему са основом *R* у *int* између *0* и *Q-1*, што значи да ћемо узети остатак при дељењу са *Q*. Ми користимо произвољан прост број *Q* и узимамо највећу могућу вредност док избегавамо прекорачење. Ова метода је најједноставнија за схватање за мало *Q* и *R=10*, као што је показано на примеру. Да би пронашли узорак 2 6 5 3 5 у тексту 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3, ми користимо табелу величине *Q* (на пример 997), одредимо хеш вредност 26535 % 997 = 613, а затим тражимо подударање тако што израчнамо хеш вредност за сваки петоцифрени подстринг у тексту. У овом примеру ми добијамо хеш вредности 508, 201, 715, 971, 442 и 929 пре проналаска решења 613.



Да би креирали хеш функцију користићемо Хорненров метод који представља метод за трансформацију полинома у облик погоднији за израчунавање. Представићемо број од *M* цифара у систему са основом *R* као низ вредонсти типа *char.* За сваку цифру у броју, ми је помножимо са *R*, додамо следећу цифру, и узимамо остатак при дељењу са *Q*. Иста

```
for (i = 0; i < M; i++)

    patHash = (R*patHash + pat[i])%Q;
```

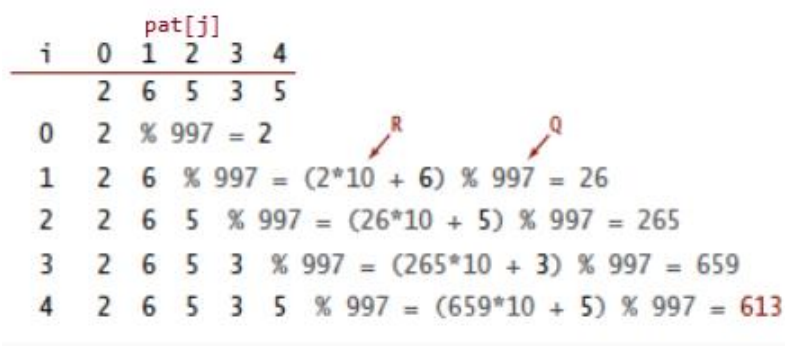
метода може се користити и у тексту, али много времена ће узимати сабирање, множење и рачунање остатка за сваки карактрер што значи *MN* операција у најгорем случају, што није ништа боље од тривијалног алгоритма.
Овај метод можемо представити и математички. Користићемо ознаку t_i за *txt.charAt(i).* Број који одговара подстрингу од *M* цифара и почиње од места *i* је:

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^{0}$$

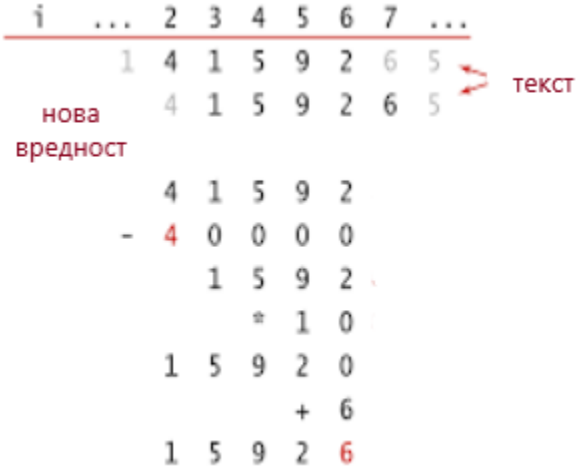
Можемо да претпоставимо да знамо вредност $h(x_i) = x_i \bmod Q$. Померајући у тексту за једно место у десно замењујемо x_i са:

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M}$$

Ми одузимамо прву цифру, помножимо са R , а затим додајемо нову цифру. Поента је да не морамо да задржавамо вредности бројева, већ само њихове остатке када се поделе са *Q*.

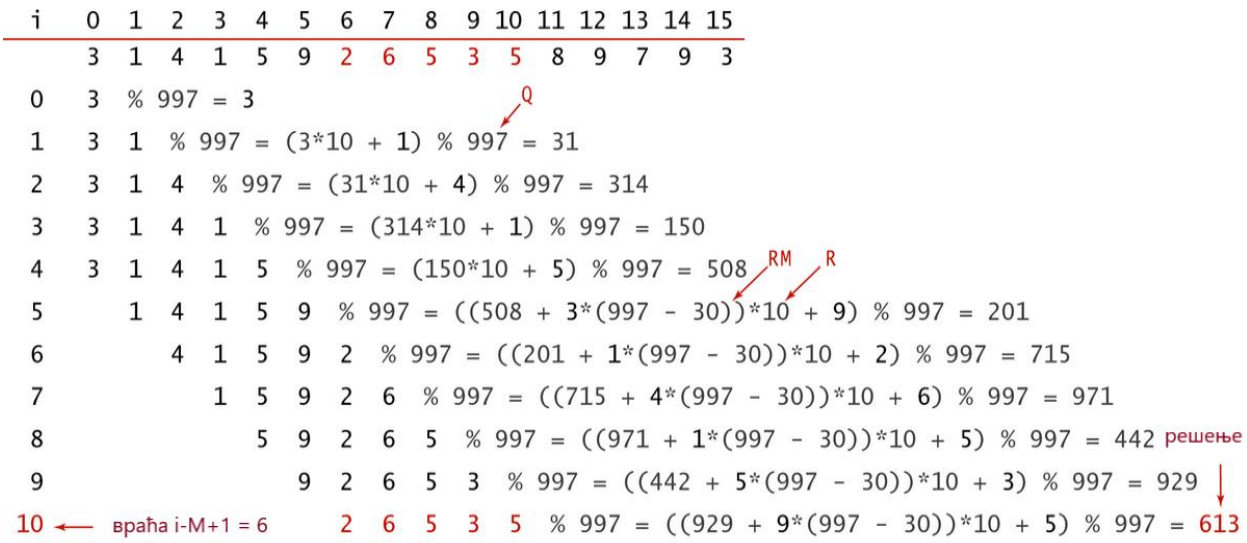


Конструктор рачуна хеш вредност *patHash* за узорак. Он такође рачуна вредност $R^{M-1} \bmod Q$ у варијабли *RM*. *HashSearch()* метод почиње рачунањем хеш функције за првих *M* карактера у тексту и пореди вредности са хеш вредности у узорку. Ако се не подударају, он наставља да иде кроз текст, користећи технику која одржава хеш функцију за *M* карактера почевши од позиције *i* за свако *i* у варијабли *txtHash* и пореди сваку нову хеш вредност са *patHash*.



Након што смо пронашли хеш вредост за подстинг од *M* карактера у тексту који одговара хеш вредности у узорку, ми нећемо поредити карактере у тексту са карактерима у узорку да бисмо проверили да ли је дошло до подударања јер би то тражило враћање у тексту. Уместо тога, ми правимо хеш табелу величине које желимо пошто нећемо заправо конструисати табелу већ само тестирамо за неподударност са узорком. Користићемо *long* вредност већу од 10^{20}, тако да то оставља вероватноћу да смо пронашли погрешно решење

мању од 10^{-40}, што је довољно мало. Овај алгоритам је познат пример Монте Карло алгоритма који гарантује добру брзину алгоритма, али постоји мала вероватноћа да одреди погрешно решење. Постоји још један метод за проверу подударања али може бити спор, али је увек тачан. Овај алгоритам је познат као Лас Вегас алгоритам. Монте Карло верзија Рабин-Карп претраживања је у линеарном времену и постоји велика вероватноћа да буде тачан, а Лас Вегас верзија је увек тачна и скоро увек у линеарном времену.



```
void RabinKarp(char pat[], char txt[] )
{
    long patHash=0;
    long txtHash=0;
    int M = strlen(pat) ;
    int N = strlen(txt) ;
    long Q = 997 ;
    int R=256;
    long RM = 1;
    int i,j;

    for (i = 0; i < M; i++)
    {
        patHash = (R*patHash + pat[i])%Q;
        txtHash = (R*txtHash + txt[i])%Q;
    }
   for (i = 1; i <= M-1; i++)
        RM = ( R * RM ) % Q;
   for (i = 0; i <= N - M; i++)
    {
        if ( patHash == txtHash )
        {
            for (j = 0; j < M; j++)
            {
                if (txt[i+j] != pat[j])
                    break;
            }
            if (j == M)

                printf("Pattern found at index %d \n", i);

        }

        if ( i < N-M )

        {

            txtHash = (R*(txtHash - txt[i]*RM) + txt[i+M])%Q;

            if (txtHash < 0)

            txtHash = (txtHash + Q);

        }  }

}
```

-15-

Z АЛГОРИТАМ

Овај алгоритам проналази сва појављивања узорка у линеарном времену. Ако је дужина текста *N*, а дужина узорка *M*, сложеност алгоритма је *O(M+N)*. Одатле можемо видети да су временска и просторна сложеност овог алгоритма исти као код КМП алгоритма, само што се овај алгоритам може сматрати једноставнијим за схватање.

У овом алгоритму ћемо користити низ *Z* који је исте дужине као текст *txt[0..n-1]*. Елемент *Z[i]* у овом низу садржи дужину од најдужег подстринга почевши од *txt[i]*, који је такође префикс *txt[0..n-1]*. Први елемент од *Z* низа је заправо небитан пошто је цео стринг увек свој подстринг.

Можемо видети представаљање *Z* низа на следећем примеру:

Пример:

Индекс	0	1	2	3	4	5	6	7	8	9	10	11
Текст	a	a	b	c	a	a	b	x	a	a	a	z
Z вредности	X	1	0	0	3	1	0	0	2	2	1	0

Пример:

str = "aaaaaa"

Z[] = {x, 5, 4, 3, 2, 1}

str = "aabaacd"

Z[] = {x, 1, 0, 2, 1, 0, 0}

str = "abababab"

Z[] = {x, 0, 6, 0, 4, 0, 2, 0}

Сада можемо посматрати како нам овај низ може помоћи у тражењу узорка. Идеја је да направимо стринг који повезује текст и узорак. Тај стринг ће бити облика *“P$T”*, где је *P* узорак, *$* је неки карактер који се не би појављивао у тексту и *T* је текст. Након тога ћемо

да направимо низ *Z* за овај стринг. У низу, ако је *Z* вредност једнака дужини узорка, онда смо пронашли решење.

Пример:

Pat = "aab", Txt = "baabaa"

Str= "aab$baabaa"

Z [] = {x, 1, 0, 0, 0, 3, 1, 0, 2, 1}.

Пошто је дужина узорка 3, вредност 3 у низу *Z* означава присуство узорка у тексту.

Конструкција Z низа

Једноставно решење је да покренемо две петље, једна у другој. Спољашња иде кроз сваки индекс, а унутрашња петља проналази дужину најдужег префикса који одговара подстрингу који почиње од тог индекса. Временска сложеност овог решења је $O(n^2)$.

Можемо и конструисати низ *Z* у линеарном времену. Идеја је да имамо интервал *[L, R]* где је *R* максимално тако да је интервал префикс.

Ако је *i < R* онда не постоји префикс који почиње пре *i* и завршава се после *i*, тако да ми ресетујемо *L* и *R* и правимо нови интервал *[L, R]* тако што поредимо *str[0..]* и *str[i..]* и добијамо *Z[i] = R – L + 1*.

Ако је *i <= R* онда нека је *K= i - L* , а *Z[i] >= min (Z[K], R-i+1)* зато што се *str[i..]* подудара са *str[K..]* за макар *R-i+1* карактера

Ако је *Z[k]< R-i+1* онда нема префикс који почиње од *str[i]* (иначе би *Z[K]* било веће) тако да је *Z[i]=Z[K]* , а интервал остаје исти.

Ако је *Z[K]>= R-i+1* онда је могуће да проширимо интервал *[L, R]* тако што поставимо *L* на *i* и затим почињемо да упоређујемо од *str[R]* и добијамо ново *R* и тако ћемо добити нови интервал *[L, R]* и израчунати *Z[i]= R-L+1*.

```
void search(string text, string pattern)
{
    string concat = pattern + "$" + text;
    int l = concat.length();
    int Z[l];  // Konstruišemo niz Z
    getZarr(concat, Z);
    for (int i = 0; i < l; ++i)
    {
        if (Z[i] == pattern.length())
       cout << "Pattern found at index "
                 <<  i - pattern.length() -1 << endl;
    }
}

void getZarr(string str, int Z[])
{
    int n = str.length();
    int L, R, k;
    L = R = 0;
    for (int i = 1; i < n; ++i)
    {
        if (i > R)
        {
            L = R = i;
            while (R<n && str[R-L] == str[R])  R++;
            Z[i] = R-L;
            R--;
        }
        else
        {
            k = i-L;
            if (Z[k] < R-i+1)
                 Z[i] = Z[k];
            else
            {
                L = i;
                while (R<n && str[R-L] == str[R])  R++;
                Z[i] = R-L;
                R--;
            }
        }
    }
}
```

Референце

- Sedgewick, R., Wayne, K. (2011) *Algorithms*, 4th edn., Boston: Addison-Wesley.
- Živković, M. (2000) *Algoritmi*, Beograd: Matematički fakultet.
- https://www.coursera.org/learn/algorithms-part2/lecture/n3ZpG/introduction-to-substring-search
- https://www.geeksforgeeks.org/z-algorithm-linear-time-pattern-searching-algorithm/
- https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/
- https://en.wikipedia.org/wiki/String_searching_algorithm
- https://ivanyu.me/blog/2013/10/15/z-algorithm/
- https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm
- https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm
- https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm