String Matching

String Matching

Consiste em encontrar um padrão dentro de um texto

Padrão -> char P[ ]

Texto -> char T[ ]

Busca Ingênua

Extremamente lenta (O(tamT * tamP))

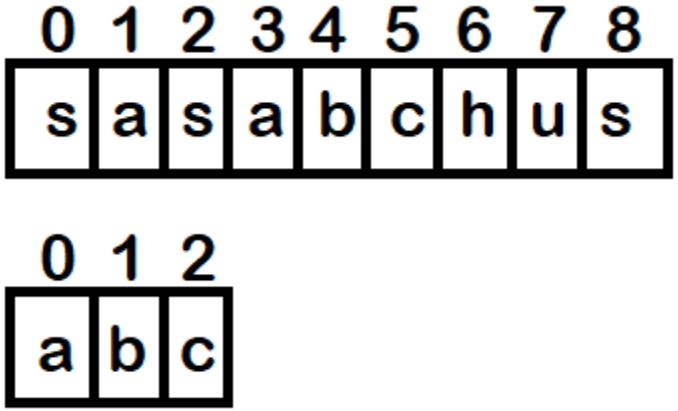
P[] = “abc”

T[] = “sasabchus”

Busca Ingênua

P[] = “abc”

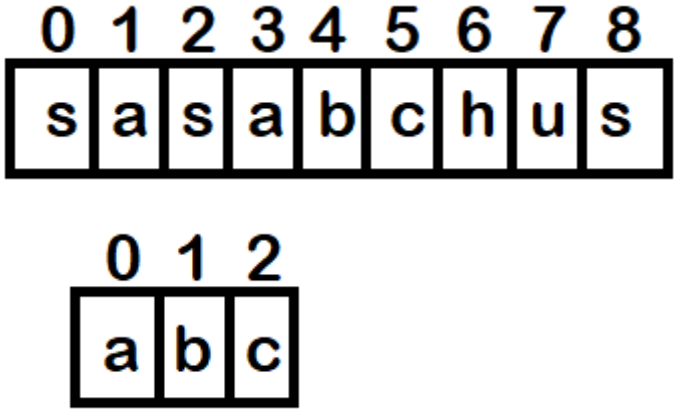
T[] = “sasabchus”



Busca Ingênua

P[] = “abc”

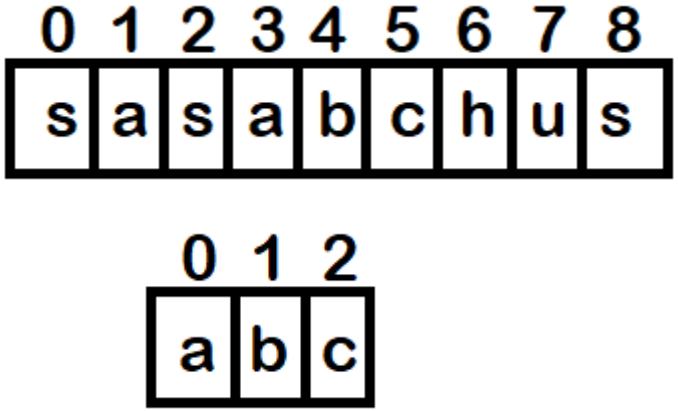
T[] = “sasabchus”



Busca Ingênua

P[] = “abc”

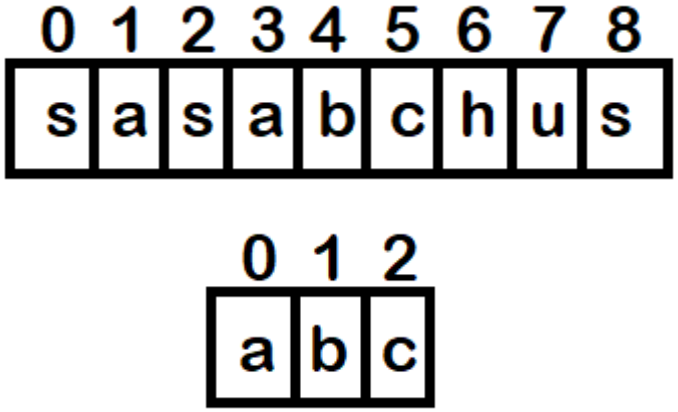
T[] = “sasabchus”



Busca Ingênua

P[] = “abc”

T[] = “sasabchus”

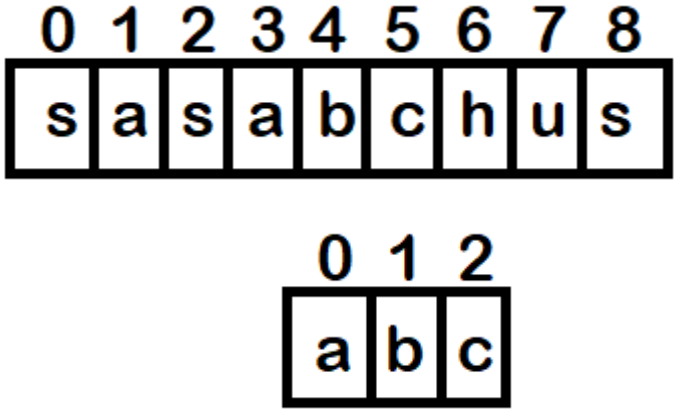


Padrão encontrado na posição i = 3

Busca Ingênua

P[] = “abc”

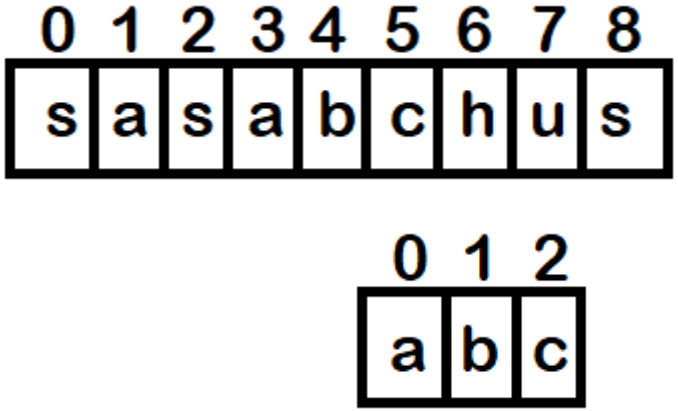
T[] = “sasabchus”



Busca Ingênua

P[] = “abc”

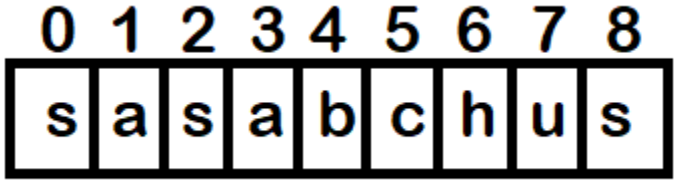
T[] = “sasabchus”

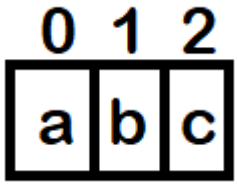


Busca Ingênua

P[] = “abc”

T[] = “sasabchus”





Busca Ingênua

```
for(int i = 0;i <= tamT-tamP; i++)
{
        for(int j = 0;j < tamP;j++)
         {
                if(T[i+j] != P[j])    // Se houver um mismatch
                        break;
                if(j == tamP-1)    // Padrão encontrado
                        cout << i << endl;
        }
}
```

Algoritmo de Rabin-Karp

- Mais rápido que a busca ingênua (contudo, ainda tem complexidade O(tamT*tamP));
- “É mais rápido comparar números que comparar strings completas”;
- Calcula valor de hash do padrão e de cada parte do texto;
- Só iremos comparar a STRING, se o valor de hash for igual!!!

Algoritmo de Rabin-Karp

P[ ] = {3,1,4,1,5} Padrão

T[ ] = {2,3,5,9,0,2,3,1,4,1,5,2,6,7,3,9,9,2,1} Texto

d = 10 Cardinalidade -> quantidade de caracteres diferentes

q = 13 Valor primo grande
Será utilizado para calcular o valor de hash

Algoritmo de Rabin-Karp

Convertendo o padrão para inteiro mod 13:

P[ ] = “31415”

p = 3 * 10^4 + 1 * 10^3 + 4 * 10^2 + 1 * 10^1 + 5 * 10^0

Convertendo a primeira parte do texto para inteiro mod 13:

T[ ] = “23590” -> vamos chamar de “t0”

t = 3 * 10^4 + 1 * 10^3 + 4 * 10^2 + 1 * 10^1 + 5 * 10^0

Algoritmo de Rabin-Karp

```
// lembrando que  d = 10 e q = 13
for(int i = 0; i < tamP; i++)
{
        p = (d * p + (P[i] - '0') ) % q;
        t = (d * t + (T[i] - '0') ) % q;
}
```

Dica: '0' é diferente de 0

'0' vale 48 (valor da tabela ASCII)

Algoritmo de Rabin-Karp

Para que o código não fique lento, a conversão de um pedaço de texto para outro deve ser com uma operação rápida, não podemos calcular um novo t_i a cada operação.

T[ ] = {2,3,5,9,0,2,3,1,4,1,5,2,6,7,3,9,9,2,1}

t0 = 23590 mod 13 = 8

t1 = 35902 mod 13 = 9

Como alterar de t0 para t1 com 1 operação?

Algoritmo de Rabin-Karp

t0 = 23590 mod 13 = 8

t1 = 35902 mod 13 = 9

h = $d^{tamP-1} mod\ q$

h = $10^4\ mod\ 13$ = 3

t_{i+1}= $(d * (t_i - (T[i] -' 0') * h) + (T[i + tamP] -' 0'))\ mod\ q$

t_1= $(10 * (8 - 3 * 2) + 2) mod\ 13$ = 9

* Se o valor de t calculado for negativo, deve-se somar q

Algoritmo KMP

- Desenvolvido por Donald Knuth, James Morris e Vaughan Pratt;
- Também tem como objetivo encontrar as ocorrências de um padrão em um texto;
- Complexidade: O(n), sendo n o tamanho do texto.

Prefixo

Prefixo de uma string são os caracteres iniciais dessa string, por exemplo, para a string "ABABA", os prefixos são:

"A"

"AB"

"ABA"

"ABAB"

"ABABA"

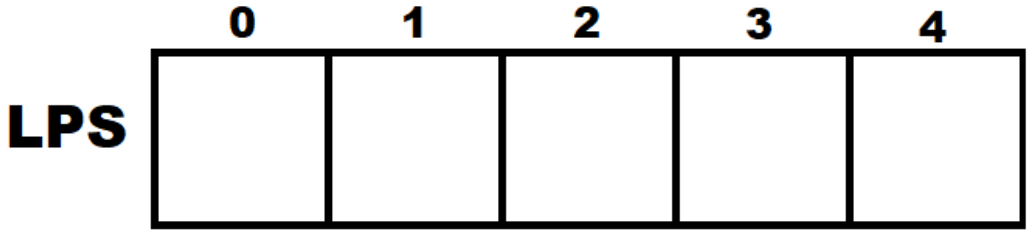
Sufixo

Sufixo de uma string são os caracteres finais dessa string, por exemplo, para a string "ABABA", os sufixos são:

"A"
"BA"
"ABA"
"BABA"
"ABABA"

LPS

- LPS é um vetor que guarda o tamanho maior prefixo que também é sufixo considerando sua string da posição 0...i e desconsiderando a string completa para a sua construção;
- Ele é construído com base no padrão a ser procurado;
- Por exemplo, considerando o padrão "ABABA", vamos montar o vetor LPS:

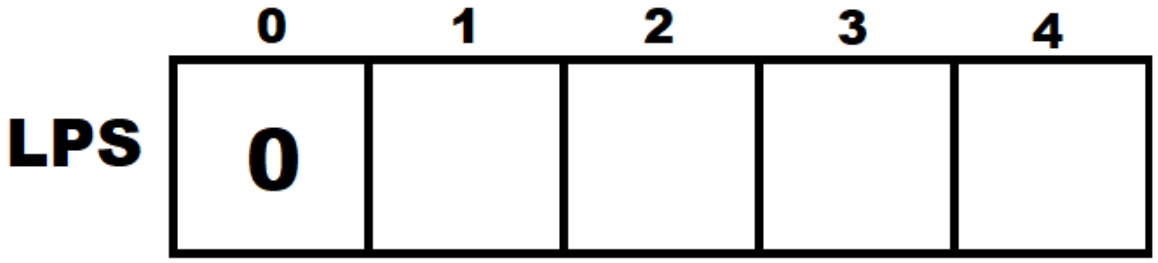


LPS

Para i=0, temos a string "A", desconsiderando a string completa:

Prefixos
"A"

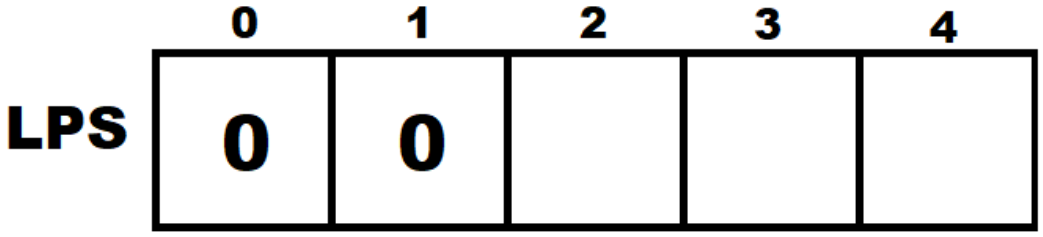
Sufixos
"A"



LPS

Para i=1, temos a string "AB", desconsiderando a string completa:

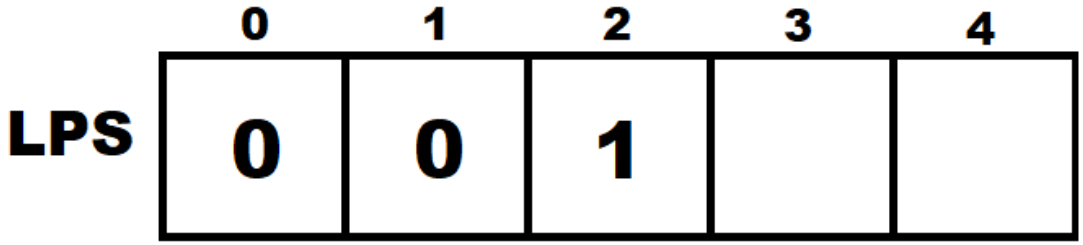
Prefixos	Sufixos
"A"	"B"
~~"AB"~~	~~"AB"~~



LPS

Para i=2, temos a string "ABA", desconsiderando a string completa:

Prefixos	Sufixos
"A"	"A"
"AB"	"BA"
~~"ABA"~~	~~"ABA"~~



LPS

Para i=3, temos a string "ABAB", desconsiderando a string completa:

Prefixos	Sufixos
"A"	"B"
"AB"	"AB"
"ABA"	"BAB"
~~"ABAB"~~	~~"ABAB"~~

	0	1	2	3	4
LPS	0	0	1	2	

LPS

Para i=4, temos a string "ABABA", desconsiderando a string completa:

Prefixos	Sufixos	Prefixos	Sufixos
"A"	"A"	~~"ABABA"~~	~~"ABABA"~~
"AB"	"BA"		
"ABA"	"ABA"		
"ABAB"	"BABA"		

	0	1	2	3	4
LPS	0	0	1	2	3

```
void calcula_LPS(int lps[], char P[], int tamP){
    int tam = 0;

    lps[0] = 0;

    int i = 1;
    while (i < tamP)
    {
        if (P[i] == P[tam])
        {
            tam++;
            lps[i] = tam;
            i++;
        }
        else
        {
            if (tam != 0)
                tam = lps[tam - 1];
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

Algoritmo KMP

A ideia principal do algoritmo é:

- Nunca voltar no seu texto;

- Se ocorrer um mismatch (uma posição do padrão e do texto não baterem), eu posso aproveitar o que já comparei.

Algoritmo KMP

Ex.:

Padrão: ABABA

	0	1	2	3	4
LPS	0	0	1	2	3

	0	1	2	3	4	5	6	7	8	9	10	11
Texto:	C	A	B	A	A	B	A	B	A	A	B	C

Algoritmo KMP

i = 0 // pos. no texto

j = 0 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Mismatch: j = 0 => i++

Algoritmo KMP

i = 1 // pos. no texto

j = 0 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Match: i++ & j++

Algoritmo KMP

i = 2 // pos. no texto

j = 1 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Match: i++ & j++

Algoritmo KMP

i = 3 // pos. no texto

j = 2 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Match: i++ & j++

Algoritmo KMP

i = 4 // pos. no texto

j = 3 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Mismatch: j != 0 => j = lps[j-1] = lps[2]

Algoritmo KMP

i = 4 // pos. no texto

j = 1 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

-> A B A B A -> aproveita o 'A' que já foi comparado

Algoritmo KMP

i = 4 // pos. no texto

j = 1 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Mismatch: j != 0 => j = lps[j-1] = lps[0]

Algoritmo KMP

i = 4 // pos. no texto

j = 0 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Match: i++ & j++

Algoritmo KMP

i = 5 // pos. no texto

j = 1 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Match: i++ & j++

Algoritmo KMP

i = 6 // pos. no texto

j = 2 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Match: i++ & j++

Algoritmo KMP

i = 7 // pos. no texto

j = 3 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Match: i++ & j++

Algoritmo KMP

i = 8 // pos. no texto

j = 4 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Match: i++ & j++

Algoritmo KMP

i = 9 // pos. no texto

j = 5 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

A B A B A

j = 5 (tamanho do padrão)

ENCONTREI O PADRÃO

NA POSIÇÃO 4 (pos: i-tamP)

Encontrei: j = lps[j-1] = lps[4]

Algoritmo KMP

i = 9 // pos. no texto

j = 5 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

-> A B A B A

Encontrei: j = lps[j-1] = lps[4]

Algoritmo KMP

i = 9 // pos. no texto

j = 3 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Mismatch: j = lps[j-1] = lps[2]

Algoritmo KMP

i = 9 // pos. no texto

j = 3 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

-> A B A B A

Mismatch: j = lps[j-1] = lps[2]

Algoritmo KMP

i = 9 // pos. no texto

j = 1 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Mismatch: j = lps[j-1] = lps[0]

Algoritmo KMP

i = 9 // pos. no texto

j = 0 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Match: i++ & j++

Algoritmo KMP

i = 10 // pos. no texto

j = 1 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Match: i++ & j++

Algoritmo KMP

i = 11 // pos. no texto

j = 2 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Mismatch: j = lps[j-1] = lps[1]

Algoritmo KMP

i = 11 // pos. no texto

j = 0 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

Mismatch: j = 0 => i++

Algoritmo KMP

i = 12 // pos. no texto

j = 0 // pos. no padrão

	0	1	2	3	4
LPS	0	0	1	2	3

C A B A A B A B A A B C

|

A B A B A

i >= tamT portanto, FIM

```
void kmp(char T[], char P[],int tamT, int tamP)
{
    int lps[tamP];
    calcula_LPS(lps, P, tamP);

    int j = 0;
    int i = 0;
    while(i < tamT){
        if(T[i] == P[j]) // match
        {
            j++;
            i++;
            if(j == tamP){
                cout << "Padrao em " << i-tamP << endl;
                j = lps[j-1];
            }
        }
        else // mismatch
        {
            if(j != 0)
                j = lps[j-1];
            else
                i++;
        }
    }
}
```