

UNIVERSIDADE DO ESTADO DO AMAZONAS - UEA
ESCOLA SUPERIOR DE TECNOLOGIA - EST

Prof SERGIO CLEGER TAMAYO

GABRIEL SENA SAN MARTIN
GONZALO IVAN DOS SANTOS PORTALES
VICTOR YAN PEREIRA E LIMA

Galil-Seiferas, Optimal Mismatch, Two-Way e String Matching On
Ordered Alphabets

MANAUS
2021

**Gabriel Sena San Martin, Gonzalo Ivan dos Santos Portales, Victor Yan Pereira
e Lima**

**Galil-Seiferas, Optimal Mismatch, Two-Way e String Matching On
Ordered Alphabets**

Trabalho solicitado como forma de avaliação para obtenção de nota parcial para os alunos de Sistemas de Informação, na disciplina de Algoritmos e Estrutura de Dados II, ministrada pelo professor Sergio Cleger Tamayo.

**Manaus, AM
2021**

1. Introdução

O objetivo deste artigo é descrever

2. Algoritmo de Galil-Seiferas

2.1. Definição

Galil-Seiferas, ou simplesmente GS, é um algoritmo de correspondência de string com espaço constante e tempo linear para alfabetos não-ordenáveis. Foi criado por Zvi Galil e Joel Seiferas.

Não tão distante de outros algoritmos de correspondência, o algoritmo GS possui duas fases: pré-processamento e busca. A partir desse momento, *match* irá se referir a *string* que está sendo buscada dentro de um texto e *source* vai se referir ao texto onde a *string match* está sendo buscada.

A fase de pré-processamento consiste em basicamente achar uma decomposição uv de tal forma que u tenha pelo menos um período do prefixo e $|u| = O(per(v))$. Tal decomposição é chamada de fatoração perfeita.

No que diz respeito à fase de busca, ela consiste em escanear a *string source* para cada ocorrência de v . Ao achar o v , ele verifica se o pedaço da decomposição u de uv está logo atrás na *string source*.

2.2. Complexidade

Durante a fase de pré-processamento, ele possui complexidade de tempo $O(m)$, sendo m igual ao tamanho da string *source*, e complexidade de espaço constante. Já durante a fase de busca, o algoritmo possui uma complexidade de tempo $O(n)$, sendo n análogo ao tamanho da string *match*.

2.3. Como e onde usar

Seu principal caso de uso é quando não é possível ordenar alfabeticamente os elementos do texto.

2.4. Implementação em C

Na implementação em C abaixo, a fase de pré-processamento é descrita pelas funções *newP1*, *newP2* e *parse*, enquanto a fase de busca é realizada pela função *search*.

Além disso, para realizar a tarefa são utilizadas variáveis globais como *x* e *y*, que representam, respectivamente, *match* e *source*. Já as variáveis *n* e *m* representam o tamanho de *match* e *source*, ou seja, a quantidade de caracteres em cada *string*. Já as variáveis posteriores são utilizadas para determinar o início e fim de cada parte da decomposição *uv*, definida durante a fase de pré-processamento, e quantidade de caracteres de cada componente da decomposição.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *x, *y;
int k, m, n, p, p1, p2, q, q1, q2, s;

void search() {
    while (p <= n - m) {
        while (p + s + q < n && x[s + q] == y[p + s + q])
            ++q;

        if (q == m - s && memcmp(x, y + p, s + 1) == 0)
            printf("%d\n", p);

        if (q == p1 + q1) {
            p += p1;
            q -= p1;
        }
        else {
            p += (q/k + 1);
            q = 0;
        }
    }
}
```

```

void parse() {
    while (1) {
        while (x[s + q1] == x[s + p1 + q1])
            ++q1;

        while (p1 + q1 >= k*p1) {
            s += p1;
            q1 -= p1;
        }

        p1 += (q1/k + 1);
        q1 = 0;

        if (p1 >= p2) break;
    }
    newP1();
}

```

```

void newP2() {
    while (x[s + q2] == x[s + p2 + q2] && p2 + q2 < k*p2)
        ++q2;

    if (p2 + q2 == k*p2)
        parse();
    else
        if (s + p2 + q2 == m)
            search();
        else {
            if (q2 == p1 + q1) {
                p2 += p1;
                q2 -= p1;
            }
            else {
                p2 += (q2/k + 1);
                q2 = 0;
            }
            newP2();
        }
}

```

```
}
```

```
void newP1() {  
    while (x[s + q1] == x[s + p1 + q1])  
        ++q1;  
  
    if (p1 + q1 >= k*p1) {  
        p2 = q1;  
        q2 = 0;  
        newP2();  
    }  
    else {  
        if (s + p1 + q1 == m)  
            search();  
        else {  
            p1 += (q1/k + 1);  
            q1 = 0;  
            newP1();  
        }  
    }  
}
```

```
void GS(char *argX, int argM, char *argY, int argN) {  
    x = argX;  
    m = argM;  
    y = argY;  
    n = argN;  
    k = 4;  
    p = q = s = q1 = p2 = q2 = 0;  
    p1 = 1;  
    newP1();  
}
```

```
int main() {  
    GS("GCAGAGAG", 8, "GCATCGCAGAGAGTATACAGTACG", 24);  
    return 0;  
}
```

2.5. Questão Resolvida - URI - 2651

A questão do URI Online Judge escolhida para ser resolvida com o algoritmo Galil-Seiferas foi a questão de número 2651 que leva o título de “Link Bolado”.

O enunciado da questão é descrito da seguinte forma: “Link é um herói famoso e por isso recebe diversas cartas de seus fãs. Porém mesmo sendo famoso, todos continuam o chamando de Zelda. Por causa disso Link está muito bolado, tão bolado que sempre que recebe uma carta ele confere como o seu fã se referiu a ele na carta, e caso ele perceba o trecho “zelda” no nome ele fica bolado e joga a carta fora. Sua tarefa é determinar se Link ficará bolado com a forma que seu fã o chamou na carta ou não.”

A entrada é composta por uma *string* S. Essa *string* é composta por letras maiúsculas e minúsculas.

Após o processamento, a saída precisa imprimir a mensagem “Link Bolado” caso a *string* S contenha o trecho “zelda”. Caso contrário, apenas imprimir “Link Tranquilo”.

Para resolver o problema, apenas adicionou-se uma variável global do tipo inteira chamada *found*, que representa um valor lógico indicando se o trecho “zelda” foi achado dentro de S ou não.

PROBLEMA:	2651 - Link Bolado
RESPOSTA:	Accepted
LINGUAGEM:	C++17 (g++ 7.3.0, -std=c++17 -O2 -lm) [+0s]
TEMPO:	0.000s

3. Optimal Mismatch

3.1. Descrição

O algoritmo foi desenvolvido por Daniel M. Sunday e publicado no artigo *A very fast substring search algorithm* em 1990. É um algoritmo de correspondência de strings que compara os caracteres mais raros primeiro. Quando um caractere não corresponde, o próximo caractere no texto além da string de pesquisa determina onde a próxima correspondência possível começa.

Sunday projetou um algoritmo onde os caracteres do padrão são escaneados do menos frequente ao mais frequente. Fazendo isso, pode-se esperar que haja uma incompatibilidade na maioria das vezes e, assim, digitalizar todo o texto muito rapidamente. Além disso, é preciso saber as frequências de cada um dos caracteres do alfabeto. O algoritmo é também uma variante do algoritmo Quick Search;

3.2. Complexidade

A fase de pré-processamento possui complexidade de $O(m^2 + \sigma)$ em tempo e $O(m + \sigma)$ em espaço, enquanto a fase de busca possui complexidade de $O(mn)$ em tempo, em que m é o tamanho da string a ser buscada, n o tamanho do texto a ser percorrido, e σ o tamanho do alfabeto;

3.3. Exemplo gráfico

A figura abaixo ilustra a fase de pré-processamento do Optimal Mismatch. Nessa fase, ocorre a classificação dos caracteres padrão em ordem decrescente de suas frequências e é construída as tabelas de bad character da Quick Search e de good suffix adaptada à ordem de varredura dos caracteres do padrão. Considera-se o padrão sendo analisado a string GCAGAGAG.

<i>c</i>	A	C	G	T
<i>freq[c]</i>	8	5	7	4
<i>qsBc[c]</i>	2	7	1	9

<i>i</i>	0	1	2	3	4	5	6	7
<i>x[i]</i>	G	C	A	G	A	G	A	G
<i>pat[i].loc</i>	1	7	5	3	0	6	4	2
<i>pat[i].c</i>	C	G	G	G	G	A	A	A

<i>i</i>	0	1	2	3	4	5	6	7	8
<i>adaptedGs[i]</i>	1	3	4	2	7	7	7	7	7

A figura abaixo ilustra a fase de busca do Optimal Mismatch. Nessa fase, ao percorrer o texto as comparações com o padrão ocorrem sempre iniciando com os caracteres menos frequentes na string de busca. As tabelas de bad character e good suffix determinam os saltos que irão ocorrer no caso de erro na comparação. Considera-se o texto a ser analisado a string GCATCGCAGAGAGTATACAGTACG.

First attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
	1						2																
G	C	A	G	A	G	A	G																

Shift by: 3 (*adaptedGs*[1])

Second attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
				1		4		3		2													
				G	C	A	G	A	G	A	G												

Shift by: 2 (*qsBc*[A]=*adaptedGs*[3])

Third attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
					5	1	8	4	7	3	6	2											
					G	C	A	G	A	G	A	G											

Shift by: 9 (*qsBc*[T])

Fourth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
														1									
														G	C	A	G	A	G	A	G		

Shift by: 7 (*qsBc*[C])

Neste exemplo, o Optimal Mismatch realiza 15 comparações de caracteres.

3.4. Código comentado em C

```
#include <stdio.h>
#include <string.h>
#define ASIZE 100000
#define XSIZE 100000

typedef struct patternScanOrder {
    int loc;
    char c;
} pattern;

int freq[ASIZE];

// Constrói a tabela de troca de bad character
```

```

void preQsBc(char *x, int m, int qsBc[]) {
    int i;

    for (i = 0; i < ASIZE; ++i) qsBc[i] = m + 1;
    for (i = 0; i < m; ++i) qsBc[x[i]] = m - i;
}

/* Constrói um pattern ordenado a partir de uma string. */
void orderPattern(char *x, int m, int (*pcmp)(), pattern *pat) {
    int i;

    for (i = 0; i <= m; ++i) {
        pat[i].loc = i;
        pat[i].c = x[i];
    }
    qsort(pat, m, sizeof(pattern), pcmp);
}

/* função de comparação de patterns do Optimal Mismatch. */
int optimalPcmp(pattern *pat1, pattern *pat2) {
    float fx;

    fx = freq[pat1->c] - freq[pat2->c];
    return (fx ? (fx > 0 ? 1 : -1) : (pat2->loc - pat1->loc));
}

/* Encontrar o próximo deslocamento para a esquerda
   para os primeiros elementos do padrão ploc após
   uma troca atual(shift) ou lshift */
int matchShift(char *x, int m, int ploc, int lshift, pattern *pat)
{
    int i, j;

    for (; lshift < m; ++lshift) {
        i = ploc;
        while (--i >= 0) {
            if ((j = (pat[i].loc - lshift)) < 0) continue;
            if (pat[i].c != x[j]) break;
        }
        if (i < 0) break;
    }
}

```

```

    return (lshift);
}

/* Constrói a tabela de troca de bons sufixos
   a partir de uma string ordenada. */
void preAdaptedGs(char *x, int m, int adaptedGs[], pattern *pat) {
    int lshift, i, ploc;

    adaptedGs[0] = lshift = 1;
    for (ploc = 1; ploc <= m; ++ploc) {
        lshift = matchShift(x, m, ploc, lshift, pat);
        adaptedGs[ploc] = lshift;
    }
    for (ploc = 0; ploc <= m; ++ploc) {
        lshift = adaptedGs[ploc];
        while (lshift < m) {
            i = pat[ploc].loc - lshift;
            if (i < 0 || pat[ploc].c != x[i]) break;
            ++lshift;
            lshift = matchShift(x, m, ploc, lshift, pat);
        }
        adaptedGs[ploc] = lshift;
    }
}

/* Algoritmo de correspondência de strings Optimal Mismatch. */
void OM(char *x, int m, char *y, int n) {
    int i, j, adaptedGs[XSIZE], qsBc[ASIZE];
    pattern pat[XSIZE];
    /* Preprocessing */
    orderPattern(x, m, optimalPcmp, pat);
    preQsBc(x, m, qsBc);
    preAdaptedGs(x, m, adaptedGs, pat);
    /* Searching */
    j = 0;
    while (j <= n - m) {
        i = 0;
        while (i < m && pat[i].c == y[j + pat[i].loc]) ++i;
        if (i >= m) printf("%d\n", j);
        int max = adaptedGs[i] > qsBc[y[j + m]] ? adaptedGs[i] :
qsBc[y[j + m]];

```

```

        j += max;
    }
}

int main() {
    char *source = "GCATCGCAGAGAGTATACAGTACG";
    char *pattern = "GCAGAGAG";
    OM(pattern, 8, source, 24);
    return 0;
}

```

3.5. Questão resolvida

A questão do URI Online Judge escolhida com o algoritmo Optimal Mismatch foi a questão 3300, com título “Números Má Sorte Recarregados”.

Segundo o enunciado: “Um número número 3 é de má sorte se contém um 1 seguido por um 3 entre seus dígitos. Por exemplo, o número 341329 é de má sorte, enquanto o número 26771 não é. Dado um inteiro N, seu programa terá que determinar se N é azarado ou não.”

A entrada consiste em um número positivo N ($0 \leq N \leq 10^{100}$).

Na saída deve ser impressa a mensagem "N es de Mala Suerte" se N é de má sorte, caso contrário imprima "N NO es de Mala Suerte".

Para resolver o problema foi preciso apenas acrescentar uma variável do tipo *boolean* ao código, de modo que ela recebe o valor *true* quando o padrão de busca “13” fosse encontrado na string. A figura abaixo evidencia a aprovação do algoritmo na resolução do problema no URI.

SUBMISSÃO # 23684038	
PROBLEMA:	3300 - Números Má Sorte Recarregados
RESPOSTA:	Accepted
LINGUAGEM:	C++17 (g++ 7.3.0, -std=c++17 -O2 -lm) [+0s]
TEMPO:	0.000s
TAMANHO:	3,29 KB
MEMÓRIA:	-
CODE GOLF:	0 caracteres (+0 que a mediana)
SUBMISSÃO:	20/07/2021 22:32:36

4. Two-way

4.1. Descrição

O algoritmo foi criado por Maxime Crochemore e Dominique Perrin em 1991. O Two-Way, Um algoritmo de correspondência de string que particiona o padrão x em dois, esquerda x_L e direita x_R para otimizar a busca.

Em seguida, compara x_R da esquerda para a direita. Se o padrão casar, compara x_L da direita para a esquerda. A fase de pré-processamento do algoritmo consiste em escolher uma boa fatorização $x_L x_R$, e o algoritmo requer também um alfabeto ordenado.

Alguns conceitos devem ser definidos para melhor entendimento do algoritmo:

Fatorização: uma string é considerada fatorada quando é dividida em duas metades. Suponha que uma string x seja dividida em duas partes (u, v) , então (u, v) é chamada de fatoração de x .

Período: um período p para uma string x é definido como um valor tal que para qualquer inteiro $0 < i \leq |x| - p$, $x[i] = x[i + p]$. Em outras palavras, " p é um período de x se duas letras de x na distância p sempre coincidem". O período mínimo de x é um número inteiro positivo denotado como $p(x)$.

Uma **repetição** w em (u, v) é uma substring de x em que:

- w é um sufixo de u ou u é um sufixo de w ;
- w é um prefixo de v ou v é um prefixo de w ;

Em outras palavras, w ocorre em ambos os lados do corte com um possível overflow em qualquer um dos lados. Cada fatoração (u, v) de x tem pelo menos uma repetição. Pode-se ver facilmente que $1 \leq r(u, v) \leq |x|$.

Período local: é o tamanho de uma repetição em (u, v) . O menor período local em (u, v) é denotado como $r(u, v)$. Para qualquer fatoração, $0 < r(u, v) \leq |x|$

Uma fatoração (u, v) de x tal que $r(u, v) = p(x)$ é chamada de **fatoração crítica** de x .

O algoritmo Two Way escolhe a fatoração crítica (x_L, x_R) tal que $|x_L| < p(x)$ e $|x_L|$ é mínimo.

4.2. Complexidade

A fase de pré-processamento possui complexidade de $O(m)$ em tempo e complexidade linear em espaço, enquanto a fase de busca possui complexidade de $O(n)$ em tempo, considerando que m é o tamanho da string a ser buscada, n o tamanho do texto a ser percorrido, e σ o tamanho do alfabeto. O algoritmo realiza $2n-m$ comparações de caracteres no pior caso.

4.3. Aplicação real

Ele é o algoritmo selecionado da *glibc*(biblioteca padrão do C do projeto GNU) e *musl*(biblioteca padrão C destinada a sistemas operacionais baseados no kernel Linux) para as famílias de funções de substring *memmem* e *strstr*.

4.4. Exemplo gráfico

A figura abaixo ilustra a fase de pré-processamento do Two-Way. Nessa fase, ocorre consiste em calcular fatorização $x_L x_R$ que melhor favorece as comparações. Considera-se o padrão sendo analisado a string GCAGAGAG.

x		G	C	A	G	A	G	A	G
local period	1	3	7	7	2	2	2	2	1
$x_\ell =$	GC								
$x_r =$					AG	AG	AG	AG	

A figura abaixo ilustra a fase de busca do Two-Way. Esta fase consiste em comparar os caracteres de x_R da esquerda para a direita e em seguida, se não ocorrer uma não-correspondência nessa primeira etapa, comparar os caracteres de x_L da direita para a esquerda. Considera-se o texto a ser analisado a string GCATCGCAGAGAGTATACAGTACG.

Fifth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
					7	8	1	2	3	4	5	6											
					G	C	A	G	A	G	A	G											

Shift by: 7

Sixth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
														1	2								
														G	C	A	G	A	G	A	G		

Shift by: 2

Seventh attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G	
																	1	2						
																	G	C	A	G	A	G	A	G

Shift by: 2

Eighth attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G	
																			1	2	3			
																			G	C	A	G	A	G

Shift by: 3

Neste exemplo o algoritmo Two-Way realiza 20 comparações de caracteres.

4.5. Código comentado em C

```
#include <stdio.h>
#include <string.h>

/* Cálculo do sufixo máximo para <= */
int maxSuf(char *x, int m, int *p) {
    int ms, j, k;
    char a, b;

    ms = -1;
    j = 0;
    k = *p = 1;
    while (j + k < m) {
        a = x[j + k];
```

```

    b = x[ms + k];
    if (a < b) {
        j += k;
        k = 1;
        *p = j - ms;
    } else if (a == b)
        if (k != *p)
            ++k;
        else {
            j += *p;
            k = 1;
        }
    else { /* a > b */
        ms = j;
        j = ms + 1;
        k = *p = 1;
    }
}
return (ms);
}

```

```

/* Cálculo do sufixo máximo para >= */
int maxSufTilde(char *x, int m, int *p) {
    int ms, j, k;
    char a, b;

    ms = -1;
    j = 0;
    k = *p = 1;
    while (j + k < m) {
        a = x[j + k];
        b = x[ms + k];
        if (a > b) {
            j += k;
            k = 1;
            *p = j - ms;
        } else if (a == b)
            if (k != *p)
                ++k;
            else {
                j += *p;

```



```

        k = 1;
    }
    else { /* a < b */
        ms = j;
        j = ms + 1;
        k = *p = 1;
    }
}
return (ms);
}

```

/* Algoritmo de correspondência de strings Two Way. */

```

void TW(char *x, int m, char *y, int n) {
    int i, j, ell, memory, p, per, q;

    /* Pré-processamento */
    i = maxSuf(x, m, &p);
    j = maxSufTilde(x, m, &q);
    if (i > j) {
        ell = i;
        per = p;
    } else {
        ell = j;
        per = q;
    }

    /* Buscando */
    if (memcmp(x, x + per, ell + 1) == 0) {
        j = 0;
        memory = -1;
        while (j <= n - m) {
            int max = ell > memory ? ell : memory;
            i = max + 1;
            while (i < m && x[i] == y[i + j]) ++i;
            if (i >= m) {
                i = ell;
                while (i > memory && x[i] == y[i + j]) --i;
                if (i <= memory) printf("%d\n", j);
                j += per;
                memory = m - per - 1;
            } else {

```

```

        j += (i - ell);
        memory = -1;
    }
}
} else {
    int max = ell + 1 > m - ell - 1 ? ell + 1 : m - ell - 1;
    per = max + 1;
    j = 0;
    while (j <= n - m) {
        i = ell + 1;
        while (i < m && x[i] == y[i + j]) ++i;
        if (i >= m) {
            i = ell;
            while (i >= 0 && x[i] == y[i + j]) --i;
            if (i < 0) printf("%d\n", j);
            j += per;
        } else
            j += (i - ell);
    }
}
}

int main() {
    char *source = "GCATCGCAGAGAGTATACAGTACG";
    char *pattern = "GCAGAGAG";
    TW(pattern, 8, source, 24);
    return 0;
}

```

4.6. Questão resolvida

A questão do URI Online Judge escolhida com o algoritmo Optimal Mismatch foi a questão 2356, com título “Bactéria I”.

A entrada consiste em vários casos de teste. Cada caso de teste contém duas strings, D e S, cada qual em uma linha, e representam o DNA da bactéria e a sequência de código genético que leva à resistência. $1 \leq |D|, |S| \leq 100$. As strings são compostas apenas pelos caracteres: A, C, G, T.

Na saída deve ser uma linha por cada caso teste, contendo a string "Resistente" (sem aspas) caso a bactéria possua o código genético requerido em seu DNA, ou a string "Nao resistente" (sem aspas) caso contrário.

Para resolver o problema foi preciso apenas acrescentar uma variável do tipo *boolean* ao código, de modo que ela recebe o valor *true* quando o padrão de busca (subsequência de DNA que leva à resistência) fosse encontrado na string (sequência de DNA da bactéria). A figura abaixo evidencia a aprovação do algoritmo na resolução do problema no URI.

SUBMISSÃO # 23684798	
PROBLEMA:	2356 - Bactéria I
RESPOSTA:	Accepted
LINGUAGEM:	C++17 (g++ 7.3.0, -std=c++17 -O2 -lm) [+0s]
TEMPO:	0.000s
TAMANHO:	2,57 KB
MEMÓRIA:	-
SUBMISSÃO:	21/07/2021 00:32:53

5. String Matching On Ordered Alphabets

5.1. Descrição

É um algoritmo de correspondência de string com tempo linear e espaço constante que explora a ordem do alfabeto;

Durante uma tentativa em que o index está posicionado no fator de texto y [$j \dots j + m - 1$], quando um prefixo u de x foi correspondido e ocorre uma incompatibilidade entre os caracteres a em x e b em y (veja na figura), o algoritmo tenta calcular o período de ub , se não conseguir encontrar o período exato, ele calcula uma aproximação dele.



O algoritmo calcula o sufixo máximo do prefixo correspondente do padrão anexado ao caractere incompatível do texto após cada tentativa. Ele evita computá-lo do zero após uma mudança de comprimento ter sido executada.

A correspondência de strings em alfabetos ordenados não precisa de fase de pré-processamento.

5.2 Complexidade

O algoritmo não precisa de uma fase de pré-processamento, porém ele precisa de uma lista/array com as letras do alfabeto em ordem para funcionar, fora isso, a fase de busca pode ser feita em complexidade de tempo $O(n)$ usando um espaço extra constante. O algoritmo não realiza mais do que $6n + 5$ comparações de caracteres de texto.

5.3 Exemplo gráfico



5.4 Código em C

```
/* Calcula o próximo sufixo máximo. */  
  
void nextMaximalSuffix(char *x, int m,  
                       int *i, int *j, int *k, int *p) {  
  
    char a, b;  
  
    while (*j + *k < m) {  
        a = x[*i + *k];  
        b = x[*j + *k];  
  
        if (a == b)  
            if (*k == *p) {  
                (*j) += *p;  
                *k = 1;  
            }  
            else  
                ++(*k);  
        else  
            if (a > b) {  
                (*j) += *k;  
                *k = 1;  
                *p = *j - *i;  
            }  
            else {  
                *i = *j;  
                ++(*j);  
            }  
    }  
}
```

```

        *k = *p = 1;
    }

}

}

/* Correspondência de strings no algoritmo de alfabetos ordenados.
*/

void SMOA(char *x, int m, char *y, int n) {
    int i, ip, j, jp, k, p;

    ip = -1;

    i = j = jp = 0;

    k = p = 1;

    while (j <= n - m) {
        while (i + j < n && i < m && x[i] == y[i + j])
            ++i;

        if (i == 0) {
            ++j;

            ip = -1;

            jp = 0;

            k = p = 1;
        }

        else {
            if (i >= m)

                OUTPUT(j);

            nextMaximalSuffix(y + j, i+1, &ip, &jp, &k, &p);
        }
    }
}

```

```

if (ip < 0 ||
    (ip < p &&
     memcmp(y + j, y + j + p, ip + 1) == 0)) {
    j += p;
    i -= p;
    if (i < 0)
        i = 0;
    if (jp - ip > p)
        jp -= p;
    else {
        ip = -1;
        jp = 0;
        k = p = 1;
    }
}

else {
    j += (MAX(ip + 1, MIN(i - ip - 1, jp + 1)) + 1);
    i = jp = 0;
    ip = -1;
    k = p = 1;
}

}

}

```

5.5 Questão Resolvida: Questão do URI - 1241 - Encaixa ou Não II

Paulinho tem em suas mãos um novo problema. Agora a sua professora lhe pediu que construísse um programa para verificar, a partir de dois valores muito grandes, A e B, se B corresponde aos últimos dígitos de A.

A entrada consiste de vários casos de teste. A primeira linha de entrada contém um inteiro **N** que indica a quantidade de casos de teste. Cada caso de teste consiste de dois valores **A** e **B** maiores que zero, cada um deles podendo ter até 1000 dígitos. Para cada caso de entrada imprima uma mensagem indicando se o segundo valor encaixa no primeiro valor, conforme exemplo apresentado.

PROBLEMA:	1241 - Encaixa ou Não II
RESPOSTA:	Accepted
LINGUAGEM:	C++17 (g++ 7.3.0, -std=c++17 -O2 -lm) [+0s]
TEMPO:	0.044s
TAMANHO:	2,22 KB
MEMÓRIA:	-
SUBMISSÃO:	21/07/2021 01:05:36

6. Referências bibliográficas

GALIL Z., SEIFERAS J., 1983, Time-space optimal string matching, *Journal of Computer and System Science* 26(3):280-294.

SUNDAY M., 1990. A very fast substring search algorithm. *Commun. ACM* 33, 8 (Aug. 1990), 132–142. DOI:<https://doi.org/10.1145/79173.79184>.

CROCHEMORE M., PERRIN D., 1991, Two-way string-matching, *Journal of the ACM* 38(3):651-675.

CROCHEMORE M., PERRIN D., 1992, String Matching on Ordered Alphabet, *Theoretical Computer Science, Journal of the ACM* 38(3):651-675.