



UNIVERSIDADE FEDERAL DE VIÇOSA - UFV - CAMPUS FLORESTAL

## **Trabalho Prático 3**

### **Projeto e Análise de Algoritmos**

#### **Algoritmos para a manipulação de caracteres**

Aymê Faustino dos Santos - 4704

Matheus Nascimento Peixoto - 4662

Matheus Nogueira Moreira - 4668

Florestal - MG

2023

# Sumário

<b>1) Introdução</b>	<b>3</b>
<b>2) Desenvolvimento</b>	<b>3</b>
2.1) Algoritmo Força Bruta	3
2.2) Algoritmo Knuth-Morris-Pratt (KMP)	4
2.3) Arquivo 'auxiliares' e 'menu'	5
2.4) Cifra de Deslocamento	6
2.4.1) Organização dos arquivos	6
2.4.2) Função Menu	7
2.4.3) Função Criptografar	7
2.4.4) Função Descriptografar	8
2.4.5) Exibir Frequências	9
2.4.6) Encontrar Chave Aleatória	10
<b>3) Comandos para a execução</b>	<b>11</b>
<b>4) Resultados</b>	<b>11</b>
4.1 Tarefa A	12
4.2 Tarefa B	15
<b>5) Conclusão</b>	<b>17</b>
<b>6) Referências</b>	<b>17</b>

## 1) Introdução

Esta documentação refere-se ao Trabalho Prático Final da disciplina de Projeto e Análise de Algoritmos. O projeto concentra-se na aplicação prática de algoritmos de manipulação de caracteres, destacando o Casamento Exato Força Bruta, o algoritmo de Knuth-Morris-Pratt (KMP) e a Cifra de Deslocamento. O Casamento Exato Força Bruta busca identificar todas as ocorrências exatas de um padrão em um texto, percorrendo-o caracter por caracter. O Algoritmo de Knuth-Morris-Pratt (KMP) destaca-se pela eficiência na localização exata de padrões em textos, utilizando uma tabela de prefixo para otimizar as comparações. A Cifra de Deslocamento é uma técnica clássica de criptografia, aqui explorada na manipulação de caracteres ao deslocar cada caractere do texto por um número fixo de posições no alfabeto. Esta documentação oferece uma análise da implementação de cada parte do trabalho, visando a compreensão prática dos algoritmos discutidos ao longo do curso.

## 2) Desenvolvimento

A seguir, neste capítulo, serão apresentados os algoritmos utilizados para o desenvolvimento do trabalho, explicando suas principais características.

### 2.1) Algoritmo Força Bruta

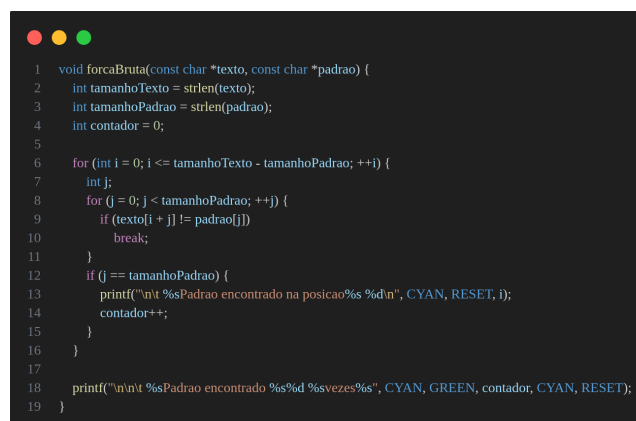
Implementando o método de Força Bruta para o problema do Casamento de Caracteres, a função `forcaBruta()`, encontrada no arquivo de mesmo nome, recebe por parâmetro o texto e o padrão que será procurado.

A seguir, descobre o tamanho de cada um, a fim de utilizar esse valor para o laço de repetição `for` criado a seguir.

Este é usado para comparar cada caractere do padrão com os caracteres correspondentes na string texto a partir da posição atual `i`. Caso haja diferença, em qualquer posição, o loop é interrompido.

Para saber se o objetivo foi alcançado, isto é, a descoberta do padrão buscado no texto, é verificado o valor de uma variável `j`, verificando se é igual ao tamanho do padrão passado. Caso isso seja verdade, significa que uma ocorrência foi obtida e esta é apresentada ao usuário.

Em sequência, na **Figura 1**, é possível observar a função `forcaBruta()`.



```
1 void forcaBruta(const char *texto, const char *padrao) {
2     int tamanhoTexto = strlen(texto);
3     int tamanhoPadrao = strlen(padrao);
4     int contador = 0;
5
6     for (int i = 0; i <= tamanhoTexto - tamanhoPadrao; ++i) {
7         int j;
8         for (j = 0; j < tamanhoPadrao; ++j) {
9             if (texto[i + j] != padrao[j])
10                break;
11        }
12        if (j == tamanhoPadrao) {
13            printf("\n\t %sPadrao encontrado na posicao %s %d\n", CYAN, RESET, i);
14            contador++;
15        }
16    }
17
18    printf("\n\t %sPadrao encontrado %s %d %s vezes %s", CYAN, GREEN, contador, CYAN, RESET);
19 }
```

**Figura 1: Função para o método de Força Bruta**

## 2.2) Algoritmo Knuth-Morris-Pratt (KMP)

O algoritmo KMP é uma técnica eficiente para buscar todas as ocorrências de um padrão em um texto, evitando comparações desnecessárias por meio do uso do array LPS (Longest Prefix Suffix). A implementação visa oferecer uma solução eficaz para essa tarefa de processamento de strings.

A implementação utiliza uma estrutura básica de dados, composta principalmente por arrays e variáveis de controle. A utilização de índices  $i$  e  $j$  controla o progresso na busca pelo texto, enquanto as variáveis  $len$  e  $M$  são empregadas no preenchimento do array LPS.

A função *preencherLPS* é uma parte fundamental da implementação do algoritmo Knuth-Morris-Pratt (KMP) para casamento exato de padrões em strings. Seu propósito é calcular o array LPS (Longest Prefix Suffix), que armazena informações sobre o maior sufixo que também é um prefixo para cada posição no padrão. Essa informação é crucial para otimizar a busca subsequente do padrão no texto, reduzindo comparações desnecessárias.

A função inicia definindo uma variável  $len$  que representa o comprimento do prefixo atualmente igual ao sufixo, e o valor inicial de  $lps[0]$  é configurado como zero, representando o caso base. Em seguida, um loop `while` é empregado para percorrer o padrão, começando da segunda posição ( $i = 1$ ). Dentro desse loop, ocorrem duas condições principais.

Se o caractere na posição  $i$  do padrão for igual ao caractere na posição  $len$ , a função incrementa  $len$ , atribui esse valor a  $lps[i]$ , e avança para a próxima posição no padrão. Essa etapa é fundamental para reconhecer padrões de repetição e construir o array LPS.

Caso os caracteres em  $padrao[i]$  e  $padrao[len]$  não sejam iguais, a função verifica se  $len$  é diferente de zero. Se for verdadeiro,  $len$  é ajustado para  $lps[len - 1]$ , retrocedendo no array LPS para encontrar o maior sufixo que também é um prefixo. Se  $len$  for zero, significa que não há um sufixo comum até a posição  $i$ , e  $lps[i]$  é configurado como zero. O loop continua até que todas as posições no padrão sejam consideradas.

A função principal *buscarKMP* emprega os resultados do preenchimento do array LPS para realizar uma busca eficiente em um texto. Inicialmente, a função determina os comprimentos do padrão ( $M$ ) e do texto ( $N$ ), e cria um array  $lps$  utilizando a função *preencherLPS* para calcular os valores do Longest Prefix Suffix.

Dentro do loop principal, a função percorre o texto, utilizando dois índices:  $i$ , que avança pelo texto, e  $j$ , que percorre o padrão. A cada iteração, há comparações entre caracteres do padrão e do texto. Se uma correspondência é encontrada, ambos os índices são incrementados.

Quando uma correspondência completa é identificada ( $j$  atinge o comprimento do padrão  $M$ ), a função imprime a posição da ocorrência no texto. Em seguida,  $j$  é ajustado utilizando o array LPS, retrocedendo para a posição adequada. No caso de não haver uma correspondência ( $j$  não atinge  $M$ ), a função utiliza o array LPS para determinar se deve retroceder  $j$  ou avançar  $i$ . Essa abordagem eficiente evita comparações desnecessárias, otimizando o processo de busca.

Na sequência é possível observar, na **Figura 2**, a função principal para o método de Knuth-Morris-Pratt.

```

1 void buscarKMP(char *texto, char *padrao) {
2     int M = strlen(padrao);
3     int N = strlen(texto);
4     int contador = 0;
5
6     // Criar e preencher o array LPS
7     int lps[M];
8     preencherLPS(padrao, M, lps);
9
10    int i = 0; // Índice para o texto
11    int j = 0; // Índice para o padrão
12
13    while (i < N) {
14        if (padrao[j] == texto[i]) {
15            j++;
16            i++;
17        }
18
19        if (j == M) {
20            printf("\n\t %sPadrao encontrado na posicao%s %d\n", MAGENTA, RESET, i - j);
21            contador++;
22            j = lps[j - 1];
23        } else if (i < N && padrao[j] != texto[i]) {
24            if (j != 0) {
25                j = lps[j - 1];
26            } else {
27                i++;
28            }
29        }
30    }
31
32    printf("\n\n\t %sPadrao encontrado %s%d %svezes%s", MAGENTA, GREEN, contador, MAGENTA, RESET);
33 }

```

**Figura 2: Algoritmo para a implementação do método KMP**

### 2.3) Arquivo ‘auxiliares’ e ‘menu’

No primeiro arquivo, ‘auxiliares’, existem as funções que utilizam cada um dos dois algoritmos apresentados anteriormente e são utilizadas no menu que é apresentado ao usuário.

Em cada uma das funções, ‘realizarForcaBruta’ e ‘realizarKMP’, é recebido o nome de um arquivo de entrada bem como do padrão a ser buscado. O arquivo é aberto lido e todas as suas palavras passam por uma transformação para minúsculo, a fim de que o algoritmo não diferencie palavras que comecem ou estejam completamente com letras maiúsculas de minúsculas, através da função ‘textoMinusculo’, a qual utiliza a função ‘tolower’ da biblioteca ‘ctype.h’.

Já no arquivo ‘menu’ são apresentadas as opções ao usuário e as solicitações de entrada.

Inicialmente é apresentado ao usuário se este deseja seguir por um método de arquivo, o qual foi criado visando maior agilidade nos testes, utilizando o arquivo ‘ArquivoApresentação.txt’, que se encontra na pasta ‘Textos’, assim como todos os demais arquivos de texto utilizados, ou se deseja seguir um modo interativo.

Para a escolha do arquivo a ser utilizado é preciso que o usuário passe apenas o nome do arquivo, e que este esteja obrigatoriamente na pasta ‘Textos’.

A seguir, o padrão é solicitado, o qual também é convertido para minúsculo. Seguindo o algoritmo, o usuário será solicitado a escolher qual método deseja seguir, escolhendo entre o Força Bruta ou o KMP.

Será apresentado, a seguir, visando melhor ilustrar o que foi explicado nesse subcapítulo, a função `realizarForcaBruta`, na **Figura 3**. Esta e a `realizarKMP` se assemelham em muitos aspectos, por isso apenas uma foi utilizada para ilustração.

```
1 void realizarForcaBruta(const char *arqEntrada, char *padrao) {
2
3     FILE *arquivo = fopen(arqEntrada, "r");
4
5     if (arquivo == NULL) {
6         perror("\n\tErro ao abrir o arquivo");
7         return;
8     }
9
10    // Ler o conteúdo do arquivo
11    fseek(arquivo, 0, SEEK_END);
12    long tamanho = ftell(arquivo);
13    fseek(arquivo, 0, SEEK_SET);
14
15    char texto[tamanho + 1];
16    fread(texto, 1, tamanho, arquivo);
17    texto[tamanho] = '\0'; // Adicionar o terminador nulo
18
19    fclose(arquivo);
20
21    textoMinusculo(texto);
22
23    // Realizar a busca força bruta e medir o tempo
24    clock_t inicio = clock();
25    forcaBruta(texto, padrao);
26    clock_t fim = clock();
27    double tempo_execucao = ((double)(fim - inicio)) / CLOCKS_PER_SEC;
28
29    printf("\n\t %sTempo de execucao: %f segundos%s\n", CYAN, tempo_execucao, RESET);
30 }
```

**Figura 3: função auxiliar para a realização do método Força Bruta**

## 2.4) Cifra de Deslocamento

### 2.4.1) Organização dos arquivos

O algoritmo de cifra de deslocamento está organizado na pasta 'CifraDeslocamento' em poucos arquivos, todas as funções estão sendo declaradas em 'criptografar.h' na pasta 'Headers' e desenvolvidas em 'criptografar.c' na pasta 'src'. No arquivo .h ainda há a declaração de um TAD chamado de 'LetraFrequencia', o qual terá sua aplicação explicada nos próximos tópicos. Além disso, os arquivos de texto com possíveis arquivos de entradas e de saídas estão na pasta 'Arquivos'. Tudo isso está junto também com um MAKEFILE para a execução do programa e a main que começa a execução do programa. A seguir uma foto para exemplificar:

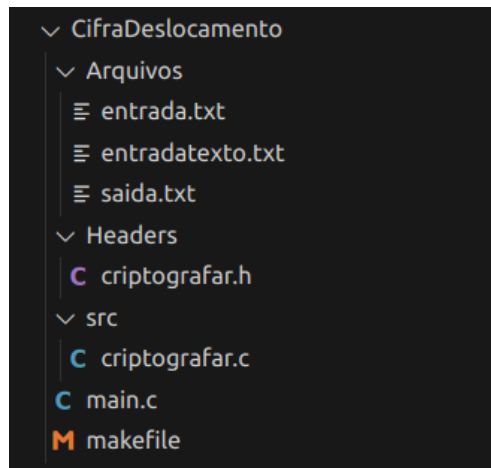


Figura 4: Organização do algoritmo cifra de deslocamento.


### 2.4.2) Função Menu

A execução inicia-se com a chamada do método 'Menu' do arquivo 'criptografar.c' na 'main.c'. A partir disso, a função é executada e o usuário é inicialmente solicitado a fornecer os nomes dos arquivos de entrada e saída. Após essa etapa, o usuário tem a opção de escolher entre utilizar uma chave definida por ele mesmo ou gerá-la aleatoriamente. Indiferente da escolha, uma nova entrada é solicitada, desta vez para determinar qual comando o usuário deseja executar: criptografar ou descriptografar. Se a opção do usuário for gerar uma chave aleatória, a chave receberá um valor que varia de 1 a 26 e ao final da execução, uma tabela contendo as frequências de cada caractere no arquivo criptografado é gerada. Além disso, o algoritmo realiza uma tentativa de 'adivinhar' a chave gerada, processo que será detalhado posteriormente. A chave "adivinhada" pelo algoritmo é exibida, juntamente com a verdadeira chave aleatória gerada anteriormente.

### 2.4.3) Função Criptografar

A função Criptografar implementa a cifra de deslocamento e recebe os seguintes parâmetros: FILE \*ArquivoEntrada: Ponteiro para o arquivo de entrada contendo o texto original. FILE \*ArquivoSaida: Ponteiro para o arquivo de saída onde o texto criptografado será escrito. int chave: Número inteiro que representa a quantidade de posições que cada letra será deslocada no alfabeto. Se a chave for positiva, o deslocamento é para a direita; se negativa, o deslocamento é para a esquerda.

A função lê caracteres do arquivo de entrada um por um até atingir o final do arquivo. Para cada caractere lido, a função verifica se é uma letra maiúscula ('A' a 'Z') ou minúscula ('a' a 'z'). Caracteres que não são letras não são modificados. Se o caractere é uma letra, a função realiza a criptografia, adicionando a chave ao valor da letra. O resultado é calculado sempre levando em consideração os limites do alfabeto, por isso o operador % 26 é usado para garantir que o deslocamento não ultrapasse os limites do alfabeto. O caractere criptografado é então escrito no arquivo de saída. Durante o processo, os caracteres criptografados também são exibidos no console usando putchar. E no final, a função 'Reiniciar Leitura' é chamada e tem um funcionamento simples, recebe os ponteiros para os arquivos de entrada e de saída e usa o 'fseek' para posicionar o ponteiro de leitura na posição inicial do arquivo para que os arquivos sejam lidos novamente, caso necessário, durante a execução. Aqui uma imagem da função em questão:



```


1 void Criptografar(FILE *ArquivoEntrada, FILE *ArquivoSaida, int chave) {
2     char caractere;
3     while ((caractere = fgetc(ArquivoEntrada)) != EOF) {
4         char caractereCriptografado;
5
6         if (caractere >= 'A' && caractere <= 'Z') {
7             caractereCriptografado = (caractere + chave - 'A' + 26) % 26 + 'A';
8         } else if (caractere >= 'a' && caractere <= 'z') {
9             caractereCriptografado = (caractere + chave - 'a' + 26) % 26 + 'a';
10        } else {
11            caractereCriptografado = caractere;
12        }
13
14        fputc(caractereCriptografado, ArquivoSaida);
15        putchar(caractereCriptografado);
16    }
17
18    ReiniciarLeitura(ArquivoEntrada, ArquivoSaida);
19 }

```

**Figura 5: Função `Criptografar`**

#### 2.4.4) Função Descriptografar

A função 'Descriptografar' tem um funcionamento muito semelhante à função 'Criptografar', com as únicas diferenças sendo que, caso receba uma chave positiva, os caracteres serão decrementados, e caso receba uma chave negativa, eles serão acrescentados. Aqui uma imagem da função 'Descriptografar':



```

1 void Descriptografar(FILE *ArquivoEntrada, FILE *ArquivoSaida, int chave) {
2     char caractere;
3     while ((caractere = fgetc(ArquivoEntrada)) != EOF) {
4         char caractereDescriptografado;
5
6         if (caractere >= 'A' && caractere <= 'Z') {
7             caractereDescriptografado = (caractere - chave - 'A' + 26) % 26 + 'A';
8         } else if (caractere >= 'a' && caractere <= 'z') {
9             caractereDescriptografado = (caractere - chave - 'a' + 26) % 26 + 'a';
10        } else {
11            caractereDescriptografado = caractere;
12        }
13
14        fputc(caractereDescriptografado, ArquivoSaida);
15        putchar(caractereDescriptografado);
16    }
17
18    ReiniciarLeitura(ArquivoEntrada, ArquivoSaida);
19 }

```

**Figura 6: Função `Descriptografar`**



### 2.4.5) Exibir Frequências

A função 'ExibirFrequencias()' é chamada em 'Menu()' ,após o usuário escolher a opção de gerar uma chave aleatória, e recebe como parâmetro um ponteiro para o arquivo de saída e o valor inteiro que pode variar de 1 a 2, que serão utilizados para encontrar a chave aleatória gerada anteriormente. Dentro da função 'ExibirFrequencias()' são declarados dois vetores de flutuantes , o primeiro é 'frequencias' que armazenará a frequência de cada letra no arquivo de saída,após o processo de criptografia ou descriptografia, e outro 'VetorFrequencias' que será utilizado como parâmetro para a função 'EncontrarChaveAleatoria()' que será explicado no próximo tópico. Ambos apresentam 26 posições ,representando uma letra do alfabeto(a→0,b'→1), e são inicializados com zero.Logo após isso, o arquivo de saída é lido e os caracteres são analisados um por um até o final do arquivo. Durante esse processo, a função verifica se o caractere é uma letra maiúscula ou minúscula do alfabeto. Se for uma letra, incrementa o contador total de caracteres e atualiza a contagem de frequência da letra correspondente no vetor 'frequencias'. Após a leitura do arquivo, a função imprime uma tabela de frequências, exibindo o caractere, sua frequência e o percentual em relação ao total de caracteres. Essa tabela fornece informações estatísticas sobre a distribuição das letras no texto após a operação de criptografia ou descriptografia.A última parte da função calcula o percentual de cada letra em relação ao total de caracteres e armazena esses percentuais normalizados no vetor 'VetorFrequencias'. Em seguida. é chamado a função 'EncontrarChaveAleatoria()' passando como parâmetro o vetor 'VetorFrequencias' e o valor de uma flag, que determina a abordagem específica para encontrar a chave aleatória.A seguir uma imagem com a tabela gerada após a execução da função 'ExibirFrequencias()' :

Tabela de Frequencias:		
Caractere	Frequencia	Percentual
A	0	0.00%
B	0	0.00%
C	0	0.00%
D	0	0.00%
E	6	18.75%
F	8	25.00%
G	0	0.00%
H	1	3.13%
I	1	3.13%
J	4	12.50%
K	0	0.00%
L	0	0.00%
M	0	0.00%
N	1	3.13%
O	1	3.13%
P	0	0.00%
Q	0	0.00%
R	0	0.00%
S	0	0.00%
T	3	9.38%
U	2	6.25%
V	0	0.00%
W	3	9.38%
X	0	0.00%
Y	2	6.25%
Z	0	0.00%

Figura 7: Tabela de Frequências

#### 2.4.6) Encontrar Chave Aleatória

A função ``EncontrarChaveAleatoria()`` é chamada ao final do método ``ExibirFrequencias()``, visando recuperar a chave aleatória previamente gerada. Essa busca é conduzida com base em dois vetores de frequências: ``VetorFrequencias``, que retém as frequências das letras após o processo de criptografia ou descriptografia, e ``VetorPesos``, que armazena as frequências esperadas conforme uma tabela especificada na documentação desse trabalho prático.

No início da função, são instanciados dois vetores de estruturas ``LetraFrequencia``, nomeados ``AUX01`` e ``AUX02``, cada um inicializado com as frequências correspondentes nos vetores ``VetorPesos`` e ``VetorFrequencias``, juntamente com os caracteres associados a cada frequência. Posteriormente, ambos os vetores são ordenados em ordem crescente de frequência por meio da função ``qsort``, utilizando o comparador ``compararFrequencias``.

A lógica do bloco condicional no interior da função é condicionada ao valor do parâmetro ``flag``, fornecido pelo usuário durante a execução da função ``Menu()``, recebendo 1 caso o usuário tenha escolhido criptografar o arquivo e 2 para descriptografar. Esse valor é transmitido inicialmente para ``ExibirFrequencias`` e, em seguida, passado como parâmetro para ``EncontrarChaveAleatoria()``. Se ``flag`` for igual a 1, a função calcula a diferença entre as maiores frequências presentes nos dois vetores. Caso ``flag`` seja 2, um loop é executado, começando pela letra com a maior frequência no arquivo criptografado até alcançar o maior valor presente no arquivo de pesos.

Contudo, se ``flag`` for diferente de 1 ou 2, a função atribui o valor 26 à variável ``MaiorChave``, indicando que, dado que os caracteres não sofreram alterações, logo, o único valor possível da chave para isso acontecer é 26.

Ao final, a função imprime a chave encontrada, representando a diferença entre as maiores frequências nos vetores ``VetorFrequencias`` e ``VetorPesos``, ou o número de iterações necessárias para atingir o mesmo caractere nas maiores frequências, conforme o valor de ``flag``.

A estrutura da função é esquematizada na imagem a seguir.

```

1 void EncontrarChaveAleatoria(double VetorFrequencias[26],int flag){
2     int MaiorChave = 0;
3     double VetorPesos[] = {
4         0.1463, 0.0104, 0.0388, 0.0499, 0.1257, 0.0102, 0.0130, 0.0128, 0.0618, 0.0040,
5         0.0002, 0.0278, 0.0474, 0.0505, 0.1073, 0.0252, 0.0120, 0.0653, 0.0781, 0.0434,
6         0.0463, 0.0167, 0.0001, 0.0021, 0.0001, 0.0047
7     };
8     LetraFrequencia *AUX01 = malloc(26 * sizeof(LetraFrequencia));
9     LetraFrequencia *AUX02 = malloc(26 * sizeof(LetraFrequencia));
10    for(int i = 0; i < 26; i++) {
11        AUX01[i].frequencia = VetorPesos[i];
12        AUX02[i].frequencia = VetorFrequencias[i];
13        AUX01[i].caractere = 'a' + i;
14        AUX02[i].caractere = 'a' + i;
15    }
16
17    qsort(AUX01, 26, sizeof(LetraFrequencia), compararFrequencias);
18    qsort(AUX02, 26, sizeof(LetraFrequencia), compararFrequencias);
19
20    if(flag == 1){
21        MaiorChave = AUX02[25].caractere - AUX01[25].caractere;
22    }else if(flag == 2){
23        while(AUX02[25].caractere!=AUX01[25].caractere){
24            MaiorChave +=1;
25            AUX02[25].caractere+=1;
26            if(AUX02[25].caractere>'z'){
27                AUX02[25].caractere = 'a';
28            }
29        }
30    }else{
31        MaiorChave = 26;
32    }
33    printf("Chave Chute: %d\n", MaiorChave);
34 }

```

**Figura 8: Função para adivinhar a chave.**

### 3) Comandos para a execução

Para a execução, basta estar na pasta “CasamentoExato” ou na pasta “CifraDeslocamento” e executar os seguintes comandos no terminal:

```
make all
```

e, a seguir, dependendo do Sistema Operacional utilizado,

```
make Linux
o
make Windows
```

### 4) Resultados

Para a execução da parte B, foram escolhidos textos que estivessem em inglês como arquivos de entrada, a fim de evitar possíveis contratempos envolvendo acentos e outros caracteres especiais, como `ç`. A seguir será exibido saídas para cada um dos algoritmos.

## 4.1 Tarefa A

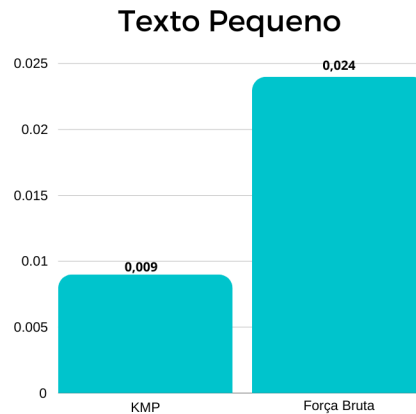
Para a análise da Tarefa A, usamos diferentes tamanhos de texto em português para testar os dois algoritmos, os quais podem ser encontrados no seguinte arquivo do [Google Drive](#) (basta clicar no texto destacado para ter acesso a esses arquivos) com os respectivos nomes: arquivo pequeno: “Pequeno.txt”; arquivo médio: “Medio.txt”; arquivo grande: “script-shrek.txt”.

Os resultados obtidos serão apresentados e debatidos a seguir:

Texto Pequeno, busca pela palavra "tecnologia"

KMP: 0,009 segundos

Força Bruta: 0,024 segundos

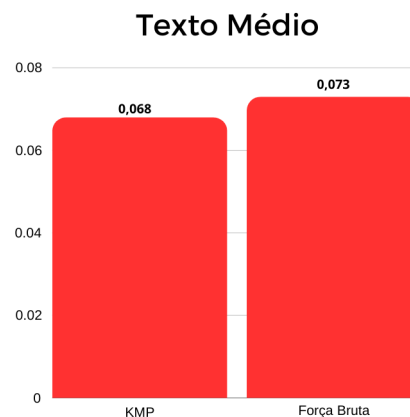


**Figura 9 - Resultados sobre o texto pequeno**

Texto Médio, busca pela palavra “filho”

KMP: 0,068 segundos

Força Bruta: 0,073 segundos

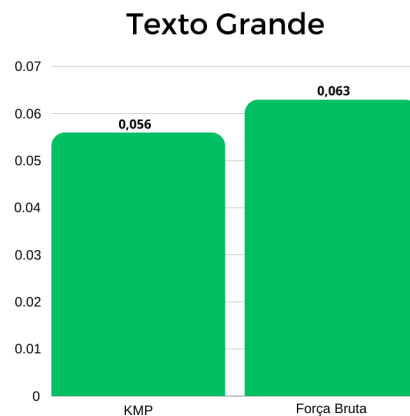


**Figura 10 - Resultados sobre o texto médio**

Texto Grande, busca pela palavra "amor" com 21 ocorrências

KMP: 0,056 segundos

Força Bruta: 0,063 segundos

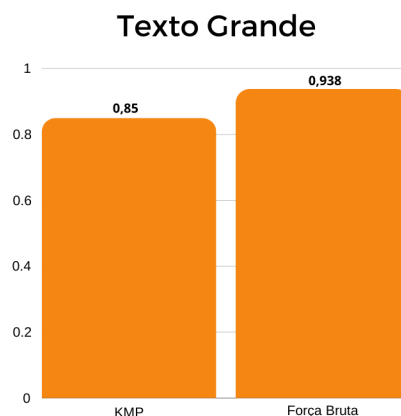


**Figura 11 - Resultados sobre o texto grande com a palavra amor**

Texto Grande, busca pela palavra "ele" com 252 ocorrências

KMP: 0,85 segundos

Força Bruta: 0,938 segundos



**Figura 12 - Resultados sobre o texto grande com a palavra ele**

Sobre as nossas análises em relação aos resultados obtidos:

#### **KMP:**

O algoritmo KMP demonstrou consistente eficiência em todos os cenários, corroborando com sua complexidade linear ( $O(n)$ ), onde 'n' é o tamanho do texto. Isso significa que o tempo de execução aumenta de maneira linear com o tamanho do texto.

Texto Pequeno (Busca por "tecnologia"): 0,009 segundos - Eficiência notável, destacando a agilidade do KMP mesmo em textos pequenos.

Texto Médio (Busca por "filho"): 0,015 segundos - A eficiência do KMP é mantida, evidenciando sua adaptabilidade a diferentes tamanhos de texto.

Texto Grande (Busca por "amor" com 21 ocorrências): 0,056 segundos - Mesmo em um contexto de múltiplas ocorrências, o KMP mantém uma performance eficiente.

Texto Grande (Busca por "ele" com 252 ocorrências): 0,85 segundos - Lida eficientemente com múltiplas ocorrências, mantendo-se ágil mesmo em textos grandes.

### Força Bruta:

O algoritmo de Força Bruta apresentou resultados mais variados, conforme sua complexidade quadrática ( $O(m * n)$ ), onde 'm' é o tamanho do padrão e 'n' é o tamanho do texto. Isso implica que o tempo de execução aumenta de forma quadrática com o aumento do tamanho do texto e do padrão.

Texto Pequeno (Busca por "tecnologia"): 0,024 segundos - Aceitável para textos pequenos, mas superior ao desempenho do KMP.

Texto Médio (Busca por "filho"): 0,073 segundos - Desempenho estável, mas ainda mais lento que o KMP.

Texto Grande (Busca por "amor" com 21 ocorrências): 0,063 segundos - Desempenho competitivo com o KMP, mas inferior.

Texto Grande (Busca por "ele" com 252 ocorrências): 0,938 segundos - Aceitável para múltiplas ocorrências, mas menos eficiente em textos grandes em comparação com o KMP.

Em resumo, as análises refletem a superioridade do KMP em termos de eficiência, especialmente em textos maiores e em casos com múltiplas ocorrências. E ao implementar algoritmos de casamento exato de padrões, é essencial considerar não apenas os tempos de execução, mas também as complexidades algorítmicas subjacentes. A complexidade linear do KMP se destaca como uma solução eficiente e escalável, enquanto a Força Bruta, embora simples, pode apresentar desafios em contextos mais exigentes. A escolha entre os dois depende da natureza específica dos dados.

A seguir, para ilustrar a execução dessa primeira parte do Trabalho Prático 03, serão apresentadas duas figuras, **14** e **15**, que representam, respectivamente, a execução do algoritmo Força Bruta e do KMP para um texto contendo o script do filme “Shrek” e buscando pelo padrão “cavaleiro”. Na **Figura 13** será possível observar o menu criado para a utilização dos algoritmos.

```
+-----+
| Insira qual método deseja seguir: |
| 1 - Arquivo                       |
| 2 - Interativo                    |
| 3 - Sair                          |
+-----+
2
Insira o nome do arquivo que será lido (precisa estar na pasta Textos):
shrek.txt
Insira o padrao que será pesquisado:
cavaleiro
+-----+
| Insira qual algoritmo será utilizado: |
| 1 - Forca Bruta                      |
| 2 - Knuth-Morris-Pratt               |
+-----+
```

**Figura 13: Menu para a “Tarefa A”**

```

Algoritmo de Força Bruta escolhido:

Padrao encontrado na posicao 254
Padrao encontrado na posicao 19685
Padrao encontrado na posicao 20571
Padrao encontrado na posicao 28497
Padrao encontrado na posicao 31661
Padrao encontrado na posicao 31771
Padrao encontrado na posicao 32939
Padrao encontrado na posicao 33272
Padrao encontrado na posicao 36917
Padrao encontrado na posicao 37135
Padrao encontrado na posicao 37614

Padrao encontrado 11 vezes
Tempo de execucao: 0.000721 segundos

Deseja realizar o processo novamente?
[1] - SIM || [2] - NAO

```

**Figura 14: Força Bruta**

```

Algoritmo de Knuth-Morris-Pratt escolhido:

Padrao encontrado na posicao 254
Padrao encontrado na posicao 19685
Padrao encontrado na posicao 20571
Padrao encontrado na posicao 28497
Padrao encontrado na posicao 31661
Padrao encontrado na posicao 31771
Padrao encontrado na posicao 32939
Padrao encontrado na posicao 33272
Padrao encontrado na posicao 36917
Padrao encontrado na posicao 37135
Padrao encontrado na posicao 37614

Padrao encontrado 11 vezes
Tempo de execucao: 0.001086 segundos

Deseja realizar o processo novamente?
[1] - SIM || [2] - NAO

```

**Figura 15: KMP**

Nesse caso de teste foi possível observar uma variação no padrão obtido anteriormente, com um caso do “Força Bruta” executando mais rápido do que o “KMP”.

## 4.2 Tarefa B

Para as saídas a seguir geradas pelo algoritmo de cifra de deslocamento, foi usado esse texto como entrada:

É recomendável inserir textos em português e sem acentos.

```

Terceiro Trabalho Pratico na disciplina de concepcao e analise de algoritmos.
As cores vibrantes do por do sol pintaram o ceu em tons de laranja e rosa.
Explorando as ruas pitorescas da cidade velha, me deparei com uma pequena livraria charmosa.
A melodia suave do piano encheu a sala, criando uma atmosfera de tranquilidade.
A medida que as ondas batiam na costa, senti a areia sob meus pes e a brisa fresca do oceano em meu rosto.
No coracao da movimentada cidade, um jardim escondido oferecia um retiro tranquilo do caos urbano.
O aroma do cafe acabado de fazer flutuava no ar, atraindo todos no cafe.
Perdido nas paginas de um romance cativante, o tempo pareceu parar por um momento.
Um rastro de luzes decorativas adornava as arvores, criando um ambiente magico no jardim.
Numa preguiçosa tarde de domingo, fiz um piquenique no parque com os amigos, saboreando guloseimas deliciosas.
O horizonte da cidade brilhava com luzes ao cair da noite, lancando um belo reflexo no rio.

```

**Figura 16.1: Arquivo de entrada utilizado para os testes.**

Saída após o processo de criptografia com a chave valendo 5:

```

Digite o arquivo de entrada:
Arquivos/entrada.txt
Digite o arquivo de saída:
Arquivos/saida.txt
[1] Digitar uma propria chave
[2] Usar chave aleatoria
1
Entre com uma chave:
5
O que deseja fazer?
[1] Criptografar
[2] Descriptografar
1
Yjwhjnw t Ywfgfqmt Uwfynht sf inxhnuqnsf ij htshjuhft j fsfqnxj ij fqltwnyrtx.
Fx htwjx angwfsyjk it utw it xtg unsyfwfr t hjz jr ytsx ij qfwfsof j wtxf.
Jcuqtwfsit fx wzfx unytwjxhfx if hnifij ajqmf, rj ijufwjn htr zrf ujvzjsf qnawfwf hmfwrwtxf.
F rjqtinf xzfaj it unfst jshmjz f xqf, hwnfsit zrf fyrtxkjwf ij ywfsvznqnfij.
F rjinif vzj fx tsifx gfyfyr sf htxyf, xjsyn f fwjnf xtg rjzx ujk j f gwnxf kwjxhf it thjfst jr rjz wtxyt.
St htwfhft if rtanrjsyfif hnifij, zr ofwinr jxhtsinit tkjwjhnf zr wjynwt ywfsvznqt it hftx zwgfst.
T fwrtrf it hfkj fhfgfif ij kfejw kqzyzfaf st fw, fywfsit ytix st hfkj.
Ujwinit sfx uflnsfx ij zr wtrfshj hfynafsyj, t yjrut ufwhjz ufww utw zr rtrjsyt.
Zr wfxwyt ij qzejx ijhtwfyfafx fitwsfaf fx fwtwtx, hwnfsit zr frgnjsyj rflnht st ofwinr.
Szrf uwjlzhtxf yfwij ij itrnsit, kne zr unvzjsnvzj st ufwwzj htr tx frnltx, xfgtwjfsit lzqtxjnrfx ijqnhtxfx.
T mtwnetsyj if hnifij gwnqmfaf htr qzejx ft hfnw if stnyj, qfshfsit zr giat wjkajct st wnt.

```

**Figura 16.2: Saída do programa após o processo de criptografia com chave igual a 5.**

Saída após o usuário escolher o processo de criptografia com chave randômica gerada:

```

Digite o arquivo de entrada:
Arquivos/entrada.txt
Digite o arquivo de saída:
Arquivos/saida.txt
[1] Digitar uma propria chave
[2] Usar chave aleatoria
2
O que deseja fazer?
[1] Criptografar
[2] Descriptografar
1
Panyaenk Pnwwhdk Lnwpeyk jw zeoyelhejw za ykjyalywk a wjwheoa za whcknepiko.
Wo yknao rexnwpao zk lkn zk okh lejpmwi k yaq ai pkjo za hwnwjfw a nkow.
Atlhknwjz wo nqwo lepnaoywo zw yezwza rahdw, ia zalwnae yki qiw lamqajw hernwnew ydwnikow.
W iahkzew oqwa zk lewjz ajydaq w owhw, ynewjz qiw wpikobanw za pnwjmgehezwa.
W iazew mqa wo kjzwo xpwewi jw ykopw, oajpe w wnaew okx iaqo lao a w xneow bnaoyw zk kyawjk ai iaq nkopk.
Jk yknwywk zw ikreiajpzw yezwza, qi fwnzei aoykjzezk kbanayew qi napenk pnwjmgehk zk ywko qnxwjz.
K wniw zk ywba wywxwzk za bwvan bhqpqrw jk wn, wpnwejkz pkzko jk ywba.
Lanzezk jwo lwcejwo za qi nkiwja ywperwpa, k pailk lwnayaq lwnwn lkn qi ikiajpk.
Qi nwopnk za hqvao zayknwperwo wzknjwrv wo wnrknao, ynewjz qi wixejpa iwceyk jk fwnzei.
Jqiw lncqeykow pwnza za zkiejck, bev qi lemqaemqa jk lwnmqa yki ko wiecko, owxknawjz cqhkoaeiwo zaheyekowo.
K dknevkjpa zw yezwza xnehdwrv yki hqvao wk ywen zw jkepa, hwjywjz qi xahk nabhatk jk nek.

```

**Figura 16.3: Execução da “Tarefa B” escolhendo um modo de chave aleatória e a opção de “Criptografar”**



Caractere	Frequencia	Percentual
A	80	10.09%
B	9	1.13%
C	7	0.88%
D	6	0.76%
E	55	6.94%
F	3	0.38%
G	0	0.00%
H	22	2.77%
I	40	5.04%
J	45	5.67%
K	90	11.35%
L	20	2.52%
M	7	0.88%
N	59	7.44%
O	46	5.80%
P	30	3.78%
Q	32	4.04%
R	11	1.39%
S	0	0.00%
T	2	0.25%
U	0	0.00%
V	5	0.63%
W	122	15.38%
X	11	1.39%
Y	39	4.92%
Z	52	6.56%

Chave Chute: 22  
Chave aleatoria gerada: 22

**Figura 16.4: Continuação da imagem anterior mostrando a execução**

## 5) Conclusão

Com a conclusão deste projeto, foi possível aprofundar nosso entendimento sobre a aplicação prática de algoritmos de manipulação de caracteres, em particular, o casamento exato utilizando força bruta, o algoritmo de Knuth-Morris-Pratt (KMP) e a cifra de deslocamento. Ao analisar os resultados obtidos ao término da execução do Trabalho Prático, concluímos que o grupo alcançou satisfatoriamente as metas estabelecidas, obtendo resultados precisos em diversas situações de teste. A aplicação desses algoritmos revelou-se eficaz para resolver problemas relacionados à manipulação de caracteres. Dessa forma, evidencia-se que a equipe não apenas adquiriu conhecimentos teóricos sobre as técnicas abordadas, mas também conseguiu implementá-las com sucesso.

## 6) Referências

IDE's utilizadas:

- CLion: <https://www.jetbrains.com/pt-br/clion/>
- Visual Studio Code: <https://code.visualstudio.com/>

Informações e Dicas:

- StackOverflow: <https://stackoverflow.com/>