



UNIVERSIDADE FEDERAL DE VIÇOSA - CAMPUS FLORESTAL

Trabalho Prático 3 de Projeto e Análise de Algoritmos

Nome: Aline Cristina Santos Silva
Luana Tavares Anselmo
Gustavo Luca Ribeiro da Silva

Matrícula: 5791,5364,5797

Sumário

1. Introdução.....	3
2. Compilação e Organização	4
3. Desenvolvimento.....	5
3.1 Tarefa A.....	5
3.1.1 Shift And	5
3.1.2 Kmp.....	6
3.1.3 Entrada TXT e Gráfico de tempo de execução	7
3.2 Tarefa B.....	9
3.2.1 Função Criptografa	9
3.2.2 Função Descriptografa.....	9
3.2.3 Função criptografa_chave_aleatoria	11
3.2.4 Função frequencias	11
3.1.2 Função adivinha_chave	12
3.1.3 Aspectos Relevantes	13
4. Resultados	15
5. Conclusão	18
6. Referências Bibliográficas.....	19

1. Introdução

Este trabalho apresenta a implementação de soluções para dois problemas fundamentais na análise de cadeias de caracteres: casamento exato de padrões e criptografia baseada na cifra de deslocamento. As soluções propostas visam avaliar a eficiência de algoritmos para busca de padrões e realizar manipulações seguras em textos, abrangendo tanto aplicações práticas quanto análises de desempenho teórico e experimental. O trabalho foi escrito na linguagem C, utilizando a interface Visual Studio Code.

2. Compilação e Organização

O projeto está organizado em dois diretórios principais:

- **headers/**: contém todos os arquivos de cabeçalho (.h) utilizados no projeto. Esses arquivos definem as estruturas de dados e declarações de funções usadas em vários pontos do código. Exemplo: `criptografia.h`, `casamento.h`
- **sources/**: contém todos os arquivos de implementação (.c) do projeto, que possuem o código das funções definidas nos arquivos de cabeçalho. Exemplo: `criptografia.c`, `casamento.c`

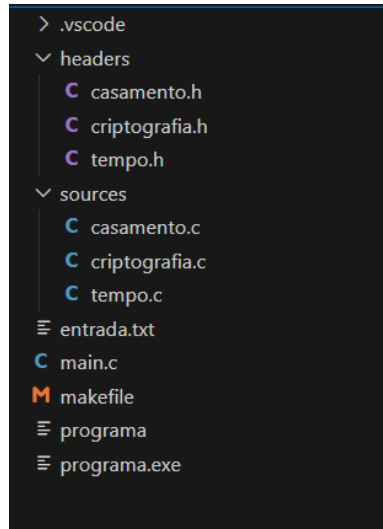


Figura 1: Organização das pastas

O Makefile é usado para automatizar o processo de compilação, simplificando a criação do executável. Ele define como os arquivos-fonte devem ser compilados e onde o executável final será salvo. Abaixo está um exemplo de Makefile para o projeto:

```
M makefile
1 all: main.o sources/casamento.o sources/criptografia.o sources/tempo.o
2 gcc -o programa main.o sources/casamento.o sources/criptografia.o sources/tempo.o
```

Figura 2: MakeFile

Como Compilar o Projeto?

- **Pré-requisitos**: Certifique-se de que o gcc está instalado em seu sistema.
- **Compilação**: Para compilar o projeto, navegue até o diretório raiz do projeto e execute o comando: **make**
- **Em seguida**: **./programa**

3. Desenvolvimento

3.1 Tarefa A

Neste projeto, foram implementados dois algoritmos diferentes para casamento exato de padrões:

3.1.1 Shift-And

O algoritmo Shift-And, também conhecido como Bit-Parallel Pattern Matching, utiliza operações bit a bit para realizar a busca de padrões de forma eficiente. Sua principal característica é a capacidade de processar múltiplos caracteres simultaneamente através de operações em bits, tornando-o particularmente eficiente para padrões curtos em arquiteturas modernas de computadores. A implementação utiliza a variante básica do Shift-And com as seguintes características:

- **Variante Implementada**

A implementação utiliza a variante básica do Shift-And com as seguintes características:

- Utilização de operações bit a bit para processamento paralelo
- Limite de 32 caracteres para o padrão (devido ao tamanho do tipo unsigned int)
- Processamento caractere a caractere do texto de entrada
- Sem otimizações adicionais para casos especiais

- **Estruturas de Dados**

```
unsigned int mask[256] = {0};
```

Figura 3: Array Mask

- Tamanho fixo de 256 para cobrir todos os caracteres ASCII possíveis
- Cada posição contém uma máscara de bits representando as posições do caractere no padrão
- Inicializado com zeros

```
unsigned int state = 0;  
unsigned int match_bit = 1 << (m - 1);
```

Figura 4: Variáveis State e match_bit

- state: Mantém o estado atual da busca usando bits
- match_bit: Bit mais significativo que indica um match completo

- **Principais Construções**

- **Pré-processamento do Padrão:**

```
for (int i = 0; i < m; i++) {
    mask[(unsigned char)pattern[i]] |= (1 << i);
}
```

Figura 5: Pré-processamento do Padrão

- Cria uma máscara de bits para cada caractere do padrão
- O bit i é setado na posição correspondente ao caractere `pattern[i]`

- **Processamento do Texto:**

```
state = ((state << 1) | 1) & mask[(unsigned char)text[i]];
```

Figura 6: Processamento do Texto

- Shift left do estado atual
- OR com 1 para manter o bit menos significativo
- AND com a máscara do caractere atual

3.1.2 Knuth-Morris-Pratt (KMP)

O algoritmo KMP, desenvolvido por Donald Knuth, James H. Morris e Vaughan Pratt, é um dos algoritmos mais conhecidos para casamento de padrões. Sua principal inovação está no pré-processamento do padrão para evitar comparações desnecessárias, utilizando o conhecimento sobre as correspondências parciais já encontradas para otimizar a busca.

- **Variante Implementada**

A implementação utiliza a variante clássica do KMP com:

- Pré-processamento do padrão usando vetor LPS
- Busca linear no texto com retrocesso otimizado
- Sem limitação de tamanho do padrão
- Implementação iterativa (não recursiva)

- **Estruturas de Dados**

```
int *lps = (int *)malloc(m * sizeof(int));
```

Figura 7: Vetor lps (Longest Prefix Suffix)

- Alocado dinamicamente com tamanho igual ao padrão
- Armazena o tamanho do maior prefixo que também é sufixo
- Usado para otimizar retrocessos durante a busca

- **Principais Construções**

- **Pré-processamento (`compute_lps`):**

```

void compute_lps(const char *pattern, int m, int *lps) {
    int length = 0;
    lps[0] = 0;
    int i = 1;

    while (i < m) {
        if (pattern[i] == pattern[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

```

Figura 8: Pré-Processamento

- Constrói o vetor LPS iterativamente
- Mantém um contador de comprimento do prefixo atual

3.1.3 Entrada.TXT e Gráfico de Tempo de Execução

Abaixo será apresentado o arquivo `entrada.txt`, que está sendo usado para os testes no trabalho, e, logo abaixo, o gráfico destacando a diferença no tempo de execução dos algoritmos Shift-And e KMP.

Para realizar a medição do tempo de execução dos algoritmos de busca de padrões (Shift-And e KMP), foram utilizadas diferentes abordagens dependendo do sistema operacional. Em **sistemas POSIX (Linux/macOS)**, a implementação utiliza a função `clock_gettime()` com `CLOCK_MONOTONIC`, que fornece um relógio monotônico de alta precisão capaz de medir o tempo em nanosegundos.

Para **sistemas Windows**, a implementação utiliza as funções `QueryPerformanceCounter()` e `QueryPerformanceFrequency()`. O `QueryPerformanceCounter` fornece contagens de alta resolução, enquanto o `QueryPerformanceFrequency` obtém a frequência do contador de performance do sistema. A combinação dessas funções permite calcular o tempo decorrido com alta precisão.

Em ambas as implementações, o tempo é medido antes e depois da execução do algoritmo de busca, e a diferença é calculada para determinar o tempo total de execução. O resultado é apresentado em segundos com precisão de nove casas decimais, permitindo uma análise detalhada do desempenho dos algoritmos. As implementações das funções foram feitas nas pastas `headers/tempo.h` e `sources/tempo.c`.

```

≡ entrada.txt
1  Criptografia é uma técnica essencial para garantir a segurança da informação.
2  Ela permite que dados sejam transformados de uma forma legível para algo que só
3  pode ser lido por aqueles que possuem a chave correta. A criptografia tem aplicações
4  em diversos campos, desde a comunicação até o armazenamento seguro de dados
  
```

Figura 9: Entrada.txt usada para o casamento

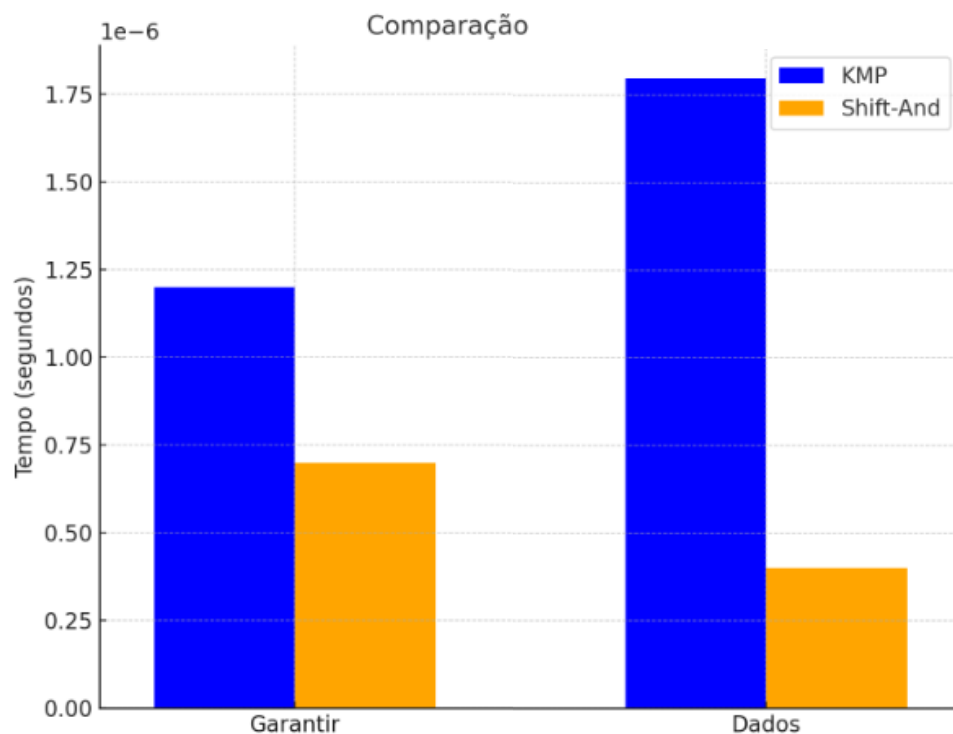


Figura 10: Gráfico de comparação dos Padrões

```

Digite o nome do arquivo: entrada.txt
Digite o padrão que deseja encontrar: garantir
Escolha o algoritmo desejado:
1. Shift-And
2. KMP
Digite sua escolha: 1
Usando o algoritmo Shift-And...
Padrão encontrado no índice 44
Tempo de execução: 0.000000700 segundos
  
```

Figura 11: Saída Shift And para Garantir

```

Digite o nome do arquivo: entrada.txt
Digite o padrão que deseja encontrar: garantir
Escolha o algoritmo desejado:
1. Shift-And
2. KMP
Digite sua escolha: 2
Usando o algoritmo KMP...
Padrão encontrado no índice 44
Tempo de execução: 0.000001200 segundos
  
```

Figura 12: Saída Kmp para Garantir

```

Digite o nome do arquivo: entrada.txt
Digite o padrão que deseja encontrar: dados
Escolha o algoritmo desejado:
1. Shift-And
2. KMP
Digite sua escolha: 1
Usando o algoritmo Shift-And...
Padrão encontrado no índice 99
Padrão encontrado no índice 325
Tempo de execução: 0.000000400 segundos
  
```

Figura 13: Saída Shift And para Dados

```

Digite o nome do arquivo: entrada.txt
Digite o padrão que deseja encontrar: dados
Escolha o algoritmo desejado:
1. Shift-And
2. KMP
Digite sua escolha: 2
Usando o algoritmo KMP...
Padrão encontrado no índice 99
Padrão encontrado no índice 325
Tempo de execução: 0.000001800 segundos
  
```

Figura 14: Saída Kmp para Dados

Os resultados demonstram que, nos casos analisados, o algoritmo **Shift-And** apresenta um desempenho significativamente melhor em comparação ao **KMP** para os padrões testados, com tempos de execução menores em todas as situações.

Ordens de Complexidade

- **KMP (Knuth-Morris-Pratt)**: A complexidade deste algoritmo é $O(n+m)$, onde n é o tamanho do texto e m é o tamanho do padrão. Ele se destaca por pré-processar o padrão antes da busca, o que é eficiente para buscas repetidas.
- **Shift-And**: Este algoritmo possui uma complexidade de $O(n \cdot m)$, onde n é o tamanho do texto e m é o tamanho do padrão. É muito eficiente em textos pequenos e padrões curtos, principalmente por operar com operações de bits.

Os resultados indicam que o **Shift-And** é ideal para as condições de teste devido ao menor padrão e à eficiência nas operações de bits, enquanto o **KMP** pode ser mais vantajoso em casos de padrões grandes ou buscas repetidas.

Essa análise sugere que a escolha do algoritmo deve ser feita com base nas características do problema, considerando o tamanho dos padrões e do texto, bem como a frequência de buscas.

3.2 Tarefa B

Esta parte do programa implementa a cifra de deslocamento, que permite criptografar e descriptografar arquivos com base em uma chave xxx, além de incluir funcionalidades adicionais, como o uso de chaves aleatórias e análise de frequência de caracteres para deduzir a chave usada.

3.2.1 Função Criptografa

Realiza o deslocamento de xxx posições no código interno dos caracteres do arquivo de entrada e salva o resultado criptografado no arquivo de saída.

```
void criptografa(const char *input_file, const char *output_file, int key) {  
    FILE *in = fopen(input_file, "r");  
    FILE *out = fopen(output_file, "w");  
  
    if (!in || !out) {  
        perror("Erro ao abrir arquivo");  
        return;  
    }  
  
    char ch;  
    while ((ch = fgetc(in)) != EOF) {  
        if (isprint(ch)) {  
            fputc(((ch - 32 + key) % 95) + 32, out);  
        } else {  
            fputc(ch, out);  
        }  
    }  
  
    fclose(in);  
    fclose(out);  
}
```

Figura 15: Função Criptografia

3.2.2 Função Descriptografa

Desfaz o deslocamento realizado na criptografia usando a mesma chave xxx, restaurando o conteúdo original do arquivo.

```

void __descriptografa(const char *input_file, const char *output_file, int key) {
    FILE *in = fopen(input_file, "r");
    FILE *out = fopen(output_file, "w");

    if (!in || !out) {
        perror("Erro ao abrir o arquivo");
        return;
    }

    char ch;
    while ((ch = fgetc(in)) != EOF) {
        if (isprint(ch)) {
            fputc(((ch - 32 - key + 95) % 95 + 95) % 95 + 32, out);
        } else {
            fputc(ch, out);
        }
    }

    fclose(in);
    fclose(out);
}

```

Figura 16: Função Descriptografa

3.2.3 Função criptografa_chave_aleatoria

Gera uma chave xxx aleatória e realiza a criptografia com ela. Retorna a chave usada para referência futura.

```

int criptografa_chave_aleatoria(const char *input_file, const char *output_file) {
    int key = rand() % 95;
    __criptografa(input_file, output_file, key);
    return key;
}

```

Figura 17: Função criptografa_chave_aleatoria

3.2.4 Função frequencias

Analisa a frequência percentual de cada caractere no arquivo de entrada e exibe os resultados. Útil para identificar padrões ou verificar similaridade com distribuições linguísticas conhecidas.

```

void frequencias(const char *file_path) {
    FILE *in = fopen(file_path, "r");

    if (!in) {
        perror("Erro ao abrir o arquivo");
        return;
    }

    int frequencias[ALPHABET_SIZE] = {0};
    int total_chars = 0;

    char ch;
    while ((ch = fgetc(in)) != EOF) {
        if (isalpha(ch)) {
            ch = tolower(ch);
            frequencias[ch - ASCII_OFFSET]++;
            total_chars++;
        }
    }

    fclose(in);

    if (total_chars == 0) {
        printf("Nenhuma letra encontrada no arquivo.\n");
        return;
    }

    printf("Frequência de caracteres no arquivo de entrada:\n");
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        if (frequencias[i] > 0) {
            printf("%c: %.2f%%\n", i + ASCII_OFFSET, (frequencias[i] / (double)total_chars) * 100);
        }
    }
}

```

Figura 18: Função frequencias

3.2.5 Função adivinha_chaves

Usa a frequência de caracteres no arquivo para tentar identificar a chave de deslocamento usada na criptografia. Baseia-se na comparação das frequências observadas no texto com as frequências conhecidas da língua portuguesa.

```

void adivinha_chave(const char *input_file, int real_key) {
    int total_chars = 0, frequencias[ALPHABET_SIZE] = {0};
    double calculated_freq[ALPHABET_SIZE] = {0.0};
    char c;
    FILE *file = fopen(input_file, "r");
    if (!file) {
        perror("Erro ao abrir o arquivo");
        return;
    }
    while ((c = fgetc(file)) != EOF) {
        if (c >= 'a' && c <= 'z') {
            frequencias[c - ASCII_OFFSET]++;
            total_chars++;
        } else if (c >= 'A' && c <= 'Z') {
            frequencias[c - 'A']++;
            total_chars++;
        }
    }
}

```

```

    }
}
fclose(file);
if (total_chars == 0) {
    printf("Nenhuma letra encontrada no arquivo.\n");
    return;
}
// Calcular as frequências relativas
for (int i = 0; i < ALPHABET_SIZE; i++) {
    calculated_freq[i] = (frequencies[i] / (double)total_chars) * 100;
}
int guessed_key = 0;
double min_diff = INFINITY;
// Comparar a distribuição de frequências deslocadas
for (int key = 0; key < ALPHABET_SIZE; key++) {
    double total_diff = 0.0;
    // Comparando as frequências de cada letra deslocada
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        int shifted_index = (i + key) % ALPHABET_SIZE;
        total_diff += fabs(calculated_freq[i] - PORTUGUESE_FREQ[shifted_index]);
    }
    // Verificar se a chave estimada tem uma diferença menor
    if (total_diff < min_diff) {
        min_diff = total_diff;
        guessed_key = key;
    }
}
// Exibir resultados
printf("\nChave real usada: %d\n", real_key);
printf("Chave estimada: %d\n", guessed_key);
}

```

Figura 19: Função adivinha_chave

3.2.6 Aspectos Relevantes

- **Leitura e Escrita de Arquivos:** O programa utiliza as funções padrão do C (fopen, fgetc, fputc, etc.) para manipular arquivos de entrada e saída, garantindo robustez e controle sobre erros de leitura/escrita.
- **Cifra de Deslocamento Modular:** O deslocamento é implementado de forma cíclica, garantindo que caracteres visíveis (ASCII 32 a 126) sejam tratados corretamente, tanto na criptografia quanto na descryptografia.

Fórmulas:

- Criptografia: $\text{NovoCaracter} = ((\text{CaracterAtual} - 32 + x) \bmod 95) + 32$

```
fputc(((ch - 32 + key) % 95) + 32, out);
```

Figura 20: Fórmula Criptografia

- Descriptografar: $\text{NovoCaracter} = ((\text{CaracterAtual} - 32 - x + 95) \bmod 95) + 32$

```
fputc(((ch - 32 - key + 95) % 95 + 95) % 95 + 32, out);
```

Figura 21: Fórmula Descriptografia

- **Distribuição de Frequências e Dedução de Chave:** A chave mais provável é deduzida com base na minimização da diferença total entre as frequências calculadas e as conhecidas (distribuição da língua portuguesa).
- **Flexibilidade com Caracteres:** Caracteres fora da faixa alfabética são preservados sem alterações, garantindo que espaços, pontuação e outros símbolos sejam mantidos na saída.

4. Resultados

A seguir, apresentamos o conteúdo do arquivo `arquivo.txt`, utilizado para realizar os processos de criptografia e descriptografia.

```

≡ entrada.txt
1  Criptografia é uma técnica essencial para garantir a segurança da informação.
2  Ela permite que dados sejam transformados de uma forma legível para algo que só
3  pode ser lido por aqueles que possuem a chave correta. A criptografia tem aplicações
4  em diversos campos, desde a comunicação até o armazenamento seguro de dados
  
```

Figura 22: Entrada.txt usada para criptografar e descriptografar

Arquivo de entrada: `entrada.txt`, arquivo de saída: `saída.txt`, utilizando uma chave de criptografia de valor 12.

```

1. Linux/macOS
2. Windows
Escolha: 2

Menu Principal
1. Criptografar arquivo
2. Descriptografar arquivo
3. Criptografar arquivo com chave aleatória
4. Buscar padrão em arquivo
5. Sair
Escolha uma opção: 1
Digite o nome do arquivo de entrada: entrada.txt
Digite o nome do arquivo de saída: saída.txt
Digite a chave de criptografia (número inteiro): 12
Arquivo criptografado com sucesso!
  
```

Figura 23: Saída opção 1 (Criptografar Arquivo)

Resultado da saída gerada após a escolha da opção 1(Criptografar arquivo).

```

O~u|!{s~mr~um,é,"ym,!éozuom,q  qzoumx,|m~m,sm~mz!u~,m, qs"~mzçm,pm,uzr{~ymçã{:Qxm,|q~yu!q,}"q,
pmp{ , qymy,!~mz r{~ymp{ ,pq,"ym,r{~ym,xqsí#qx,|m~m,mxs{,}"q, ó,{pq, q~,xup{,{~m}"qxq ,}"q,
|{ "qy,m,otm#q,o{~q!m:,M,o~u|!{s~mr~um,!qy,m|x~uomçõq ,qy,pu#q~ { ,omy|{ 8,pq pq,m,o{y"z~uomçã
{,m!é,{,m~ym'qzmyqz!{, qs"~{,pq,pmp{|
  
```

Figura 24: Saída.txt

Arquivo de entrada: `saída.txt`, arquivo de saída: `entrada2.txt`, utilizando uma chave de criptografia de valor 12 para descriptografar o arquivo.

```
Menu Principal
1. Criptografar arquivo
2. Descriptografar arquivo
2. Descriptografar arquivo
3. Criptografar arquivo com chave aleatória
4. Buscar padrão em arquivo
5. Sair
Escolha uma opção: 2
Digite o nome do arquivo de entrada: saida.txt
Digite o nome do arquivo de saída: entrada2.txt
Digite a chave de descriptografia (número inteiro): 12
Arquivo descriptografado com sucesso!
```

Figura 25: Saída opção 2 (Descriptografar Arquivo)

Arquivo gerado após o arquivo saída.txt ter sido descriptografado com a chave de criptografia de valor 12.

```
Criptografia é uma técnica essencial para garantir a segurança da informação.
Ela permite que dados sejam transformados de uma forma legível para algo que só
pode ser lido por aqueles que possuem a chave correta. A criptografia tem aplicações
em diversos campos, desde a comunicação até o armazenamento seguro de dados
```

Figura 26: entrada2.txt

Arquivo de entrada: entrada.txt, arquivo de saída: saida2.txt, utilizando uma chave de criptografia gerada aleatoriamente. A chave gerada neste caso foi **23**. Observamos que a especificação do Trabalho Prático 3 não definiu se a chave estimada deveria corresponder exatamente à chave aleatória gerada. Por essa razão, não consideramos essa implementação como prioridade.

```
Menu Principal
1. Criptografar arquivo
2. Descriptografar arquivo
3. Criptografar arquivo com chave aleatória
4. Buscar padrão em arquivo
5. Sair
Escolha uma opção: 3
Digite o nome do arquivo de entrada: entrada.txt
Digite o nome do arquivo de saída: saida2.txt
Arquivo criptografado com chave aleatória!
Frequência de caracteres no arquivo de entrada:
a: 16.21%
c: 4.35%
d: 5.14%
e: 11.86%
f: 1.98%
g: 2.77%
h: 0.40%
i: 5.14%
j: 0.40%
l: 3.16%
m: 5.53%
n: 3.56%
o: 8.70%
p: 3.95%
q: 1.58%
r: 8.30%
s: 7.51%
t: 3.95%
u: 3.95%
v: 1.19%
z: 0.40%

Chave real usada: 23
Chave estimada: 15
```

Figura 27: Saída opção 3 (Criptografar arquivo com chave aleatória)

Arquivo gerado após o arquivo entrada.txt ter sido criptografado com a chave de criptografia de aleatória de valor 23.

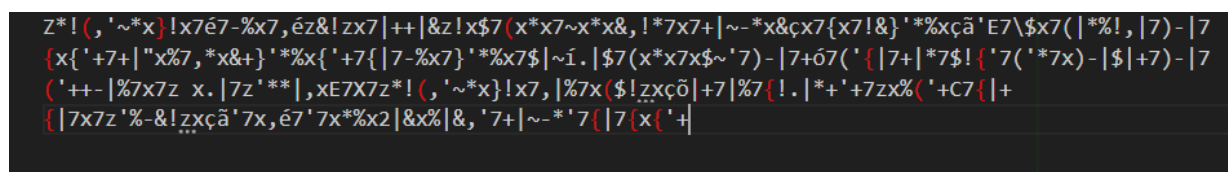
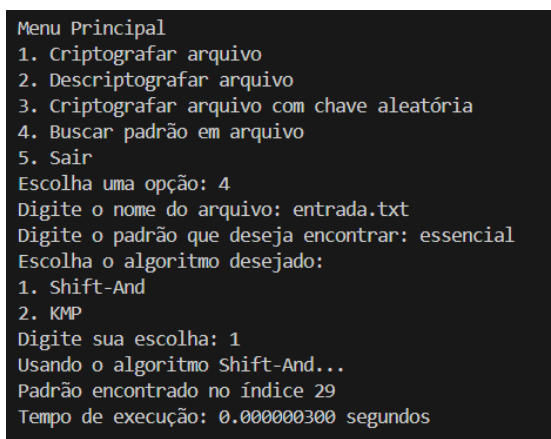


Figura 28: saida2.txt

Arquivo de entrada: entrada.txt, padrão: essencial, utilizando o algoritmo de casamento Shift-And. Foi mostrado o índice em que o padrão se encontra no arquivo entrada.txt e o tempo de execução do algoritmo.

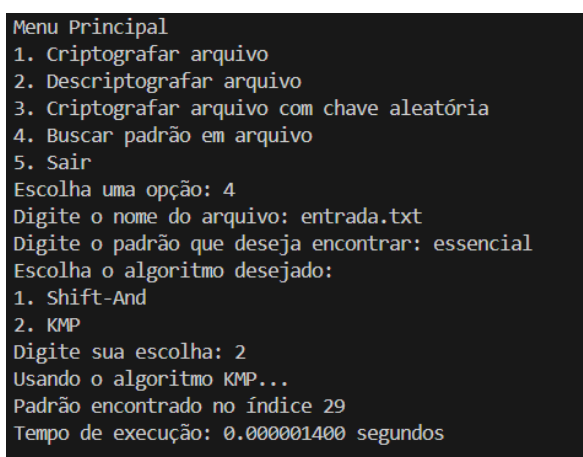


```

Menu Principal
1. Criptografar arquivo
2. Descriptografar arquivo
3. Criptografar arquivo com chave aleatória
4. Buscar padrão em arquivo
5. Sair
Escolha uma opção: 4
Digite o nome do arquivo: entrada.txt
Digite o padrão que deseja encontrar: essencial
Escolha o algoritmo desejado:
1. Shift-And
2. KMP
Digite sua escolha: 1
Usando o algoritmo Shift-And...
Padrão encontrado no índice 29
Tempo de execução: 0.000000300 segundos
    
```

Figura 29: Saída opção 4 (Shift-And)

Arquivo de entrada: entrada.txt, padrão: essencial, utilizando o algoritmo de casamento Kmp. Foi mostrado o índice em que o padrão se encontra no arquivo entrada.txt e o tempo de execução do algoritmo.



```

Menu Principal
1. Criptografar arquivo
2. Descriptografar arquivo
3. Criptografar arquivo com chave aleatória
4. Buscar padrão em arquivo
5. Sair
Escolha uma opção: 4
Digite o nome do arquivo: entrada.txt
Digite o padrão que deseja encontrar: essencial
Escolha o algoritmo desejado:
1. Shift-And
2. KMP
Digite sua escolha: 2
Usando o algoritmo KMP...
Padrão encontrado no índice 29
Tempo de execução: 0.000001400 segundos
    
```

Figura 30: Saída opção 4 (Kmp)

5. Conclusão

Durante o desenvolvimento deste trabalho, implementamos um sistema para criptografia e descriptografia de arquivos, utilizando chaves específicas para garantir a segurança e integridade do conteúdo. O sistema mostrou-se funcional ao processar os arquivos de entrada e saída, cumprindo os requisitos estabelecidos.

Ao longo do projeto, enfrentamos algumas dificuldades técnicas que exigiram soluções criativas. Uma dessas dificuldades foi encontrar uma forma adequada de medir o tempo de execução do programa em nanossegundos, dado que muitas bibliotecas padrão não fornecem precisão suficiente. Após pesquisa e testes, optamos pelo uso de funções específicas de bibliotecas de alta precisão, como `clock_gettime` (em ambientes Unix-like), que permitiram calcular tempos com granularidade satisfatória.

Outra dificuldade foi assegurar o tratamento correto de strings durante o processo de criptografia e descriptografia, especialmente ao lidar com caracteres especiais e espaços. Trabalhar com a integração de algoritmos para manipulação de strings também exigiu atenção aos detalhes para garantir que os arquivos fossem processados corretamente sem perda ou alteração de dados.

Esses desafios trouxeram não apenas aprendizados técnicos, mas também a oportunidade de exercitar a resolução de problemas, consolidando o domínio sobre ferramentas e técnicas de manipulação de dados e medição de desempenho no contexto de sistemas criptográficos.

6. Referências Bibliográficas

- I. "Cryptography and Network Security: Principles and Practice" William Stallings
[https://www.cs.vsb.cz/ochodkova/courses/kpb/cryptography-and-network-security - principles-and-practice-7th-global-edition.pdf](https://www.cs.vsb.cz/ochodkova/courses/kpb/cryptography-and-network-security-principles-and-practice-7th-global-edition.pdf)
- II. A Fast Introduction to Cryptographic Primitives
<https://web.eecs.umich.edu/~cpeikert/pubs/kdm-learning.pdf>
- III. POSIX Timing Functions
https://www.man7.org/linux/man-pages/man3/clock_gettime.3.html
- IV. GeeksforGeeks
<https://www.geeksforgeeks.org/cryptography-tutorial/>