

Relatório do Trabalho 1 de INF 1010 - Estruturas de Dados Avançadas

Trabalho 1 - Compressão e Descompressão

Data: 13/05/2024

Alunos:

Lucas Lucena - matrícula 2010796

Luísa Silveira - matrícula 2210875

Objetivo:

O objetivo deste trabalho é criar e implementar um compactador e um descompactador de arquivos de texto com base no algoritmo de Huffman. Para compactar e descompactar o arquivo implemente uma tabela contendo o caractere e o número de ocorrências do caractere no texto e outra contendo o caractere, o código binário correspondente e o seu tamanho em bits.

Estrutura do Programa:

Inclusão de bibliotecas utilizadas;

Define TAM como 128(tamanho da tabela ascii) e BYTE como 8 (tamanho do número de bits em um byte);

Criação de estruturas de dados:

- NoFreq (utilizado para lista de frequências lidas)
- FREQUENCIA (lista de nós NoFreq de frequências)
- NoTrie (nó da árvore de Huffman)
- CaractereCodigo (armazena caractere e código para compressão)

Criação de funções para Frequência;

Criação de funções para Árvore de Huffman;

Criação de funções para Compactação e Descompactação;

Função principal (main), que abre os arquivos utilizados, e utiliza as outras funções e estruturas criadas.;

Solução:

1. Criação das estruturas de dados:

```
typedef struct NoFreq {
    unsigned char simbolo;
    int frequencia;
    struct NoFreq* proximo;
} NoFreq;

typedef struct {
    NoFreq* primeiro;
    int tamanho;
} FREQUENCIA;

typedef struct NoTrie {
    char caractere;
    int frequencia;
    struct NoTrie* esquerda;
    struct NoTrie* direita;
} NoTrie;

typedef struct {
    char caractere;
    unsigned int codigo;
    int tamanho;
} CaractereCodigo;
```

Struct NoFreq: utilizada para representar um nó na lista de frequência. Cada nó possui dois campos: o símbolo (caractere) e sua frequência no arquivo de texto. Além disso, possui um ponteiro para o próximo nó na lista, permitindo a construção de uma lista encadeada. Ela é importante para contar as frequências dos caracteres e ordená-los posteriormente.

Struct FREQUENCIA: lista de frequência dos caracteres no arquivo de texto. Ela contém um ponteiro para o primeiro nó da lista e um inteiro para armazenar o tamanho da lista.

Struct NoTrie: representa um nó na árvore de Huffman. Cada nó possui um caractere (ou um caractere especial para os nós intermediários), sua frequência, e dois ponteiros para os filhos esquerdo e direito na árvore.

Struct CaractereCodigo: armazena o caractere, seu código binário correspondente na árvore de Huffman e o tamanho desse código e é usada na fase de compactação e descompactação.

2. Funções para frequência:

```
int* criaVetorFrequencia(FILE* arqTexto, int vetor[]) {
    int c;
    for (int i = 0; i < TAM; i++) {
        vetor[i] = 0;
    }
    while ((c = fgetc(arqTexto)) != EOF) {
        vetor[c]++;
    }
    return vetor;
}

FREQUENCIA* inicializa_lista_frequencia() {
    FREQUENCIA* lista = (FREQUENCIA*)malloc(sizeof(FREQUENCIA));
    if (lista == NULL) {
        printf("Erro ao alocar memória.\n");
        exit(1);
    }
    lista->primeiro = NULL;
    lista->tamanho = 0;
    return lista;
}

void insereOrdenado(FREQUENCIA* lista, NoFreq* no) {
    if (lista->primeiro == NULL || no->frequencia < lista->primeiro->frequencia)
    {
        no->proximo = lista->primeiro;
        lista->primeiro = no;
    }
}
```

```

else {
    NoFreq* anterior = lista->primeiro;
    NoFreq* atual = lista->primeiro->proximo;
    while (atual != NULL && atual->frequencia < no->frequencia) {
        anterior = atual;
        atual = atual->proximo;
    }
    no->proximo = atual;
    anterior->proximo = no;
}
lista->tamanho++;
}

void criar_lista_frequencia(int vetor[], FREQUENCIA* lista) {
    for (int i = 0; i < TAM; i++) {
        if (vetor[i] > 0) {
            NoFreq* aux = (NoFreq*)malloc(sizeof(NoFreq));
            if (aux == NULL) {
                printf("Erro ao alocar memória.\n");
                exit(1);
            }
            aux->simbolo = i;
            aux->frequencia = vetor[i];
            aux->proximo = NULL;
            insereOrdenado(lista, aux);
        }
    }
}

// Adicionando manualmente o caractere '!' com frequência 0 - serve como eot
NoFreq* eot = (NoFreq*)malloc(sizeof(NoFreq));
if (eot == NULL) {
    printf("Erro ao alocar memória.\n");
    exit(1);
}
eot->simbolo = '!';
eot->frequencia = 0;
eot->proximo = NULL;
insereOrdenado(lista, eot);

```

```

}

void liberaLista(FREQUENCIA* lista) {
    NoFreq* atual = lista->primeiro;
    while (atual != NULL) {
        NoFreq* temp = atual;
        atual = atual->proximo;
        free(temp);
    }
    free(lista);
}

void imprime_frequencia(FREQUENCIA* lista) {
    NoFreq* atual = lista->primeiro;
    while (atual != NULL) {
        printf("%c'- %d\n", atual->simbolo, atual->frequencia);
        atual = atual->proximo;
    }
}

```

criaVetorFrequencia: Lê o arquivo de texto de entrada e cria um vetor de frequência, onde cada posição do vetor representa um caractere ASCII e o valor é a frequência de ocorrência desse caractere no texto.

inicializa_lista_frequencia: Inicializa e retorna uma lista de frequência vazia.

insereOrdenado: Insere um novo nó na lista de frequência de forma ordenada, utilizando o conceito de heap. Ela começa verificando se a lista está vazia ou se o nó a ser inserido possui uma frequência maior que o primeiro nó da lista. Se for verdadeiro uma das condições, o novo nó é inserido no início da lista. Se não for, a função percorre a lista, com dois ponteiros, sendo um para o nó anterior e outro para o atual, até chegar na condição que a frequência do atual for menor que a do nó. Quando encontra a posição adequada, o novo nó é inserido entre o nó anterior e o nó atual. Após a inserção, o tamanho da lista é aumentado.

criar_lista_frequencia: Cria lista de frequência a partir do vetor de frequência, adicionando os caracteres que aparecem no texto e suas frequências, de maneira ordenada. Ela também adiciona manualmente o caractere '!' com frequência 0, que vai servir como eot(end of transmission), ou seja, o caractere que indica o fim do texto.

liberaLista: Libera a memória alocada para a lista de frequência

imprime_frequencia: Imprime a lista de frequência, mostrando o número de repetições de cada caractere.

3. Funções para Árvore de Huffman:

```
NoTrie* criarNoTrie(char caractere, int frequencia) {
    NoTrie* novoNo = (NoTrie*)malloc(sizeof(NoTrie));
    novoNo->caractere = caractere;
    novoNo->frequencia = frequencia;
    novoNo->esquerda = NULL;
    novoNo->direita = NULL;
    return novoNo;
}

NoTrie* construirArvoreHuffman(FREQUENCIA* lista_frequencia) {
    int tamanho = lista_frequencia->tamanho;
    NoTrie** fila = (NoTrie**)malloc(tamanho * sizeof(NoTrie*));
    if (fila == NULL) {
        printf("Erro ao alocar memoria.\n");
        exit(1);
    }

    // Preenchimento inicial da fila com os nós da lista de frequência

    NoFreq* atual = lista_frequencia->primeiro;
    for (int i = 0; i < tamanho; ++i) {
        fila[i] = criarNoTrie(atual->simbolo, atual->frequencia);
        atual = atual->proximo;
    }

    // Construção da árvore de Huffman
    for (int i = 0; i < tamanho - 1; ++i) {
        int indiceMenor1 = 0, indiceMenor2 = 1;
        for (int j = 2; j < tamanho - i; ++j) {
            if (fila[j]->frequencia < fila[indiceMenor1]->frequencia) {
                indiceMenor2 = indiceMenor1;
                indiceMenor1 = j;
            }
        }
    }
}
```

```

    }
    else if (fila[j]->frequencia < fila[indiceMenor2]->frequencia) {
        indiceMenor2 = j;
    }
}

NoTrie* novoPai = criarNoTrie('\0', fila[indiceMenor1]->frequencia +
fila[indiceMenor2]->frequencia);
novoPai->esquerda = fila[indiceMenor1];
novoPai->direita = fila[indiceMenor2];
fila[indiceMenor1] = novoPai;
fila[indiceMenor2] = fila[tamanho - i - 1];
}

NoTrie* raiz = fila[0];
free(fila);
return raiz;
}

void geraCodigoRecursivo(NoTrie* raiz, unsigned int codigo, int tamanho,
CaractereCodigo caracteresCodificados[], int* indiceCaracteresCodificados) {
    if (raiz == NULL)
        return;

    if (raiz->caractere != '\0') {
        caracteresCodificados[*indiceCaracteresCodificados].caractere =
raiz->caractere;
        caracteresCodificados[*indiceCaracteresCodificados].codigo = codigo;
        caracteresCodificados[*indiceCaracteresCodificados].tamanho = tamanho;
        ++(*indiceCaracteresCodificados);
        return;
    }

    geraCodigoRecursivo(raiz->esquerda, codigo << 1, tamanho + 1,
caracteresCodificados, indiceCaracteresCodificados);

    geraCodigoRecursivo(raiz->direita, (codigo << 1) | 1, tamanho + 1,
caracteresCodificados, indiceCaracteresCodificados);
}

```

```
void codificarHuffman(NoTrie* raiz, CaractereCodigo caracteresCodificados[],int*
indiceCaracteresCodificados) {
    unsigned int codigo = 0;
    int tamanho = 0;
    geraCodigoRecursivo(raiz, codigo, tamanho,
caracteresCodificados,indiceCaracteresCodificados);
}
```

criarNoTrie: Cria e retorna um novo nó para a árvore de Huffman, dados caractere e frequência

contruirArvoreHuffman: Constroi e retorna árvore de Huffman a partir de uma lista de frequência, inicialmente preenchendo a fila com os nós da lista de frequência. Em seguida realiza um loop no qual os dois nós com menor frequência são combinados em um novo nó pa. Em cada iteração, os dois nós com as menores frequências são removidos da fila. Um novo nó é criado para representar o pai desses dois nós, com sua frequência sendo a soma das frequências dos nós filhos, sendo, depois, adicionado na fila. O processo continua até que apenas um nó, a raiz da árvore permaneça na fila.

geraCodigoRecursivo: Gera os códigos para compactação recursivamente tendo a árvore. Se o nó atual não for nulo, ela verifica se o nó contém um caractere. Se sim, armazena o caractere, junto com o caminho percorrido (o código para compactação) no array caracteresCodificados. Caso o nó não tenha um caractere, chama a si mesma recursivamente para o filho esquerdo e para o filho direito, acumulando o código que percorrerá (1 para a direita e zero para a esquerda).

codificarHuffman: Função de apoio que chama a geraCodigoRecursivo para gerar os códigos para compactação.

4. Função de Compactação:

```
void compacta(FILE* arqTexto, FILE* arqBin, CaractereCodigo* v, FREQUENCIA* freq)
{
    unsigned int byte = 0; // vai ser o nosso buffer
```



```

int contbits = 0;

//adicionando a tabela de codigo no arquivo
int tamanhoStruct = freq->tamanho;
fwrite(&tamanhoStruct, sizeof(int), 1, arqBin);

for (int i = 0; i < tamanhoStruct; i++) {
    fwrite(&(v[i].caractere), sizeof(char), 1, arqBin);
    fwrite(&(v[i].codigo), sizeof(unsigned int), 1, arqBin);
    fwrite(&(v[i].tamanho), sizeof(int), 1, arqBin);
}

while (!feof(arqTexto)) {    // lemos o arquivo
    unsigned char letra = fgetc(arqTexto);    // pegamos a letra
    for (int i = 0; i < tamanhoStruct; i++) {    // percorremos o vetor até
achar um símbolo que corresponde a letra

        if (v[i].caractere == letra) {    // quando encontra, adiciona ao byte
na posição correspondente
            int tamanho = v[i].tamanho;
            contbits += tamanho;
            byte = byte << tamanho;
            byte |= (int)v[i].codigo;

            while (contbits >= BYTE) {    // se o byte estiver cheio, com 8 ou
mais bits, escreve. Isso se mantém até o contbits for menor que 8, não estiver
cheio

                unsigned int novoCod = (byte >> (contbits - BYTE)) & 0xFF;
                fwrite(&novoCod, 1, 1, arqBin);
                contbits -= BYTE;    // reduz de 8 o contbits
            }

        }
    }
}

// escrevendo o EOT (!), fazemos um processo semelhante ao anterior, mas
precisamos percorrer o arquivo para saber o código do "!"

```

```

unsigned char eot = '!';
for (int i = 0; i < TAM; i++) {
    if (v[i].caractere == eot) {
        int tamanho = v[i].tamanho;
        contbits += tamanho;
        byte = byte << tamanho;
        byte |= v[i].codigo; // Adiciona o código do EOT ao byte
        while (contbits > 0) {
            unsigned int novoCod = byte >> (contbits - BYTE);
            novoCod &= 0xFF;
            if (contbits < BYTE) {
                novoCod &= (0xFF >> (BYTE - contbits));
            }
            fwrite(&novoCod, 1, 1, arqBin);
            contbits -= BYTE;
        }
    }
}
}
}

```

compacta: Compacta um arquivo binário usando os códigos gerados através da árvore de Huffman. Inicialmente, insere um byte contendo a quantidade de caracteres diferentes lidos, e então o código para cada um dos caracteres e o tamanho dos códigos, criando assim uma tabela de codificação no arquivo em binário. Ela lê o arquivo de texto caractere por caractere e, para cada caractere lido, busca o código de Huffman correspondente na struct CaractereCodigo. Os códigos binários resultantes são escritos no arquivo binário de saída. Como a compactação é feita por bit, se utiliza um buffer (variável byte) para armazenar temporariamente os códigos compactados antes de escrevê-los no arquivo binário. Ela tem um contador (contbits) para saber quantos bits foram adicionados ao buffer. Quando o buffer está cheio (8 bits), seu conteúdo é escrito no arquivo binário, e o contador é reiniciado. Depois de compactar todos os caracteres do arquivo, a função adiciona manualmente o código de fim de texto ("!") no arquivo binário.

5. Função de Descompactação:

```

void descompacta(FILE* arqBin, FILE* arqTexto) {

```

```

unsigned int buffer = 0;
int contbits = 0;
unsigned char hexa;
int total_caracteres;

//lendo a tabela de codigo
fread(&total_caracteres, sizeof(int), 1, arqBin);
CaractereCodigo* traduz = (CaractereCodigo*)malloc(total_caracteres *
sizeof(CaractereCodigo));
for (int i = 0; i < total_caracteres; i++) {
    fread(&(traduz[i].caractere), 1, 1, arqBin);
    fread(&(traduz[i].codigo), sizeof(unsigned int), 1, arqBin);
    fread(&(traduz[i].tamanho), sizeof(int), 1, arqBin);
}

while (!feof(arqBin)) { // Enquanto não chegarmos ao final do arquivo binário
    fread(&hexa, 1, 1, arqBin); // Lê um byte do arquivo binário

    for (int i = 7; i >= 0; i--) {
        unsigned int novoCod = 0;
        novoCod |= (hexa >> i) & 1; // Extrai um bit do byte
        buffer |= (novoCod); // Adiciona o bit lido ao buffer
        contbits++;

        for (int j = 0; j < total_caracteres; j++) {
            if (traduz[j].codigo == buffer && contbits == traduz[j].tamanho)
            {
                if (traduz[j].caractere == '!') // EOT
                    return;
                fputc(traduz[j].caractere, arqTexto);
                buffer= 0;
                contbits = 0;
            }
        }
        buffer <<= 1;
    }
}
}

```

descompacta: Descompacta um arquivo binário usando os códigos binários gerados a partir da árvore de Huffman. Primeiramente, acessa a tabela de codificação do arquivo .bin, que contém a quantidade de caracteres diferentes codificados, o símbolo de cada caractere, o código deles e o tamanho de cada um, tornando a descompacta independente da compacta. Durante a leitura dos códigos binários no arquivo binário, a função mantém um buffer para armazenar temporariamente os bits lidos do arquivo binário. Quando um código válido é encontrado no buffer, é achado um caractere que tenha o mesmo código e tamanho do código, o caractere é escrito no arquivo de texto de saída. O processo continua até que o código do EOT seja encontrado no arquivo binário, indicando o final do texto compactado.

6. Função adicional (para imprimir binários):

```
void imprimeBinario(unsigned int num, int tamanho) {
    for (int i = tamanho - 1; i >= 0; i--) {
        printf("%d", (num >> i) & 1);
    }
}
```

imprimeBinario: imprime um número em binário

1. Função principal do código (main):

```
int main(void) {
    FILE* arqTexto;
    arqTexto = fopen("texto.txt", "r");
    if (arqTexto == NULL) {
        printf("Erro ao abrir o arquivo.");
        return 1;
    }

    FILE* arqBin;
    arqBin = fopen("texto_bin.bin", "wb");
    if (arqBin == NULL) {
        printf("Erro ao criar o arquivo binário.");
        exit(1);
    }
}
```

```

}

// Criação do vetor de frequência e contagem das frequências
int vetorFrequencia[TAM];
criaVetorFrequencia(arqTexto, vetorFrequencia);
fclose(arqTexto);

// Inicialização da lista de frequência e preenchimento
FREQUENCIA* lista_frequencia = inicializa_lista_frequencia();
criar_lista_frequencia(vetorFrequencia, lista_frequencia);

// imprimindo a lista de frequencia
printf("Lista de Frequencia:\n");
imprime_frequencia(lista_frequencia);

// Construção da árvore de Huffman
NoTrie* raizHuffman = construirArvoreHuffman(lista_frequencia);

// Geração dos códigos de Huffman
CaractereCodigo caracteresCodificados[TAM];
int indiceCaracteresCodificados = 0;
codificarHuffman(raizHuffman,
caracteresCodificados,&indiceCaracteresCodificados);

// Impressão dos códigos de Huffman em hexadecimal e binario
printf("\nCodificacao de Huffman:\n");
for (int i = 0; i < indiceCaracteresCodificados; ++i) {
    printf("%c'- %u (" , caracteresCodificados[i].caractere,
caracteresCodificados[i].codigo);
    imprimeBinario(caracteresCodificados[i].codigo,
caracteresCodificados[i].tamanho);
    printf(") - %d\n", caracteresCodificados[i].tamanho);
}

// Compactação do arquivo
arqTexto = fopen("texto.txt", "r");
if (arqTexto == NULL) {
    printf("Erro ao abrir o arquivo.");
    exit(1);
}

```

```

    }
    compacta(arqTexto, arqBin, caracteresCodificados, lista_frequencia);
    fclose(arqBin);

    // Descompactação do arquivo
    arqBin = fopen("texto_bin.bin", "rb");
    FILE* arqTextoOut = fopen("texto_descompactado.txt", "w");
    if (arqBin == NULL || arqTextoOut == NULL) {
        printf("Erro ao abrir os arquivos.");
        exit(1);
    }

    descompacta(arqBin, arqTextoOut);

    fclose(arqBin);
    fclose(arqTextoOut);
    liberaLista(lista_frequencia);

    return 0;
}

```

main: Abre arquivo de texto que será lido, cria o vetor de frequência, inicializa e cria a lista de frequência, constrói a árvore de Huffman, codifica os caracteres, compacta o arquivo de texto em um arquivo binário, descompacta o arquivo binário em um novo arquivo teste, e por fim libera as memórias alocadas.

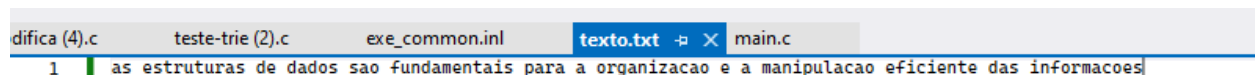
Observações e conclusões:

Fizemos diversos testes diferentes para ter certeza de que o código funciona certo. Para todos os testes, todas as saídas (prints com as frequências no texto original, print da tabela de conversão, o arquivo binário e o arquivo decodificado) foram as esperadas. Abaixo seguem alguns de nossos testes.

Vale considerar que as funções codifica e decodifica foram feitas para o trabalho de software básico nesse mesmo período. A fim de testá-las, usamos o linux com os comandos `hexdump -C "nome do arquivo texto de entrada"`, `hexdump -C "nome do arquivo binário"` e `hexdump -C "nome do arquivo texto de saída"` no terminal do linux.

- 1 teste:

Foi utilizado o seguinte arquivo texto:



```
difica (4).c  teste-trie (2).c  exe_common.inl  texto.txt  main.c
1 | as estruturas de dados sao fundamentais para a organizacao e a manipulacao eficiente das informacoes
```

A saída do programa foi:

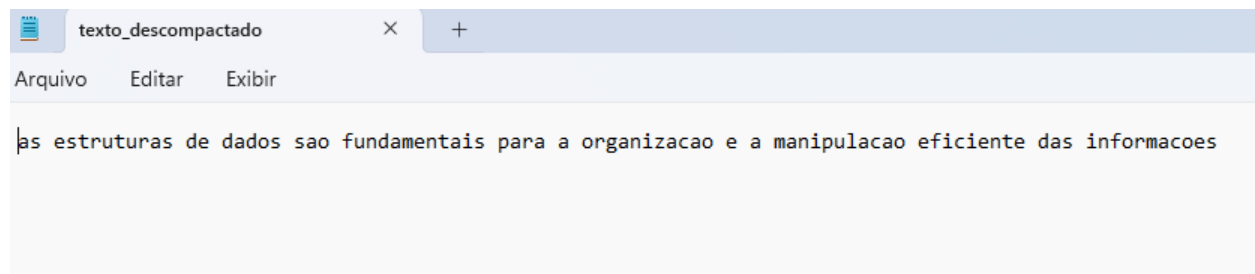
Lista de Frequencia:

```
'!'- 0
'g'- 1
'z'- 1
'l'- 1
'p'- 2
'm'- 3
'f'- 3
'u'- 4
't'- 4
'c'- 4
'r'- 5
'd'- 5
'n'- 6
'i'- 6
'o'- 7
's'- 8
'e'- 8
' '- 14
'a'- 18
```

Codificacao de Huffman:

```
'e'- 0 (0000) - 4
's'- 1 (0001) - 4
'u'- 4 (00100) - 5
't'- 5 (00101) - 5
'c'- 6 (00110) - 5
'p'- 14 (001110) - 6
'l'- 30 (0011110) - 7
'!'- 124 (001111100) - 9
'g'- 125 (001111101) - 9
'z'- 63 (00111111) - 8
'a'- 1 (01) - 2
'r'- 8 (1000) - 4
'd'- 9 (1001) - 4
'n'- 10 (1010) - 4
'm'- 22 (10110) - 5
'f'- 23 (10111) - 5
'i'- 12 (1100) - 4
'o'- 13 (1101) - 4
' '- 7 (111) - 3
```

O arquivo descompactado:



- 2 teste:

Foi utilizado o arquivo texto da música posta para testarmos no ead, A saída do arquivo texto descompactado deu como o esperado, resultando no mesmo conteúdo.

Lista de Frecuencia:

'!'- 0
'j'- 1
'z'- 1
'p'- 31
'v'- 39
'b'- 41
'g'- 52
'm'- 64
'f'- 78
'c'- 84
'w'- 105
'u'- 117
'k'- 124
'd'- 144
'y'- 146
's'- 160
'l'- 161
'r'- 171
'h'- 175
'
'- 189
'i'- 227
't'- 283
'n'- 302
'a'- 306
'o'- 370
'e'- 383
' '- 799

Codificacao de Huffman:

```
'r'- 0 (00000) - 5
'h'- 1 (00001) - 5
'o'- 1 (0001) - 4
'e'- 2 (0010) - 4
,
'- 6 (00110) - 5
'b'- 28 (0011100) - 7
'g'- 29 (0011101) - 7
'w'- 15 (001111) - 6
'i'- 8 (01000) - 5
'u'- 18 (010010) - 6
'k'- 19 (010011) - 6
'm'- 40 (0101000) - 7
'!'- 656 (01010010000) - 11
'j'- 657 (01010010001) - 11
'z'- 329 (0101001001) - 10
'p'- 165 (010100101) - 9
'v'- 83 (01010011) - 8
'd'- 21 (010101) - 6
't'- 11 (01011) - 5
'n'- 12 (01100) - 5
'y'- 26 (011010) - 6
's'- 27 (011011) - 6
'a'- 14 (01110) - 5
'l'- 30 (011110) - 6
'f'- 62 (0111110) - 7
'c'- 63 (0111111) - 7
' '- 1 (1) - 1
```