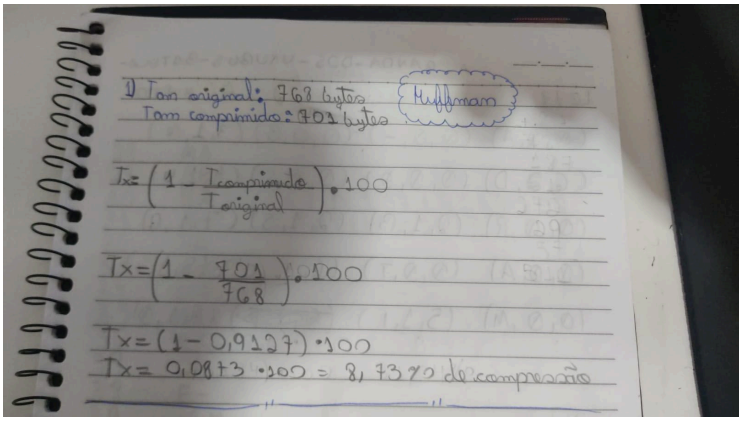
Pontifícia Universidade Católica de Minas Gerais Instituto de Ciências Exatas e Informática

Nome: Gabriel Henrique Vieira de Oliveira, Vinicius Cezar Pereira Menezes
Data: 25/11/2025
Matéria: Algoritmos e Estruturas de Dados 3
Link Github: https://github.com/GabrielHenrique20/TP_AEDS3

Formulário

1. Qual foi a taxa de compressão obtida com o algoritmo de Huffman?

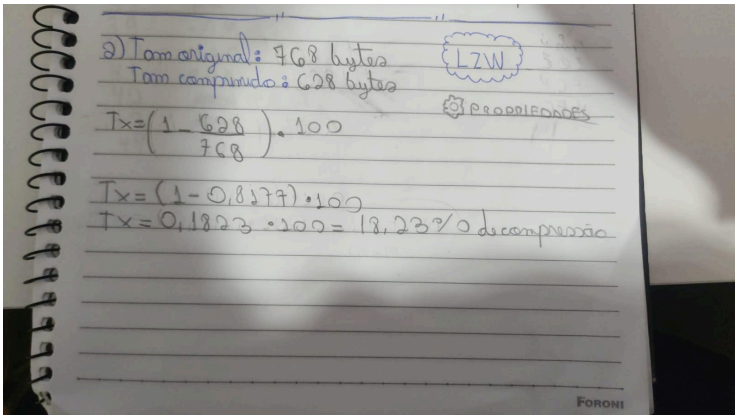
a. Tamanho do arquivo original;
b. Tamanho do arquivo comprimido;
c. Cálculo da taxa;



d. Interpretação do resultado;
O Huffman conseguiu comprimir os arquivos originais presentes em nosso projeto com uma taxa de 9% de compressão. Entretanto, há algumas observações a serem feitas: a compressão não foi feita de maneira eficiente como gostaríamos, tendo em vista que, em nosso código, o algoritmo precisa guardar metadados (tabelas de códigos das relações, em binário, cabeçalhos, etc.) que ocupam espaço mais do que era necessário (um dos erros presentes no nosso código, apontados pelo professor).

2. Qual foi a taxa de compressão obtida com o algoritmo de LZW?

a. Tamanho do arquivo original;
b. Tamanho do arquivo comprimido;
c. Cálculo da taxa;



d. Interpretação do resultado;
O LZW conseguiu comprimir a imagem de backup com uma taxa melhor que a anterior: de 18% de compressão. Porém, o ganho com a compactação poderia ter sido maior, tendo em vista que a quantidade de dados é bem pequena, diferentemente do que é melhor para o LZW onde, em volumes maiores, a tendência é aproveitar melhor os padrões repetidos e melhorar a taxa.

3. Quais dificuldades surgiram ao implementar Huffman e LZW e como você resolveu?

Definitivamente, uma das maiores dificuldades na implementação do Huffman e LZW foi entender como poderíamos criar uma classe capaz de elaborar métodos de compactação e descompactação dos dois algorítmos (entender como fazer todo esse conjunto em uma única classe).

Inicialmente, atribuímos em nosso projeto as classes de compactação disponibilizadas pelo professor Kutova em seu GitHub (Huffman, LZW e VetorDeBits que sofreram pequenas alterações, relacionadas aos bytes, para funcionarem corretamente dentro do nosso trabalho). Depois, pensamos na elaboração do MENU BACKUP, onde usuário seria capaz de selecionar: **compactar ou descompactar com ambos os algoritmos**. Porém, percebemos que era necessário uma classe backup capaz de armazenar e elaborar todas essas etapas. Teríamos que pensar um meio do código analisar todos os .db presentes no projeto, transformá-los em uma única imagem, colocá-los em um único vetor de bytes e realizar a compactação exclusivamente em nível arquivo, funcionando como um backup completo, o que de começo, foi complicado.

Contudo, após algumas tentativas, pesquisas com vídeos e sites e reuniões com colegas de fora, chegamos a um resultado satisfatório da nossa classe backup.java, que possui métodos de compactação e descompactação bem definidos, além de outras funções auxiliares extremamente importantes (principalmente **construirImagemBackup()**, responsável por montar um único vetor de bytes com todos os arquivos .db utilizados no sistema e seu complemento **listarArquivosDados()**, capaz de listar todos os arquivos de dados usados pelo sistema de viagens).

4. Justifique a escolha da estrutura de dados usada para armazenar as tabelas, dicionários e árvores utilizados pelos algoritmos.

<u>HUFFMAN: elaborados pelo professor Kutova</u>

Tabelas de frequências e códigos:

Usam mapas/dicionários que mapeiam cada byte -> frequência e byte -> código em bits.

- Motivos da escolha:
  Acesso rápido aos dados para uma busca por chave bem eficiente.
  Só armazenam os símbolos que realmente aparecem no arquivo (vai economizar espaço na parte lógica do código).
  Deixam o código mais simples e legível para contar frequências e consultar códigos na hora de comprimir.

Árvore de Huffman

Representada como uma árvore binária encadeada (nós com filho esquerdo e direito);
Construída usando uma fila de prioridade (**PriorityQueue**) ordenada pela frequência;

- Motivos da escolha:

O algoritmo de Huffman precisa, o tempo todo, dos dois menores nós; a fila de prioridade resolve isso naturalmente;

A árvore binária é a forma mais direta de representar os códigos prefixos (0 = esquerda, 1 = direita);
Facilita tanto a geração dos códigos quanto a decodificação dos arquivos.

```java
public static HashMap<Byte, String> codifica(byte[] sequencia) {
    HashMap<Byte, Integer> mapaDeFrequencias = new HashMap<>();
    for (byte c : sequencia) {
        mapaDeFrequencias.put(c, mapaDeFrequencias.getOrDefault(c, defaultValue: 0) + 1);
    }

    PriorityQueue<HuffmanNode> pq = new PriorityQueue<>();
    for (Byte b : mapaDeFrequencias.keySet()) {
        pq.add(new HuffmanNode(b, mapaDeFrequencias.get(b)));
    }

    while (pq.size() > 1) {
        HuffmanNode esquerdo = pq.poll();
        HuffmanNode direito = pq.poll();

        HuffmanNode pai = new HuffmanNode((byte) 0, esquerdo.frequencia + direito.frequencia);
        pai.esquerdo = esquerdo;
        pai.direito = direito;

        pq.add(pai);
    }

    HuffmanNode raiz = pq.poll();
    HashMap<Byte, String> codigos = new HashMap<>();
    constroiCodigos(raiz, codigo: "", codigos);

    return codigos;
}
```

```java
private static void constroiCodigos(HuffmanNode no, String codigo, HashMap<Byte, String> codigos) {
    if (no == null) {
        return;
    }

    // ignorava o byte 0 (alteramos aqui).
    // Nó folha: ambos os filhos são nulos. Isso permite códigos para qualquer byte,
    // incluindo o valor 0.
    if (no.esquerdo == null && no.direito == null) {
        codigos.put(no.b, codigo);
    }

    constroiCodigos(no.esquerdo, codigo + "0", codigos);
    constroiCodigos(no.direito, codigo + "1", codigos);
}
```

```java
// Versão buscando na tabela de códigos.
public static byte[] decodifica(String sequenciaCodificada, HashMap<Byte, String> codigos) {
    ByteArrayOutputStream sequenciaDecodificada = new ByteArrayOutputStream();
    StringBuilder codigoAtual = new StringBuilder();

    for (int i = 0; i < sequenciaCodificada.length(); i++) {
        codigoAtual.append(sequenciaCodificada.charAt(i));
        for (byte b : codigos.keySet()) {
            if (codigos.get(b).equals(codigoAtual.toString())) {
                sequenciaDecodificada.write(b);
                codigoAtual = new StringBuilder();
                break;
            }
        }
    }
    return sequenciaDecodificada.toByteArray();
}
```

LZW: elaborados pelo professor Kutova

Dicionário LZW (ArrayList<ArrayList<Byte>>):

Implementado como uma estrutura indexada por inteiros, onde cada posição guarda uma sequência de bytes correspondente a um código.

- Motivos da escolha:

No LZW, os códigos são números inteiros sequenciais (0, 1, 2, …); acessar por índice é direto e rápido.
O dicionário só cresce por inserção no final, o que combina bem com esse tipo de estrutura.

Cada entrada é uma “palavra” de tamanho variável (sequência de bytes), e a estrutura permite montar e estender essas sequências com facilidade.
Assim, o ArrayList<ArrayList<Byte>> casa perfeitamente com a lógica do LZW: dicionário indexado por código inteiro, que cresce linearmente ao longo da compressão/ descompressão.

```java
public static byte[] codifica(byte[] msgBytes) throws Exception {

    // Cria o dicionário e o preenche com os 256 primeiros valores de bytes
    // O dicionário será um vetor (ArrayList) de vetores de bytes.
    ArrayList<ArrayList<Byte>> dicionario = new ArrayList<>();
    ArrayList<Byte> vetorBytes; // elemento (vetor de bytes) para inclusão no dicionário
    int i, j;
    byte b;
    for (j = -128; j < 128; j++) { // Usamos uma variável int para o laço, pois, em uma variável byte,
                                   // 127 + 1 == -128
        b = (byte) j;
        vetorBytes = new ArrayList<>(); // Cada byte será encaixado no dicionário como um vetor de um único elemento
        vetorBytes.add(b); // Não é necessária a conversão explícita de byte para Byte
        dicionario.add(vetorBytes);
    }

    // Vetor de inteiros para resposta
    ArrayList<Integer> saida = new ArrayList<>();
// FASE DE CODIFICAÇÃO

i = 0;
int indice; // Índice: posição do vetor de bytes no dicionário
int ultimoIndice; // Indica o último índice encontrado no dicionário
while (i < msgBytes.length) {

    // Cria um novo vetor de bytes para acumular os bytes
    // Quanto maior for a sequência encontrada no dicionário, melhor será a
    // compressão
    vetorBytes = new ArrayList<>();

    // Adiciona o próximo byte da mensagem ao vetor de bytes, para busca no
    // dicionário
    // Obviamente, é esperado que esse vetor de um único byte já exista no
    // dicionário,
    // já que, algumas linhas acima, criamos o dicionário com todos os bytes
    // individuais possíveis.
    b = msgBytes[i];
    vetorBytes.add(b);
    indice = dicionario.indexOf(vetorBytes);
    ultimoIndice = indice;

    // Tenta acrescentar mais bytes ao vetor de bytes
    while (indice != -1 && i < msgBytes.length - 1) {

        i++;
        b = msgBytes[i];
        vetorBytes.add(b);
        indice = dicionario.indexOf(vetorBytes); // Faz nova busca

        if (indice != -1)
            ultimoIndice = indice;
    }

    // Acrescenta o último índice encontrado ao vetor de índices a ser retornado
    saida.add(ultimoIndice);

    // Acrescenta o último índice encontrado ao vetor de índices a ser retornado
    saida.add(ultimoIndice);

    // Acrescenta o novo vetor de bytes, com o último caráter que provocou a
    // falha na busca, ao dicionário (se couber)
    if (dicionario.size() < (Math.pow(a: 2, BITS_POR_INDICE) - 1))
        dicionario.add(vetorBytes);

    // Testa se os bytes acabaram sem provocar a codificação anterior
    if (indice != -1 && i == msgBytes.length - 1)
        break;
}

// Transforma o vetor de índices como uma sequência de bits
// Para facilitar a operação, escrevi o vetor do fim para o início
VetorDeBits bits = new VetorDeBits(saida.size() * BITS_POR_INDICE);
int l = saida.size() * BITS_POR_INDICE - 1;
for (i = saida.size() - 1; i >= 0; i--) {
    int n = saida.get(i);
    for (int m = 0; m < BITS_POR_INDICE; m++) { // apenas um contador de bits
        if (n % 2 == 0)
            bits.clear(l);
        else
            bits.set(l);
        l--;
        n /= 2;
    }
}

// Imprime os índices
System.out.println(x: "Índices: ");
System.out.println(saida);
System.out.println(x: "Vetor de bits: ");
System.out.println(bits);

// Retorna o vetor de bits
return bits.toByteArray();
```

Vetor de bits: elaborados pelo professor Kutova

Para guardar o resultado da compressão (no caso, todos os os bits gerados pelos algoritmos), foi usado um vetor de bits.

- Motivos da escolha:
  Armazena os dados de forma compacta, bit a bit, sem desperdício de espaço como ocorreria com String ou boolean[].
  Fornece operações prontas para ler, escrever e manipular bits em posições específicas.
  Isola a parte “baixo nível” (manipulação de bits e conversão para bytes), deixando o código dos algoritmos mais limpo e fácil de entender.

5. Qual campo foi escolhido para criptografia? Por quê?
O campo escolhido para a criptografia foi o email do usuário. Inicialmente, tivemos a ideia de criar um atributo senha para o nosso sistema, porém, analisando melhor e vendo que seriam necessárias várias mudanças na base do nosso código, o que poderia nos complicar a longo prazo, optamos por selecionar o email.

Motivos:

- O email só é exibido ou alterado nas telas de usuário (MenuUsuario), não participa de buscas, índices ou relações, ou seja, criptografar não quebra nada;
- É um texto curto, perfeito para RSA (não precisa de operações matemáticas como comparação/soma que você teria com orçamento ou datas);
- Ótimo para um CRUD simples e por ser um dos únicos dados do sistema claramente identificável/sensível ("afeta" o usuário).

6. Descreva como o RSA foi implementado no projeto.
Algoritmo retirado do site GeeksForGeeks (análise feita sobre ele):

a. Estrutura das chaves pública e privada;
Na classe criptografia.RSA:

São usados números inteiros grandes (BigInteger):
p = 7919 e q = 1009 (primos fixos);
n = p * q -> módulo da chave;
phi = (p - 1) * (q - 1);

A partir de phi:

Escolhe-se tal que 1 < e < phi e gcd(e, phi) = 1 (laço que incrementa o e até achar um primo próximo);
Calcula-se d = e^{-1} mod phi usando modInverse;

Estruturas:

Chave pública: par (e, n)
armazenada em publicE e modulusN;

Chave privada: par (d, n)
armazenada em privateD e modulusN;

```
// RSA Key Generation
static void generateKeys(BigInteger[] keys) {
    BigInteger p = new BigInteger(val: "7919");
    BigInteger q = new BigInteger(val: "1009");

    BigInteger n = p.multiply(q);
    BigInteger phi = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));

    // Choose e, where 1 < e < phi(n) and gcd(e, phi(n)) == 1
    BigInteger e = BigInteger.ZERO;
    for (e = new BigInteger(val: "2"); e.compareTo(phi) < 0; e = e.add(BigInteger.ONE)) {
        if (e.gcd(phi).equals(BigInteger.ONE)) {
            break;
        }
    }

    // Compute d such that e * d ≡ 1 (mod phi(n))
    BigInteger d = modInverse(e, phi);

    keys[0] = e; // Public Key (e)
    keys[1] = d; // Private Key (d)
    keys[2] = n; // Modulus (n)
}
```

b. Como e onde foram armazenadas;
As chaves não são gravadas em nenhum arquivo. Elas ficam somente em memória, na própria classe RSA, como atributos static:

```
// Deixar as chaves fixas durante a execução do programa
private static BigInteger publicE;
private static BigInteger privateD;
private static BigInteger modulusN;
```

Ou seja, o "armazenamento" das chaves é estático em memória, enquanto o programa está rodando.

c. Como foram carregadas pelo sistema;
Existe um bloco estático no RSA:

```
static {
    // Gera as chaves uma única vez, com p e q fixos (ver generateKeys)
    BigInteger[] keys = new BigInteger[3];
    generateKeys(keys);
    publicE = keys[0];
    privateD = keys[1];
    modulusN = keys[2];
}
```

Ou seja:

Na primeira vez que a classe RSA é carregada, o bloco estático roda;
Ele chama generateKeys(keys), gera -> e / d / n, a partir de p e q fixos;
Preenche os atributos publicE, privateD e modulusN.

Depois, qualquer chamada das classes criadas encryptString(String texto) e decryptString(String cifra) já usa essas chaves prontas.

d. Tamanho das chaves escolhidas e justificativa;
TAMANHO: p = 7919 e q = 1009:
n = 7.990.271, que tem 23 bits de comprimento (módulo pequeno para padrões reais de segurança).

- Esse tamanho reduzido é suficiente para demonstrar o algoritmo RSA funcionando;
- Mantém as operações (m^e mod n, c^d mod n) rápidas e fáceis de executar;
- Evita trabalhar com blocos grandes ou esquemas mais complexos de partição da mensagem.

e. Em qual momento a criptografia do(s) campo(s) ocorre (no CRUD);
A criptografia acontece na serialização do email, dentro da classe Usuario.java.

```
// email alteração feita para o RSA funcionar e criptografar o email
String emailClaro = (this.email == null ? "" : this.email);
String emailCriptografado = RSA.encryptString(emailClaro);
byte[] bytesEmail = emailCriptografado.getBytes(charsetName: "UTF-8");
dos.writeShort(bytesEmail.length);
dos.write(bytesEmail);

return baos.toByteArray();
```

No momento em que o CRUD grava o usuário no arquivo , o toByteArray() é chamado e o email é criptografado antes de ir para o .db (ocorre nas operações de inclusão e alteração do usuário, quando o objeto é convertido para bytes e persistido no arquivo .db).

f. Em qual momento ocorre a descriptografia;
A descriptografia acontece na desserialização do email, também dentro da classe Usuario.java.

```
// email lido criptografado e depois, descriptografado com RSA para funcionar
short tamEmail = dis.readShort();
byte[] bytesEmail = new byte[tamEmail];
dis.readFully(bytesEmail);
String emailCriptografado = new String(bytesEmail, charsetName: "UTF-8");
this.email = RSA.decryptString(emailCriptografado);
```

Em fromByteArray(), depois de ler id, nome e telefones, o código realiza um tratamento do email criptografado. Então, o email é lido no arquivo ainda criptografado (como string de números), depois é descriptografado com a classe criada descryptString() e por fim, o atributo this.email já fica em texto na instância de usuário (caso o usuário queira buscar por um Usuário, o email vai aparecer normalmente)

g. Conversões realizadas (ex.: string → bytes → blocos);
O fluxo de conversões acontece da seguinte maneira em cada um dos processos:

CRIPTOGRAFIA:

1) String (texto do email) -> vetor de byte (byte[]);
2) Para cada byte:
    - Converte o byte para um inteiro sem sinal (0 a 255);
    - Cria um BigInteger a partir desse inteiro;
    - Aplica o RSA (c = m^e mod n);
    - Converte c para string decimal e adiciona num StringBuilder, separando com espaço;
3) No fim, gera um string cifrada do tipo "12345 67890 43210 ...", onde cada número representa um caractere do e-mail cifrado.
4) Na classe Usuário, essa string cifrada vira novamente um vetor de byte para ser escrita no arquivo

```
public static String encryptString(String texto) {
    if (texto == null || texto.isEmpty())
        return "";
    try {
        byte[] bytes = texto.getBytes(charsetName: "UTF-8");
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < bytes.length; i++) {
            int unsigned = bytes[i] & 0xFF; // garante valor positivo 0..255
            BigInteger m = BigInteger.valueOf(unsigned);
            BigInteger c = encrypt(m, publicE, modulusN);
            sb.append(c.toString());
            if (i < bytes.length - 1)
                sb.append(str: " ");
        }
        return sb.toString();
    } catch (Exception e) {
        throw new RuntimeException(message: "Erro ao criptografar string com RSA", e);
    }
}
```

DESCRIPTOGRAFIA:

1) String cifrada lida do arquivo -> String em memória;
2) Quebra a string nos espaços;
3) Para cada parte quebrada:
    - Converte o número decimal para BigInteger;
    - Aplica RSA: m = c^d mod n;
    - Converte o resultado para byte;
4) No final, reconstrói a string original em UTF-8;

```java
/**
 * Descriptografa a String produzida acima restaurando o texto
 * original.
 */
public static String decryptString(String cifra) {
    if (cifra == null || cifra.isEmpty())
        return "";
    try {
        String[] partes = cifra.split(regex: " ");
        byte[] bytes = new byte[partes.length];
        for (int i = 0; i < partes.length; i++) {
            if (partes[i].isEmpty()) {
                bytes[i] = 0;
                continue;
            }
            BigInteger c = new BigInteger(partes[i]);
            BigInteger m = decrypt(c, privateD, modulusN);
            bytes[i] = (byte) m.intValue();
        }
        return new String(bytes, charsetName: "UTF-8");
    } catch (Exception e) {
        throw new RuntimeException(message: "Erro ao descriptografar string com RSA", e);
    }
}
```