UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
ESCOLA DE INFORMÁTICA APLICADA

ALGORITMO DE HUFFMAN:
IMPLEMENTAÇÃO E TESTES EM JAVA

JUAN GOMES SANT'ANNA
GABRIEL RAMIRO MESQUITA

RIO DE JANEIRO
AGOSTO DE 2022

SUMÁRIO

1 INTRODUÇÃO

Este trabalho foi desenvolvido pelos discentes Juan Gomes Sant'anna e Gabriel Ramiro Mesquita em cumprimento ao trabalho final de Estrutura de Dados I, disciplina ministrada pelo professor Pedro Nuno de Souza Moura no semestre letivo de 2022.1.

Esta atividade tem como objetivo o estudo do *Huffman Algorithm* e sua implementação na linguagem de programação Java, além de nossas observações em relação a complexidade computacional obtida a partir da forma de implementação abordada

2 ALGORITMO DE HUFFMAN E IMPLEMENTAÇÃO

Nesta seção, iremos descrever como funciona e qual é a finalidade do algoritmo de Huffman, e também como ele foi implementado usando a linguagem Java.

2.1 ALGORITMO DE HUFFMAN

O algoritmo de Huffman comprime dados com uma frequência que pode variar de 20% a 90% , dependendo das características dos dados que estão sendo comprimidos. Os dados são considerados como uma sequência de caracteres, ou seja, *strings*. Dessa forma cada caractere ocorre n vezes em todo o dado que vai ser comprimido, então n é denominado como a frequência do caractere (ou símbolo). É baseado nessas frequências que o algoritmo de Huffman elabora um modo ótimo para representar os símbolos como cadeia binária (CORMEN et al, 2012).

Este algoritmo representa os dados como cadeia binária com comprimento variável, onde cada caractere é representado de acordo com sua frequência. Por exemplo, caracteres mais frequentes em uma determinada cadeia de caracteres são representados com menos bits, enquanto que outros menos frequentes podem ser representados com cadeias de bits maiores. Sendo assim, uma forma mais otimizada de representar esses dados.

Além disso, os código binários gerados pelo algoritmo de Huffman são considerados como códigos prefixos, isso significa que nenhum código binário é um prefixo de outro código. Como nenhuma representação binária de um símbolo é prefixo de outro, isso elimina qualquer ambiguidade na hora de descompactar o arquivo.

O algoritmo usa uma fila de prioridade mínima (MinHeap), onde cada elemento do vetor é uma árvore binária e nó guarda os valores da frequência e o símbolo, então os elementos com a menor frequência são identificados e atribuídos como filhos de uma nova árvore onde a raiz guarda o valor da soma de suas frequências, e então é adicionada novamente à MinHeap. Esse processo ocorre até que reste apenas um nó da árvore na Heap, e no final teremos uma árvore onde os símbolos estão guardados nas folhas.

Por fim, o código binário de cada símbolo é construído como o caminho da raiz até o símbolo onde cada vez que o caminho vai para um filho a esquerda é adicionado o bit “0” ao código, e cada vez que o caminho vai para um filho a direita é adicionado o bit “1”

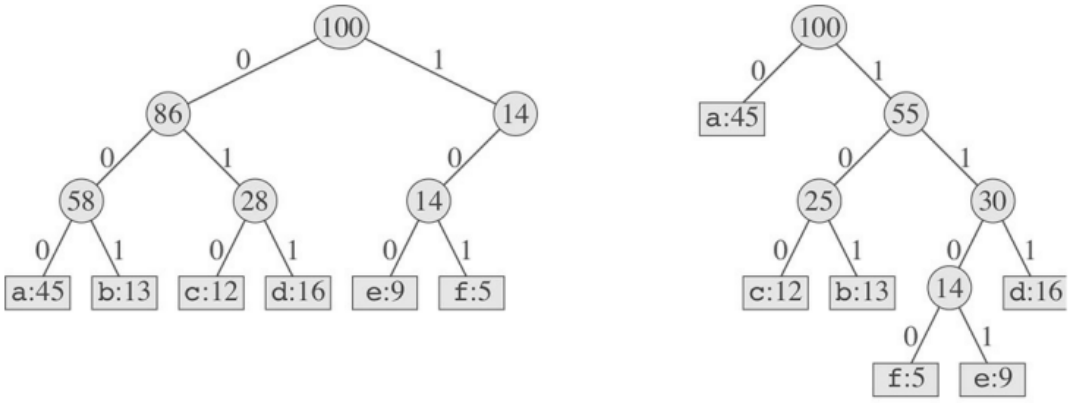


Figura 1 - Exemplo de árvore após utilização do algoritmo de Huffman

2.2 IMPLEMENTAÇÃO E MÉTODOS

Nesta seção, serão apresentados os métodos utilizados para a implementação do algoritmo de Huffman. A implementação foi feita na linguagem de programação Java.

2.2.1 Implementação da classe *Arvbin*

A classe *Arvbin* possui o atributo símbolo do tipo char, frequência do tipo int, e 2 atributos, esq e dir do tipo Arvbin. Os métodos construtores variam sendo um que recebe como parâmetro o símbolo e a frequência, e o segundo recebe como parâmetro além do símbolo e frequência, as árvores binárias esq e dir.

```
public class Arvbin
{
    protected char simbolo;
    protected int frequencia;
    protected Arvbin esq, dir;


    public Arvbin(char simbolo, int frequencia)
    {
        this.simbolo = simbolo;
        this.frequencia = frequencia;

    }

    public Arvbin(char simbolo, int frequencia, Arvbin esq, Arvbin dir)
    {
        this.simbolo = simbolo;
        this.frequencia = frequencia;
        this.esq = esq;
        this.dir = dir;
```

Figura 2 - *Print* dos métodos construtores de ***Arvbin***

O Método *mostra()* é um método recursivo que percorre a árvore em pré-ordem e imprime o valor da frequência de cada nó e caso esse nó seja uma folha além da frequência ele imprime o símbolo.

```
public void mostra()
{
    System.out.print(" ( "+ frequencia);
    if(esq == null && dir == null)
        System.out.print(" "+ simbolo);

    if(esq != null){
        this.esq.mostra();
    }
    if(dir != null){
        this.dir.mostra();
    }
    System.out.print(" ) ");

}
```

Figura 3 - Implementação do método ***mostra( )***

O método *mostraCodigo( )* recebe como parâmetro uma String (vazia) e concatena “0” ou “1” a essa string dependendo do caminho que faz até a folha (para esquerda concatena “0” e para direita concatena “1”). Com isso, ao chegar nas folhas, é exibida no *console* o símbolo e sua codificação associada.

```
public void mostraCodigo(String str)
{
    if(this.esq == null && this.dir == null){
        System.out.println("Simbolo: "+ simbolo + " Codificacao: " + str);
        return;
    }
    if(esq != null)
        esq.mostraCodigo(str + "0");
    if(dir != null)
        dir.mostraCodigo(str + "1");
}
```

Figura 4 - Implementação do método ***mostraCodigo( )***

2.2.2 Implementação da classe *MinBinaryHeap*

A classe *MinBinaryHeap* é o nome da classe que utilizamos para implementar os métodos e atributos referentes a Heap Mínima utilizada para realizar a execução do algoritmo de Huffman

```
public class MinBinaryHeap
{
    private int n;
    private int tamMaximo;
    private Arvbin[] vetor;
    private Arvbin raiz = null;

    public MinBinaryHeap(int tamanho)
    {
        n = 0;
        tamMaximo = tamanho;
        vetor = new Arvbin[tamanho+1];

    }

    public MinBinaryHeap(int tamanho, Arvbin[] v)
    {
        tamMaximo = tamanho;
        vetor = new Arvbin[tamanho+1];
        n = tamanho;

        for( int i = 0; i < tamanho; i++ )
            vetor[ i + 1 ] = v[ i ];

        constroiHeap();
    }
```

Figura 5 - Implementação de atributos e métodos construtores da MinBinaryHeap

O método *imprime()* itera sobre o vetor da Heap e imprime cada elemento contido nela. Sendo assim, exibindo símbolo e a frequência

```
public void imprime()
{
    System.out.println("Heap Binária: ");

    for(int i = 1; i <= n; i++)
        System.out.println("v["+ i +"] " + vetor[i].simbolo + " " + vetor[i].frequencia);

    System.out.println();
}
```

Figura 6 - Implementação do método ***imprime( )***

O método *removeMin()* retira e retorna o menor elemento da *Heap Mínima*, ou seja, sua primeira posição. Ele será utilizado para realizar as operações do algoritmo de Huffman

```
public Arvbin removeMin()
{
    Arvbin itemMin;

    if(this.vazia())
    {
        System.out.println("Fila de prioridades vazia!");
        return null;
    }

    itemMin = vetor[1];
    vetor[1] = vetor[n];
    n--;
    refaz( i: 1);

    return itemMin;
}
```

Figura 7 - Implementação do método ***removeMin( )***

O método *constroiHeap()* chama o método *refaz()* e passa como parâmetro a posição do vetor de i = n/2 até i= 1

```
private void constroiHeap()
{
    for( int i = n / 2; i > 0; i-- )
        refaz(i);
}
```

O método *refaz()* recebe como parâmetro um inteiro i, passado pelo método *constroiHeap( )*. Dessa forma o método refaz testa para o elemento na i-ésima posição do vetor, se ele satisfaz a condição de Heap Mínima, comparando seu valor com os seus filhos. Caso a condição não se satisfaça ele troca de posição com o filho de menor valor.

```
private void refaz(int i)
{
    Arvbin x = vetor[ i ];

    while(2*i <= n)
    {
        int filhoEsq, filhoDir, menorFilho;

        filhoEsq = 2*i;
        filhoDir = 2*i + 1;
        menorFilho = filhoEsq;

        if(filhoDir <= n)
        {
            if(vetor[ filhoDir ].frequencia < vetor[ filhoEsq ].frequencia)
                menorFilho = filhoDir;
        }
        if(vetor[ menorFilho ].frequencia < x.frequencia)
            vetor [ i ] = vetor[ menorFilho ];
        else
            break;

        i = menorFilho;
    }

    vetor[ i ] = x;
}
```

Figura 8 - Implementação do método ***refaz( )***

O método insere() recebe como parâmetro uma Arvbin, esse método verifica se a heap está cheia, caso esteja exibe uma mensagem ao usuário e caso não esteja ele acrescenta a árvore na heap e executa um loop para torná-la uma heap mínima novamente.

```
public void insere(Arvbin x)
{
    if (tamMaximo == n)
    {
        System.out.println("Fila de prioridades cheia!");
        return;
    }
    n++;
    int pos = n;

    vetor[0] = x;

    while(x.frequencia < vetor[pos/2].frequencia)
    {
        vetor[pos] = vetor[ pos/2 ];
        pos /= 2;
    }
    vetor[pos] = x;

}
```

Figura 9 - Implementação do método ***insere( )***

Este método chama um outro método na classe *Arvbin* que exibe a codificação encontrada. *mostraCodigo( )* foi descrito anteriormente com maiores detalhes

O método *carregaDados( )* é o método que, de forma manual, preenche a *heap mínima* com os símbolos e sua frequência. Nele, existe uma estrutura de repetição que irá ser repetida até que o contador i tenha um valor igual ao do tamanho da heap.

```
public void carregaDados() {
    Scanner scan = new Scanner(System.in);

    for(int i = 0; i < tamMaximo; i++){
        System.out.print("Digite o símbolo: ");
        char simbolo = scan.next().charAt(0);
        System.out.print("Digite a frequência: ");
        int frequencia = scan.nextInt();

        insere(new Arvbin(simbolo, frequencia));
    }
}
```

Figura 10 - Implementação do método ***carregaDados( )***

O método *aplicaHuffman( )* é o método que implementamos para executar o algoritmo de huffman. Nele, há um loop executado enquanto a quantidade de elementos na heap seja maior que 1. Nesse loop, o método remove os dois menores elementos da heap e os guarda em duas variáveis do tipo Arvbin. O primeiro elemento a ser removido, será o filho esquerdo e o segundo será o filho direito de uma nova árvore (Arvbin) que é criada representando a soma das frequências dos seus filhos. No algoritmo, essa árvore é inserida na *heap mínima,* através do método insere() e o atributo raiz da heap aponta para essa árvore criada. O tempo de execução dos passos citados anteriormente são medidos pela diferença dos valores das variáveis que guardam o tempo atual, em milisegundos (ms), uma variável é iniciada imediatamente antes do início do algoritmo de Huffman, e a segunda imediatamente depois. Por fim, o método retorna o somatório desse tempo medido todas as vezes que o algoritmo é executado, para fins de testes. Esses testes serão abordados mais à frente neste trabalho.

O método tem complexidade O(n log n), onde n é o número de símbolos (caracteres) distintos. Podemos observar que, se temos n caracteres distintos, o método *removeMin( )* obtemos uma complexidade computacional O(log n), pois a cada elemento retirado, a heap deve ser reconstruída. Ademais, é importante

salientar que o método *removeMin()* é chamado 2 * (n - 1) vezes em todo percurso do algoritmo de Huffman, fazendo a complexidade do algoritmo de huffman ser O(n log n).

Com isso, podemos perceber a partir da sua complexidade na notação big O que a quantidade de caracteres únicos (n) influenciam em seu tempo de execução.

```
public long aplicaHuffman() {
    long somatorio = 0;

    while (n > 1){
        long inicio = System.currentTimeMillis();
        Arvbin esquerdaElem = removeMin();
        Arvbin direitaElemen = removeMin();
        int somaFrequencia = esquerdaElem.frequencia + direitaElemen.frequencia;
        Arvbin novoNo = new Arvbin( simbolo: ' ', somaFrequencia);
        novoNo.esq = esquerdaElem;
        novoNo.dir = direitaElemen;
        insere(novoNo);
        raiz = novoNo;
        long termino = System.currentTimeMillis();
        somatorio += (termino - inicio);
        imprime();
    }
    raiz.mostra();
    System.out.println();
    return somatorio;
}
```

Figura 11 - Implementação do método ***aplicaHuffman( )***

O método *geraMinBinaryHeap()* recebe como parâmetro uma frequência total que todos os símbolos da Heap somados corrente devem ter. Então esse método automatiza a atribuição de símbolos e frequência para cada elemento da heap, limitando os caracteres de “!” até “~”. O método confere se já existe alguma árvore com o símbolo gerado, e caso haja ele refaz o símbolo. Caso não haja, ele cria uma nova árvore com o símbolo e frequência gerados automaticamente e a insere na heap através do método *insere().*

```
public void geraMinBinaryHeap (int frequenciaTotal){
    Random random = new Random();
    int frequencia, somaFrequencia = 0;
    char simbolo;
    boolean repetido = false;
    int aux = tamMaximo;
    int limite = frequenciaTotal - aux;
    for (int i = 0; i < tamMaximo; i++){
        do{
            if (aux > 1) {
                simbolo = (char) (random.nextInt( bound: 93) + '!');
                frequencia = random.nextInt( bound: (limite) + 1) + 1;
            }
            else {
                simbolo = (char) (random.nextInt( bound: 93) + '!');
                frequencia = frequenciaTotal - somaFrequencia;
            }
            for (int k = 1; k <= tamMaximo; k++){
                if (vetor[k] != null) {
                    if (Character.valueOf(vetor[k].simbolo).compareTo(simbolo) == 0) {
                        repetido = true;
                        break;
                    }

                    else
                        repetido = false;
                }
                else
                    break;
            }
        } while (repetido == true);
        somaFrequencia += frequencia;
        aux --;
        limite = frequenciaTotal - aux - somaFrequencia;
        Arvbin arvore = new Arvbin(simbolo, frequencia);
        insere(arvore);
    }
}
```

Figura 12 - Implementação do método ***geraMinBinaryHeap( )***

O método *repeteGeracao()* recebe como parâmetro um inteiro de quantidade e um inteiro como tamanho. Ele é responsável por automatizar as repetições dos testes de aplicação do algoritmo de Huffman. Esse método cria uma *minBinaryHeap* automática através do método *geraMinBinaryHeap()* de tamanho passado como parâmetro e executa o algoritmo de Huffman para essa *Heap Mínima* pela quantidade de vezes que for passada por parâmetro também. Além disso, ele guarda o tempo total que todas as repetições do algoritmo de Huffman levaram para serem executadas e retorna esse valor.

```java
public long repeteGeracao( int quantidade, int tamanho){
    long somatorio = 0;

    for (int i = 0 ; i < quantidade; i++) {
        geraMinBinaryHeap(tamanho);
        long tempoHuffman = aplicaHuffman( teste: "teste");
        this.n = 0;
        this.vetor = new Arvbin[this.tamMaximo + 1];
        somatorio += tempoHuffman;
    }
    return somatorio;
}
```

Figura 13 - Implementação do método ***repeteGeracao( )***

O método *imprimeTeste()* tem a finalidade de mostrar o resultado dos testes realizados. Ele cria 9 heap diferentes cada uma com a quantidade de símbolos especificadas no enunciado do trabalho (exceto a de 93 símbolos que foi criada, pois não existem 100 símbolos diferentes como pedido no enunciado). Então, ele executa o método *repeteGeracao()* para cada uma das heaps e o valor de tempo retornado pelo método é guardado em uma variável para cada heap. Os valores de repetição foram definidos como 100 mil repetições, para uma boa amostragem e o total de frequência dos símbolos em 100. Por fim, os valores são impressos.

```java
static void imprimeTeste(){
    MinBinaryHeap heap5 = new MinBinaryHeap( tamanho: 5);
    MinBinaryHeap heap10 = new MinBinaryHeap( tamanho: 10);
    MinBinaryHeap heap12 = new MinBinaryHeap( tamanho: 12);
    MinBinaryHeap heap15 = new MinBinaryHeap( tamanho: 15);
    MinBinaryHeap heap20 = new MinBinaryHeap( tamanho: 20);
    MinBinaryHeap heap30 = new MinBinaryHeap( tamanho: 30);
    MinBinaryHeap heap50 = new MinBinaryHeap( tamanho: 50);
    MinBinaryHeap heap80 = new MinBinaryHeap( tamanho: 80);
    MinBinaryHeap heap93 = new MinBinaryHeap( tamanho: 93);

    long tempo5 = heap5.repeteGeracao( quantidade: 100000,  tamanho: 100);
    long tempo10 = heap10.repeteGeracao( quantidade: 100000,  tamanho: 100);
    long tempo12 = heap12.repeteGeracao( quantidade: 100000,  tamanho: 100);
    long tempo15 = heap15.repeteGeracao( quantidade: 100000,  tamanho: 100);
    long tempo20 = heap20.repeteGeracao( quantidade: 100000,  tamanho: 100);
    long tempo30 = heap30.repeteGeracao( quantidade: 100000,  tamanho: 100);
    long tempo50 = heap50.repeteGeracao( quantidade: 100000,  tamanho: 100);
    long tempo80 = heap80.repeteGeracao( quantidade: 100000,  tamanho: 100);
    long tempo93 = heap93.repeteGeracao( quantidade: 100000,  tamanho: 100);
    System.out.println("O tempo total de 100 mil ciclos para 5 simbolos diferentes e de: "+tempo5+"ms");
    System.out.println("O tempo total de 100 mil ciclos para 10 simbolos diferentes e de: "+tempo10+"ms");
    System.out.println("O tempo total de 100 mil ciclos para 12 simbolos diferentes e de: "+tempo12+"ms");
    System.out.println("O tempo total de 100 mil ciclos para 15 simbolos diferentes e de: "+tempo15+"ms");
    System.out.println("O tempo total de 100 mil ciclos para 20 simbolos diferentes e de: "+tempo20+"ms");
    System.out.println("O tempo total de 100 mil ciclos para 30 simbolos diferentes e de: "+tempo30+"ms");
    System.out.println("O tempo total de 100 mil ciclos para 50 simbolos diferentes e de: "+tempo50+"ms");
    System.out.println("O tempo total de 100 mil ciclos para 80 simbolos diferentes e de: "+tempo80+"ms");
    System.out.println("O tempo total de 100 mil ciclos para 93 simbolos diferentes e de: "+tempo93+"ms");
}
```

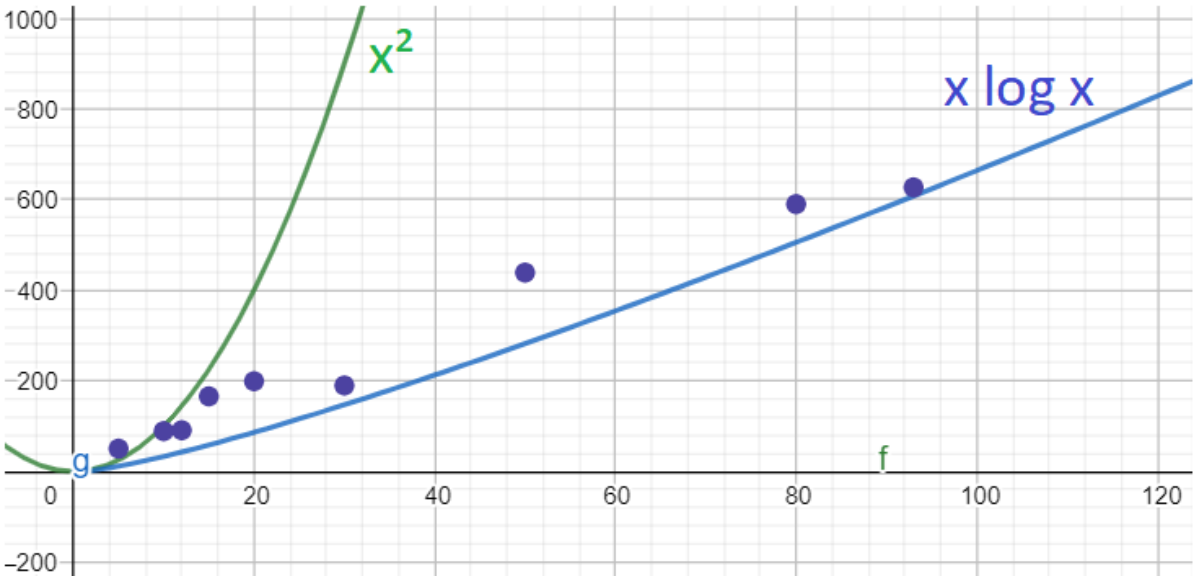
Figura 14 - Implementação do método ***imprimeTeste( )***

3 RESULTADOS

.A seguir encontra-se a tabela com o tempo de execução do algoritmo de Huffman para cada heap de quantidade de símbolos diferentes. O teste foi realizado 5 vezes e o tempo médio obtido.

Resultados Teste Trabalho ED1 100 mil ciclos por teste						
Quantidade de simbolos	**Teste 1 Tempo (ms)**	**Teste 2 Tempo (ms)**	**Teste 3 Tempo**	**Teste 4 Tempo (ms)**	**Teste 5 Tempo (ms)**	**Média**
5	53	51	48	57	43	50,4
10	87	98	80	86	94	89
12	79	91	88	89	108	91
15	130	143	173	176	207	165,8
20	186	182	180	231	217	199,2
30	191	196	188	168	206	189,8
50	285	303	304	980	323	439
80	705	531	631	502	582	590,2
93	659	576	684	592	625	627,2

A seguir esses dados foram plotados no geogebra assim como as curvas de x^2 e xlog(x), para que os resultados fossem comparados a essas curvas e a complexidade de execução fosse analisada. No eixo vertical encontram-se os valores de tempo em milissegundos e no eixo horizontal a quantidade de símbolos.



4 CONCLUSÕES

Ao analisar o gráfico gerado verificamos que os resultados se aproximam da curva de x log x e não de x^2, indicando que a complexidade esperada foi almejada. Porém, é observável também que os valores registrados encontram-se acima da curva, o que não deveria ocorrer, já que o limite assintótico superior é O(n log n). Uma das possíveis causas para o tempo de execução ter essa variedade pode se dar pelos processos que o computador poderia estar executando no momento dos testes executados no nosso trabalho.

Sendo n como n caracteres distintos presentes na heap, é interessante ressaltar que a frequência dos caracteres não influencia no tempo de execução analisado.

Ademais, é importante ressaltar uma curiosidade na execução deste trabalho. No enunciado do trabalho, foram requeridas entradas de até 100 caracteres distintos, porém ao verificar na tabela ASCII, verifica-se que há somente 93 caracteres, sendo os outros representações de comandos, o que não é interessante para abordagem deste trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/
https://www.delftstack.com/pt/howto/java/java-random-character/
https://www.programiz.com/dsa/huffman-coding
https://computerscience360.files.wordpress.com/2018/02/algoritmos-teoria-e-prc3a1tica-3ed-thomas-cormen.pdf.