Algoritmos e Estruturas de Dados

1ª Série

(parte 1)

Primeira série de problemas

N. 45827 Nome Daniel Azevedo

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2019/2020

13/04/2020

Índice

Conteúdo

Introdução

Este trabalho foi proposto no âmbito da unidade curricular de Algoritmos e Estruturas de Dados com o intuito de abordar a criação de algoritmos de uma forma mais eficiente, tendo em conta os seus custos.

Este trabalho encontra-se dividido em 3 partes. Na primeira parte, era necessário criar alguns algoritmos propostos, tendo em conta o custo exigido ou preferencial.

Na segunda parte, tive que responder a algumas perguntas sobre:

- Equações de recorrência;
- Complexidade de algoritmos;
- Custo dos algoritmos;

Na terceira parte, foi-me pedido para resolver o problema “as k palavras mais próximas”. Este teria que seguir um conjunto de regras lógicas descritas no enunciado.

1. Algoritmos Elementares

Exercício 1

Considere o método

public static int printEachThreeElementsThatSumTo(int[] v, int l, int r, int s)

que dado o sub*array* (v, l, s) e o inteiro s, apresenta na consola todos os triplos de elementos distintos desse *array* tal que a sua soma seja igual a s. O método retorna o número de triplos encontrados.

1.1) Realize uma implementação usando um algoritmo com custo assimptótico $O(n^3)$.

- **Análise do problema**

Para este exercício foi utilizada uma cadeia de três *for* para percorrer o array e fazer um somatório de três elementos diferentes, o qual no final é comparado com o valor do *int s*. Caso seja semelhante, é incrementado o valor *int count.*

Tal como solicitado, este código têm um custo $O(n^3)$. Sendo n o número de elementos do *array*, obtêm-se a seguinte expressão:

*O(n)*O(n)*O(n)*

Como tal o resultado final será $O(n^3)$

- **Código Java**

```
if(l>=r) return 0;
        int count = 0;
        for (int i=l; i <= r; i++) {
           if(v[i] < s) {
              for(int j=i+1; j<=r; j++) {
                 if(v[i]+v[j] <= s) {
                    for (int k=j+1;k <= r; k++) {
                       if(v[i]+v[j]+v[k] == s) {
                          count++;
                       }
                    }
                 }
              }
           }
        }
        return count;
```

1.2) Realize uma implementação usando um algoritmo com custo assimptótico $O(n^2 \log n)$.

- **Análise do problema (parte 1)**

Este exercício ficou dividido em três fases (métodos) diferentes. Comecei por ordenar o *array* utilizando o algoritmo *InsertionSort*. Este algoritmo têm um custo $O(n^2)$. No pior caso tanto o *for* como o *while* vão percorrer todo o *array*. Nesta situação temos o seguinte custo:

O(n)*O(n)

Abaixo segue o código do algoritmo de ordenação *sort*.

- **Código Java(parte 1)**

```
public static int[] sort (int[]v, int l, int r){
        for(int j=2; j<v.length-1; j++) {
           int key=v[j], i=j-1;
           while (i>0 && v[i]>key) {
              v[i+1] = v[i];
              i=i-1;
```

```
            }
            v[i+1] = key;
        }
        return v;
    }
```

- **Análise do problema (parte 2)**

Seguidamente foi colocado dois *for* encadeados, o primeiro *for* irá percorrer o *array a* começar em *l*. O segundo *for* bloqueia o valor seguinte. Depois é chamado a função do algoritmo *binarySearch,* o qual foi alterado de forma a adaptar-se a este exercício. O algoritmo *searchLast* têm um custo O(log n), como um *binarySearch*.

Como tal, a equação final do custo deste exercício será:

$O(n^2)+O(n)*O(n)*O(\log n) = O(n^2)+O(n^2 \log n) = O(n^2 \log n)$

Abaixo poderá ser verificado os códigos para o algoritmo *searchLast*.

- **Código Java (parte 2)**

```
public static int searchLast(int[] v, int l, int r, int first, int second, int s) {
    int m = (l+r)/2;
    if(l>=r) {
       if(first + second + v[m] == s && v[m] != first && v[m] != second) return 1;
       else return 0;
    }
    if (first + second + v[m] == s) return 1;
    else if (first + second + v[m] < s) return searchLast(v, m+1, r, first, second, s);
    else return searchLast(v, l, m-1, first, second, s);
 }
```

- **Código Java (parte 3)**

```
if(l>=r) return 0;
int count = 0;
v = sort (v,l,r);
for(int i=l; i<r;i++) {
    if(v[i]<s) {
        for (int j=i+1;j<r;j++) {
            if(v[i]+v[j]<= s && v[i] != v[j]) {
                count+=searchLast(v,j+1,r,v[i],v[j],s);
             }
        }
     }
}
return count;
```

1.3) Realize uma implementação usando um algoritmo com custo assimptótico O(n2).

Análise do problema

Neste exercício tentei replicar o mecanismo do 1.1. Usei dois for para percorrer o array, e uma variável fixa (que incrementava quando os dois *for* acabassem de percorrer) com a mesma função do primeiro *for* do exercício 1.1. Este algoritmo têm um custo $O(n^2)$. Nesta situação temos o seguinte custo:

O(n)*O(n)

- **Código Java**

```
if(l>=r) return 0;
int count = 0;
int aux = l;
for(int i = aux+1; i <= r; i++){
    if(i >= r-1 && aux <= r){
        aux++;
        i = aux+1;
    }
    if(v[aux]+v[i] <= s) {
            for (int j = i+1; j <= r; j++) {
                    if (v[j] + v[i] + v[aux] == s) {
                            count++;
                    }
            }
        }
}
return count;
```

Exercício 2

Realize o método

public static int removeIndexes(int[] v, int l, int r, int[] vi, int li, int ri)

que retira da sequência representada pelos inteiros do *subarray* (v, l, r) os elementos cujos índices originais estão presentes no *subarray* (vi, li, ri), que está ordenado de forma crescente. A sequência resultante fica contida de forma contígua nas primeiras posições do *subarray* (v, l, r) mantendo a ordem relativa dos elementos. O valor retornado pelo método é a dimensão da sequência resultante. Valorizam-se as soluções que não criem *arrays* auxiliares e que tenham complexidade O(n), com n = r − l + 1.

- **Análise do problema**

Inicialmente implementei o código para as exceções a partir de dois *if*. Ambos os *if* vão verificar se os *arrays* se encontram vazios, retornando zero ou o tamanho do array v, no caso do array v ou vi estarem vazios.

Depois criei a variável driver, a qual compensa o número de vezes que foi efetuado uma remoção. A variável index, que irá servir de índice para percorrer o array vi.

Criei um *for* para percorrer o *array* v entre os dois parâmetros l e r. Sempre que o valor da variável que percorre o array v é diferente do valor que estamos a ler no array vi, este será guardado no array v, na posição bus.
Caso contrário, verifico se a variável index é menor que a variável ri, e em caso positivo incremento o index. No final o valor de retorno é bus.

O algoritmo desenvolvido têm um custo O(n), tendo em conta que n é igual a r menos l.
Como o enunciado solicita, não foi utilizado *arrays* auxiliares para obter esta solução.

- **Código Java**

```
public static int removeIndexes(int v[], int l, int r, int[] vi, int li, int ri) {
        if (v.length <= 0) {
            return 0;
        }
        if (li > ri) {
            return v.length;
        }
        int bus = 0;
        int index = li;
        for (int j = l; j <= r; j++) {
                if (vi[index] != j) {
                    v[bus] = v[j];
                    bus++;
                }
                else if(index<ri){
                    index++;
                }
         }
          return bus;
}
```

Exercício 3

Realize o método

public static String greaterCommonPrefix(String[] v, int l, int r, String word)

que, dado o subarray (v, l, r), ordenado de modo crescente e a palavra word, retorna a palavra presente no subarray (v, l, r) que contenha o maior prefixo que seja comum a um prefixo da palavra word. Em caso de empate, retorna a string que se encontra na posição de maior valor no subarray (v, l, r). Caso o subarray (v, l, r), não contenha palavras, o método deverá retornar null.

- **Análise do problema**

Criei inicialmente o caso excecional com um *if.* Este exigia na hipótese de ser enviado um *array* vazio (como parâmetro) para a função, esta retornar null.

Criei 3 variáveis, 2 do tipo *string* e uma do tipo *int*. A *string* test vai servir como teste durante a comparação (word e *string* do *array*), e concatenação (caso a letras correspondam a word). O *int* letra irá incrementar de forma a percorrer letra a letra as palavras a serem comparadas.
A *string* finalWord serve como variável de retorno para a palavra assumida com solução.

Foi utilizado dois *for* encadeados para a solução do problema. O primeiro vai desde l até r (no entanto foi verificado pelos testes que mesmo a começar em zero os resultados deveriam ser positivos).

No primeiro ciclo verifico se a primeira letra da palavra em array v e da palavra word são equivalentes. Em caso afirmativo, entra-se num *if* que verifica se o tamanho da palavra que irá ser comparada, é maior do que a palavra que se encontra na variável test (melhor resultado até à data).

O segundo ciclo é composto por um *while* que utiliza a variável letra para percorrer a palavra do *array,* ao mesmo tempo que a palavra word. Dentro do ciclo caso as letras correspondam, é feita a concatenação da letra à variável test.

Caso a dimensão da *string* test seja menor que a dimensão da string aux, é efectuada uma reposição com o último valor obtido como test. Igualo assim a varíavel test a aux, e coloco a variável letra novamente a 0, interrompendo o ciclo. Caso esse *else if* não se verifique, apenas é incrementado o valor de letra.

Na parte final do programa verifico se a dimensão de test é maior que zero e, no caso de ser, atribui-se a finalWord o que está contido em test.
Por fim retornamos a variável finalWord, ou a última palavra do array v no caso de finalWord estar vazia.

Ao analisarmos o custo deste algoritmo verificamos que no primeiro ciclo iremos percorrer o array de l a r, e no segundo ciclo será percorrido o tamanho da palavra word.

Considerando n o tamanho do *array*, o primeiro ciclo terá um custo O(n). A este multiplica-se o custo do segundo ciclo.

O custo do segundo ciclo irá depender do tamanho da palavra. No entanto para n com valor alto, a palavra irá ser menor que o tamanho do *array*. Como é um valor menos em comparação com n, associo a O(1).

Teremos então:

$$O(n)*O(1) = O(n)$$

Logo para n grande, este algoritmo terá um custo assimptótico O(n) (é um Algoritmo Linear).

- **Código Java**

```
public static String greaterCommonPrefix(String[] v, int l, int r, String word) {
    if(l>r) return null;
    int letra=0;
    String test="", resultWord="";
    for (int i = 0; i < r; i++) {
        if (v[i].charAt(0) == word.charAt(0)) {
            if (v[i].length() > test.length()) {
                String aux = test;
                test = "";
                while (letra < v[i].length() && letra < word.length()) {
                    if (v[i].charAt(letra) == word.charAt(letra)) {
                        test += v[i].charAt(letra);
                    }
                    else if (test.length() < aux.length()) {
                        test = aux;
                        letra = 0;
                        break;
                    }
                    letra++;
                }
                if (test.length() > 0) {
                    resultWord = v[i];
                    letra = 0;
                }
            }
        }
    }
    if (resultWord == "")
        return v[v.length - 1];
    return resultWord;
}
```

Exercício 4

Realize o método

public static int sumGivesN(int n)

que, dado um número inteiro positivo n, escreve no standard output todas as sequências de números consecutivos cuja soma dê n. Este método retorna o número de resultados encontrados. Por exemplo, quando n é 24, existem
duas sequências: a sequência 7, 8 e 9, e a sequência 24.
Valorizam-se as soluções com tempo O(n).

- **Análise do problema**

Visto que o próprio n será sempre uma sequência, e n poderá ser 0, então inicializei a variável de retorno a 1.

Dentro do *for*, itero todos os números de ordem crescente até a soma dar n ou maior que n. Se for n, é impressa a sequência de números e incrementado o contador de sequências. Se for maior, apenas incrementa o primeiro número que irá formar a nova sequência a testar.

Como é um *for,* que itera n números e um *while* que incrementa n vezes perfaz um algoritmo com custo O(n)*O(n) = O(n^2).

- **Código Java**

```
public static int sumGivenN(int n){
	int res=1;
	for(int i=1; i<n; i++) {
		int sum=0, number=i;
		String sequence = "";
		while((sum+=number++)<n){
		   sequence += number+" ";
		};
		if(sum==n) {
		   System.out.println(sequence);
		  res++;
		}
	}
	System.out.println(n);
	return res;
}
```

Exercício 5

Realize o método estático

public static int deleteMin(int[] maxHeap, int sizeHeap)

que, recebendo um max-heap de elementos distintos e de dimensão sizeHeap, remove se possível, o menor elemento presente no heap. O método retorna a nova dimensão do heap. O array maxHeap poderá ser modificado. Indique justificando, a complexidade do algoritmo.

- **Análise do problema**

Para resolver este exercício contruí uma classe Heap, que possui os algoritmos dispostos no PowerPoint (HeapSort) da cadeira. Usei dois algoritmos do heapSort (ordena) e maxHeapify (constrói maxHeap).

Criei uma condição inicial para verificar o tamanho do heap fornecido como parâmetro, caso a dimensão fosse inferior a 1, este retornaria 0.

Posteriormente adotei o algoritmo heapSort de forma a ordenar por ordem crescente o heap fornecido. A seguir utilizo a variável int aux para guardar o valor da última posição do *array* (maior número). Depois copio esse valor para a primeira posição do *array*, e chamo maxHeapify para reconstruir o heap.

Tive insucesso em dois testes, por consequência de posições no heap estarem ordenadas de forma diferente.

No final retorno o tamanho do sizeHeap menos um (valor mínimo alterado).

Tendo em conta que os algoritmos heapSort e maxHeapify têm um custo O(n log n) e O (log n) respectivamente, temos então:

O(n log n) + O (log n) = O(n log n)

Tendo como custo final deste algoritmo O(n log n).

- **Código Java**

```
public static int deleteMin(int[] maxHeap, int sizeHeap) {
    if(sizeHeap<=1)return 0;
    Heap.heapSort(maxHeap, sizeHeap);
    int aux=maxHeap[sizeHeap-1];
    maxHeap[sizeHeap-1]=maxHeap[0];
    maxHeap[0]=aux;
    --sizeHeap;
    return sizeHeap;
}
```

- **Código Java Heap**

```
public static void maxHeapify(int[] h, int i, int n) {
   int l = 2 * i + 1;
   int r = 2 * i + 2;
   int largest;
   if (l < n && h[l] > h[i])
      largest = l;
   else largest = i;
   if (r < n && h[r] > h[largest])
      largest = r;
   if (largest != i) {
      exchange(h, i, largest);
      maxHeapify(h, largest, n);
   }
}

private static void exchange(int[] h, int i, int j) {
   int aux = h[i];
   h[i] = h[j];
   h[j] = aux;
}

public static void buildMaxHeap(int[] h, int n) {
   for (int i = n / 2 - 1; i >= 0; i--)
      maxHeapify(h, i, n);
}

public static void heapSort(int[] h, int n) {
   buildMaxHeap(h, n);
   for (int i = n - 1; i >= 1; i--) {
      exchange(h, i, 0);
      maxHeapify(h, 0, i);
   }
```

2.Análise de Desempenho

Exercício 1

1.1

1. $\begin{cases} C(0) = C(1) = O(1) & ,n = 0 \\ C(n) = C\left(\frac{N}{2}\right) + C\left(\frac{N}{2}\right) + O(n) & ,n > 0 \end{cases}$

2. $\begin{cases} C(0) = C(1) = O(1) & ,n = 0 \\ C(n) = C\left(\frac{N}{2}\right) + O(n) & ,n > 0 \end{cases}$

1.2

Algoritmos com equação de recorrência semelhantes:

1.1 MergeSort (Divide-and-Conquer)

1.2 Binary Search

Exercício 2

2.1)

Código do algoritmo	Instrução	Nº de vezes
public static long xpto(longx, long n){	----------	----------
if (n == 0) return 1;	**C1**	**1**
if (n % 2 == 0) return xpto(x, n/2) * xpto(x, n/2);	**C2**	**1**
return xpto(x, n/2) * xpto(x, n/2) * x;	**C3**	**1**
}	----------	----------

A sua equação de recorrência é:

$$\begin{cases} C(0) = 1 & ,n = 0 \\ C(n) = 2 * C(n/2) & ,n > 0 \end{cases}$$

C(n) = O(1) + 2 * C(n/2) = 2 * C(n/2)

2.2) Sim, é possível. Por exemplo, se em if(n%2 == 0) e na instrução de return usarmos apenas uma vez a expressão xpto(x, n/2), é escusado calcular o algoritmo uma segunda vez. O código ficaria desta forma:

```
public static long xpto(long x, long n){
      if (n == 0) return 1;
      int result;
      if (n % 2 == 0) return (result = xpto(x, n/2))*result;
      return (result = xpto(x, n/2)) * result * x;
}
```

Exercício 3

O Θ (theta) é a representação em simultâneo do limite superior e inferior do algoritmo ao longo do tempo consoante o número de elementos. De forma a que o algoritmo pertença a O(n^2), mas não a Θ(n^2), impõe que o seu melhor caso – Ω – seja diferente do seu pior caso. Para conseguir assim variar o Θ.

3.1

Um algoritmo com esse requisito é o QuickSort. No seu pior caso é O(n^2) e no seu melhor caso Ω(n log n), resultando assim em Θ(n log n).

3.2

Não existe nenhum algoritmo lecionado em aula (MergeSort, HeapSort..) com estas características. Todos os algoritmos de ordenação O(n log n) são Ω(n log n).

Exercício 4

Dos 4 algoritmos enunciados, é parcimonioso o:

- InsertionSort, embora cada elemento seja comparado com todos os outros, esse elemento após comparação é colocado na sua posição final, não permitindo desta forma uma nova comparação entre pares anteriormente comparados.
- SelectionSort, à medida que o array é percorrido é selecionado o menor valor respetivamente, incrementado no final o índice a percorrer. Desta mesma forma, também o SelectioSort apenas compara cada par de elementos uma única vez.

- MergeSort, após a divisão do *array* por elementos, a junção é feita através da comparação de pares uma única vez.

No HeapSort, na segunda fase de construção é necessário desenvolver max Heap (inserir o maior elemento na 1ª posição), o que requer uma nova comparação entre pares já comparados. Logo, não pode ser parcimonioso.

3.Problema: as k palavras mais próximas

Descrição da aplicação

Esta aplicação permite determinar um número de palavras (definido pelo utilizador) mais próximas de um certo prefixo (também definido pelo utilizador). Estas podem estar num ou mais ficheiros.

Foi utilizado para o desenvolvimento desta aplicação o algoritmo de minHeap, adaptado para *Strings.* As verificações que ocorrem em BuildMinHeap e MinHeapify são feitas com base na variável int prefixCount de cada palavra.

Existe também a class Organizer, que têm como objetivo fazer pesquisa dentro do array organizado de MinHeap. Esta decisão foi tomada de forma a diminuir o custo da aplicação, pois em toda ela não são utilizados algoritmos de ordenação, apenas métodos de construção de Heaps. Este método têm um custo O(n), sendo n o tamanho do array.

Apesar destas alterações, nenhum destes métodos teve o seu custo alterado, ou seja, BuildMinHeap continua a ter um custo de O(n) e o método MinHeapify um custo O(log n).

- **Class teste**

Este código é responsável por inicializar o programa com as respetivas diretorias transformadas em string, para serem enviadas argumentos para a função MaisProximas. É usado a biblioteca java.util.Date de forma a termos o tempo de execução do programa.

```
public static void main (String [] args) throws IOException{
    long initTime = System.currentTimeMillis();
    String f1path = "C:\\Users\\azeve\\Desktop\\LEIC\\Universidade\\4º Semestre\\AED\\FilesEx3\\f1.txt";
    String f2path = "C:\\Users\\azeve\\Desktop\\LEIC\\Universidade\\4º Semestre\\AED\\FilesEx3\\f2.txt";
    String f3path = "C:\\Users\\azeve\\Desktop\\LEIC\\Universidade\\4º Semestre\\AED\\FilesEx3\\f3.txt";
    String outpath = "C:\\Users\\azeve\\Desktop\\LEIC\\Universidade\\4º Semestre\\AED\\FilesEx3\\output.txt";
    String[] s = {"1000000", "lua", outpath, f1path, f2path, f3path};
    MaisProximas.main(s);
```

```
    long elapsedTime = System.currentTimeMillis()-initTime;

    System.out.println(new SimpleDateFormat("mm:ss.SSS").format(new Date(elapsedTime)));
  }
```

- **Class AvaliadorDePalavra**

Este é o algoritmo que têm como função contar o número de letras semelhantes (int prefixCounter) ao prefixo em casa palavra.

```
public class AvaliadorDePalavra {
  private String word;
  private int prefixCounter;

  public AvaliadorDePalavra(String  word, String prefix) {
    this.setWord(word);
    this.setPrefixCounter(prefixCounter(word, prefix));
  }

  public int prefixCounter(String word, String prefix) {
    int value=1;
    for(int i=1; i<prefix.length();i++) {
      if(word.charAt(i)==prefix.charAt(i)) value++;
      else break;
    }
    return value;
  }

  public int getPrefixCounter() {
    return prefixCounter;
  }

  private void setPrefixCounter(int prefixCounter) {
    this.prefixCounter = prefixCounter;
  }

  public String getWord() {
    return word;
  }

  public void setWord(String word) {
```

```
        this.word = word;
    }
}
```

- **Class MaisProximas**

```
public static void main (String []args) throws IOException{

    int count = Integer.parseInt(args[0]);

    final String prefix = args[1];

    AvaliadorDePalavra[] end = new AvaliadorDePalavra[count];

    for (int i = 3; i < args.length; i++) {
        BufferedReader reader = new BufferedReader(new FileReader(args[i]));
        try {
            String line="";
            int lineDelimiter=0;
            while ((line=reader.readLine()) != null) {

                System.out.println("Line: "+lineDelimiter);
                lineDelimiter++;
                if(line.charAt(0)==prefix.charAt(0)) {
                    AvaliadorDePalavra word = new AvaliadorDePalavra(line,prefix);
                    if (count>0) {

                        end[--count]=word;
                        if(count==0) StringHeap.buildMinHeap(end, end.length);
                    }
                    else {
                        if (word.getPrefixCounter() > StringHeap.minimum(end)) {
                            if (StringHeap.organizer(end, 0, word) == false) {
                                end[0] = word;

                                StringHeap.minHeapify(end, 0, end.length);
                            }
                        }
                    }
                }
```

```
            }
        } finally {
            reader.close();
        }
    }
    PrintWriter fileWriter = new PrintWriter(args[2]);

    for(int z=0; z<end.length-1; z++) {
        if(end[z]==null){
            ++z;
        }else{
            fileWriter.println(end[z].getWord());
        }
    }
    fileWriter.close();
}
```

- **Class StringHeap**

```
public static int minimum (AvaliadorDePalavra[] h) {
    return h[0].getPrefixCounter();
}

private static void exchange(AvaliadorDePalavra[] h, int i, int j) {
    AvaliadorPalavra aux = h[i];
    h[i] = h[j];
    h[j] = aux;
}

public static void buildMinHeap(AvaliadorDePalavra[] h, int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        minHeapify(h, i, n);
}

public static void minHeapify(AvaliadorDePalavra[] h, int i, int n) {
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    int smallest;

    if (l < n && h[l].getPrefixCounter() < h[i].getPrefixCounter())
        smallest = l;
```

```
    else smallest = i;
    if (r < n && h[r].getPrefixCounter() < h[smallest].getPrefixCounter())
        smallest = r;
    if (smallest != i) {
        exchange(h, i, smallest);
        minHeapify(h, smallest, n);
    }
}


public static boolean organizer(AvaliadorDePalavra[] h, int i, AvaliadorDePalavra word) {

    if(i>=h.length)return false;
    int l= 2 * i + 1;
    int r= 2 * i + 2;
    boolean equalPrefix = false;

    if (h[i].getPrefixCounter()== word.getPrefixCounter()) {
        equalPrefix =h[i].getWord().equalsIgnoreCase(word.getWord());
        if(equalPrefix==false) {
            equalPrefix = organizer(h,l,word);
            if(equalPrefix==false)equalPrefix = organizer(h,r,word);
        }
        else return equalPrefix;
    }
    else if(h[i].getPrefixCounter()<word.getPrefixCounter()) {

        equalPrefix = organizer(h,l,word);
        if(equalPrefix==false)equalPrefix = organizer(h,r,word);
    }
    else return equalPrefix;

    return equalPrefix;
}
```

Resultados experimentais

A avaliação experimental da aplicação foi realizada com os três ficheiros disponibilizados.

Abaixo segue os resultados finais.

Nº de ficheiros	1	2	3
Tempo (segundos) 1ºteste	1.540	3.029	4.166
Tempo (segundos) 2ºteste	1.554	2.941	4.175
Tempo (segundos) 3ºteste	1.487	2.837	4.146
Média	1,527	2.94	4.16

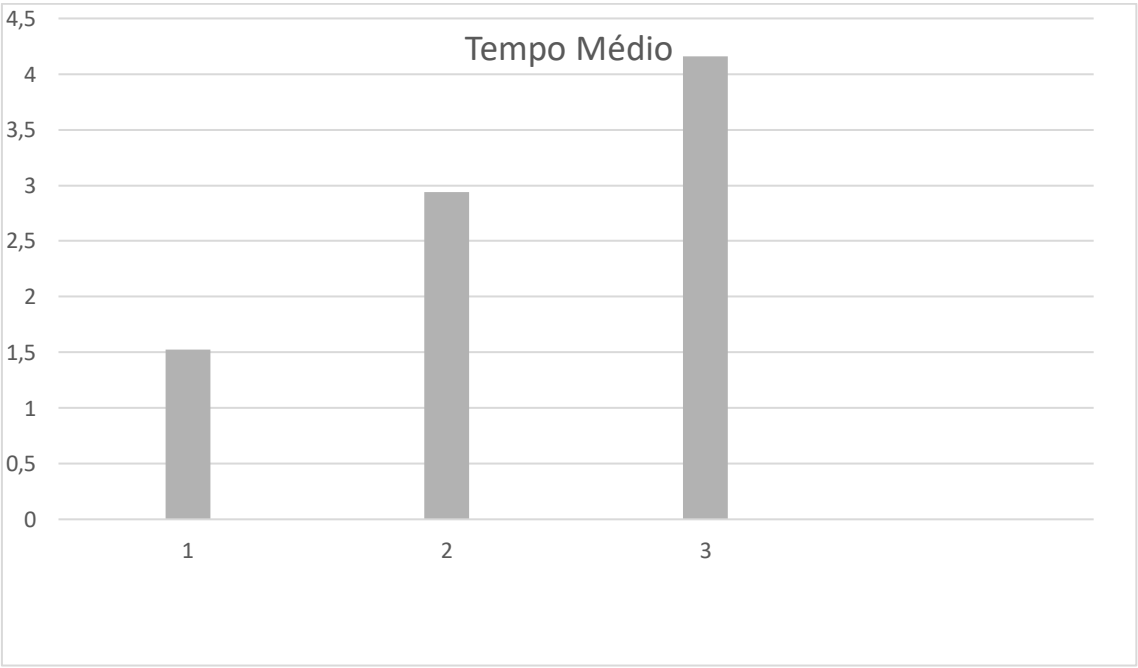

Figura 2 – Tempo médio (segundos) de leitura por nº de ficheiro(s)

Cada ficheiro têm 419846 palavras.

1. Conclusões

Com a realização deste trabalho pude me aperceber da dificuldade que é garantir o custo de um algoritmo. E independentemente das diversas tentativas, não consegui garantir todos os testes exigidos para os algoritmos.

As variações nos resultados experimentais podem ser consequência dos diferentes processos a serem executados pelo CPU na altura em que o programa está a decorrer. O crescimento é uniforme, pois à medida que se aumenta o número de ficheiros, é possível observar um padrão.