

**Algoritmos e Estruturas de Dados   
1ª Série**

**(Primeira série de problemas)**

|  |  |
| --- | --- |
|  |  |

**Junção ordenada de ficheiros, sem repetições**

| 50461 | João Freitas |
| --- | --- |
| 50457 | João Lopes |

Licenciatura em Engenharia Informática e de Computadores  
Semestre de Verão 2022/2023

16/4/2023

**Índice**

[**1.**](#_gjdgxs) **Introdução 2**

[**2.**](#_30j0zll) **Junção ordenada de ficheiros, sem repetições 3**

[2.1](#_1fob9te) Análise do problema 3

[2.2](#_3znysh7) Estruturas de Dados 4

[2.3](#_tyjcwt) Algoritmos e análise da complexidade 5

[**3.**](#_1t3h5sf) **Avaliação Experimental** 6

[**4.**](#_3rdcrjn) **Conclusões 8**

# Introdução

Este relatório tem como objetivo a aplicação de conhecimentos dados em aula, como o uso e alterações de diferentes algoritmos de ordenação, o cálculo da complexidade temporal e espacial tal como fazer funções com uma complexidade específica.

Como problema principal é necessário fazer um programa que recebe um determinado número de ficheiros, *n*, e considerando que cada ficheiro apenas tem uma palavras por linha, e que os junte lexicograficamente num único ficheiro (assumindo que dentro de cada ficheiro já está por ordem) sem repetidos, de forma a ter, de preferência, complexidade temporal *O(m \* logn)*, sendo *m* o número total de palavras e *n* o número de ficheiros. Também é necessário notar que é obrigatório uma complexidade espacial de *O(n)*.

Na primeira parte do problema apenas podemos usar o Priority Queue de array, sendo necessário implementarmos as funções como por exemplo, *offer()* que adiciona um elemento ao Priority Queue e como a *minHeapify()*. Na segunda parte é proibido usar arrays exceto para receber como argumentos de main os ficheiros, além disso não devemos usar a implementação da parte anterior, usando as já implementadas na biblioteca de java.

O relatório irá estar dividido da seguinte forma: Problema (análise, estrutura de dados, algoritmos e análise da complexidade), Avaliação Experimental do Problema e Conclusões sobre o trabalho.

# Junção ordenada de ficheiros, sem repetições

Iremos, nos tópicos a seguir, mostrar a nossa análise do problema com detalhe, as estruturas de dados utilizadas no nosso programa e os algoritmos e a sua complexidade, tanto temporal como espacial.

Para atingirmos uma implementação com respeito às especificações de complexidade temporal e espacial, optamos por usar uma data class PriorityQ, Priority Queue, que na primeira implementação é constituída por uma data classe que contém tanto a word a ser analisada como o *BufferedReader*, e que na segunda implementação é apenas usado o PriorityQ da biblioteca java, usando um *Pair<BufferedReader,String?>*.

## Análise do problema

Como referido na introdução, este problema tem como objetivo a programação de uma aplicação que permita juntar de forma ordenada os dados provenientes de vários ficheiros, produzindo um novo ficheiro de texto ordenado de modo crescente e sem repetições. Os ficheiros originais encontram-se ordenados de modo crescente, contendo uma palavra por linha. Queremos o programa , se possível, com a complexidade temporal de *O(m \* logn)*.

O algoritmo é utilizado com o objetivo de escolher a menor palavra entre as palavras de n ficheiros, logo optamos por um usar uma versão alterada de Priority Queue visto que apenas precisamos do elemento de maior prioridade, ou seja a menor palavra, e que é o único algoritmo de ordenação até a data com complexidade temporal *O(logn)*. Sendo m o número de palavras dos ficheiros, a complexidade a usar o PriorityQueue seria teoricamente a pretendida.

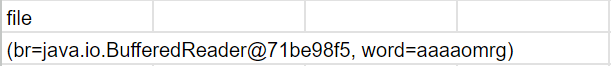
Assim sendo, o programa iria, na primeira parte do problema, criar um array onde iria armazenar inicialmente cada palavra inicial de cada ficheiro. Tirando proveito das funcionalidades do PriorityQueue, seria apenas necessário colocar a menor palavra lexicograficamente no lugar *heap[0]*, escrever a palavra no ficheiro output, substituindo a palavra pela próxima do mesmo ficheiro. Ao pensarmos nisso, notamos que a forma mais fácil para saber o ficheiro de dada palavra, seria apenas necessário o PriorityQueue ter o array de *Files(heap)*, este contendo a palavra sendo avaliada e o ficheiro de origem.

Na segunda parte do problema utilizamos o tipo Priority Queue já implementada na biblioteca java e, em vez de usarmos uma data class *File*, utilizamos o tipo *Pair*, onde no primeiro elemento armazena a string e no segundo o *BufferedReader*, tal como o *File*.

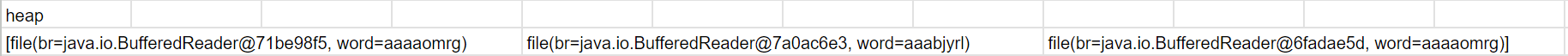
Com o objetivo de não adicionar palavras repetidas no mesmo ficheiro ao *heap*, criamos uma variável que memoriza a última palavra que foi adicionada ao array, e no caso de a palavra a seguir ser a mesma, a função next que vai substituindo a palavra pela próxima do mesmo ficheiro até ter uma palavra diferente da anterior. Isto também acontece em palavras iguais entre ficheiros, dessa vez implementada dentro do bloco while no main, visto que depois da palavra ser escrita no ficheiro de output, irá ser guardada como a última palavra e será substituída em next por outra palavra do mesmo ficheiro que a repetida.

## Estruturas de Dados

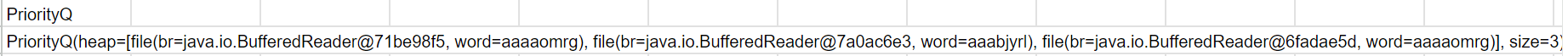
Na primeira parte deste problema foi utilizado a data class *PriorityQ(val heap:Array<File?>, var size:Int = 0)* e a data class *File(val br: BufferedReader, var word:String?)*. A variável *br* de *File* é o *BufferedReader*, que serve como a identificação de um ficheiro e também para avançar dentro do próprio, enquanto que a variável word é apenas a palavra que está a ser lida daquele ficheiro. A variável *heap* de Priority Queue é um array de *File* que vai armazenar a palavra em ordem lexicograficamente crescente de cada file a ser lida naquele momento e os seus respectivos *BufferedReaders*, enquanto que size vai armazenar o size do heap, sendo atualizado sempre que um novo elemento é adicionado ou removido de *heap*. *Heap* utiliza o tamanho de forma a verificar se existe espaço para adicionar novos elementos e para evitar leitura de posições *null*.  
Na segunda parte deste problema foi usado a PriorityQueue da biblioteca java, esta sendo do tipo *Pair<BufferedReader, String?>*.



*Figura 1: Exemplo da estrutura de dados File*



*Figura 2: Exemplo da estrutura de dados array, mais especificamente, heap*



*Figura 3: Exemplo da estrutura de dados PriorityQ*

## Algoritmos e análise da complexidade

Neste problema, devido à necessidade de uma complexidade temporal *O(m \* log n)*, usamos o Priority Queue, devido a ser o único algoritmo capaz de resolver o problema que aprendemos até à data de resolução deste trabalho com complexidade temporal *O(log n)*. Para colocarmos apenas o primeiro elemento por ordem usamos a função *minHeapify*, que irá comparar qual das palavras vem primeiro lexicograficamente e ordena-lhes ao chamar a função exchange. O *m* da complexidade vem do facto de metermos em ordem o primeiro algoritmo *m* vezes, visto que o ficheiro tem *m* palavras.

Em termos de espaço a complexidade é *O(n)*, visto que o *heap* só suporta *n* elementos cada um composto pela palavra e pelo seu respectivo *BufferedReader*.

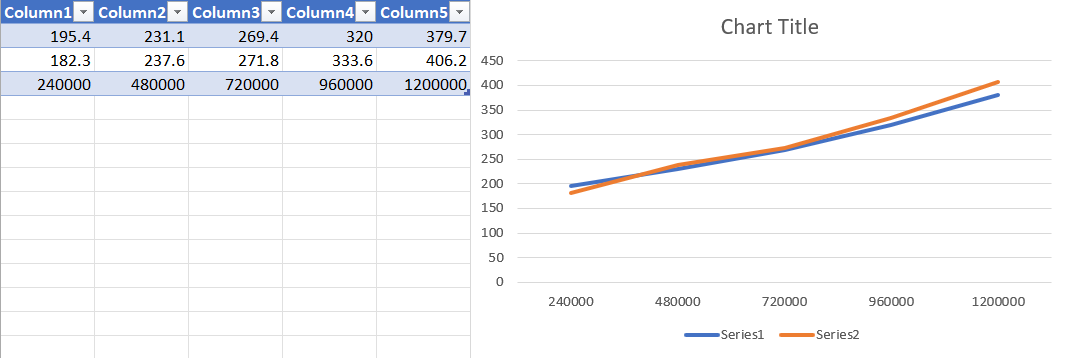
# Avaliação Experimental

Após verificarmos a funcionalidade do programa pretendida, devemos testar se respeita as complexidades temporais e espaciais pretendidas. Com este propósito, foram feitas tabelas com os resultados do tempo de execução do PriorityQueue, sendo feitos diversos testes para confirmar que o programa funcionava de forma eficiente, ao medir o tempo de execução na variação da quantidade de palavras (*m*) e também do número de ficheiros (*n*).

O primeiro teste consiste na variação de *m*, mantendo o mesmo número de ficheiros *n*. É de relevância notar que cada valor da tabela e gráfico na avaliação experimental foi feito através da média de tempo do teste, visto que cada teste foi realizado 10 vezes.

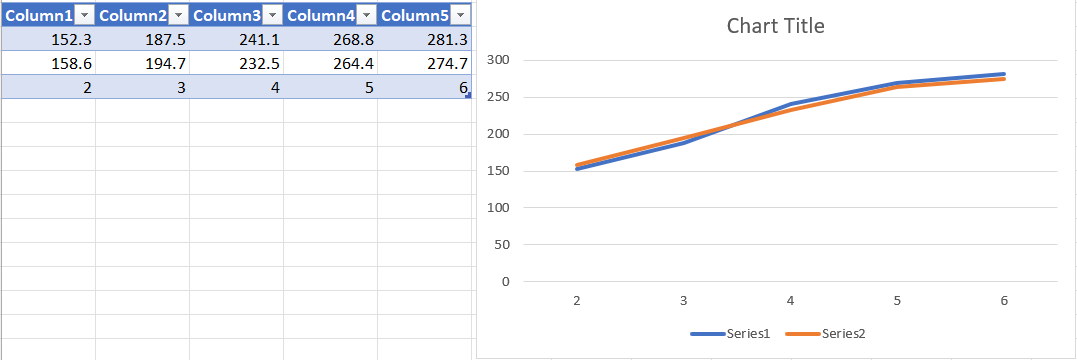
| **n = 3** | 240000 | 480000 | 720000 | 960000 | 1200000 |
| --- | --- | --- | --- | --- | --- |
| **JoinFiles1** | 195.4 | 231.1 | 269.4 | 320 | 379.7 |
| **JoinFiles2** | 182.3 | 237.6 | 271.8 | 333.6 | 406.2 |

Tabela 1: Resultados do tempo de execução do programa considerando 3 ficheiros, variando m



| **m = 120000** | 2 | 3 | 4 | 5 | 6 |
| --- | --- | --- | --- | --- | --- |
| **JoinFiles1** | 152.3 | 187.5 | 241.4 | 268.8 | 281.3 |
| **JoinFiles2** | 158.6 | 194.7 | 232.5 | 264.4 | 274.7 |

Tabela 2: Resultados do tempo de execução do programa considerando 240000 palavras, variando n



O segundo teste consiste no oposto do primeiro. Tendo um *m* que não varia, ir alterando o número de ficheiros.

Ao analisarmos os gráficos, é possível tirarmos muita informação. Podemos ver que, tal como previsto, no primeiro gráfico, ao aumentarmos o número de palavras o tempo cresce de forma linear, visto que *n* é sempre igual a 3, então a complexidade temporal ficaria *O(m\*log 3)* e, sendo *log 3* uma constante, é possível a retirarmos. Assim a complexidade ficaria *O(m)*.

No segundo gráfico, como previsto, ao aumentarmos o número de ficheiros o tempo cresce de forma logarítmica, visto que *m* é sempre igual a 240000, então a complexidade temporal ficaria(240000) e, sendo 240000 uma constante, é possível a retirarmos. Assim a complexidade ficaria *O(log n)*. Desta maneira podemos concluir que, se nem *n* nem *m* for constante, que a complexidade *O(m \* log n*).

# Conclusões

Este trabalho teve como objetivo consolidar algoritmos aprendidos em aula e a sua alteração de forma mais benéfica para com um determinado problema, tal como o cálculo da complexidade temporal de programas. De forma mais específica foi bem consolidada a PriorityQueue e a sua alteração, que nos permitiu ordenar o primeiro elemento do array da maneira pretendida.

Além disso, comparando as duas implementações é possível concluir que apesar de terem tempos de execução bastante similares, a segunda implementação que utiliza uma biblioteca da priority queue em java foi a mais eficiente das duas de forma geral, isso provavelmente devido ao facto que a nossa implementação poderá não estar a mais eficiente.

**Resposta ao tópico 2, análise de desempenho:**