

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

**(Problema)**

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

**Exercício 2**

|  |  |
| --- | --- |
| 46973 | Jorge Alexandre Luzio Simões |
| 47199 | Tiago Luís Lima da Silva |

Licenciatura em Engenharia Informática e de Computadores  
Semestre de Inverno 2020/2021

16/11/2020

# Introdução

Pretende-se desenvolver uma aplicação que permita determinar as k palavras que ocorrem mais vezes em n ficheiros ordenados lexicograficamente de modo crescente, em que o número de palavras total em todos os ficheiros é m. Assume-se que m >= 1.

O problema é descrito por:

* Um conjunto F = {f1, ..., fn} de n ficheiros de texto, em que n > 0;
* Cada linha de um ficheiro fn tem uma palavra.

O objetivo da aplicação a desenvolver é a produção de um novo ficheiro de texto contendo as k palavras distintas que têm um maior número de ocorrências. Em caso de empate, é indiferente.

# Maior número de ocorrências

A resolução deste exercício foi divida em três partes analise do problema, estruturas de dados utilizadas e analise de complexidade.

## Análise do problema

Numa primeira analise ao problema a primeira solução encontrada, foi a de ler cada ficheiro até ao fim, guardando cada valor encontra e o seu número de ocorrências dentro de um dicionário.

Esta implementação traz problemas com ficheiros muito grandes, onde cada palavra ocorra uma única vez, estamos a guardar em memoria todas essas entradas correndo o risco de ficar sem memoria.

Para resolver esta questão, na segunda solução para resolver o problema de ficar sem memoria para guardar as palavras, optámos por ler a palavra alfabeticamente menor no conjunto de ficheiros de forma a que seja preciso apenas guardar a palavra corrente e a sua contagem mais n palavras, sendo n o número de ficheiros, em memoria. Com esta palavra corrente assim que termine as suas ocorrências podemos comparar esta contagem com a menor contagem da entre as palavras anteriormente guardadas para decidir qual manter.

## Estruturas de Dados

Para as palavras de cada ficheiro e os apontadores para os mesmos é usado um *array* para cada.

Para guardar a palavra e o seu número de ocorrência foi criada uma classe auxiliar que tem dois atributos públicos para guardar os mesmos.

Para facilitar a comparação da palavra corrente com a palavra com menos ocorrências foi utilizada uma classe que implemente a estrutura de *heap*, de forma a ser mais rápida a obtenção da palavra com menor ocorrência até ao momento.

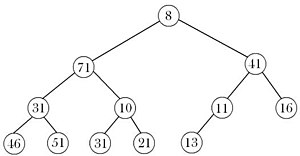


Figura 1 - Imagem de *heap*

## Algoritmos e análise da complexidade

Na primeira versão do algoritmo usamos a *PriorityQueue* do java que implementa a estrutura *heap*, onde recebe um comparador no seu construtor.

No caso da segunda implementados a nossa própria *heap* com base a *PriorityQueue* onde recebe um comparado, mas com métodos extras focados para o aumento da eficiência para este problema.

Um dos métodos implementados foi o *replaceHead* (Listagem TODO), com o objetivo de diminuir por metade o número de *siftDown* executados, visto que para a implementação com a *PriorityQueue* para trocarmos a posição mínima é preciso fazer um *poll* (que vai remover o elemento mais pequeno, colocar a ultima folha na primeira posição e reorganizar a *heap*) e depois é preciso adicionar o elemento novo com *add* (que vai adicionar o novo elemento a posição seguinte da ultima folha e reorganizar a *heap*), com o *replaceHead* apenas trocamos o elemento mais pequeno com o novo elemento e reorganizamos a *heap*.

**public void replaceHead(T head){**

**array[0] = head;**

**siftDown(1);**

**}**

O outro método é *changeComparator* que muda o comparador recebido no construtor e reconstrói a *heap* com base no novo comparador. Este método foi implementado para evitar a criação de um novo *PriorityQueue* com um comparador diferente e copia dos elementos do *PriorityQueue* antigo para o novo.

**public void changeComparator(Comparator<T> comparator){**

**this.comparator = comparator;**

**heapify();**

**}**

A nível complexidade tanto a versão com *PriorityQueue* como a versão da nossa *heap* tem uma complexidade dada pela expressão abaixo, sendo o N igual ao número de ficheiros e K igual ao número palavras com mais ocorrências.

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

Visto para cada ficheiro nos reorganizamos a *heap* caso adicionemos algum elemento ou haja alguma remoção da *heap*.

Apesar de terem a mesma complexidade, a versão com a *heap* implementada é ligeiramente mais rápida devido aos métodos referidos anteriormente, como pode ser visto na secção avaliação experimental.

# Avaliação Experimental

A avaliação experimental foi feita com os seguintes parâmetros:

* A *heap* do java foi colocada com um máximo de 32m de espaço.
* Mudanças no número de ficheiros (n) e o número de palavras a guardar (k).
* Cada algoritmo (*SORT* e *GET*) foi corrido cinco vezes para cada conjunto de n e k.

Comparação da implementação com as classes do Java e a nossa implementação da *heap*.

Tabela 1 - Tempos em milissegundos do algoritmo GET da implementação em Java para 3, 6 e 9 ficheiros com k = 3

|  |  |  |  |
| --- | --- | --- | --- |
| Java GET | 3 | 6 | 9 |
| 1 | 941.05 | 1821.14 | 2634.98 |
| 2 | 922.63 | 2157.76 | 2783.26 |
| 3 | 1164.7 | 1867.61 | 2622.03 |
| 4 | 906.26 | 1845.94 | 2695.44 |
| 5 | 913.8 | 1825.45 | 2640.79 |
| Average | 969.688 | 1903.58 | 2675.3 |

Tabela 2 - Tempos em milissegundos do algoritmo SORT da implementação em Java para 3, 6 e 9 ficheiros com k = 3

|  |  |  |  |
| --- | --- | --- | --- |
| Java SORT | 3 | 6 | 9 |
| 1 | 1094.53 | 1886.23 | 2631.01 |
| 2 | 939.82 | 1995.22 | 2752.24 |
| 3 | 956.19 | 1796.86 | 2668.2 |
| 4 | 945.21 | 1817.74 | 2738.92 |
| 5 | 921.03 | 1852 | 2567.66 |
| Average | 971.356 | 1869.61 | 2671.606 |

Tabela 3 - Tempos em milissegundos do algoritmo GET da nossa implementação para 3, 6 e 9 ficheiros com k = 3

|  |  |  |  |
| --- | --- | --- | --- |
| Heap GET | 3 | 6 | 9 |
| 1 | 868.25 | 1712.56 | 2641.7 |
| 2 | 862.77 | 1856.81 | 2755.66 |
| 3 | 838.53 | 1737.5 | 2639.37 |
| 4 | 830.96 | 1735.32 | 2631.18 |
| 5 | 828.04 | 1764.94 | 2612.93 |
| Average | 845.71 | 1761.426 | 2656.168 |

Tabela 4 - Tempos em milissegundos do algoritmo SORT da nossa implementação para 3, 6 e 9 ficheiros com k = 3

|  |  |  |  |
| --- | --- | --- | --- |
| Heap SORT | 3 | 6 | 9 |
| 1 | 878.58 | 1711.07 | 2574.29 |
| 2 | 914.77 | 1933.72 | 2686.05 |
| 3 | 871.94 | 1719.36 | 2585.56 |
| 4 | 874.76 | 1729.03 | 2588.63 |
| 5 | 888.17 | 1792.09 | 2583.86 |
| Average | 885.644 | 1777.054 | 2603.678 |

Figura 2 - Gráfico de comparativo entres os tempos nas tabelas anteriores.

Como se pode verificar com a implementação dos dois métodos referidos na secção anterior, existe um melhoramento dos tempos de execução.

Tabela 5 - Tempos em milissegundos do algoritmo GET da implementação em Java para 3, 6 e 9 ficheiros com k = 100 000

|  |  |  |  |
| --- | --- | --- | --- |
| Java GET | 3 | 6 | 9 |
| 1 | 1271.45 | 2033.15 | 2771.61 |
| 2 | 1121.74 | 2118.64 | 2858.7 |
| 3 | 1160.95 | 2049.16 | 2797.89 |
| 4 | 1097.41 | 1978.44 | 2933.42 |
| 5 | 1115.08 | 2014.11 | 3116.74 |
| Average | 1153.326 | 2038.7 | 2895.672 |

Tabela 6 - Tempos em milissegundos do algoritmo SORT da implementação em Java para 3, 6 e 9 ficheiros com k = 100 000

|  |  |  |  |
| --- | --- | --- | --- |
| Java SORT | 3 | 6 | 9 |
| 1 | 1188.29 | 2041.42 | 2799.88 |
| 2 | 1548.16 | 2141.55 | 2779.24 |
| 3 | 1409.06 | 2180.64 | 2937.12 |
| 4 | 1220.64 | 2086.06 | 2860.77 |
| 5 | 1275.9 | 2102.34 | 2760.41 |
| Average | 1328.41 | 2110.402 | 2827.484 |

Tabela 7 - Tempos em milissegundos do algoritmo GET da nossa implementação para 3, 6 e 9 ficheiros com k = 100 000

|  |  |  |  |
| --- | --- | --- | --- |
| Heap GET | 3 | 6 | 9 |
| 1 | 1181.62 | 1944.53 | 2719.96 |
| 2 | 1143.75 | 2109.8 | 2772.27 |
| 3 | 1140.8 | 2058.78 | 2778.93 |
| 4 | 1098.6 | 1961.21 | 2817.82 |
| 5 | 1144.86 | 1960.44 | 2765.11 |
| Average | 1141.926 | 2006.952 | 2770.818 |

Tabela 8 - Tempos em milissegundos do algoritmo SORT da nossa implementação para 3, 6 e 9 ficheiros com k = 100 000

|  |  |  |  |
| --- | --- | --- | --- |
| Heap SORT | 3 | 6 | 9 |
| 1 | 1204.38 | 1997.44 | 2950.64 |
| 2 | 1299.72 | 1969.59 | 2917.39 |
| 3 | 1203.89 | 2042.03 | 2735.75 |
| 4 | 1170.25 | 2016.82 | 2793.15 |
| 5 | 1133.78 | 2019.47 | 2759.19 |
| Average | 1202.404 | 2009.07 | 2831.224 |

Figura 3 - Gráfico de comparativo entres os tempos nas tabelas anteriores.

Nas figuras anteriores (2 e 3) podemos notar que não existe um aumento de tempo de execução significante com o aumento de K.

# Conclusões

Através deste trabalho concluímos que ao acrescentar métodos específicos para o problema conseguimos reduzir o tempo de execução do problema.

# Anexo

## Exercício 1

### findMinDifference

### lowerBound

### countSubKSequences

### mostLonely

### median