INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Algoritmos e Estruturas de Dados

Primeira Série de Problemas

Trabalho realizado por:

Nome:	Jorge Silva	Nº 49504
Nome:	Pedro Malafaia	Nº 49506
Nome:	Pedro Pacheco	Nº 47229

Docente: Engenheira Cátia Vaz

Resumo

No âmbito da UC Algoritmos e Estrutura de Dados, foi realizado este trabalho, onde, numa primeira fase, fomos encarregues de produzir vários algoritmos elementares. Para tal, foram usadas funções aprendidas ao decorrer das aulas, de modo a algoritmos de melhor eficácia. Ao chegarmos à segunda fase desta série, foi necessário estudar a complexidade de algoritmos, decifrando assim a complexidade de dois excertos de códigos. Chegando então à última fase da série, o problema: Maior número de Ocorrências, onde se pretende desenvolver uma aplicação que permite determinar as *k* palavras que ocorrem mais vezes em *n* ficheiros ordenados lexicograficamente (ordem alfabeticamente) de modo crescente. Essa fase possui duas implementações, a primeira utiliza as estruturas presentes em **kotlin.collections** ou **java.util**, já a segunda não utiliza essas estruturas.

Índice

1. Introdução

O trabalho tem como objetivo a consolidação da matéria estudada em aula. Esta matéria pode-se dividir em três partes: Algoritmos de Ordenação Elementar, Algoritmos de Pesquisa e Complexidade Assintótica. Algoritmos de Ordenação Elementar são algoritmos que colocam elementos de um certo dado em sequência de modo crescente. Estes são tanto usados em sequências numéricas como léxicas. Alguns exemplos destes algoritmos são *InsertionSort*, *BubbleSort*, *MergeSort*, *SelectionSort*. Algoritmos de Pesquisa são algoritmos capazes de a partir de um problema, procurar em uma sequência, um certo valor/dado, retornando assim esse valor procurado. Um Algoritmo de Pesquisa é, por exemplo, a Pesquisa Binária, ou *BinarySearch*. Esta, ao receber dois vetores e um elemento, retorna o índice do elemento pedido. A Complexidade Assintótica, é a análise dos algoritmos de modo a descobrir a sua complexidade em termos de tempo, ou seja, quanto tempo esse algoritmo demora a executar. A Complexidade Assintótica também se pode dividir em duas partes: Algoritmos Iterativos e Algoritmos Recursivos. Nos Algoritmos Recursivos existem duas formas para facilitar a sua análise: o Método de Substituições Recursivas, que vai aplicando sucessivamente a definição da equação até encontrar um candidato que satisfaça a equação, chegando assim a um valor, e existe também o Teorema Mestre, utilizado em equações do tipo dividir pra conquistar.

2. Análise de desempenho

A complexidade do primeiro exercício (figura 1) desta parte, é O(n*log n). A função *binarySearch* possui uma complexidade de O(log n), e esta é acedida por este código a cada *n*, através do *if (n >0)*. Logo, a complexidade de *xpto* será:

O(n)*O(log n) = O(n* log n)

```
 val k = a.size
 val n = b.size
 println(xpto(a, b, n))
...
```

em que

```
fun xpto(a: IntArray, b: IntArray, n: Int): Int {
    var n = n
    if (n > 0) {
        if (binarySearch(a,b[--n]) != -1)
         return 1 + xpto(a, b, n)
        /*o algoritmo pesquisa binária retorna -1 se o elemento não estiver presente em a*/
    }
    return 0
}
```

Figura 1 – Exercicio 1 (função xpto)

A complexidade do segundo exercício (figura 2) é $O(\log_m n)$. A função xpto neste caso recorre à recursividade $\log_m n$. Logo, se m=2 como sugere a primeira alínea deste exercício, então a complexidade do programa seria log n.

```
fun xpto(n: Int, m: Int): Int {
    return if (n / m == 0) 0 else 1 + xpto(n / m, m)
}
```

Figura 2 – Exercicio 2

3. Maior número de Ocorrências

Neste problema, o desafio proposto é fazermos um programa que corra n ficheiros e que encontre as k palavras mais repetidos nesses ficheiros. Para além disso, é necessário que a implementação utilize nos VM Parameters a opção -Xmx32m, que estabelece um heap size máximo de 32 MB.

3.1 Análise do Problema

Para resolver o problema, utilizamos o seguinte caminho de pensamento:

- Para obter o número de k palavras mais repetidas, será necessário um mapa (*hashMap*) em que cada *key* será uma palavra e com o respetivo *value* que corresponde ao número de vezes repetidas. Ou seja, iremos correr todas as palavras dos ficheiros, colocando-as no *hashMap*. Se a *key* já existir, ou seja, se a palavra já estiver nessa lista, adiciona-se 1 ao contador dessa palavra.
- De seguida, é necessário pegar nesse *hashMap* e introduzi-lo numa *PriorityQueue(HeapNode)* sendo que *HeapNode* foi implementado como um *maxHeap*. Deste modo, obtemos uma *PriorityQueue* de *HeapNodes*, ordenada do maior número de contagem (palavra mais frequente) para o menos número de contagem (palavras menos frequentes).
- Por fim, é realizado um for *loop* de 0 até k (não inclusive), dando *poll* em cada iteração, retirando assim do *heap* as palavras mais repetidas e introduzindo-as numa lista, sendo apenas necessário *printar* no output file cada elemento dessa lista.

3.2 Estruturas de Dados

Tal como disse anteriormente, foram utilizadas duas estruturas de dados: um *Heap* e uma *PriorityQueue*.

Um heap é uma árvore binária em que, dependendo da sua implementação, cada node terá um elemento, dependendo do pretendido. Neste caso, utilizando a PriorityQueue, obtemos um *maxHeap*.

Uma *PriorityQueue* é tal como o nome indica uma fila de prioridade, sendo que, neste caso, a maior prioridade vai para a palavra que tem o maior número de ocorrências, a segunda maior prioridade vai para a segunda palavra que ocorre mais vezes e por aí em diante.

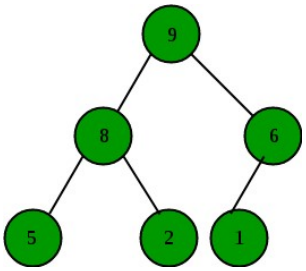

Figura 3 – Representação de um maxHeap

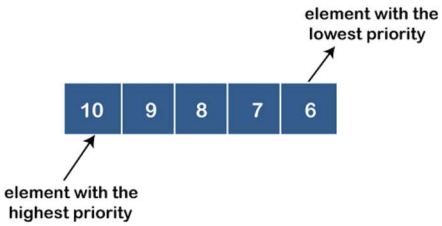

Figura 4 – Representação de uma *PriorityQueue*

3.3 Análise da Complexidade

Considerando que ***n*** é o número de palavras nos vários ficheiros, visto que temos de correr cada palavra do ficheiro com a finalidade de introduzir cada palavra no *hashMap* e obter como *value* o número de vezes que cada palavra foi repetida, obtemos assim uma complexidade em termos de tempo de O(n).

Já para colocarmos as palavras no *heap* e retirá-las, obtemos uma complexidade de O(k log k).

No troço de código a seguir é possível observar a criação da *PriorityQueue*, bem como a leitura das palavras do(s) ficheiro(s) e introduzindo no *hashMap*.

```
val heap = PriorityQueue<HeapWords>()
inputFiles.forEach{ file ->
    file.forEachLine { word ->
        mappedList[word] = (mappedList.getOrPut(word) { 0 } as Int) +
1
    }
}
```

No troço de código a seguir, é possível observar a adicionar ao *maxheap* cada elemento, bem como, retirar do heap k elementos. Como a *PriorityQueue* ficou definida como um maxHeap, o seu elemento mais à esquerda, e por isso, o primeiro a ser retirado, é a palavra que foi repetida mais vezes.

```
mappedList.forEach{ entry -> heap.add(HeapWords(entry.key,
entry.value))}

val kWordsArray = mutableListOf<String>()

for (i in 0 until kWords){
    kWordsArray.add(heap.poll().string)
}
```

4. Avaliação Experimental

Nesta fase do relatório iremos demonstrar algumas avaliações experimentais.

Na primeira avaliação experimental, irei procurar as 5 palavras mais repetidas, primeiro de apenas um ficheiro, depois de dois ficheiros e por fim de três ficheiros.

Figura 5 – Primeira avaliação experimental

Como é possível observar na figura 5, quantos mais ficheiros for necessário ler, maior vai ser o tempo que o programa demora a obter as 5 palavras mais repetidas.

De seguida, iremos realizar o mesmo procedimento, mas desta vez para as 10 palavras mais repetidas.

Figura 5 – Segunda avaliação experimental

Como é possível observar na figura 5, e como seria de esperar devido a, tal como referido anteriormente no relatório colocar palavras no *heap* e retirar tem uma complexidade de O(k log k) quando se aumenta o número de palavras a pesquisar, o tempo de procura é praticamente o mesmo devido a esta implementação.

Assim, é possível concluir que o que provoca um maior aumento no tempo de execução do programa é o aumento do número de ficheiros que é necessário ler pelo simples facto que iremos ter de ler todas as palavras de todos os ficheiros.