Algoritmos e Estruturas de Dados

2ª Série

Agrupar Palavras

Nº 50493 Bernardo Pereira

Nº 50512 António Paulino

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2022/2023

12/5/2023

Índice

1. Introdução

Pretende-se desenvolver uma aplicação que recebe como input um ficheiro de texto com **n** palavras e que retorna um ficheiro de texto em que as palavras que contenham os mesmos caracteres estão agrupadas por linhas (separadas por uma vírgula).

2. Agrupar palavras

2.1 Análise do problema

O problema é constituído por um ficheiro, em que cada linha do ficheiro tem múltiplas palavras. As palavras não estão ordenadas, e o número total de palavras é n.

As operações necessárias são as seguintes:

1. Criação de um HashMap que armazene o grupo de palavras com os mesmos caracteres
2. Leitura do ficheiro de input.
3. Extração das palavras de cada linha.
4. Armazenamento do conjunto de palavras com os mesmos caracteres num HashMap
5. Escrita dos conjuntos num ficheiro output

A abordagem implementada para as operações foi a seguinte:

1. É criado um HashMap com tamanho inicial 11. Este HashMap irá ter como key um código único gerado para todas as palavras que partilhem os mesmos caracteres, e como value irá ter esse grupo de palavras.

2. É criado um "leitor" para ler o ficheiro.

3. É percorrida a linha, enquanto não são encontrados caracteres, continua-se. Assim que são encontrados caracteres, são adicionados a uma string que irá conter a palavra até não se encontrarem mais caracteres. Depois de ser encontrada a palavra é armazenada no HashMap.

4. Para gerar a key da palavra, é usado um algoritmo de ordenação nos caracteres da palavra, assim assegura-se que quando os caracteres das palavras são os mesmos, a palavra ordenada será sempre a mesma, e o subsequente hashcode da palavra será sempre o mesmo, o que facilita a criação do conjunto de palavras. Depois de ser ordenada a palavra, verifica-se existe algum valor na key dada pela palavra ordenada. Caso exista, é adicionada a palavra ao conjunto já existente, caso não, é criada uma nova entrada no HashMap com o conjunto da key e da palavra.

5. Depois de serem percorridas todas as linhas do ficheiro, é percorrido o HashMap e são colocados no ficheiro de output todos os conjuntos de palavras com os mesmos caracteres.

2.2 Estruturas de Dados

Neste trabalho utilizamos as seguintes estruturas de dados:

Array

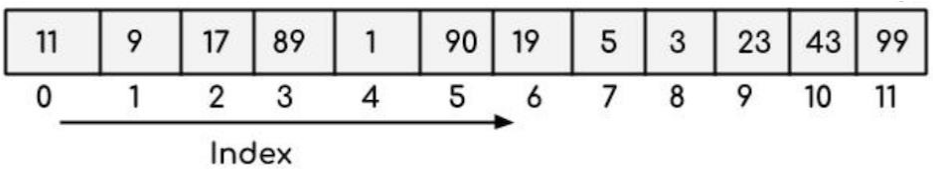


Figura 1: Exemplo da estrutura de dados *array*.

HashMap

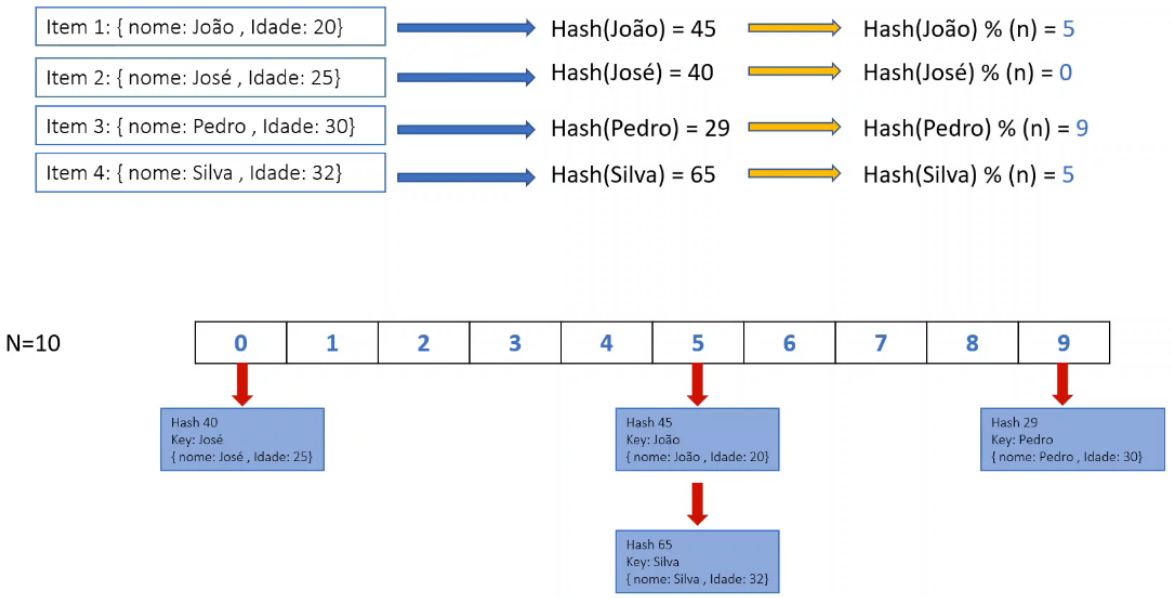


Figura 2: Exemplo da estrutura de dados *HashMap*.

2.3 Algoritmos e análise da complexidade

Principais algoritmos utilizados nesta implementação:

HashNode

O HashNode implementa a seguinte função que é utilizada nesta implementação

setValue(newValue : V)

```
oldval = value
value = newValue
return oldval
```

Esta função tem complexidade temporal O(1), já que realiza operações constantes.

HashMap (mapeamento de elementos através de keys)

O HashMap implementa os seguintes algoritmos/funções que são utilizados nesta implementação

```
fun hash(k : Int): Int,
    idx = k.hashcode() % dimTable
    if(idx < 0)  idx  dimTable
    return idx
```

Esta função tem complexidade temporal O(1), já que realiza operações constantes.

```
fun put(key : K, value: V): V?
     idx = hash(key)
     curr = table[idx]
          while(curr != null)
               if(key = curr.key)
                    return curr.setValue(value)
               curr = curr.next
    if(size / dimTable >= 0.75) resize()
    newNode = HashNode(key, value)
    newNode.next = table[idx]
    table[idx] = newNode
    size++
    return null
```

Esta função tem complexidade temporal O(N) no pior caso, em que n é o tamanho da Table atual, já que chama a função resize que cria uma nova Table com todos os elementos da atual. e não for necessário fazer resize, No melhor caso, tem complexidade O(1), se não existir lista na posição da table, caso contrário, tem complexidade O(l) em que l é o tamanho da lista ligada.

```
fun resize()
        dimTable = dimTable x 2
        newTable = arrayOfNulls<HashNode<K,V>?>(dimTable)
                for(idx in table)
                        curr = table[i]
                        while(curr != null)
                                table[i] = table[i].next
                                idx = hash(curr.key)
                                curr.next  = newTable[idx]
                                newTable[idx] = curr
                                curr = table[i]
        table = newTable
```

Esta função tem complexidade temporal O(N), em que n é a quantidade de elementos da Table atual já que cria uma nova table e tem que dispersar novamente todos os elementos ao utilizar a função hash.

```
fun get(key: K): V?
      idx = hash(key)
      curr = table[idx]
      while (curr != null)
          if (key = curr.key)
              return curr.value
         curr = curr.next
      return null
```

Esta função tem complexidade temporal O(N) no pior caso, em que n é a quantidade de elementos na lista simplesmente ligada associada ao índice da table, já que pode ter que percorrer todos os elementos até encontrar a key. No melhor caso, é O(1), se a função encontrar a key na primeira iteração pela lista.

```
fun containsKey(k: K): Boolean
    idx = hash(k)
    curr = table[idx]
    while (curr != null)
        if (k != null && k == curr.key) return true
        curr = curr.next
    return false
```

Esta função tem complexidade temporal O(N) no pior caso, em que N é o tamanho da lista ligada, se não existir o elemento. No melhor caso, o elemento é o primeiro da lista ligada, e a função tem complexidade temporal O(1)

Adicionalmente, são utilizadas as funções de extensão

```
operator fun <K, V> AEDMutableMap<K, V>.set(key: K, value: V)
  put(key, value)
```

```
operator fun <K,V> AEDMutableMap<K, V>.contains(k:K):Boolean
   return containsKey(k)
```

Que têm a complexidade temporal das funções put e containsKey, respectivamente.

Finalmente, é utilizada uma versão do algoritmo quicksort para caracteres.

```
fun sortWord(a: CharArray, left: Int, right: Int)
   if (left < right)
      val i = partitionChar(a, left, right)
      sortWord(a, left, i - 1)
      sortWord(a, i + 1, right)
```

```
fun partitionChar(a: CharArray, left: Int, right: Int): Int {
    var i = left - 1
    var j = right
    val pivot = a[right]
    while (true)
        while (i < right and a[++i] < pivot)
        while (j > left and a[--j] > pivot)
        if (i >= j) break
        exchangeChar(a, i, j)
    exchangeChar(a, i, right)
    return i


fun  exchangeChar(a: CharArray, i: Int, j: Int)
    val x = a[i]
    a[i] = a[j]
    a[j] = x
```

Este algoritmo, como já foi analisado nas aulas, tem complexidade temporal O(n^2), no pior caso em que as partições são desequilibradas. No caso médio, que é mais comum, tem complexidade temporal O(n lg n), quando as partições são mais equilibradas e os elementos estão distribuídos aleatoriamente.

Algoritmo principal de agrupamento de palavras(pseudo-código):

```
inputreader = createReader(args[1])
outputwriter = createWriter(args[0])
wordGroups = AEDHashMapAED<String, String>()

line = inputreader.readLine()
while (line != null) {
   word = ""
   i = 0
   while (i < linesize)
      while (i < linesize && line[i] == ' ')
         i++
      while (i < linesize && line[i] != ' ')
         word = word + line[i]
         i++

      size = word.length
      wordArr = CharArray(size) { word[it] }
      sortWord(wordArr, 0, size - 1)
      sortedWord = String(wordArr)

      if (word.isNotEmpty())
         group = wordGroups[sortedWord]
         if (group != null)
            wordGroups[sortedWord] = "$group, $word"
         else
            wordGroups[sortedWord] = word
         word = ""

   line = inputreader.readLine()

for (wordGroup in wordGroups)
   outputwriter.println("[${wordGroup.value}]")

outputwriter.close()
```

Análise da complexidade temporal do algoritmo:

$$T(n) = O(n) * (2\,O\,(m) + O(m\,lg\,m) + O(k)) + O(w)$$
$$= O(n) * (O\,(m\,lg\,m) + O(k)) + O\,(w)$$

N é a quantidade de palavras, m o tamanho das palavras em média, k o tamanho da lista ligada ao elemento no HashMap e w a quantidade de elementos no HashMap. Como este algoritmo cria sempre a mesma key para palavras com os mesmos caracteres, irá sempre sobrepor o primeiro elemento da lista ligada, em que n é a quantidade de palavras. Tendo isto em conta, podemos considerar k = 1 e pode simplificar-se a expressão:

$$T(n) = O(n) * (O(m\,lg\,m) + O(1)) + +O(w)$$
$$= O(nm\,lg\,m) + O(w)$$

Este algoritmo tem uma complexidade temporal aproximadamente logarítmica linear, sendo que cresce linearmente com a quantidade de palavras. Em termos de complexidade espacial, será O (m + w), isto porque para cada iteração do loop é necessário armazenar temporariamente os caracteres da palavra num array para utilizar o algoritmo de quicksort, e armazenar os conjuntos de palavras no HashMap.

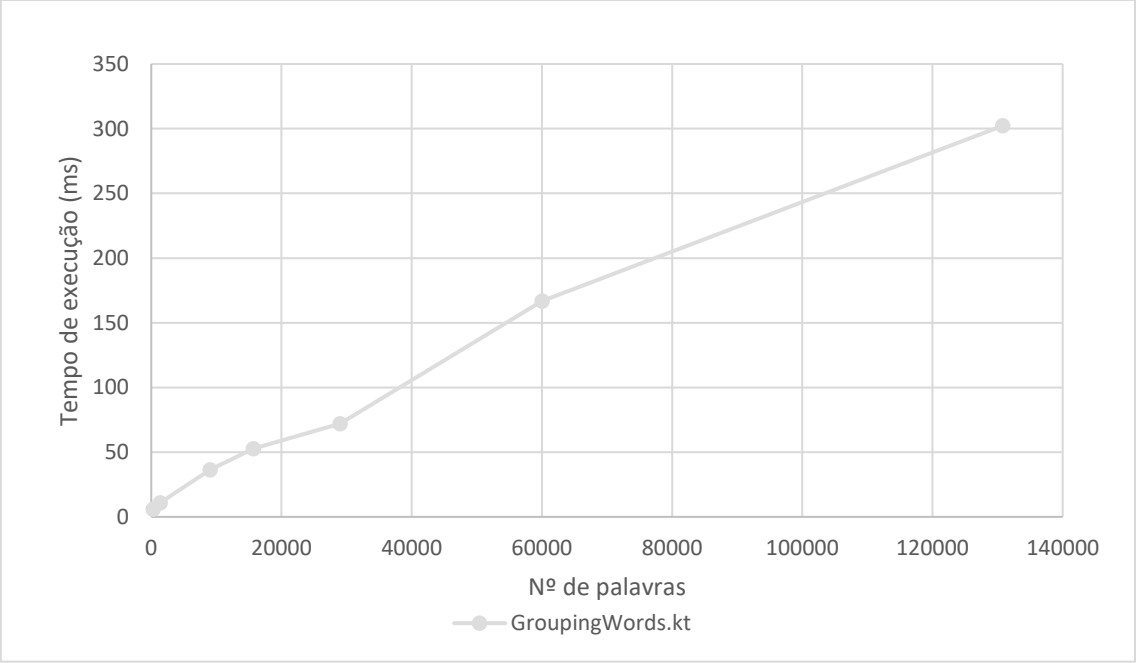
3 Avaliação Experimental

Para testar o algoritmo, serão utilizadas amostras de 270, 1370, 9000, 15645, 29000, 60000 e 130765 palavras, sendo que esses ficheiros serão os ficheiros book, book2, book3, book4, book5, book6 e book7.

Os testes serão realizados numa máquina com o processador Intel I7 12650h, 16 Gb ram 4800mhz

	Nº de palavras a agrupar						
	270	1370	9000	15645	29000	60000	130765
GroupingWords.kt	5.983	10.996	36.454	52.746	71.982	166.859	302.207

Tabela 1: Resultados do tempo em milissegundos de execução de algoritmos de ordenação considerando várias amostras. Média de 5 valores por amostra



Como é possível observar no gráfico e na tabela 1, a implementação aparenta ter um crescimento logarítmico linear/ linear, quando comparada com o gráfico da figura 4. Então verifica-se experimentalmente a análise do algoritmo na secção 2.3.

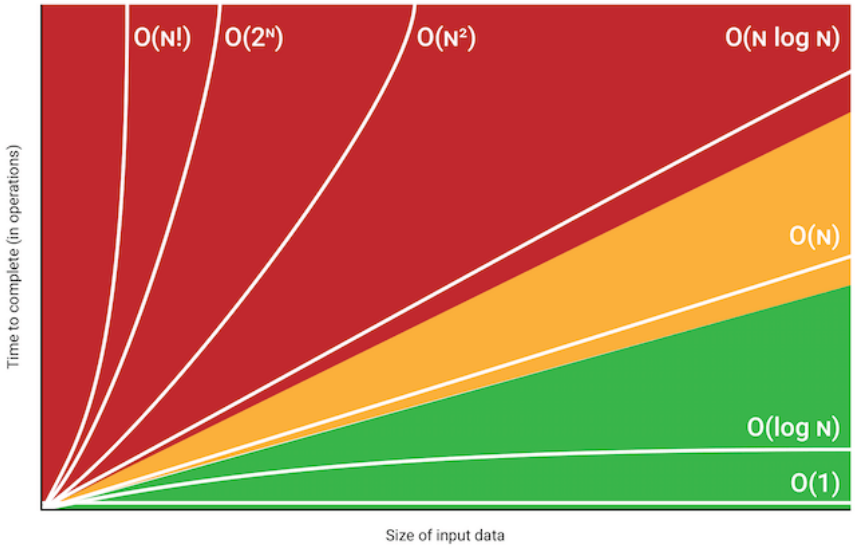


Figura 4: Gráfico de notação O-Grande

4 Conclusões

Neste trabalho foi implementado um algoritmo que agrupa múltiplas palavras que partilham os mesmos caracteres, criando um ficheiro com um conjunto por linha. Foi possível implementar um algoritmo com complexidade temporal aproximadamente linear logarítmica e complexidade espacial O(m + w) em que m é o tamanho das palavras em média e w a quantidade de conjuntos existentes.