Algoritmos e Estruturas de Dados

1ª Série

(parte 2)

Filtrar entre Palavras

N. 49431 Tiago Neto
N. 49423 Rafael Pegacho

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2021/2022

10/04/2022

1. Introdução

- Introdução

  Pretende-se desenvolver uma aplicação que permita juntar de forma ordenada os dados provenientes de vários ficheiros, compreendidos entre duas palavras introduzidas pelo utilizador, produzindo um novo ficheiro de texto ordenado de modo crescente. Os ficheiros originais encontram-se ordenados de modo crescente e contêm uma palavra por linha.

- Principais objetivos

  O objetivo da aplicação a desenvolver é a produção de um novo ficheiro de texto, ordenado de modo crescente, que contenha as palavras dos ficheiros pertencentes ao conjunto F, sem repetições e que estejam compreendidas entre as palavras word1 e word2. O ficheiro produzido deverá também conter uma palavra por linha.

- Organização do relatório (resumo das secções em que se divide)

  O relatório divide-se em 4 grandes pontos:

  Introdução, Análise e Estrutura do problema, Avaliação Experimental e Conclusão.

  Ao longo da Análise e Estrutura do problema serão enunciados os métodos de resolução deste e será feita a análise de eficiência, enquanto na Avaliação Experimental será feita uma avaliação dos resultados obtidos.

2. Filtrar entre Palavras

2.1 Análise do problema

Para a resolução deste problema vamos precisar de ler vários ficheiros de texto, para isso faremos uma função chamada “fileRead” que retornará uma mutableList<String> com todas as “Strings” existentes nos ficheiros. Nesta função precisamos de passar como parâmetro uma lista de Strings com vários objetos, estes objetos são argumentos passados na execução do programa e são os nomes ficheiros que vão ser lidos, de seguida implementa-se um for loop que faz uma leitura linha a linha de cada ficheiro e adiciona o conteúdo de cada linha como objeto a uma á lista mutável que no fim será retornada

Tendo agora todas as “Strings” dos ficheiros que se prentede ler numa lista mutável, será implementada a função “fileOrganize” que começa por chamar a função “fileRead” e guardar o retorno desta função numa variável chamada "list". Sendo esta variável o retorno da função “fileRead”, será do tipo MutableList<String>, então, adiciona-se agora a esta lista a word1 e word2, parâmetros da função “fileOrganize” que têm o seu valor ditado nos argumentos da função “main”. Já temos todos os objetos que precisamos dentro da lista mutável, apenas

falta ordená-la, o que será feito usando o algoritmo MergeSort na segunda versão de resolução e usando a função .sort() na primeira versão da resolução.

Antes de imprimir a lista no ficheiro de saída, precisamos de remover da lista as palavras não compreendidas entre a word1 e word2 e as palavras repetidas, mas mais eficiente que isso será criar uma nova lista. Sabendo que a lista está já ordenada, criaremos uma variável do tipo MutableList<String> sem qualquer objeto e chamaremos esta variável de “newlList”. Recorrendo novamente a um for loop iremos alguns dos elementos de "list” a “newList” começando no índice correspondente à word1 + 1 e acabando no índice da word2, não incluindo o mesmo. Para além disso, para cada elemento compreendido entre estes índices ser adicionado, é necessário garantir que não é igual ao elemento que o sucede. Concluímos assim esta função retornando “newList”.

Por fim, só falta imprimir os resultados retornados da função “fileOrganize” num ficheiro de texto, criamos a função "fileWriter", com apenas dois parâmetros, a lista que iremos imprimir o ficheiro de destino. No corpo desta função começamos por criar o ficheiro destino e em seguida adicionar a uma variável vazia do tipo “String” cada um dos objetos existentes na lista, dando uma linha de intervalo entre cada um. Quando a String tiver completa finalizamos imprimindo-a no ficheiro destino.

2.2 Estruturas de Dados

Na Figura 1 mostram-se as estruturas de dados utilizadas nas diversas fases de ordenação de um array de inteiros (considerando como exemplo o array {9,7,5,11,12,2,14,3,10,6}, através do algoritmo quicksort. (utilizado por definição quando chamada a função .sort())

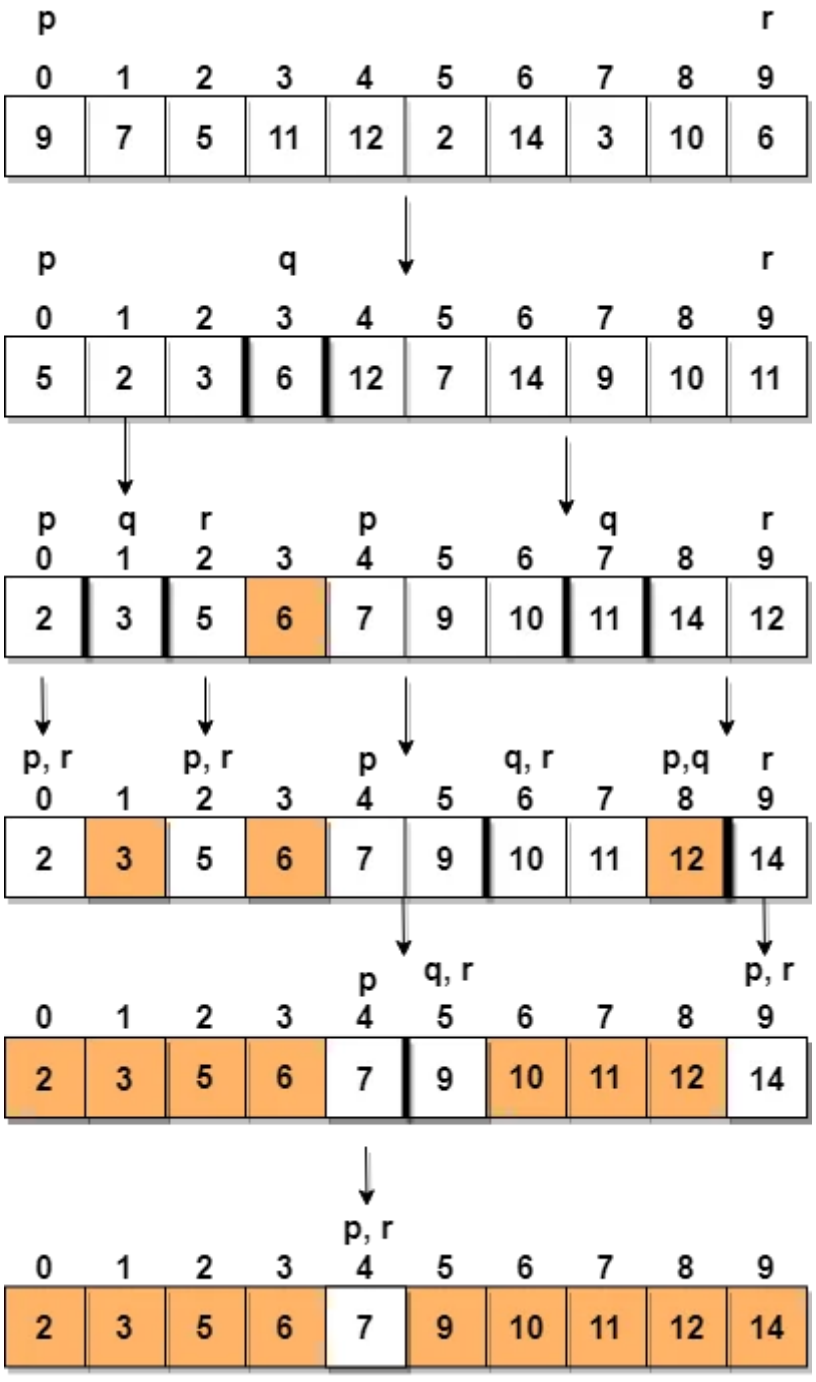

Figura 1 – Exemplo das estruturas utilizadas para armazenar os elementos quando o input é um um Array*{9,7,5,11,12,2,14,3,10,6}*

2.3 Algoritmos e análise da complexidade

A implementação do merge-sort realizada neste trabalho encontra-se na figura 2.

```kotlin
fun mergeSort(words: Array<String>) {
    if(words.size >= 2) {
        val left = Array(words.size / 2) {""}
        val right = Array(words.size - words.size / 2) {""}
        for (i in left.indices) left[i] = words[i]
        for (i in right.indices) right[i] = words[i + words.size / 2]
        mergeSort(left)
        mergeSort(right)
        merge(words, left, right)
    }
}

fun merge(words: Array<String>, left: Array<String>, right: Array<String>) {
    var a = 0
    var b = 0
    for(i in words.indices) {
        if(b >= right.size || (a < left.size && left[a] < right[b])) {
            words[i] = left[a]
            a++
        } else {
            words[i] = right[b]
            b++
        }
    }
}
```

Figura 2: Implementação em Kotlin do algoritmo merge-sort.

O diagrama apresentado na Figura 2 descreve o processo completo *do merge-sort* para o *array* {38, 27, 43, 3, 9, 82, 10}. Os números indicam a ordem nos quais os passos são processados.

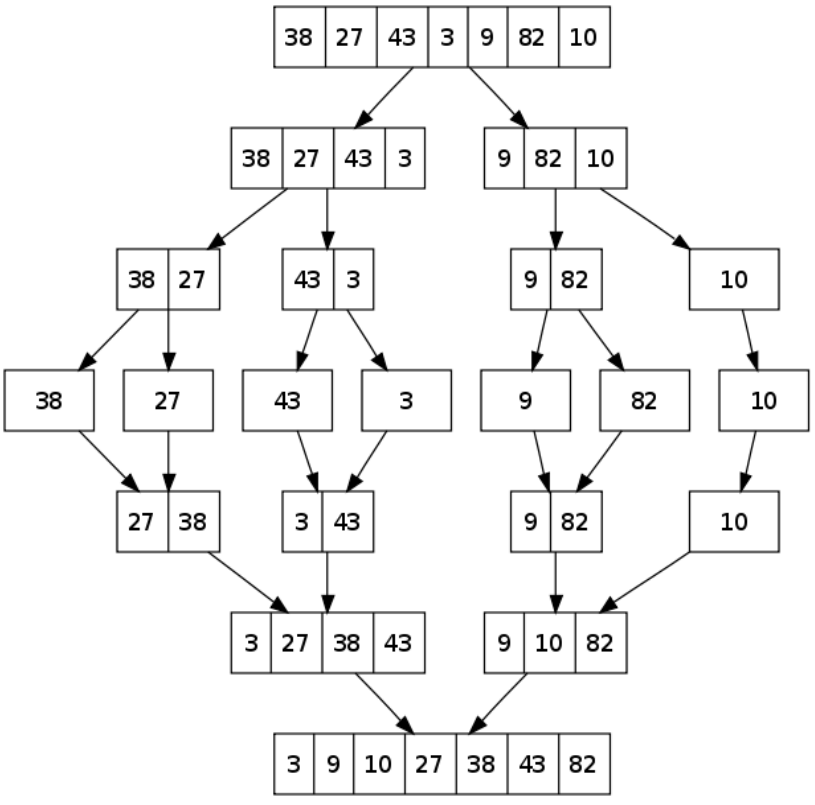

Figura 3: Exemplo de execução para um exemplo.

A expressão (2.1) representa a recorrência do custo do algoritmo Merge-Sort, sendo a expressão (2.2), o seu resultado.

$$C(N) = 2C\left(\frac{N}{2}\right) + N \quad (2.1)$$

$$C(N) = O\left(Nlog_2 N\right) \quad (2.2)$$

3. Avaliação Experimental

Processador do computador utilizado para os teste:
AMD Ryzen 7 4800HS with Radeon Graphics
2.90 GHz 16 CPUS

Na tabela 1 está representada uma tabela com os tempos de execução médios (em milissegundos) de 7 tentativas do programa recorrendo ao .sort() e ao kotlin.collections(). Estas tentativas foram feitas para casos com 3 ficheiros de texto com arrays de 3.000, 5.000, 15.000, 100.000 e 200.000 “Strings”, um total de 9.000, 15.000, 45.000, 300.000, 600.000 palavras.

	9.000 words	15.000 words	45.000 words	300.000 words	600.000 words
MergeSort	149 ms	330 ms	535 ms	24 906 ms	169 028 ms
.Sort()	63 ms	65 ms	264 ms	24 327 ms	176 334 ms

Tabela 1: Tabela de Tempos de execução de algoritmos.

Na Figura 4 estão representados os 10 pontos num gráfico de tempo de execução em função do número palavras. os pontos a Laranja representam o MergeSort e os pontos a Azul representam o .sort()

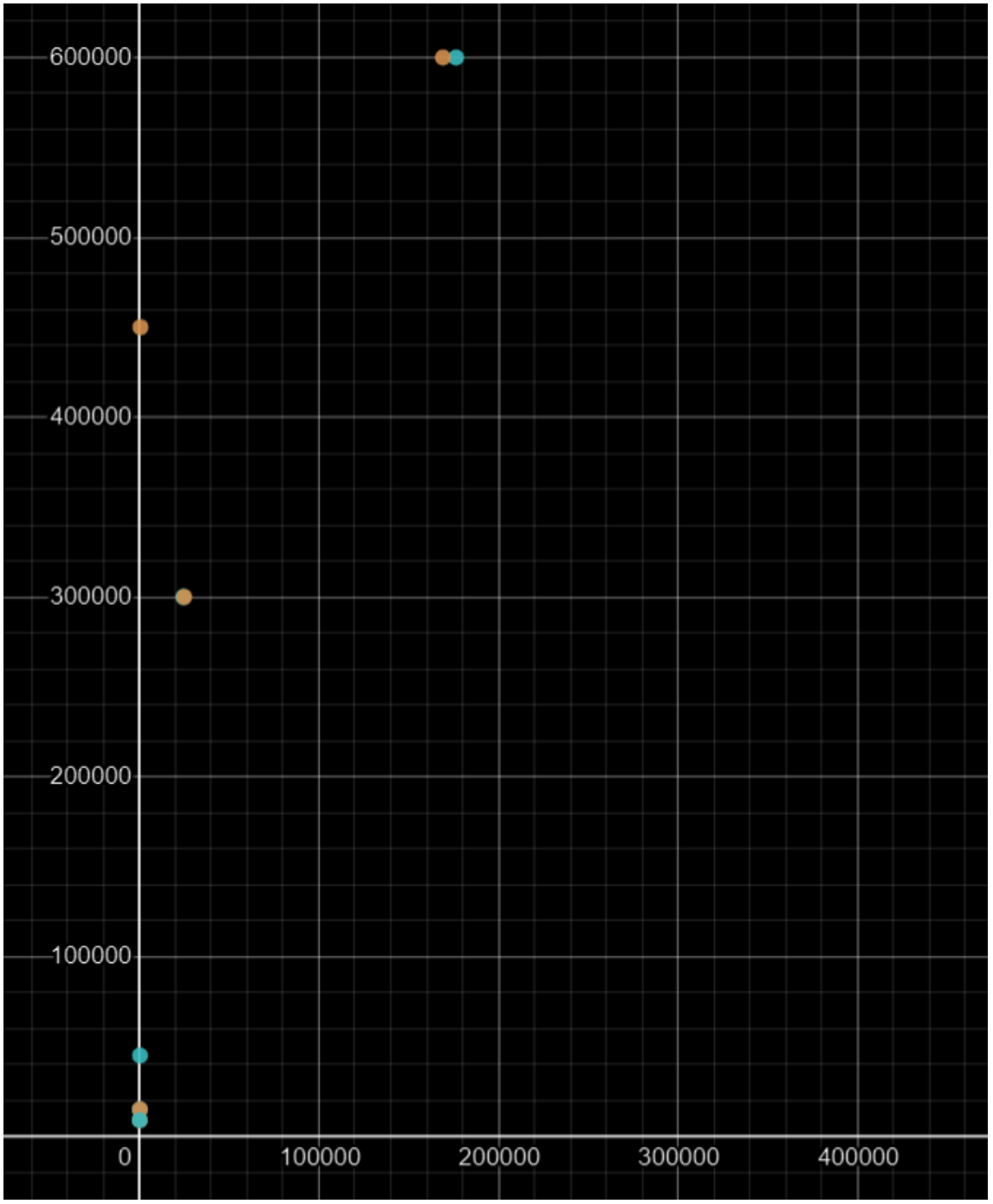

Figura 4: Implementação em Kotlin do algoritmo merge-sort.

4. Conclusões

Para finalizar, podemos concluir que a utilização da função .sort() da biblioteca kotlin.collections é mais efetiva em casos de arrays com menos objetos. Conforme os objetos vão aumentando, mais se torna eficaz e vantajoso a utilização do algoritmo MergeSort para a ordenação de Arrays. Sendo as diferenças de tempo maiores em grandes arrays e mais significativas, a utilização MergeSort acaba por ser vantajosa.

Referências

[1] “Disciplina: Algoritmos e Estruturas de Dados - 2122SV,” *Moodle 2021/2022*. [Online]. Available: https://2122.moodle.isel.pt. [Accessed: 10-April-2022].