Departamento de Engenharia Eletrónica e de Telecomunicações e Computadores

Licenciatura em Engenharia Informática e de Computadores

Algoritmos e Estruturas de Dados

1ª Série

Alun@s:
A49506 - Pedro
A49418 - Roberto

Docentes:
Cátia Vaz

Abril 2022

Conteúdo

1 Introdução

Neste relatório iremos apresentar soluções, acompanhadas das respetivas explicações para os diversos exercicios propostos na primeira série de problemas. Para implementação das diversas funções foram utilizados algortimos lecionados nas aulas, com o objetivo de otimizar ao máximo em tempo e memória a execução das funções. Nesta primeira série de exercicios é importante também referir que foram criado ficheiros de benchmark para verificar e analisar os tempos de execução em cada uma das funções realizadas.

2 Algoritmos Elementares

Esta é a primeira parte desta série de exercícios, que nos desafia a implementar quatro funções: Upper Bound, Count Incresing Sub-Arrays, Count Equals e Most Lonely. Para cada uma destas funções irá ser explicada a sua respetiva resolução.

2.1 Upper Bound

```
fun upperBound(a: IntArray, l: Int, r: Int, element: Int): Int
```

Dado um *sub-array (a,l,r)* ordenado de modo crescente, retorna o último indícei tal que `array[i] <= element`. No caso do sub-array ser um *sub-array* vazio retorna -1. Note que se todos os números no sub-array forem maiores do que *element* retorna-se -1.

```
fun upperBound(a: IntArray, l: Int, r: Int, element: Int): Int {
    var lx = l
    var rx = r
    if ( lx > rx || a[lx] > element ) return -1
    while (lx <= rx){
        val mid = (lx + rx)/2
        if ( a[mid] <= element ) lx = mid + 1
        else rx = mid - 1
    }
    return rx
}
```

As duas variáveis `lx` e `rx` correspondem ao limite esquerdo e direito respetivamente. Neste problema, foi implementado um Binary Search, que é um algoritmo utilizado em arrays ordenados. Este metodo consiste em dividir repetidamente o intervalo da pesquisa ao meio, encontrando de modo mais eficaz e rápido o elelemento que estamos à procura. Enquanto o limite esquerdo for menor ou igual que o limite direito, iremos sucessivamente atribuir a `mid` o valor do limite esquerdo somado com o limite direito a dividir por 2. De seguida, verificamos se o elemento nesse indice `mid` é menor ou igual ao elemento que procuramos. Caso isso se verifique, significa que o elemento que procuramos, se encontra acima desse indice, logo, o novo valor do limite direito `lx` passa a ser `mid + 1`, e o processo é repetido, mas desta vez reduzindo o número de elementos para metade. Caso se verifique o contrário, e o elemento que procuramos seja menor do que o elemento com indíce `mid`, o novo limite direito passa a ser `mid - 1`. Por fim, quando este ciclo acaba, retornamos o valor do limite direito `rx`, sendo que neste caso, procuramos o indice de um elemento que seja menor ou igual ao elemento que é passado como parâmetro.

2.2 Count Increasing Sub-Arrays

```
fun countIncreasingSubArrays(v: IntArray): Int
```

Dado um array v de inteiros, retorna o número de *sub-arrays* estritamente crescentes presentes em v com dimensão maior ou igual a 2. Por exemplo, no caso de $v = \{1, 2, 4, 4, 5\}$ este método deverá retornar 4, visto que existem os seguintes sub-arrays estritamente crescentes: {1, 2},{1, 2, 4}, {2, 4} e {4, 5}.

```
fun countIncreasingSubArrays(v: IntArray): Int {
    var (count, len) = Pair(0, 1)

    for (i in 1 until v.size) {
        if (v[i] > v[i - 1]) {
            count += len++
        } else {
            len = 1
        }
    }

    return count
}
```

A variável `count` representa a quantidade de *sub-arrays* estritamente crescentes presentes em `v`, enquanto que `len` representa a quantidade de *sub-arrays* possíveis de formar no *sub-array* estritamente crescente atual. A cada iteração verificamos se o elemento atual é maior que o anterior, caso se verifique, incrementamos `count` adicionando-lhe o valor de `len`, incrementando-o logo a seguir. Caso o valor atual seja inferior ou igual que o anterior, damos o valor de 1 a `len`, para "começar de novo". A notação da solução para este problema é de $O(n)$, sendo `n` o número total de elementos de `v`.

2.3 Count Equals

```
fun countEquals(points1: Array<Point>, points2: Array<Point>,
cmp:(p1:Point, p2:Point)-> Int): Int
```

Dados dois *arrays* points1 e points2, ordenados de modo estritamente crescente segundo a função de comparação cmp, retorna o número de pontos que ocorrem simultaneamente em ambos os arrays. No caso de não existirem elementos em algum dos arrays, o método retorna 0. O tipo Point está definido da seguinte forma: data class Point(var x:Int, var y:Int)

```
fun countEquals(points1: Array<Point>, points2: Array<Point>, cmp: (p1: Point, p2:
    if (points1.isEmpty() || points2.isEmpty()) return 0

    var i = 0
    var j = 0
    var count = 0

    while ( i!= points1.size  && j!= points2.size ){
```

```
        when{
            cmp(points1[i], points2[j]) < 0 -> i++
            cmp(points1[i], points2[j]) > 0 -> j++
            cmp(points1[i], points2[j]) == 0 -> {
                count += 1
                i++
                j++
            }
        }
    }

    return count
}
```

Para começar, verificamos se algum dos pontos passados por parâmetro é um array vazio. Caso se verifique, retornamos 0. De seguida, são criadas 3 variaveis, `i`, `j` e `count`. As variáveis `i` e `j` irão servir para correr os diversos pontos dos arrays, enquanto que a variável `count` irá fazer a contagem do número de pontos que ocorrem simultaneamente em ambos os arrays. Para realizarmos essa contagem, recorremos a um ciclo `while` que irá ser repetido enquanto `i` for diferente do tamanho do array de `points1` e enquanto `j` for diferente do tamanho de array de `points2`. Dentro desse ciclo, o objetivo é ir correndo os vários indices dos dois pontos e quando são iguais, somar +1 à variável `count`. Para isso, é utilizado um when, com o auxílio do `cmp`. Quando `cmp` entre dois pontos no indice `i` e `j` dá menor que 0, incrementamos o indice `i`. Quando o `cmp` entre os dois pontos dá maior que 0, incrementamos o indice `j`. Quando o `cmp` é igual a 0, significa que os pontos são iguais, logo incrementamos `count`, bem como variáveis `i` e `j`. Deste modo, é possivel correr os vários pontos de ambos os arrays, verificando o número de pontos iguais.

2.4 Most Lonely

```
fun mostLonely(a: IntArray): Int
```

Dado o array a de inteiros positivos, retorna o inteiro que se encontra mais isolado. O inteiro mais isolado num array é o inteiro que tem a maior distância `k`, sendo `k`, a menor das diferenças, em valor absoluto, entre ele e cada um dos restantes. Por exemplo, considere $a = \{10, 16, 1, 17, 5\}$. O elemento mais isolado neste caso é o elemento 10, porque $k1 = |1 - 5| = 4$; $k5 = |5 - 1| = 4$; $k10 = |10 - 5| = 5$; $k16 = |16 - 17| = 1$ e $k17 = |17 - 16| = 1$. Assuma que o array tem pelo menos dimensão 1, e no caso de ser 1, deve retornar o único elemento que existe. Em caso de empate, retorna o menor dos elementos empatados que se encontra no array.

```
fun mostLonely(a: IntArray): Int {
    mergeSort(a, a.size)

    if (a.size == 1)
        return a[0]
```

```
    // salvar 1 iteração
    var (lonely, maxK) = Pair(a[0], abs(a[0] - a[1]))

    // loop array e obter o k de a[i]
    for (i in 1 until a.size) {
        val k = if (i == a.size - 1) abs(a[i] - a[i - 1])
                else min(abs(a[i] - a[i - 1]), abs(a[i] - a[i + 1]))

        if (k >= maxK) {
            val draw = k == maxK
            maxK = if (draw) min(k, maxK) else k
            lonely = if (draw) min(lonely, a[i]) else a[i]
        }

    }

    return lonely
}
```

Primeiramente ordenamos o array de forma crescente, porquê? Porque descobrimos que ao ordenar primeiro, sabemos sempre que o `k` mínimo é a subtração absoluta entre o número e as suas vizinhanças. Portanto, após ordernarmos, verificamos se o tamanho de `a` é 1, se for returnamos o único valor existente, caso contrário, vamos correr o *array* `a` e obter o `k` do valor atual. Caso o `k` calculado seja maior que o `k` máximo até agora (`maxK`), atualizamos o seu valor e de `lonely` (valor mais isolado). Obviamente que antes de atualizar, verificamos se existe um empate, caso se verifique, atualizamos com o valor mínimo entre os dois. A notação da solução para este problema é de $O(n \log n + n)$, sendo `n` o número total de elementos de `a`, pois o algoritmo `mergeSort` tem notação $O(n \log n)$ e logo a seguir fazemos outro loop pelo array com tamanho `n`.

3 Análise de desempenho

3.1 Exercício 1

Considere as seguintes funções:

```
fun method1(n: Int) :Int {
    var s = 0
    for ( i in 1 . . n ) {
        s += i
    }
    return s
}

fun method2(n : Int) : Int {
```

```
    var s = 0
    var i = 1
    while( i <= n ){
        s += i
        i = i*2
    }
    return s;
}
```

A complexidade da função method1 é de $O(n)$, pois o loop `for` irá executar `n` vezes. Já a complexidade da função method2 é aproximadamente $O(\log_2 n)$

3.2 Exercício 2

Considere o algoritmo `xpto`, que recebe como parâmetro dois inteiros `n` e `m`.

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

Quando m = 2, a complexidade de xpto em função de n é $O(\log_2 n)$
A complexidade de xpto é $O(\log_m n)$

4 Problema: Filtrar entre palavras

O problema proposto é 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.
Os parâmetros de execução são os seguintes:

- 0 : word1
- 1 : word2
- 2 : ficheiro output
- 3..N : ficheiros input...

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.

4.1 Implementações de ambas as versões

A resolução deste problema foi realizada de forma idêntica em ambas versões, alterando apenas da segunda para a primeira implementação o uso de listas da biblioteca do Kotlin.

- Verificar se os argumentos são válidos
- Declarar um array de `BufferedReader` (representa os ficheiros), `String` (para guardar a linha atual de cada ficheiro) e `Boolean` para sabermos se queremos avançar para a próxima linha no ficheiro, isto tudo com `n` posições, sendo `n` a quantidade de ficheiros para filtrar.
- Abrir os ficheiros todos
- Iterar até chegarmos ao final de todos os ficheiros (`endOfFile`)
- A cada iteração, iteramos todos os ficheiros, armazenando a linha atual em `currentFileLine` se for válida e não repetida
- Obtemos a menor palavra e o seu indíce, com a função `getMinStr`
- Escrevemos a palavra no ficheiro output
- Damos o valor de `true` a `changed` no índice do ficheiro, para na próxima iteração sabermos que queremos obter uma palavra nova só desse ficheiro

4.2 Avaliação experimental

Nesta secção do trabalho é onde iremos proceder à avaliação experimental referente ao problema. Aqui irão ser apresentados tabelas e gráficos onde estarão expostos os resultados experimentais, isto é, o tempo em milisegundos que cada uma das implementações demorou a correr em diversos casos. Os casos submetidos a avaliação experimental variam na escolha de palavras passadas como argumento (ana..joaquim) e (ana..pedro). A outra variação foi no número de palavras a serem filtradas. Para isso, temos três variantes: apenas 1 text file, 2 text files ou 3 text files. Desse modo filtramos entre 419846, 839692 e 1259538 palavras respetivamente. Irão ser exibidas duas tabelas e dois gráficos, sendo que os testes foram realizados em ambos os computadores do membro deste grupo, de modo a verificar também a diferença que existe entre vários computadores. É importante referir as especificações de ambos os computadores:
Primeiro computador: `Intel Core i5-3320M @ 2.60GHz, 8 GB RAM, 256 SSD`
Segundo Computador: `AMD Ryzen 7 5700U @ 1.80GHz, 16 GB RAM, 1T SSD`

Número de palavras a filtrar	419 846	839 692	1 259 538
1ª Implementação (ana .. joaquim)	0	92	145
2ª Implementação (ana .. joaquim)	0	107	169
1ª Implementação (ana .. pedro)	1	110	213
2ª Implementação (ana .. pedro)	1	133	235

Figura 1: Tempo de execução em milisegundos de cada implementação, consoante o número de palavras a filtrar no primeiro computador

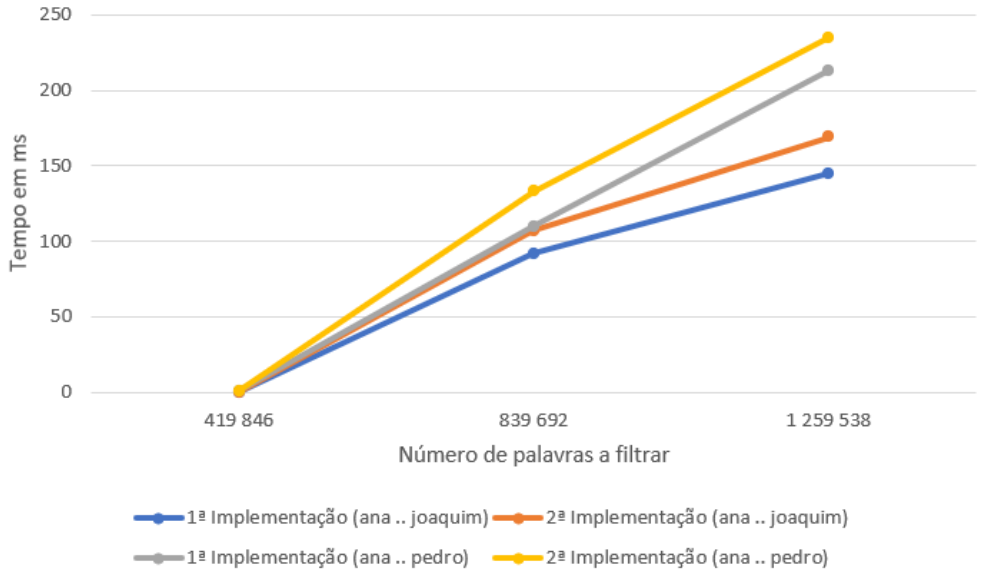

Figura 2: Gráfico representativo da variação de tempo em milisegundos em função do número de palavras a filtrar no primeiro computador

Número de palavras a filtrar	419 846	839 692	1 259 538
1ª Implementação (ana .. joaquim)	1	70	91
2ª Implementação (ana .. joaquim)	1	76	109
1ª Implementação (ana .. pedro)	1	68	124
2ª Implementação (ana .. pedro)	1	85	148

Figura 3: Tempo de execução em milisegundos de cada implementação, consoante o número de palavras a filtrar no segundo computador

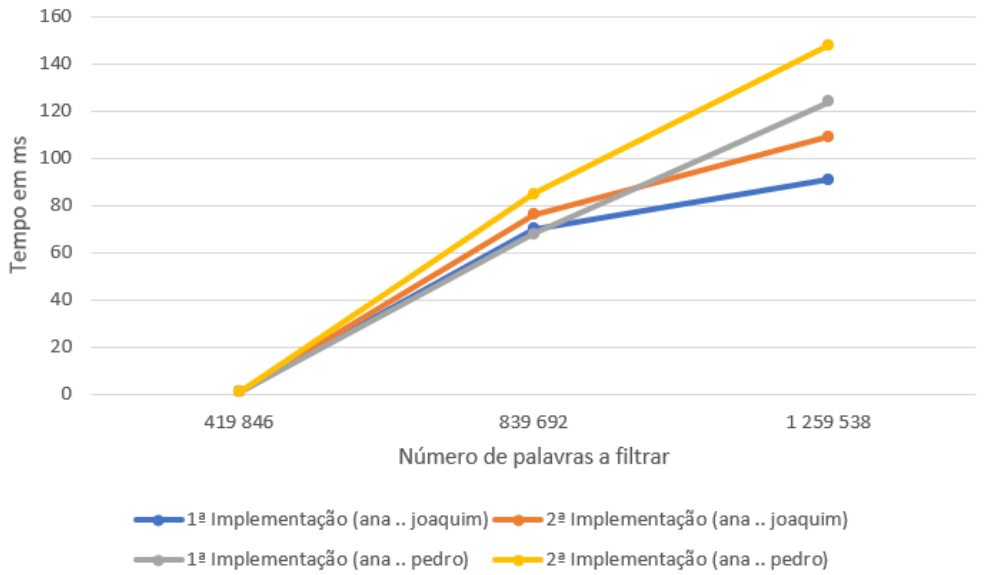

Figura 4: Gráfico representativo da variação de tempo em milisegundos em função do número de palavras a filtrar no segundo computador

5 Conclusão

Com esta série de exercicios, foi possível aprofundar o conhecimento sobre algoritmos e perceber a sua importância no desenvolvimento de programas, de modo a que sejam mais rápidos e ocupem menos memória.