Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

ALGORITMOS E ESTRUTURAS DE DADOS

Relatório da 1ª Série de Problemas

Docente

Prof.ª Cátia Vaz

Alunos

36368	Mihail Cirja
43552	Samuel Costa

Março 2018

Índice

Introdução

Este relatório, referente ao primeiro de três trabalhos práticos, foi elaborado no âmbito da unidade curricular de Algoritmos e Estruturas de Dados e acompanha a resolução de uma série de exercícios. Esta série de exercícios constituiu uma aplicação de técnicas de algoritmia, o uso de algoritmos de ordenação e pesquisa estudados nas aulas, a medição do tempo de execução de um algoritmo através do custo assintótico, e a comparação de algoritmos com base no seu grau de crescimento e outras características como a sua estabilidade.

Ao longo do documento apresentam-se a resolução de quatro exercícios com recurso a algoritmos elementares; três estudos de desempenho; e, por fim, uma breve reflexão sobre o trabalho realizado em que se discutem os resultados obtidos, explicitando-se, conclusões referentes ao trabalho desenvolvido.

1 Algoritmos Elementares

Foi proposta a resolução de quatro exercícios. Remete-se a descrição dos problemas para o enunciado do trabalho.

Exercício 1
Foi considerado o método com a seguinte a assinatura:

```
public static int removeIf(Integer[] v, int l, int r,
Predicate<Integer> predicate)
```

Atribuiu-se a uma variável auxiliar i o valor de l. Realizou-se um ciclo em que, a cada iteração se incrementou l, fixando-se como caso de paragem l>r. Contou-se o número de vezes que o elemento considerado v[l] cumpre o predicado predicate, marcando-se a posição i de v com o elemento v[l], e avançando o índice i. No final, a contagem resultante é retornada.

Exercício 2

Considerou-se o método:

```
public static int maximum (int[] v, int left, int right)
```

Sabe-se que dados:

- 5 inteiros left, mid, right, i, j, tais que left < i < mid < j < right;
- o array de inteiros (v,left,right);
- os *sub-arrays* a (v,left,mid), b (v,mid+1,right), e c (v,i,j),

se v tem um máximo, então este pertence a um dos 3 *sub-arrays* a, b ou c.

Sabe-se ainda que existe um inteiro pos tal que o *sub-array* (v, left, pos) é estritamente crescente e o *sub-array* (v, pos, right) é estritamente decrescente, sendo que *left* ≤ *pos* ≤ *right*. Portanto, dado um elemento v[*pos*] com left<pos<right, uma das três seguintes desigualdades é verdadeira:

[1] v[pos-1] < v[pos] > v[pos+1]
[2] v[pos-1] < v[pos] < v[pos+1]
[3] v[pos-1] > v[pos] > v[pos+1]

Nas condições descritas, pode dizer-se que no caso [1], v[pos] é máximo do sub-array (v, left, right). Nos casos [2] e [3], no entanto, a pesquisa continua, para tal sendo efetuada chamada recursiva do método `maximum`. No caso [2], sabe-se que o máximo se encontra na metade direita do array, no *sub-array* (v,mid+1,right), e, por fim, no caso [3], o máximo encontra-se necessariamente no *sub-array* (v,left,mid).

Dão-se ainda os casos em que pos é igual a left ou right, que são casos especiais da circunstância em que o *sub-array* considerado tem menos de 3 posições. Nesse caso, basta determinar qual dos elementos é maior e retorná-lo.

Por razões de coerência, deve considerar-se que se o protocolo de chamada da função não for cumprido é retornado -1.

Exercício 3

Considerou-se o método:

```
public static int[] countLessOrEqual(int[] a, int[] b)
```

Como os arrays a e b estão ordenados de forma crescente, sabe-se que, para qualquer posição *pos* de a, com x elementos de b menores ou iguais que a[*pos*], então, existem pelo menos x elementos em b menores ou iguais que a[*pos+1*].

Assim, o algoritmo desenvolvido para dar resposta ao problema consiste em:

- utilizar a variável auxiliar i, para percorrer o array a;
- utilizar a variável auxiliar j, para percorrer o array b, armazenando o número de elementos de b menores ou iguais que o elemento a[i] de a;

Enquanto j e i não alcançarem o tamanho do respetivo array, dá-se um de dois casos:

- o elemento na posição i de a é maior ou igual que a posição j de b, caso em que é necessário incrementar j
- pelo contrário, o elemento a[i] é menor que o elemento b[i], caso em que é necessário marcar a posição i de a com a contagem, armazenada em j, e passar ao próximo elemento de a, incrementando i.

Tendo sido ultrapassados os limites de um dos arrays, pode dar-se o caso de se ter percorrido todo o array b, significando que todos os elementos de b são menores ou iguais que os restantes de a. É então preciso marcar as posições sobrantes de a com j.

Exercício 4

O algoritmo desenvolvido para a resolução do exercício consiste em:

- ordenar os elementos do array de String v, recorrendo, para tal ao algoritmo QuickSort;
- para cada elemento v[i] de v, obter a string inversa `inv` e utilizar o algoritmo BinarySearch para pesquisar por `inv` no *sub-array* (v,i,r).
- Se for encontrada a string `inv` incrementar um contador em uma unidade.

Note-se que se a última letra do elemento v[i] for lexicograficamente inferior à primeira, não é necessário efetuar pesquisa, evitando-se chamadas redundantes aos métodos auxiliares `invert` e `binarySearch`, dado que v está ordenado alfabeticamente e que, por isso, o elemento já foi pesquisado.

2 Análise de Desempenho

Exercício 1

Escolha de candidato:

$$C_{equalToIndex}(n) = \begin{cases} O(1)\,, & se\ n = 0 \\ O(1) + C_{equalToIndex}\left(\frac{n}{2}\right), & se\ n > 0 \end{cases}$$

Resolução da equação de recorrência:

$$C_{equalToIndex}(n) = O(1) + C_{equalToIndex}\left(\frac{n}{2}\right)$$

$$C_{equalToIndex}(n) = O(1) + O(1) + C_{equalToIndex}\left(\frac{n}{4}\right)$$

$$C_{equalToIndex}(n) = O(1) + O(1) + O(1) + C_{equalToIndex}\left(\frac{n}{8}\right)$$

$$C_{equalToIndex}(n) = k.O(1) + C_{equalToIndex}\left(\frac{n}{2^k}\right); com\ k = 2log_2 n$$

$$C_{equalToIndex}(2^m) = \begin{cases} O(1), se\ m = 0 \\ O(1) + C_{equalToIndex}(2^{m-1}) \end{cases}$$

$$C_{equalToIndex}(2^m) = O(1) + O(1) + C_{equalToIndex}(2^{m-2})$$

$$C_{equalToIndex}(2^m) = O(1) + O(1) + O(1) + C_{equalToIndex}(2^{m-3})$$

$$C_{equalToIndex}(2^m) = m.O(1) + C_{equalToIndex}(2^{m-m})$$

$$C_{equalToIndex}(2^m) = O(m) + O(1) = O(m+1)$$

$$C_{equalToIndex}(2^m) = O(m)$$

$$C_{equalToIndex}(n) = O(log_2 n)$$

Exercício 2

Escolha de candidato:

$$C_{xpto}(n) = \begin{cases} O(1), se\ n = 0 \\ 2C\left(\frac{n}{2}\right) + O(1), se\ n > 0 \end{cases}$$

Resolução da equação de recorrência:

$$C_{xpto}(n) = 2C\left(\frac{n}{2}\right) + O(1)$$

$$C_{xpto}(n) = 2\left(2C\left(\frac{N}{4}\right) + O(1)\right) + O(1)$$

$$C_{xpto}(n) = 4C\left(\frac{N}{4}\right) + 2.O(1) + O(1)$$

$$C_{xpto}(n) = 4\left(2C\left(\frac{N}{8}\right) + 2.O(1)\right) + O(1) + O(1)$$

$$C_{xpto}(n) = 8C\left(\frac{N}{8}\right) + \ 4.O(1) + O(1) + O(1)$$

$$C_{xpto}(n) = 2^m C\left(\frac{N}{2^m}\right) + m.O(1)$$

Com $n = 2^m$

$$C_{xpto}(n) = 2^m C(2^{m-m}) \ + m.O(1)$$

$$C_{xpto}(n) = n.O(1) \ + \lg n\,.O(1)$$

$$C_{xpto}(n) = O(n) \ + O(\lg n) = O(n)$$

Exercício 3

Considere-se a ordenação de n inteiros armazenados no array `a`:

- primeiramente encontrando o menor elemento de `a`, e armazenando-o numa variável auxiliar;
- troquem-se todos os elementos de `a[i]` a `a[min]` uma posição para a direita, por forma a criar espaço para o menor elemento;
- coloque-se o menor elemento na posição `a[0]`;
- Proceda-se desta forma para os primeiros n-1 elementos de `a`.

Análise da complexidade quanto ao tempo:

$n.O(n)$ pesquisa pelo i-ésimo menor elemento;

$\sum_{k=0}^{n-1} k = \frac{n\ x\ (n-1)}{2}$, no melhor caso, diz respeito às trocas sucessivas para a direita, que no pior caso é $O(n).n$;

O(1) colocar o elemento na ordem correta;

Assim, o algoritmo tem complexidade quanto ao tempo de $n.O(n) + O(n).n + O(1).n = O(n^2) + O(n^2) + O(n) = O(n^2)$.

Análise de complexidade quanto ao espaço:

$O(1) + O(1) = O(1)$, pois é necessário guardar índice do i-ésimo menor elemento, para além do próprio elemento.

Conclusões

Os métodos desenvolvidos foram testados e apresentam conformidade com os testes fornecidos.

É de salientar que a elaboração deste trabalho (série e relatório), revelou-se importante na aprendizagem, uma vez que vem introduzir um critério mensurável para categorização e escolha de algoritmos. É ainda de notar a perceção tida no contexto da elaboração deste trabalho, de que as soluções mais espontâneas ou ingénuas correspondem frequentemente a algoritmos com maior custo assintótico, constituindo-se como motivação para o estudo das técnicas de algoritmia, cujo emprego possa resultar em ganhos de eficiência nos programas desenvolvidos.