

## Exercise 1

### Solution

a)

It is clear that:

$$T_1 \in O(n)$$

$$T_2 \in O(n^2)$$

Given the proper definition of the function class  $O(g)$  ("Big O") of a function  $g : \mathbb{N} \rightarrow \mathbb{R}$  as follows;

$$O(g) := \{f : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c \in \mathbb{R}, n_0 \in \mathbb{N}, f(n) \leq cg(n), \forall n > n_0\}$$

b)

We assume  $n \neq 0$ , for obviously  $T_1(0) = T_2(0)$ , so  $T_2$  is more efficient if:

$$T_1(n) > T_2(n) \Leftrightarrow 625n > n^2 \Leftrightarrow 625 > n$$

And the opposite is true for  $625 < n$ , for  $n = 625$  the algorithms perform equally.

## Exercise 2

### Solution

Algorithms  $T_1, \dots, T_5, T_7, \dots, T_9$  are trivial. for  $T_6$ , if  $n > 1$  then  $T_6 \in O(n \log^2(n))$ . For  $T_{10}$ , notice that:

$$T(2) = 2T(1) + 2 \in O(1),$$

$$T(3) = 2T(2) + 2 = 2[2T(1) + 2] + 2$$

.  
.  
.

$$T(n) = 2^n[T(1) + 1] + 2 \in O(2^n).$$

## Exercise 3

### Solution

a)

```
1 int a = 0, b = 0;
2 for (i = 0; i < n; i++) {
3     a = a + i;
4 }
5 for (j = 0; j < m; j++) {
6     b = b + j;
7 }
```

 C++

The algorithm operates in  $O(n) + O(m)$ , the first for grows linearly with  $n$  and the second one grows with  $m$ .

b)

```
1 float what2(int *arr, int n) {  
2     int a = 0;  
3     for (int i = 0; i < n; i++) {  
4         if(arr[i] > 10) {  
5             for (int j = 0; j < n; j++) {  
6                 a += n / 2;  
7             }  
8         } else {  
9             printf("ok :(")  
10        }  
11    }  
12 }
```

The algorithm has 2 for's with  $n$  operations each in a worst-case scenario, therefore it is precisely  $\in O(n^2)$ .

c)

```
1 int a = 0;  
2 for (int i = 0; i < n; i++) {  
3     for (int j = n; j > i; j--) {  
4         a += i + j;  
5     }  
6 }
```

For  $i = 0$ ,  $j$  goes from  $n$  to 0, and the operation  $a += n/2$  is executed  $n$  times, for  $i = 1$ ,  $n$  goes from  $n$  to 1 and the operation is executed  $n - 1$  times and so on, so we have:

$$\sum_{i=1}^{n-1} n - i = \frac{n(n+1)}{2}$$

So the algorithm is  $\in O(n^2)$ .

d)

```
1 float what4(int *arr, int n) {  
2     int a = 0;  
3     for (int i = 0; i < 1000; i++) {  
4         for (int j = 0; j < 5000; j++) {  
5             a += i + j;  
6         }  
7     }  
8 }
```

Both loops have constant limits, so the total complexity is  $O(1)$ .

e)

```
1 int a = 0;  
2 for (int i = n/2; i <= n; i++) {
```

```

3     for (int j = 2; j <= n; j = j * 2) {
4         a += i + j;
5     }
6 }

```

The external loop executes  $O(n)$  times, and the internal one  $O(\log n)$ . So the total complexity is  $O(n \log n)$ .

f)

```

1 int a = 0, i = n;
2 while (i > 0) {
3     a += i;
4     i /= 2;
5 }

```

C++

The only difference is that now the operation  $a += i$  is being executed when  $i = n, \frac{n}{2}, \frac{n}{4}, \dots, 0$ , which is a sum that shows the total complexity of the algorithm:

$$\sum_{i=0}^{\log(n)} \frac{n}{2^i} = 2n - 1 \in O(n).$$

g)

```

1 int a = 0, i = n;
2 while (i > 0) {
3     for (int j = 0; j < i; j++) {
4         a += i;
5     }
6     i /= 2;
7 }

```

C++

A cada iteração do while, o valor de  $i$  é dividido por 2. Na primeira iteração, o for executa  $n$  vezes, depois  $n/2, n/4, \dots$ , até  $i = 1$ . A soma total de operações é:

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = 2n - 1 \in O(n)$$

Portanto, a complexidade do algoritmo é  $O(n)$ .

h)

```

1 float soma(float *arr, int n) {
2     float total = 0;
3     for (int i = 0; i < n; i++) {
4         total += arr[i];
5     }
6     return total;
7 }

```

C++

O algoritmo percorre uma vez o vetor de tamanho  $n$ , realizando uma soma por elemento. Portanto, sua complexidade é linear:  $O(n)$ .

i)

```
1 int buscaSequencial(int *arr, int n, int x) {
2     for (int i = 0; i < n; i++){
3         if (arr[i] == x) {
4             return i;
5         }
6     }
7     return -1;
8 }
```

No pior caso (quando  $x$  não está no vetor), o algoritmo percorre todos os  $n$  elementos. Assim, sua complexidade no pior caso é  $O(n)$ .

j)

```
1 int buscaBinaria(int *arr, int x, int i, int j) {
2     if (i >= j) {
3         return -1;
4     }
5
6     int m = (i + j) / 2;
7     if (arr[m] == x) {
8         return m;
9     } else if ( x < arr[m] ) {
10        return buscaBinaria (arr, x, i, m-1);
11    } else {
12        return buscaBinaria (arr, x, m+1, j);
13    }
14 }
```

A cada chamada recursiva o espaço de busca é reduzido pela metade. Assim, a complexidade da busca binária no pior caso é  $O(\log n)$ .

k)

```
1 void multiplicaiMatriz(float **a, float **b, int n, int p, int m, float
  **x) {
2     for (int i = 0; i < n; i++) {
3         for (int j = 0; j < m; j++) {
4             x[i][j] = 0.0;
5             for (int k = 0; k < p; k++) {
6                 x[i][j] += a[i][k] * b[k][j];
7             }
8         }
9     }
10 }
```

A multiplicação de matrizes de dimensões  $np$  por  $pm$  realiza  $nmp$  multiplicações. Logo, a complexidade é  $O(nmp)$ .