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 $g : \mathbb{N} \to \mathbb{R}$ as follows;

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

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

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

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

$$\vdots$$

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

Exercise 3

Solution

a)

C++

```cpp
int a = 0, b = 0;
for (i = 0; i < n; i++) {
    a = a + i;
}
for (j = 0; j < m; j++) {
    b = b + j;
}
```

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

b)

```cpp
float what2(int *arr, int n) {
    int a = 0;
    for (int i = 0; i < n; i++) {
        if(arr[i] > 10) {
            for (int j = 0; j < n; j++) {
                a += n / 2;
            }
        } else {
            printf("ok :(")
        }
    }
}
```

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

c)

```cpp
int a = 0;
for (int i = 0; i < n; i++) {
    for (int j = n; j > i; j--) {
        a += i + j;
    }
}
```

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)

```cpp
float what4(int *arr, int n) {
    int a = 0;
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 5000; j++) {
            a += i + j;
        }
    }
}
```

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

e)

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

```cpp
    for (int j = 2; j <= n; j = j * 2) {
        a += i + j;
    }
}
```

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

f)

```cpp
int a = 0, i = n;
while (i > 0) {
    a += i;
    i /= 2;
}
```

The only difference is that now the operation a += i is being executed when $i = n, \frac{n}{2}, \frac{n}{4}, \ldots, 0$, 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)

```cpp
int a = 0, i = n;
while (i > 0) {
    for (int j = 0; j < i; j++) {
        a += i;
    }
    i /= 2;
}
```

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, ..., até i = 1. A soma total de operações é:

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

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

h)

```cpp
float soma(float *arr, int n) {
    float total = 0;
    for (int i = 0; i < n; i++) {
        total += arr[i];
    }
    return total;
}
```

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

i)

```cpp
int buscaSequencial(int *arr, int n, int x) {
    for (int i = 0; i < n; i++){
        if (arr[i] == x) {
            return i;
        }
    }
    return -1;
}
```

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)

```cpp
int buscaBinaria(int *arr, int x, int i, int j) {
    if (i >= j) {
        return -1;
    }

    int m = (i + j) / 2;
    if (arr[m] == x) {
        return m;
    } else if ( x < arr[m] ) {
        return buscaBinaria (arr, x, i, m-1);
    } else {
        return buscaBinaria (arr, x, m+1, j);
    }
}
```

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)

```cpp
void multiplicacaiMatriz(float **a, float **b, int n, int p, int m, float **x) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            x[i][j] = 0.0;
            for (int k = 0; k < p; k++) {
                x[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

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

Exercise 4

Solution

Currently empty. We encourage the reader (me lol) to do this as homework.