Lista 4 - Francisco Braz

1-)
a-)

```
função verificaSoma(S, T, tamA, tamB, x)
inicio
        i := 1
        j := tamB
        enquanto i ≤ tamA && j ≥ 1
        inicio
             soma := S[ i ] + T[ j ]
             se soma = x então retorna 1
             se não se soma > x então j = j -1
             senão i = i + 1
        fim
        retorna 0
fim

função intercala(Y, esq, meio, dir)
início
    tamA := m-esq+1; tamB := dir-meio;
    para i = 1 até tamA faça A[ i ] := Y[esq + 1]
    para j = 1 até tamB faça B[ j ] := Y[meio + 1 + j]

    i := 1; j := 1; k := esq
    enquanto (i < tamA) && (j < tamB) faça
    início
        se A[ i ] <= B[ j ] então {Y[ k ] := A[ i ]; i := i+1}
        senão {Y[ k ] := B[ j ]; j := j+1}
        k := k+1;
   fim

   enquanto i < tamA faça {Y[ k ] := A[ i ]; i := i+1; k := k+1}
   enquanto j < tamB faça {Y[ k ] := B[ j ]; j := j+1; k:= k+1}
fim
```

```
função mergeSort(S, esq, dir)
início
    se esq < dir então
        meio := ⌊ (esq + dir) / 2 ⌋
        mergeSort(X, esq, meio)
        mergeSort(X, meio+1, dir)
        intercala(X, esq, meio, dir)
fim
```

b-)

Complexidade da função verificaSoma:

De acordo com as simplificações vistas na aula 05, podemos focar nas operações principais do algoritmo, que seriam aquelas que consomem mais tempo para executar. Nessa função a operação principal seria o conjunto de tarefas feitas dentro do loop "enquanto".

Parte principal:

```
enquanto i <= tamA && j >= 1
        inicio
            soma := S[ i ] + T[ j ]
            se soma = x então retorna 1
            se não se soma > x então j = j -1
            senão i = i + 1
        fim
```

Considerando n como sendo a soma das entradas dos dois vetores, no pior caso poderíamos ter n-1 execuções para uma das variáveis e n execuções para a outra. Por exemplo, n-1 execuções para i e n execuções para a variável j (ou vice-versa). Portanto, temos:

$$\sum_{i=1}^{n-1} c1 + \sum_{j=1}^{n} c2$$

$= c_1 * ((n - 1) - 1 + 1) + c_2 * (n-1+1)$

$= c_1 * (n-1) + c_2 * n$

$= n * (c_1 + c_2) - c_1$

Para grandes valores de N, as constantes seriam relativamente insignificantes, então nossa complexidade seria O(n).

Complexidade da função merge:

```
início
    se esq < dir então
        meio := ⌊ (esq + dir) / 2 ⌋          O(1)
        mergeSort(X, esq, meio)             T( ⌊n/2⌋ )
        mergeSort(X, meio+1, dir)           T( ⌈n/2⌉ )
        intercala(X, esq, meio, dir)
fim
```

Complexidade da função intercala:

```
função intercala(Y, esq, meio, dir)
início
    tamA := m-esq+1; tamB := dir-meio;
    para i = 1 até tamA faça A[ i ] := Y[esq + 1]  n-1
    para j = 1 até tamB faça B[ j ] := Y[meio + 1 + j] n-1

    i := 1; j := 1; k := esq
    enquanto (i < tamA) && (j < tamB) faça
    início
        se A[ i ] <= B[ j ] então {Y[ k ] := A[ i ]; i := i+1}
        senão {Y[ k ] := B[ j ]; j := j+1}                          n-1
        k := k+1;
    fim

    enquanto i < tamA faça {Y[ k ] := A[ i ]; i := i+1; k := k+1} n-1
    enquanto j < tamB faça {Y[ k ] := B[ j ]; j := j+1; k:= k+1} n-1
fim
```

Portanto, teríamos:

$$\sum_{i=1}^{n-1} c1 + \sum_{j=1}^{n-1} c2 + \sum_{i=1}^{n-1} c3 + \sum_{j=1}^{n} c4 + \sum_{i=1}^{n-1} c5 + \sum_{j=1}^{n-1} c6$$

$$\sum_{i=1}^{n-1} c1 = c_1 * ((n - 1) - 1 + 1) = c_1 * (n-1)$$
$$\sum_{j=1}^{n-1} c2 = c_2 * ((n - 1) - 1 + 1) = c_2 * (n-1)$$
$$\sum_{i=1}^{n-1} c3 = c_3 * ((n - 1) - 1 + 1) = c_3 * (n-1)$$
$$\sum_{j=1}^{n} c4 = c_4 * (n - 1 + 1) = c_4 * n$$
$$\sum_{i=1}^{n-1} c5 = c_5 * ((n - 1) - 1 + 1) = c_5 * (n-1)$$
$$\sum_{j=1}^{n-1} c6 = c_6 * ((n - 1) - 1 + 1) = c_6 * (n-1)$$

Somando os resultados, teríamos:
$n * (c_1 + c_2 + c_3 + c_4 + c_5 + c_6) - c_1 - c_2 - c_3 - c_5 - c_6$

Considerando somente o termo de maior grau temos que a complexidade é O(n).

Juntando a complexidade das três funções, teríamos:
T(n) = 2T(n/2) + O(n) + O(n)
T(n) = 2T(n/2) + 2O(n)

Pelo teorema mestre:
a = 2, b = 2, f(n) = 2n

OBS: n^log$_b$a - ε = $n^{log_b a - \varepsilon}$

1ºCaso: Se f(n)∈O(n^log$_b$a - ε) então T(n) = θ(n^log$_b$a), ε > 0

2n ∈ O(n^log$_2$2 - ε)
2n ≤ c * n^log$_2$2 - ε
2n ≤ c * (n^1-ε)

Como ε precisa ser maior do que 0, nosso n ficará elevado à um número menor do que 1, portanto essa desigualdade não será verdadeira. Podemos substituir o ε por 0.5, por exemplo.

2º Caso: Se $f(n) \in \theta(n^{\log_b a})$ então $T(n) = \theta(n^{\log_b a} * \log n)$

$2n \in \theta(n^{\log_b a})$
$2n \in \theta(n^{\log_2 2})$
$c_2 * n^{\log_2 2} \leq 2n \leq c_1 * n^{\log_2 2}$
$c_2 * n^1 \leq 2n \leq c_1 * n^1$

Como as duas desigualdades são verdadeiras, o nosso segundo caso é verdadeiro. Portanto, temos:

$T(n) = \theta(n^{\log_b a} * \log n)$
$= \theta(n^{\log_2 2} * \log n)$
$= \theta(n * \log n)$

c) Implementação no arquivo somaReal.c

2-)
a-)

```
função intercala(Y, esq, meio, dir): inteiro
início
    inversoes := 0
    tamA := m-esq+1; tamB := dir-meio;
    para i = 1 até tamA faça A[ i ] := Y[esq + 1]
    para j = 1 até tamB faça B[ j ] := Y[meio + 1 + j]

    i := 1; j := 1; k := esq
    enquanto (i < tamA) && (j < tamB) faça
    início
        se A[ i ] <= B[ j ] então {Y[ k ] := A[ i ]; i := i+1}
        senão
            Y[ k ] := B[ j ]
            j := j+1
```

```
            inversoes := tamA - i
        k := k+1;
    fim

    enquanto i < tamA faça {Y[ k ] := A[ i ]; i := i+1; k := k+1}
    enquanto j < tamB faça {Y[ k ] := B[ j ]; j := j+1; k:= k+1}
    retorna inversoes
fim



função mergeSort(S, esq, dir)
início
    inversoes := 0
    se esq < dir então
    início
        meio := ⌊ (esq + dir) / 2 ⌋
        inversoes = inversoes + mergeSort(X, esq, meio)
        inversoes = inversoes + mergeSort(X, meio+1, dir)
        inversoes = inversoes + intercala(X, esq, meio, dir)
    fim
    retorna inversoes
fim
```

b-) As complexidades da função mergeSort e da função intercala já foram calculadas na primeira questão. A intercala possui complexidade de O(n) e a mergeSort possui O(1), T(⌊n/2⌋) e T(⌈n/2⌉). Logo fazendo a junção dessas complexidades, temos:

$T(n) = 2T(n/2) + O(n)$

Pelo teorema mestre:
$a = 2$, $b = 2$, $f(n) = n$

OBS: n^log$_b$a - ε = $n^{log_b a - \varepsilon}$

1ºCaso: Se $f(n) \in O(n^{\log_b a - \varepsilon})$ então $T(n) = \theta(n^{\log_b a})$, $\varepsilon > 0$

$n \in O(n^{\log_2 2 - \varepsilon})$
$n \leq c * n^{\log_2 2 - \varepsilon}$
$n \leq c * (n^{1-\varepsilon})$

Como ε precisa ser maior do que 0, nosso n ficará elevado à um número menor do que 1, portanto essa desigualdade não será verdadeira. Podemos substituir o ε por 0.5, por exemplo.

2º Caso: Se $f(n) \in \theta(n^{\log_b a})$ então $T(n) = \theta(n^{\log_b a} * \log n)$

$n \in \theta(n^{\log_b a})$
$n \in \theta(n^{\log_2 2})$
$c2 * n^{\log_2 2} \leq n \leq c1 * n^{\log_2 2}$
$c_2 * n^1 \leq n \leq c_1 * n^1$

Como as duas desigualdades são verdadeiras, o nosso segundo caso é verdadeiro. Portanto, temos:

$T(n) = \theta(n^{\log_b a} * \log n)$
$= \theta(n^{\log_2 2} * \log n)$
$= \theta(n * \log n)$

c-) Implementação no arquivo numeroInversoes.c

3-)
a-)
```
função ordenarNumeros(X, tam)
início
    i := 1;
    j := 2;
    enquanto (i <= tam-1) && (j <= tam)
    início
        se X[ i ] >= 0 && X[ j ] < 0 então
```

```
        início
            aux := X[ i ]
            X[ i ] := X[ j ]
            X[ j ] := aux
            i++
        fim
        j = j +1
    fim
    i:=1
    j:=2
    enquanto (i <= tam-1) && (j <= tam)
    início
        se X[ i ] <= 0 então i = i+1
        senão se X[ i ] > 0 && X[ j ] == 0 então
        início
            aux := X[ i ]
            X[ i ] := X[ j ]
            X[ j ] := aux
            i++
        fim
        j = j +1
    fim
fim
```

b-)

Nossas operações principais (aquelas que gastariam maior tempo de execução) estão dentro dos loops “enquanto”. Esse conjunto de operações dentro dos loops será executado no máximo n-1 vezes em ambos os casos. Ou seja, iremos ter:

$$\sum_{i=1}^{n-1} c1 + \sum_{i=1}^{n-1} c2$$

$= c_1 * ((n-1)-1+1) + c_2 * (n-1 +1)$

$= c_1 * (n-1) + c_2 * n$

$= n * (c_1 + c_2) - c_1$

Se ignorarmos as constantes e considerarmos os termos de maior grau, teríamos uma complexidade de O(n).

c-) Implementação no arquivo numPosNegNulo.c

4-)
a-)

```
função partição()
início
    pivo := X[esq]
    L := esq
    R := dir
    enquanto L < R faça
    início
        se pivo % 2 = 0 então
            enquanto (X[L] <= pivo) e (L <= dir) faça L := L+1
            enquanto (X[R] > pivo) e (R >= esq) faça R := R-1
            se (L < R) então Troca(X[L], X[R])
        senão
            enquanto (X[L] >= pivo) e (L <= dir) faça L := L+1
            enquanto (X[R] < pivo) e (R >= esq) faça R := R-1
            se (L < R) então Troca(X[L], X[R])
    fim
    pos := R
    X[esq] = X[pos]
    X[pos] = pivo
    retorne pos
fim

função separar(X, tam)
início
    i := 0; j := tam;
    enquanto (i < j )
    início
```

```
        se X[ i ] % 2 ≠ 0 e X[ j ] % 2 = 0 então
            aux :=  X[ i ]
            X[ i ] := X[ j ]
            X[ j ] := aux
            i := i + 1
            j := j - 1
        senão se (X[ i ] % 2 ≠ 0) então j := j - 1
        senão se X[ i ] % 2 = 0) então i := i + 1
    fim
fim


função quickSort(X, esq, dir)
início
  se (esq < dir) então
      pos := Particao(X, esq, dir)
      quickSort(X, esq, pos-1)
      quickSort(X, pos+1, dir)
fim
```

b-)
Para a função quickSort, considerando o pior caso, teríamos uma partição vazia e a outra com o restante dos elementos. Desse modo, estaríamos passando para as próximas chamadas n-1 elementos e assim teríamos T(n-1), enquanto nossa parte vazia teria um . Como uma das partições ficaria vazia, teríamos um T(0). Nosso T(0) seria igual a T(1) que seria igual a um tempo constante, O(1).

Para a função de partição, poderíamos analisar somente a parte que mais consumiria tempo de execução, que seria o nosso loop “enquanto”

```
enquanto L < R faça
início
    se pivo % 2 = 0 então
        enquanto (X[L] <= pivo) e (L <= dir) faça L := L+1
        enquanto (X[R] > pivo) e (R >= esq) faça R := R-1
        se (L < R) então Troca(X[L], X[R])
    senão
        enquanto (X[L] >= pivo) e (L <= dir) faça L := L+1
        enquanto (X[R] < pivo) e (R >= esq) faça R := R-1
        se (L < R) então Troca(X[L], X[R])
fim
```

Nesse caso, apesar de existir outro loop "enquanto" dentro do primeiro loop, nós percorremos os elementos do array no máximo 1 única vez, pois quando a nossa variável R não for maior do que L, esse ciclo será interrompido. Portanto, teríamos um comportamento linear, ou seja, O(n).

Juntando essas duas partes, temos:

$T(n) = T(n-1) + O(n), T(1) = c$

$$
\begin{aligned}
T(n) &= T(n-1) + n \\
&= (T(n-2) + n-1) + n \\
&= ((T(n-3) + n-2) + n-1) + n \\
&\quad . \\
&\quad . \\
&\quad . \\
&= T(1) + 2 + 3 + \ldots + n-1 + n \\
&= 1 + 2 + 3 + \ldots + n-1 + n \\
&= n(n+1)/2 \\
&= n^2/2 + n
\end{aligned}
$$

Considerando somente o termo de maior grau, temos $O(n^2)$

c-) Implementação no arquivo quickSortParImpar.c