Projeto 1
Análise de Complexidade

João Pedro Machado Silva, BV3032477

OBS: Na análise presente neste documento, o cálculo do T(n) foi feito considerando o pior caso, ou então, desconsiderando o custo das constantes, visto que são insignificantes para o resultado final.

TreeSumForcaBruta():

```
void treeSumForcaBruta(int A[], int n) {
   int contTriplas = 0;
   for (int i = 0; i < n - 2; i++) {
       for (int j = i + 1; j < n - 1; j++) {
           for (int k = j + 1; k < n; k++) {
               qtdOperacoes3SumFB
               if (A[i] + A[j] + A[k] == 0) {
                   contTriplas++;
                   printf("%d Tripla Encontrada: [%d, %d, %d]\n",
contTriplas, A[i], A[j], A[k]);
               }
           }
       }
   }
    printf("Total Triplas Encontradas pela Forca Bruta: %d\n",
contTriplas);
}
```

Ao analisar o algoritmo acima, notamos que ele possui 3 for's com complexidade n, um dependendo do outro. As outras instruções dentro do for também se repetem n vezes, temos então $T(n) = n^3 + 4n + 2$. Concluímos então que **$T(n) \in O(n^3)$.**

TreeSumMelhorado():

```
void treeSumMelhorado(int A[], int n) {
   int contTriplas = 0;

   MergeSortRecursivo(A, 0, n-1, n);
   ImprimeArray(A, "Array Ord.    []", n);

   for (int i = 0; i < n - 2; i++) {
       for (int j = i + 1; j < n - 1; j++) {
           qtdOperacoes3SumMelhorado++;
           int x = -(A[i] + A[j]);
           int k = BuscaBinaria(x, A, j + 1, n - 1);
           if (k != -1) {
               contTriplas++;
                  printf("\n%d Tripla Encontrada: [%d, %d, %d]",
contTriplas, A[i], A[j], A[k]);
           }
       }
   }

    printf("Total Triplas Encontradas pelo 3SUM Melhorado: %d\n",
contTriplas);
}
```

No algoritmo acima, temos uma chamada do MergeSort(n.lgn), 1 chamada do ImprimeArray(n), depois 2 for dependentes e uma chamada de BuscaBinaria(lgn). Tendo isso em vista e ignorando os custos constantes que são insignificantes, temos:

T(1) = 1, para n=1

T(n) = n^2.lgn + n.lgn + n, para n>1

Sendo assim, o termo de maior ordem é a complexidade do algoritmo: **T(n)** ∈ **O(n^2.lgn).**

BuscaBinaria():

```
int BuscaBinaria (int x, int A[], int inicio, int fim) {
```

```
    // Verifica se o início é menor ou igual ao fim
    if (inicio <= fim) {
        int meio = inicio + (fim - inicio) / 2;
        // Verifica se o elemento está no meio
        if (A[meio] == x) {
            return meio;
        }

        // Se o elemento está na metade esquerda do array
        if (A[meio] > x)
            BuscaBinaria(x, A, inicio, meio - 1);

        // Se o elemento está na metade direita do array
        else
            BuscaBinaria(x, A, meio + 1, fim);
    } else {
        return -1;
    }
}
```

O algoritmo possui instruções que são constantes e a chamada recursiva, em que cada vez, será chamada apenas 1 vez. Sendo assim, temos que a chamada recursiva é pela metade, T(n/2) e mais ou menos 5 instruções constantes. Sendo assim temos que:

T(n) = 1, para n=1
T(n) = T(n/2) + 5, pana n>1

Usando o método da substituição:
T(n) ∈ O(lgn)
T(n) <= c.lgn
T(n) <= 2(c.lgn/2) + 5
T(n) <= 2.c.(lgn - lg2) + 5
T(n) <= c.lgn - c + 5
T(n) <= c.lgn **-> Provado**

MergeSortRecursivo():

```
void MergeSortRecursivo(int A[], int inicio, int fim, int n) {
   int meio;

   if(inicio < fim){
       meio = (inicio + fim) / 2;
       MergeSortRecursivo(A, inicio, meio, n);
       MergeSortRecursivo(A, meio+1, fim, n);
       IntercalaSemSentinela(A, inicio, meio, fim, n);
   }
}
```

Analisando o algoritmo acima, notamos que ele possui duas chamadas recursivas dividindo pela metade e uma chamada do intercalaSemSentinela, sendo assim, as duas chamadas são 2.T(n/2) e o IntercalaSemSentinela tem complexidade n, depois temos mais uma constante 2 de custo de instrução, mas podemos desconsiderar, pois não é significante:

T(n) = 1, para n=1
T(n) = 2T(n/2) + n, para n>1

Utilizando o método da substituição:
T(n) ∈ O(n.lgn)
T(n) <= c.n.lgn
T(n) <= 2(c.n/2.lgn/2) + n
T(n) <= 2.c.n/2.(lgn - lg2) + n
T(n) <= c.n.lgn - c.n + n
T(n) <= c.b.lgn **-> Provado**

IntercalaSemSentinela():

```
void IntercalaSemSentinela(int A[], int inicio, int meio, int
fim, int n) {
   int i, j;
   int *arrayB;

   arrayB = (int*) malloc( (fim+1) * sizeof(int) );

   for(i=inicio; i<=meio; i++) {
       arrayB[i] = A[i];
   }
   for(j=meio+1; j<=fim; j++) {
       arrayB[fim+meio+1-j] = A[j];
   }

   i = inicio;
   j = fim;

   for(int k=inicio; k<=fim; k++) {
       qtdOperacoes3SumMelhorado++;
       if(arrayB[i] <= arrayB[j]) {
           A[k] = arrayB[i];
           i++;
       } else {
           A[k] = arrayB[j];
           j--;
       }
   }

   free(arrayB);
}
```

No algoritmo, notamos que existem 3 for, porém são independentes entre si, somando os custos, teremos T(n) = 7.n + 5. Independente do número multiplicador e da constante somada, temos que **T(n) ∈ O(n).**

ImprimeArray():

```
void ImprimeArray(int A[], char Msg[], int n) {
   printf("\n %s = ", Msg);

   for (int i=0; i<n; i++) {
       printf(" %d", A[i]);
   }
}
```

O algoritmo possui apenas 1 for e 1 printf, sendo assim, o for tem custo n, junto com o printf e o printf custo 1. Sendo assim, T(n) = 2n + 1, porém independente da constante que multiplica n e da constante que soma-se, temos que **T(n) ∈ O(n).**