Aluno: Rodrigo Gonçalves
Matrícula: 11325752

InsertionSort:

1) A melhor complexidade do InsetionSort, ocorre quando o vetor estar completamente ordenado, onde a sua complexidade é O(n).

Ex.:

4	5	6	7

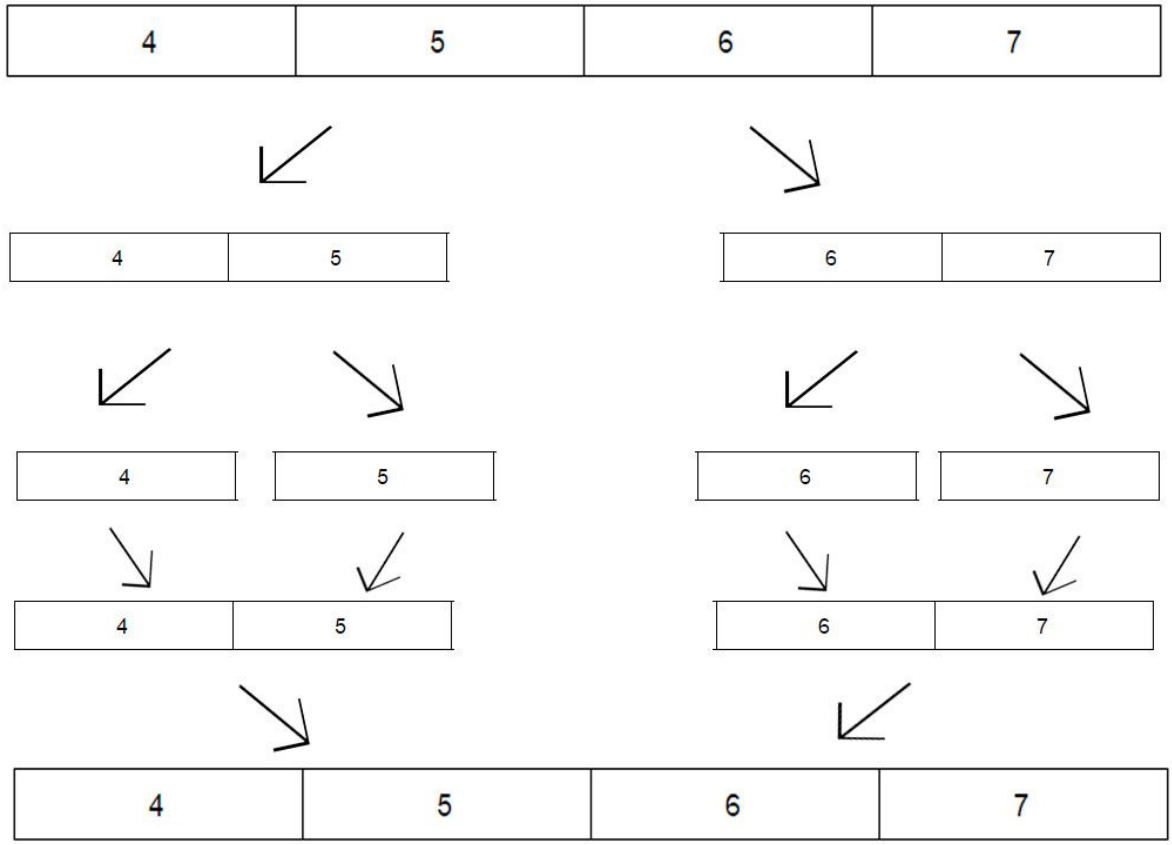
Em cada interação, o elemento atual de interação do vetor é comparado com aquele que o antecede, de modo que se o vetor estiver completamente ordenado, o algorítmo irá caminhar por todo a array apenas fazendo comparação dessa natureza, logo intuitivamente o custo desse processo é linear.
Em outras palavras: $\forall$ vetor A ^ i $\in$ N, tal que 0 <= i <= n - 1 v[1] > v[0], v[2] > v[1], … , v[n - 1] > v[n - 2] $\Rightarrow$ O(n).

MergeSort:

O mergeSort terá a mesma complexidade para todos os casos e igual a O(nlogn).

Ex.:

O exemplo mostra o processo do MergeSort em um vetor completamente ordenado.
Onde a altura do vetor é dado por log n e o número de operações em cada nível soma-se n, de modo que o mesmo processo ocorre para um vetor que não esteja ordenado, logo, a complexidade do mergeSort somada a cada nível é de O(nlogn).



QuickSort:
A complexidade do QuickSort do melhor caso é O(nlogn) que é quando a função que particiona o vetor coloca o pivô sempre no meio.

Ex.:

3	2	4

Nesse exemplo, se o pivô do vetor for o primeiro elemento(pivo = 3 na primeira recursão), podemos ver que após a aplicação da função que o particiona, o pivô passa a ocupar a posição do meio, onde a recursão em cada um dos níveis, consistirá de vetores, onde cada um terá tamanho n/2, tal que n é o números de elementos do vetor pai na árvore recursiva, localizada no nível anterior. De modo que a execução nesse caso comportar-se-á de modo assemelhará ao mergeSort.

Heapsort:

Precisa-se de O(n) para construir a *heap* e O(lg n) para manter a propriedade *heap* em cada passo de ordenação. O melhor caso é O(n lg n), bem como médio e pior caso.