Algoritmos de Ordenação e Complexidade computacional

Aluno: João Victor da Silva Prado

1- Insertion Sort: [-4, -84, 85, 88, 36]

I- [-4 | -84] 85 88 36

II- -84 [-4 | 85] 88 36

III- -84 -4 [85 | 88] 36

IV- -84 -4 85 [88 | 36]
-84 -4 [85] 88 [36]
-84 [-4] 85 88 [36]
-84 -4 36 85 88

I- Começamos com o elemento de indice 1, comparando-o com o elemento de sua esquerda. Como o "-84" é menor que "-4" trocamos eles de posição

II- Comparamos o próximo elemento com os da sua esquerda até que o mesmo esteja ordenado. Como ele já está ordenado não há troca.

III- O mesmo ocorre quando o elemento é o de índice 3.

IV- Chegando na ultima posição, comparamos o elemento com os da sua esquerda até que ele esteja ordenado (enquanto isso, remanejamos os elementos comparados para frente. Chegando no "-4" vemos que "36" é maior e o ordenamos.

2- O algoritmo em questão busca um elemento "x" no array. Ele usa uma variável auxiliar que representa o meio do array.

A primeira checagem é se o elemento que procuramos é o do meio, se for, teremos o melhor caso, em que a complexidade é constante. A segunda checagem, se verdadeira, mostra que o elemento não existe no array.

A ultima condição comparará o "x" com o meio e se x for maior faremos a recursão executando o método eliminando toda parte esquerda do array, tendo assim um novo "meio" na próxima execução. E, caso não entre nessa ultima faremos a recursão com a mesma lógica, mas eliminando a parte esquerda do array. Enquanto a 1ª ou 2ª condição não for verdadeira continuamos a recursão.

Analisando a lógica do algoritmo nota-se que o array é dividido pela metade até que "x" seja encontrado, tendo assim a complexidade O(log n)

3- Selection Sort: [-1, 101, 3, 80, 0, 5]

No Selection Sort guardaremos o valor do menor elemento numa variável auxiliar e compararemos o menor com os próximos para se encontrarmos um menor mudamos o valor da auxiliar para que só no fim remanejemos o menor para cima.

I- Na 1ª varredura o "-1" é o menor. Então não muda de lugar.

II- Na 2ª o menor é o "3" até que chegamos no "0" e ele passa a ser o menor e vai para cima (trocando de lugar com o "101").

III- Na 3ª O menor é o "3" e não precisa mudar de lugar

IV- Na 4ª o menor é o "80" até chegar no "5" e ele passa a ser o menor e vai para cima (trocando de lugar com o "80").

V- Por fim comparamos o "101" com o "80" e o segundo, por ser menor, vai para cima.

I	II	III	IV	V	VI
"-1"	-1	-1	-1	-1	-1
101	101	0	0	0	0
3	3	"3"	3	3	3
80	80	80	80	5	5
0	"0"	101	101	101	80
5	5	5	"5"	"80"	101

101 ↔ 0 80 ↔ 5 101 ↔ 80

4- Bubble sort:

-Melhor caso: Com o uso de uma flag para auxiliar o algoritmo é possível sair do loop. Assim, seria identificado que nenhuma troca ocorreu e faria um break, saindo do laço. Logo a complexidade será $O(n)$.

- Pior caso: $O(n^2)$. Pois se o array fosse varrido uma vez a complexidade seria $O(n)$, mas como no pior caso o array é varrido para cada elemento a complexidade será $O(n^2)$.

Selection sort: É mais economico do que o anterior se tratando de "swaps". Porém tem uma desvantagem sobre os demais: a sua lógica o obriga a varrer o vetor até o fim todas as vezes, independente se o vetor já estiver ordenado ou não. Logo a complexidade tanto no melhor quanto no pior caso será $O(n^2)$.

Insertion sort: Cada elemento ainda não ordenado é comparado com os elementos da sua esquerda. Então no pior caso (vetor em ordem inversa) o laço interno fará a quantidade máxima de iterações, tendo complexidade $O(n^2)$.

No melhor caso a condição do laço interno sempre será falsa, pois o vetor já estará ordenado. Logo teremos apenas as iterações do laço externo e a complexidade $O(n)$.

5- Temos 2 for's e dentro do interno há uma chamada à própria função. Como há um for dentro do outro, para um elemento no array ele percorrerá todos os n-1 que sobraram, mas ainda nessa lógica ele percorrerá todos os (n-2) na próxima iteração e assim por diante. O que resultaria em n(n-1(...(2(1)))) ou O(n!). Porém, à recursividade presente no for interno fará a chamada do metodo executando novamente não apenas o for interno como também o externo a cada iteração, resultando num algoritmo de complexidade O(n! * n!)