Trabalho Prático 1
Recursão e Complexidade de Algoritmos

BubbleSort

A função consiste em percorrer o vetor, verificando o elemento atual com o elemento anterior e caso o elemento atual seja menor que o elemento anterior, faz-se a troca dos elementos. A verificação se repete $\mathbf{n^2}$ vezes, independente de todos os elementos já estiverem sido ordenados.

A ordem de complexidade desse algoritmo é de $O(n^2)$, tanto para o pior caso, quanto para o caso médio e o melhor caso. Portanto, em ordenações de muitos elementos, esse algoritmo não é aconselhável.

Exemplo, em um vetor de 5 elementos.

4	6	1	0	3
4	6	1	0	3
4	1	6	0	3
4	1	0	6	3

4	1	0	3	6
1	4	0	3	6
1	0	4	3	6
1	0	3	4	6

1	0	3	4	6
0	1	3	4	6
0	1	3	4	6
0	1	3	4	6

0	1	3	4	6
0	1	3	4	6
0	1	3	4	6
0	1	3	4	6

0	1	3	4	6
0	1	3	4	6
0	1	3	4	6
0	1	3	4	6

SelectionSort

A função seleciona o menor elemento (ou o maior, caso seja decrescente) e coloca-o na primeira posição, em seguida, seleciona o segundo

menor, e assim sucessivamente, até que seja feita n-1 vezes, visto que o ultimo elemento estará automaticamente posicionado.

A ordem de complexidade desse algoritmo é de $O(n^2)$, tanto para o pior caso, quanto para o caso médio e o melhor caso. Portanto, em ordenações de muitos elementos, esse algoritmo não é aconselhável.

Exemplo, em um vetor de 5 elementos

4	6	1	0	3
0	6	1	4	3
0	6	1	4	3
0	1	6	4	3
0	1	6	4	3
0	1	3	4	6
0	1	3	4	6
0	1	3	4	6

InsertionSort

A função percorre um vetor de elementos da esquerda para a direita e à medida que avança deixa os elementos mais à esquerda ordenados.

No melhor caso, o algoritmo executa O(n) operações. No pior caso e no caso médio, são feitas $O(n^2)$ operações. A complexidade desse algoritmo é de Ordem de n, considerando que ele tem menor número de trocas e comparações entre os algoritmos de ordenação O(n) quando o vetor está ordenado. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

4	6	1	0	3
4	6	1	0	3
4	6	1	0	3
1	4	6	0	3
0	1	4	6	3
0	1	3	4	6

QuickSort

O QuickSort adota a estratégia de dividir para conquistar. A estratégia consiste em dividir o vetor em dois e rearranjar as chaves de modo que as elementos de menores valor que o pivô escolhido fiquem a sua esquerda e os maiores fiquem a direita.Para isso, a função utiliza-se da estratégia da recursividade. Podemos citar os passos como:

1. Escolher um elemento da lista, denominado pivô;

2. Rearranjar a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua posição final e haverá duas sublistas não ordenadas. Essa operação é denominada partição;
3. Recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores;

A base da recursão são as listas de tamanho zero ou um, que estão sempre ordenadas. O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte.

O **Quicksort** é um método de ordenação muito rápido e eficiente, tendo no pior caso, a complexidade de $O(n^2)$ e no melhor caso $O(n\log_2(n))$.

4	6	1	0	3
1	4	6	0	3
1	4	6	0	3
0	1	4	6	3
0	1	3	4	6

MergeSort

A função divide o vetor em diversos subvetores e começa organizando a partir desses subvetores, após organizar o subvetor de dois, junta-se com outro subvetor já ordenado e apenas faz uma comparação linear e ordena-os, assim, tendo um vetor de quatro elementos. Repete-se o processo até que todos os subvetores tenha sido reajuntados e consequentemente todo o vetor ordenado.

Os três passos úteis dos algoritmos dividir para conquistar, que se aplicam ao *mergesort* são:

1. Dividir: Dividir os dados em subsequências pequenas;
2. Conquistar: Classificar as duas metades recursivamente aplicando o *mergesort*;
3. Combinar: Juntar as duas metades em um único conjunto já classificado.

Sua complexidade é sempre $O(n\log_2(n))$. Tornando-se no melhor algoritmo de ordenação, porém sua complexidade de implementação não é simples, podendo ser substituído por outros métodos em pequenas operações de ordenação.

4	6	1	0	3
4	6	0	1	3
4	6	0	1	3
0	1	4	6	3
0	1	3	4	6

Novo Metodo (ShellSort)

É uma extensão do algoritmo de ordenação por inserção. Troca itens adjacentes para determinar o ponto de inserção.
São efetuadas (n – 1) comparações e movimentações quando o menor item está na posição mais à direita no vetor.
Para escolher o valor de h:

h(s) = 3h(s - 1) + 1, para s > 1
se s = 1
h(s) = 1

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
h = 4	*N*	*A*	*D*	*E*	*O*	*R*
h = 2	*D*	*A*	*N*	*E*	*O*	*R*
h = 1	*A*	*D*	*E*	*N*	*O*	*R*

A razão da eficiência do algoritmo ainda não é conhecida. Ninguém ainda foi capaz de analisar o algoritmo. A sua análise contém alguns problemas matemáticos muito difíceis. A começar pela própria seqüência de incrementos. O que se sabe é que cada incremento não deve ser múltiplo do anterior.

		Tempo(s) - Aleatorio					
		10^1	10^2	10^3	10^4	10^5	10^6
Método	Bubble Sort	0	0	0	1	82	N/A
	Selection Sort	0	0	0	0	25	N/A
	Insertion Sort	0	0	0	0	15	N/A
	Quick Sort	0	0	0	0	0	15
	Merge Sort	0	0	0	0	0	1
	Shell Sort	0	0	0	0	0	1

		Tempo(s) - Crescente					
		10^1	10^2	10^3	10^4	10^5	10^6
Método	Bubble Sort	0	0	0	0	49	N/A
	Selection Sort	0	0	0	0	25	N/A
	Insertion Sort	0	0	0	0	0	0
	Quick Sort	0	0	0	0	0	N/A
	Merge Sort	0	0	0	0	0	1

	Shell Sort	0	0	0	0	0	0

		Tempo(s) - Decrescente					
		10^1	10^2	10^3	10^4	10^5	10^6
Método	Bubble Sort	0	0	0	0	74	N/A
	Selection Sort	0	0	0	0	25	N/A
	Insertion Sort	0	0	0	0	30	N/A
	Quick Sort	0	0	0	0	N/A	N/A
	Merge Sort	0	0	0	0	0	N/A
	Shell Sort	0	0	0	0	0	N/A