Prática 07 – Estrutura de Dados

Questão 2:

BubbleSort: Faz ordenações "borbulhando" os maiores valores para o final do vetor. Seu número de comparações é C(n) = $O(n^2)$ e de movimentações é de 3 vezes o número de comparações M(n) = 3.C(n). Caso o vetor venha ordenado o número de comparações não se altera, mas reduz o número de trocas.

SelectionSort: Faz ordenações colocando o n-ésimo maior elemento na n-ésima posição, fazendo uma troca por vez. Seu número de comparações é C(n) = $O(n^2)$ e de movimentações é de 3 vezes o número de comparações M(n) = 3(n-1). É interessante para entradas grandes pois não altera o número de movimentos. O fato de o arquivo já estar ordenado não reduz o número de trocas.

InsertionSort: Faz ordenações semelhantes à ordenação de cartas de baralho na mão de um jogador, verificando se cada elemento deve ficar antes ou depois da posição que está. O melhor caso ocorre quando os elementos estão ordenados, sendo O(2n-2), e o pior caso ocorre quando os elementos estão inversamente ordenados, sendo O((n^2+3n-4) / 2). É mais eficiente quando se deseja adicionar uma pequena quantidade de elementos ao vetor.

MergeSort: Faz ordenações utilizando a estratégia de "dividir e conquistar", divide o problema em instâncias menores, os resolve e combina os resultados posteriormente. Sua complexidade de tempo é de O(n.$\log_n$), sendo bem eficiente, mas consome muita memória pois cria vários vetores auxiliares para poder dividir o vetor original em vetores menores e por ser implementado recursivamente, sendo O(n) para memória.

QuickSort: Faz ordenações escolhendo um elemento como pivô e dividindo o vetor original em vetores menores, de modo que os elementos menores que o pivô ficam à sua esquerda e os maiores à direita, repetindo o processo até atingir vetores trivialmente ordenados. Após essas divisões os resultados são combinados. O pior caso desse algoritmo ocorre quando as divisões dos vetores intermediários não são feitas de forma eficiente e equilibrada, isso ocorre quando o vetor já estiver ordenado e sua complexidade será $O(n^2)$. Já o melhor caso ocorre quando a divisão da árvore binária é equilibrada, com subvetores de tamanhos semelhantes, sendo sua complexidade O(n.$\log_n$)

ShellSort: Faz ordenações de forma semelhante ao InsertionSort, porém ordena elementos que tem uma distância *h* entre si, não elementos consecutivos, reduzindo os valores de *h* até atingir o valor um. Para *h* = 1 o algoritmo funciona de forma semelhante ao InsertionSort. Sua complexidade não é bem definida mas é eficiente para tamanhos de entrada medianos.

HeapSort: Faz ordenações utilizando o conceito de Heap. Esse conceito pode ser entendido como uma árvore binária, podendo ser máxima ou mínima. No heap máximo, os nós pais são maiores ou iguais aos os filhos e no heap mínimo ocorre o contrário. Desse modo o elemento de maior prioridade, que se encontra na raiz da árvore, será inserido na primeira posição, caso seja um heap mínimo, ou na útilma, caso seja um heap máximo. Após um elemento sair do heap ele é refeito até o último elemento. Sua complexidade é O(n.logn) independentemente do tamanho de sua entrada.