**Universidade Federal de Roraima**

**Departamento de Ciência da Computação**

**Estrutura de Dados I**

**DISCIPLINA: ESTRUTURA DE DADOS I**

**ALUNOS: Diego Santos Costa**

**Talles Bezerra de Assunção**

**Selection Sort:**

O selection sort trabalha procurando o menor valor de um vetor não ordenado e colocá-lo na frente desse vetor. Com a ideia de separar o vetor em duas partes, ordenado e não ordenado, ele seleciona o primeiro valor da parte não ordenada e compara com os outros procurando o menor entre eles, quando encontra o menor, adiciona na parte ordenada.

***Desempenho:***

Como ele sempre compara o elemento escolhido com todos os outros do vetor, o número de comparações sempre será o mesmo para todos os vetores de mesmo tamanho, desta forma não existe nem pior e nem melhor caso, sua complexidade sempre será O (n²).

***Vantagens:***

* Fácil de ser implementando;
* Sem necessidade de vetor auxiliar;
* Rápido com vetores pequenos.

***Desvantagens:***

* Não importa se o vetor está ordenado, sempre faz a mesma quantidade de comparações;
* Lento com vetores grandes.

**Insertion Sort:**

O insertion sort consiste em percorrer todo o vetor selecionando cada elemento e ir colocando na sua posição correta. Também trabalha com a ideia de separar o vetor em uma parte ordena e outra não ordenada, mas o primeiro elemento do vetor já é definido com ordenado, e percorrendo o resto do vetor comparando o elemento seguinte com a parte ordenada e colocando na sua posição correta.

***Desempenho:***

O insertion sort possui melhor, médio e pior caso, variando na forma como os elementos estão posicionados no vetor. O melhor caso é o vetor já está ordenado, tendo sua complexidade O (n). O médio caso se trata quando os elementos estão distribuídos de forma aleatória, tendo complexidade O (n²). Possui pior caso quando o vetor está com os elementos de forma invertida a qual deseja ordenar, pois o maior elemento, que está na primeira posição, irá percorrer todo o vetor até chegar na última posição, possuindo complexidade O (n²).

***Vantagens:***

* É o método a ser utilizado quando o arquivo está "quase" ordenado;
* É um bom método quando se desejar adicionar poucos elementos em um arquivo já ordenado, pois seu custo é linear;
* O algoritmo de ordenação por inserção é estável.

***Desvantagens:***

* Alto custo de movimentação de elementos no vetor.

**Bubble Sort:**

O bubble sort consiste em um processo de percorrer todo o vetor, comparando cada elemento com o seu próximo, trocando de posição se o primeiro for maior que segundo, terminando quando esse processo for executado n vezes.

***Desempenho:***

Possui melhor caso quando o vetor já está ordenado, sem a necessidade de fazer nenhuma troca, com complexidade O (n). O pior caso ocorre quando não está ordenado, com complexidade O (n²).

***Vantagens:***

* Entendimento e implementação bem simples.

***Desvantagens:***

* Não é recomendado para vetores muito grandes.

**Heap Sort:**

O heap sort trabalha de forma a rearranjar o vetor em uma heap, rearranjar a heap em ordem crescente, e para isso ele usa o conceito de árvore binária para trocar de lugar os elementos da heap, e ao final o vetor estará ordenado.

***Desempenho***:

***Número de comparações:***

2 n lg(m), o tempo do heap sort é proporcional ao número de comparações entre os elementos do vetor, e portanto proporcional a n log n no pior caso.

***Vantagens:***

* Comportamento O(n log n) no pior caso

***Desvantagens***:

* Não é estável.

**Merge sort:**

O merge sort parte do princípio de divisão e conquista, ele divide o vetor em subvetores e recursivamente vai dividindo até não for mais possível dividir, após isso esses vetores menores são ordenados e juntados novamente até formar o vetor inicial ordenado.

***Desempenho***:

O(n log n), o tempo é proporcional a quantidade de movimentações no vetor.

***Vantagens***:

* Tempos no pior caso O(n log n)

***Desvantagem***:

* Consome muito mais memória devido a utilização de vetores auxiliares.

**Quick Sort:**

Considerado um dos melhores algoritmos de ordenação, o quick sort trabalha com ideia de dividir e conquistar. A principal característica é a escolha de um elemento do vetor, o pivô, e a partir desse pivô organizar os elementos menores que o pivô a esquerda, e os maiores a direita, ordenando de maneira recursiva.

***Desempenho:***

O melhor e pior caso irá depender da escolha do pivô, se o pivô escolhido for o menor ou o maior elemento do vetor será o pior caso, com complexidade O (n²). O melhor caso ocorrerá quando as partições forem do mesmo tamanho, com complexidade O (n log n).

***Vantagens:***

* Melhor opção para ordenar vetores grandes;
* Muito rápido por que o laço interno é simples;
* Memória auxiliar para a pilha de recursão é pequena.

***Desvantagens:***

* Não é estável;
* Pior caso é quadrático.

**Radix Sort:**

O radix sort funciona com a ideia de ordenar de acordo com a base do número, fazendo ordenação através dos dígitos dos números, pegando do menos significativo para o mais significativo.

***Desempenho***:

***Número de comparações:***

Nenhuma comparação é realizada;

Inspeção de dígitos:

2\*n\*qtd\_digitor, se a quantidade de dígitos for a mesma em todos os números, o radix sort tem custo linear O(n).

***Número de trocas***:

N\*qtd\_digitos, o número de trocas também é linear O(n).

***Vantagens:***

* É estável;
* Não faz comparações.

***Desvantagens:***

* Depende do hardware, nem sempre é fácil otimizar a inspeção de dígitos.
* Serve apenas para números inteiros positivos e pequenos.

**Busca sequencial:**

Trabalha percorrendo todo o vetor comparando o valor passado com cada elemento do vetor.

***Desempenho***:

O desempenho depende a posição onde o valor procurado se encontra, tendo melhor caso quando o valor está na primeira posição, com 1 comparação, e pior caso quando o valor está na última, com n comparações.

***Vantagens:***

* Entendimento e implementação bem simples;
* O vetor não precisa estar ordenado.

***Desvantagens:***

* Lento com vetores muito grandes no pior caso.

**Busca binária:**

Consiste em fazer busca de um elemento em um vetor ordenado, começando a busca na metade do vetor e se não estiver no elemento central do vetor é então verificado se o elemento procurado está para a metade direita ou esquerda do vetor e então esse processo é repetido até encontrar o elemento desejado ou que a busca se esgote não encontrando o elemento.

***Desempenho***:

O desempenho é proporcional ao número de comparações que mesmo no pior dos casos é aproximadamente log n.

***Vantagens:***

* Mais rápido que a busca sequencial;
* Eficiente com vetores grandes.

***Desvantagens:***

* O vetor precisa estar ordenado.

**Referencias:**

http://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/radixsort.pdf

https://www.ime.usp.br/~pf/algoritmos/aulas/bubi.html

https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html

http://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/heapsort.pdf

http://www.cos.ufrj.br/~rfarias/cos121/aula\_07.html

https://www.ime.usp.br/~pf/algoritmos/aulas/mrgsrt.html

http://www.univasf.edu.br/~marcelo.linder/arquivos\_aed2/aulas/aula12.pdf

http://www.ft.unicamp.br/liag/siteEd/implementacao/selection-sort.php

http://www.ft.unicamp.br/liag/siteEd/implementacao/insertion-sort.php

http://www.ft.unicamp.br/liag/siteEd/definicao/bubble-sort.php

http://cafofodoprogramador.blogspot.com.br/2009/02/bubble-sort-em-linguagem-c.html

https://programacaodescomplicada.wordpress.com/2014/05/23/ed1-aula-52-ordenacao-quicksort/

http://wiki.icmc.usp.br/images/d/df/ICC2\_12.Busca.pdf