TABALHO PRATICO DE AEDS II/ GRUPO: Christoffer de Paula Oliveira. UFSJ

# Introdução

Um algoritmo que ordena um conjunto, geralmente representada num vetor, é chamado de algoritmo de ordenação. **Algoritmo de ordenação** em ciência da computação é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem—em outras palavras, efetua sua ordenação completa ou parcial. As ordens mais usadas são a numérica e a lexicográfica. Existem várias razões para se ordenar uma sequência. Uma delas é a possibilidade se acessar seus dados de modo mais eficiente.

Há uma grande variedade de algoritmos de ordenação, cada uma mais adequada que a outra para a massa de dados que será trabalhada, alguns algoritmos mais importantes, são *bublle sort* (ou ordenação por flutuação),*selection sort(*ou ordenação por seleção*),shell sort* , *heap sort* (ou ordenação por *heap*), *insertion sort* (ou ordenação por inserção), *merge sort*(ou ordenação por mistura) e o *quicksort*. No entanto nos concentraremos nos: selection sort, shell sort, heap sort, insertion sort, merge sort e quicksort.

## Objetivo do trabalho

O objetivo do trabalho é implementar um código em linguagem C, contendo os principais tipos de ordenação com diferentes tipos de dados, um array de inteiro ou um array de estrutura com vários outros arrays. Dessa forma deverão ser realizados experimentos que compare o desempenho dos seis algoritmos de ordenação para tamanhos de array diferentes sendo eles 20 elementos, 500 elementos, 5000 elementos, 50000 elementos, 200000 elementos, 1000000 elementos. Os testes serão realizados na sessão de analise de resultados do trabalho.

# Ambiente

A linguagem de programação C necessita em seu básico apenas um editor de textos para a digitação dos códigos e um compilador instalado na máquina para realizar a compilação.

Os testes foram realizados em um notebook com processador intel i3 de primeira geração, com 3 gb de memória RAM e sistema operacional ubuntu 15.10. Foi utilizado o editor de textos Gedit e o compilador geany para linux.

# Ordenação

Ordenação é o ato de se colocar os elementos de uma sequência de informações, ou dados, em uma ordem predefinida. O termo técnico em inglês para ordenação é sorting, cuja tradução literal é "classificação".

* 1. **Ordenação de vetores**

Entrada: vetor com os elementos a serem ordenados;   
Saída: mesmo vetor com elementos na ordem especificada;

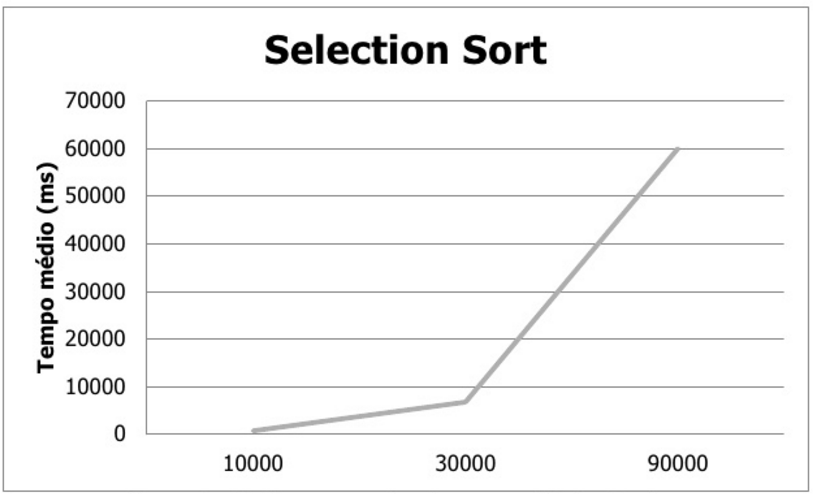
Ordenação: pode ser aplicada a qualquer dado com ordem bem definida;   
Vetores com dados complexos (structs): chave da ordenação escolhida entre os campos   
– elemento do vetor contém apenas um ponteiro para os dados   
– troca da ordem entre dois elementos = troca de ponteiro

* 1. **Ordenação *selection sort(*ou ordenação por seleção*)***

A cada passo, procura o menor valor do array e o coloca na primeira posição do array após isso descarta-se a primeira posição do array e repete-se o processo para a segunda posição e este processo e feito para todas as posições do array.

No seu melhor e pior caso tem-se : O(n^2).

Deixando-o ineficiente para grandes conjuntos de dados.



* 1. **Ordenação *Insertion Sort***

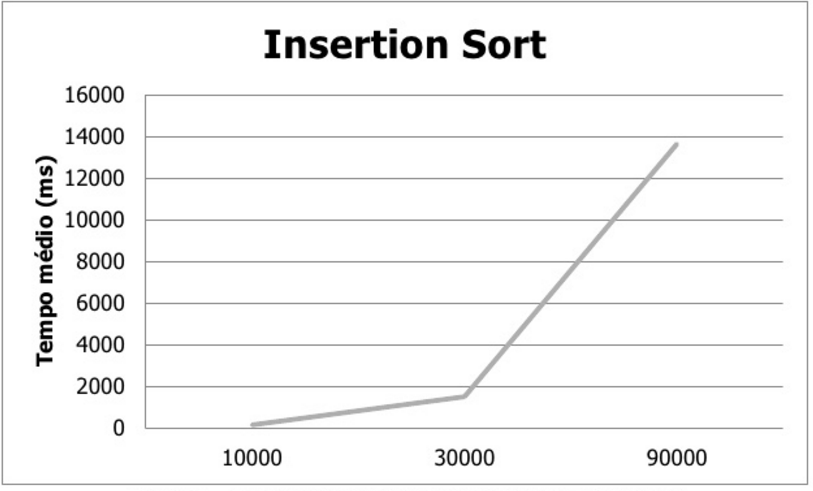
Similar a ordenação de cartas de baralho com as maõs.

Pega-se uma carta de cada ve e a coloca em seu devido lugar, sempre deixando as cartas da mão em ordem.

No seu melhor caso tem-se: O(n)

No seu melhor caso tem-se: O(n^2)

Eficiente para comjuntos pequenos de dados e estável, ou seja, não altera a ordem de dados iguais, pois ordena os dados a medida que os recebe(tempo real).

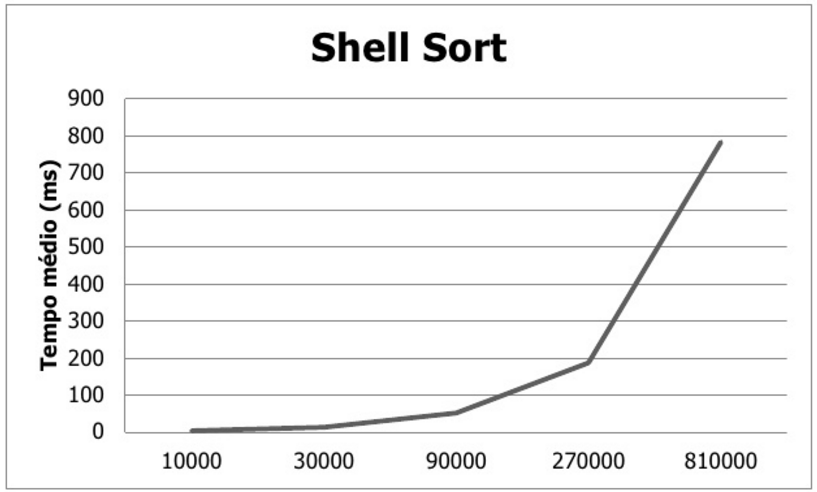


* 1. **Ordenação *Shell Sort***

O algoritmo shell sort é uma variação do algoritmo de inserção. O shell sort faz a inserção dos elementos na ordem correta, através de passos mais largos, utilizando um incremento para determinar qual item será ordenado na subsequência.

Não se sabe ainda a razão de eficiência do algoritmo

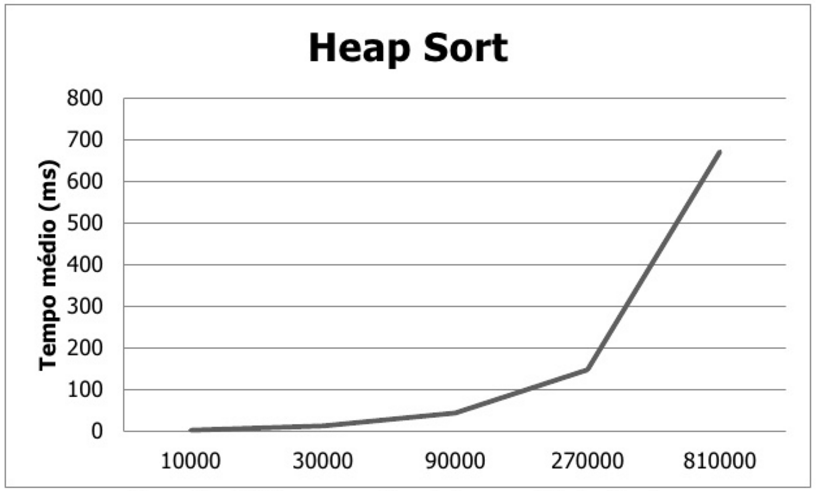
Execelente para arquivos de tamanho moderado



**3.5 Ordenação *Heap Sort(*ou ordenação por *heap)***

***Heap:*** vetor que simula uma árvore binária completa(exceção do ultimo nível). Todo elemento raiz do vetor tem dois elementos folhas.

No seu pior caso e melhor caso tem-se: O(n log n)

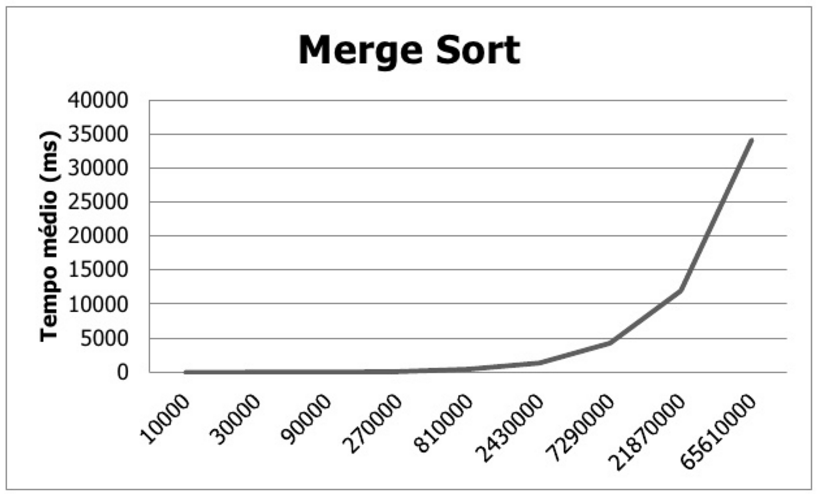


**3.6 Ordenação *Merge Sort(*ou ordenação por mistura*)***

Idéia básica : Dividir e Conquistar. Divide, recursivamente, o conjunto de dados até que cada subconjunto possua 1 elemento. Combina 2 subconjuntos de forma a dotar 1 conjunto maior e ordenado. Esse processo se repete até que exista apenas 1 conjunto

No melhor e pior caso tem-se: O(n log n)

Estável não altera a ordem de dados iguais. Desvantagem recursivo e usa um vetor auxiliar durante a ordenação.



**3.7 Ordenação por *Quick Sort***

Ideia básica : Dividir e Conquistar.

Um elemento é escolhido como pivô.

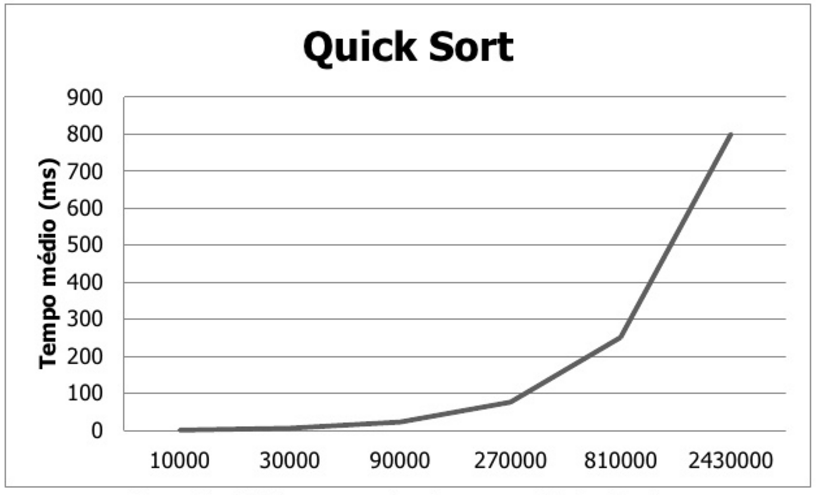
Particionar: Os dados são rerranjados(valores menores que o pivô são colocados antes dele e os maiores, depois).

Recursovamente: Ordena as duas partições.

No seu melhor caso tem-se : O(n log n)

No se pior caso tem-se: O(n^2).

Desvantegem seria de como escolher o pivô uma vez que esse interfere consideravelemente no rendimento do algoritmo.



# 3.8 Implementação

# A implementação funciona da seguinte forma, pelo menu é selecionado o tipo de ordenação, o tamanho dos dados e a forma com o que o vetor vai ser iniciado, como pode ser visto na implementação a várias funções para estes menus, no caso do tipo de ordenação todas as funções tem dois modos um para tamanho de dados pequenos e outro para tamanho de dados grandes(com struct ou sem), essas funções são duplicadas e divididas de acordo com o tamanho dos dados o restante das funções de ordenação já foram explicadas nos itens a cima. Na funções que definem como o vetor será iniciado há, uma função para gerar valores aleatórios(no caso de aleatório),uma crescente que pega de zero até o tamanho do vetor, decrescente que inicia do tamanho do vetor e vai até até zero, quase crescente e quase decrescente que são metade aleatório e a outra metade é feita com valores de 1 a 1 até o tamanho do vetor ou até zero no caso de quase decrescente.

# 

# 4 Análise de Resultados

# Foram realizado diversas baterias de testes, a fim de comparar os algoritmos implementados neste trabalho prático. Dessa forma foram comparados os algoritmos *inserction sort, seletion sort, shell sort, merge sort, heap sort* e *quick sort* com diferentes tamanhos de dados

# 

# A análise gráfica nos permite observar que a medida que a quantidade de elementos a serem ordenados cresce, o desempenho decai e o tempo aumenta. Isto significa que o desempenho dos algoritmos só é aproveitável se o conjunto possuir uma quantidade máxima de N elementos.

# Para ficar mais visível o rendimento dos algoritmos será dividido em duas partes a primeira será uma análise dos algoritmos ineficientes, ou seja, os *inserction sort* e *o seletion sort* junto com 1 eficiente o *shell sort.*

# Observe que o algoritmo insertion sort esteve abaixo dos demais ineficientes,

# 

# mostrando que no geral o inesection obteve melhor desempenho dentre os ineficientes, observe também que o shell sort teve um crescimento muito pequeno em relação aos demais deixando de fato os ineficientes com um desempenho inferior a ele. Logo os algoritmos ineficientes são inaptos para grandes quantidades de dados(N > 30 000), por crescerem muito em relação ao tempo.

# Além disso convém deixar claro que o insertion sort é o mais rápido para qualquer tamanho de dados já ordenado. Consequentemente é o mais lento para dados em ordem decrescente.

# Os algoritmos eficientes ao contrário dos ineficientes mostraram um rendimento linearmente baixo durante todo o estudo e testes.

# 

# Observando o gráfico podemos ver que o quick sort obteve um desempenho superior ao demais eficientes consequentemente maior também que o dos ineficientes.

# Apesar de o Heap sort apresentar um pior caso de O(n log n), não superou o rendimento do quick sort que tem um pior caso de O(n^2)

# Para arquivos pequenos o shell sort é mais rápido que o heap sort , porém quando o números de elementos cresce o heap sort obtém um melhor desempenho.

# O algoritmo merge sort superou ainda o algoritmo heap sort em todos os processos de testes, mais ainda sim não superou o desempenho do quick sort.

# O Shell sort obteve uma pequena diferença entre o quick sort cerca de duas vezes mais lento.

# 

# Este último gráfico mostra todos os algoritmos juntos observe como os eficientes se mantém com um tempo abaixo dos ineficientes e consequentemente com um desempenho maior.

# Conclusão

Neste trabalho foi apresentado uma análise dos principais algoritmos de ordenação . No quesito tempo de execução para qualquer tamanho com ordem aleatória o algoritmo quick sort demonstrou mais eficiência, já para qualquer tamanho com ordem crescente o algoritmo inserction sort se mostrou com maior desempenho. No quesito comparações para qualquer tamanho em qualquer ordem o algoritmo insertion sort se mostrou com maior desempenho. No quesito movimentação o selection sort se mostrou mais eficiente até mesmo que o quick sort para valores decrescentes e aleatórios de qualquer tamanho, porém em ordem decrescente de qualquer tamanho o quick sort obteve o melhor resultado.

# Referências

# “[http://pt.slideshare.net/OnOSJunior/anlise-emprica-de-algoritmos-de-ordenao” Acessado em 14/05/2016](http://pt.slideshare.net/OnOSJunior/anlise-emprica-de-algoritmos-de-ordenao).

<https://br.answers.yahoo.com/question/index?qid=20071103130826AAIA8F6>

[https://pt.wikibooks.org/wiki/Algoritmos\_e\_Estruturas\_de\_Dados/Algoritm os\_de\_Ordena%C3%A7%C3%A3o](https://pt.wikibooks.org/wiki/Algoritmos_e_Estruturas_de_Dados/Algoritm%09os_de_Ordena%C3%A7%C3%A3o)

https://programacaodescomplicada.wordpress.com/