**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA. PARAÍBA**

**Projeto de Análise de Algoritmos**

**Análise Experimental de Algoritmos**

**Aluno: Antônio de Farais Amorim**

**Matricula: 201815020016**

**Professor: José Gomes Lopes Filho**

**Curso: Análise e Desenvolvimento de Sistemas Período: 5º Turno: Noite**

Sumário

[**Introdução.** 3](#_Toc66717566)

[**Algoritmos.** 3](#_Toc66717567)

[**Bubble Sort** 3](#_Toc66717568)

[**Insertion Sort** 3](#_Toc66717569)

[**Selection Sort** 3](#_Toc66717570)

[**Quick Sort** 3](#_Toc66717571)

[**Merge Sort** 3](#_Toc66717572)

[**Implementação** 4](#_Toc66717573)

[**Bubble Sort** 4](#_Toc66717574)

[**Implementação em código.** 4](#_Toc66717575)

[**Ilustração do funcionamento.** 4](#_Toc66717576)

[**Insertion sort** 4](#_Toc66717577)

[**Implementação em código.** 5](#_Toc66717578)

[**Ilustração do funcionamento.** 5](#_Toc66717579)

[**Selection Sort** 5](#_Toc66717580)

[**Implementação em código.** 5](#_Toc66717581)

[**Ilustração do funcionamento.** 5](#_Toc66717582)

[**Quick Sort** 5](#_Toc66717583)

[**Implementação em código.** 6](#_Toc66717584)

[**Ilustração do funcionamento.** 6](#_Toc66717585)

[**Merge Sort** 6](#_Toc66717586)

[**Implementação em código.** 7](#_Toc66717587)

[**Ilustração do funcionamento.** 7](#_Toc66717588)

[**Analise gráfica.** 7](#_Toc66717589)

[**Bubble Sort** 7](#_Toc66717590)

[**Conclusão da análise.** 9](#_Toc66717591)

[**Insertion sort** 10](#_Toc66717592)

[**Conclusão da análise.** 11](#_Toc66717593)

[**Selection sort** 12](#_Toc66717594)

[**Conclusão da análise.** 13](#_Toc66717595)

[**Quick Sort** 14](#_Toc66717596)

[**Conclusão da análise.** 15](#_Toc66717597)

[**Merge Sort** 16](#_Toc66717598)

[**Conclusão da análise.** 20](#_Toc66717599)

[**Conclusão Geral** 20](#_Toc66717600)

[**Referencias** 20](#_Toc66717601)

# **Introdução.**

Este documento tem como objetivo avaliar algoritmos de ordenações (InsertionSort, SelectionSort, BubbleSort, MergeSort e QuickSort), através de analises assintóticas, número de comparações de chaves, número de movimentações de registros e tempo total gasto de execução. Para isso, serão colocados vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 elementos com suas ordens serão de arranjos ordenados, inversamente ordenados, quase ordenados e aleatórios. Assim possibilitando uma análise geral dos algoritmos.

# **Algoritmos.**

## **Bubble Sort**

É um algoritmo que percorre várias vezes um mesmo vetor com a intenção de ordena-lo, por esse movimento parecer o uma bolha por isso o nome. Sua complexidade é no pior e médio caso e no melhor caso.

## **Insertion Sort**

É um algoritmo que visa através de inserção por vez ordenar um vetor. Este algoritmo e bastante eficiente principalmente em casos pequenos, ele vai ordenando o vetor aos poucos desde do começo e ao ordenar aquela parte introduz mais um elemento do vetor para ordena-lo. Sua complexidade é no pior e médio caso e no melhor caso.

## **Selection Sort**

É um algoritmo baseado em percorrer o vetor a procura do menor elemento da vez e colocá-lo no começo do vetor, o que faz que ele seja um anime pouco eficiente. Sua complexidade é no pior, médio e no melhor caso.

## **Quick Sort**

É um algoritmo através da fragmentação do vetor e da escolha de um pivô como meta comparativa para fazer a troca de chaves e este pivô muda para ordenar aquela parte do vetor e isto vai sendo feito ate o vetor estar ordenado. Sua complexidade é no pior caso, nomédio e no melhor caso.

## **Merge Sort**

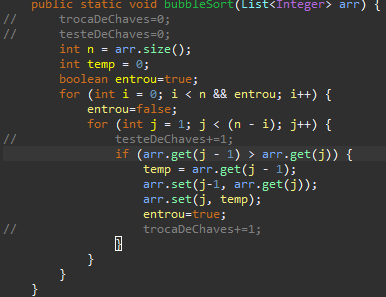
É um algoritmo que divide o vetor em várias partes até chegar em um valor mínimo e vai ordenando cada parte e assim juntar todas as partes que no final ficara todo ordenado, isso permite que este algoritmo seja bastante efetivo, mas tem um alto consumo de memória. Sua complexidade é sempre no melhor,médio e no pior caso.

# **Implementação**

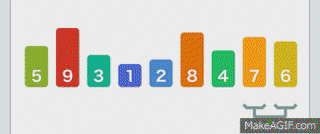
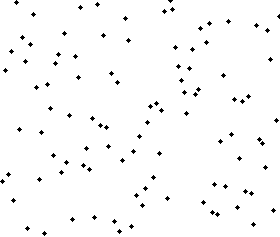
## **Bubble Sort**

Este algoritmo visa percorrer todo o vetor testando duas chaves se a segunda chave que estar a frente da primeira, e se caso a segunda chave for maior que a primeira ela é trocada isto e feito repetidas vezes sendo no caso o vetor só esteja ordenado no final da execução da função.

### **Implementação em código.**

****

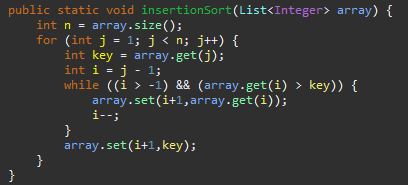
### **Ilustração do funcionamento.**



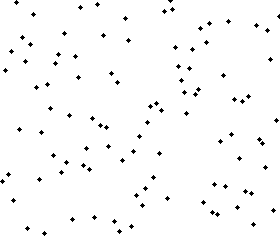
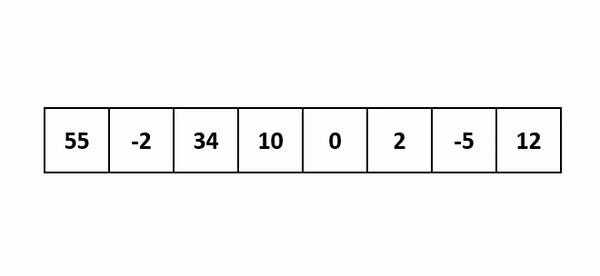
## **Insertion sort**

O Insertion Sort parte da premissa de sempre manter uma parte do vetor ordenada e através de uma inserção por vez, é introduzido um novo elemento e o ordena no vetor. Isto acontece repetidas vezes até que o vetor esteja todo ordenado.

### **Implementação em código.**



### **Ilustração do funcionamento.**

****

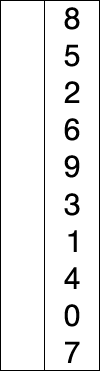
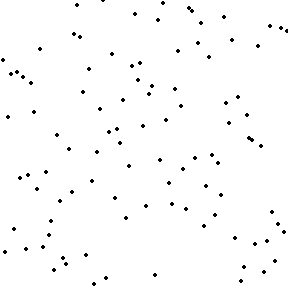
## **Selection Sort**

Na ordenação do Selection Sort o algoritmo percorre todo o vetor atras do menor valor para ordena-lo, ao terminar de percorrer o valor é colocado no começo do vetor assim, será feito no segundo slot sucessivamente até que esteja ordenado

### **Implementação em código.**

****

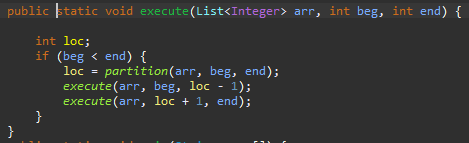
### **Ilustração do funcionamento.**

****

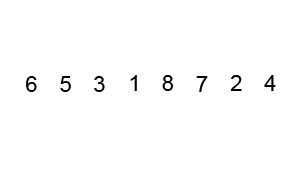
## **Quick Sort**

No Quick Sort, ao ser inserido o vetor é definido um pivô, este que é medida de comparação, os que forem menores que o pivô ficarão a esquerda, já os maiores a direito, vai-se dividindo o vetor e fazendo este mesmo processo ate que ele esteja todo ordenado.

### **Implementação em código.**



### **Ilustração do funcionamento.**

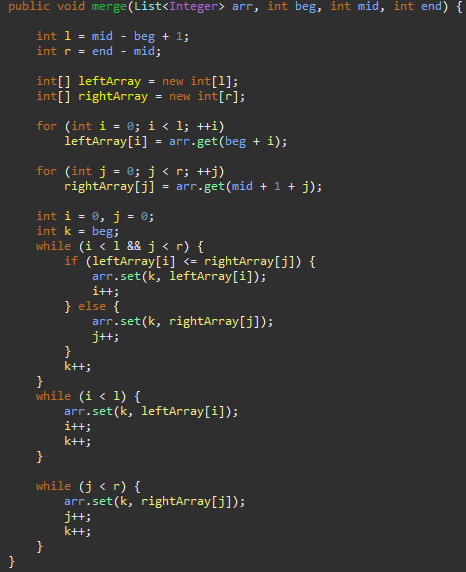
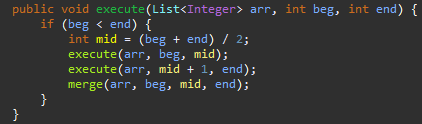
****

## **Merge Sort**

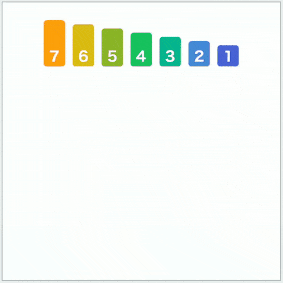
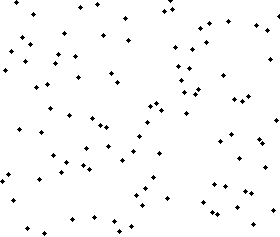
A ideia do Merge Sort é fragmentar o vetor ate que fique com o menor tamanho possível e assim ele junta todas as partes já ordenando-as. Isto garante que qualquer em qualquer cenário seja ele melhor ou pior, seu custo seja logaritmo. Este método é conhecido como “Dividir para conquistar”.

Sua ideia básica é muito fácil: criar uma sequência ordenada a partir de duas outras também ordenadas. Para isso, ele divide a sequência original em pares de dados, ordena-as; depois as agrupa em sequências de quatro elementos, e assim por diante, até ter toda a sequência dividida em apenas duas partes. (C, 2011)

### **Implementação em código.**

****

### **Ilustração do funcionamento.**

****

# **Analise gráfica.**

## **Bubble Sort**

### **Conclusão da análise.**

Assim pode-se concluir que apenas no caso de um vetor ordenado o bubble sort se sai bem, vale ressaltar que a versão usada é modificada para se caso percorrer o vetor todo sem fazer nenhuma alteração ele para, pois, já estar ordenada.

## **Insertion sort**

### **Conclusão da análise.**

Com isso podemos concluir que principalmente nos casos de vetor quase ordenado e aleatório o Insertion Sort se sobre sai em relação ao Bubble Sort, porem na ordem inversa seu desempenho e quase o mesmo, por ser o pior caso.

## **Selection sort**

### **Conclusão da análise.**

O Selection Sort mostra-se constante em todos os tipos de ordenação, porém, não é eficiente pois em qualquer caso, pois seu custo é quadrático, assim mostrando um algoritmo pouco eficiente até em casos de vetor pequeno, além de que não identifica vetores já ordenados.

## **Quick Sort**

### **Conclusão da análise.**

O Quick Sort se mostra ser muito efetivo em todos os casos, porém este algoritmo não consegue identificar quando o vetor já estar ordenado, além disso dependendo do tamanho do vetor ele pode apresentar erro de estouro na memória. No entanto o Quick Sort é muito efetivo.

## **Merge Sort**

### **Conclusão da análise.**

Dado a análise do Merge Sort, ele se mostra bem eficiente, porém uma das desvantagens deste algoritmo é o grande uso de memória, por conta de fragmentar sempre o vetor em dois ate que fique em um tamanho mínimo possível este método só pode ser feito com a criação de vários vetores menores para guardar estes dados e assim o uso de memoria é muito alto.

# **Conclusão Geral**

Assim conclui-se que tanto o Merge Sort e Quick Sort são ótimos algoritmos, mas em contra ponto o Bubble Sort, Insertion Sort e Selection sort, não são bons algoritmos, em nenhum dos casos, pelo contrário até mesmo levando até horas para ordenar certos vetores.

Percebe-se que quando chegamos em vetores a partir de 100.000(cem mil) elementos podemos perceber que o tempo de execução começa a aumentar, ate mesmo nos melhores algoritmos como o Quick Sort e Merge Sort que antes vinham mostrado desempenho muito baixo, em algumas vezes com milissegundos quase nulos, agora mostram resultados diferente, apesar de em tempo de execução não ser uma diferença tão significante ou até mesmo perceptível, mas com isso pode-se ver que quanto maior o vetor seu tempo será bem maior em comparação anterior. Isso também se reflete nos piores algoritmos, pois, apesar de seus tempos de execução serem maiores do que os melhores algoritmos nos casos de menores vetores sua a diferença é quase imperceptível, só se tornando possível comparar quando se mostra o resultado e milissegundos, porem nos casos de vetores maiores sua diferença é muito perceptível, pois quando colocado em execução os piores algoritmos podem demorar horas, enquanto os melhores demoram questão de segundos.

Nos casos de vetores quase ordenados e aleatórios os algoritmos tem quase sempre o mesmo desempenho, no caso de vetores ordenados todos os algoritmos se saem bem, sendo que o Insertion Sort como o Selection Sort seu tempo de execução em milissegundos ser bem maior que os outros, o Bubble sort em caso de vetor ordenado se sai ate melhor que o Merge Sort, isso se deve pelo fato que esta implementação do Bubble Sort é feita para entender quando o vetor já estar ordenado , porem tanto o Merge Sort quando o Quick Sort compensam este problema com sua velocidade mesmo não identificando quando o vetor já estar ordenado. No entanto o pior caso de todos é o inversamente ordenado, pois nos piores algoritmos chega a demorar hora já nos melhores demorando bem mais que nos outros tipos de vetores.

Por tanto pode-se concluir que os piores algoritmos são o Bubble Sort e o Selection Sort, e os melhores são o Merge Sort e Quick Sort, porem dado a analise o Quick se sobre sai na questão de tempo de execução, essa diferença só pode ser notada ao visualizar os dados gerados neste artigo, pois é imperceptível. Apesar de uma das desvantagens do Merge Sort ser o grande uso de memória quem mostrou problemas de estouro de memória foi o Quick Sort tendo que fazer modificações para resolver o este problema, o que não aconteceu nos outros algoritmos. Por fim, o Quick Sort mostra-se ser bem melhor que todos os outros algoritmos e por isso é o mais adequado em todas as ocasiões.

# **Referencias**

* C, P. d. (6 de 2011). *Ordenar C Blogspot*. Fonte: Ordenar C Blogspot: http://ordenarc.blogspot.com
* https://pt.wikipedia.org/wiki/Bubble\_sort#:~:text=O%20bubble%20sort%2C%20ou%20ordenação,de%20ordenação%20dos%20mais%20simples
* https://gifimage.net/bubble-sort-gif-7/
* https://pt.wikipedia.org/wiki/Insertion\_sort
* https://web.whatsapp.com/0b9ab6f1-0a10-4e09-9885-ae72e804b5cc
* https://pt.wikipedia.org/wiki/Selection\_sort
* https://pt.wikipedia.org/wiki/Quicksort
* https://www.google.com/url?sa=i&url=https%3A%2F%2Fcommons.wikimedia.org%2Fwiki%2FFile%3AQuicksort-example.gif&psig=AOvVaw3TMR2XWuqt80vJTnMd2iGx&ust=1615831043277000&source=images&cd=vfe&ved=0CAcQjhxqFwoTCLDcy\_CtsO8CFQAAAAAdAAAAABAn
* https://pt.wikipedia.org/wiki/Merge\_sort#:~:text=O%20merge%20sort%2C%20ou%20ordenação,tipo%20dividir-para-conquistar.&text=Como%20o%20algoritmo%20Merge%20Sort,muito%20eficiente%20em%20alguns%20problemas
* https://www.javatpoint.com/bubble-sort-in-java
* https://www.javatpoint.com/insertion-sort-in-java
* https://www.javatpoint.com/merge-sort
* https://www.javatpoint.com/selection-sort-in-java
* https://www.javatpoint.com/quick-sort