Algoritmos de Ordenação

Bubble, Selection, Insertion, Merge e Quick

Aluno: Gabryel Alexandre Campos da Silva

Instituto Federal da Paraíba (IFPB)

Análise e Desenvolvimento de Sistemas

Sumário

[1.Introdução 2](#_Toc69704096)

[2.Apresentação aos algoritmos 2](#_Toc69704097)

[3. Funcionamento dos Algoritmos 4](#_Toc69704098)

[4. Análise Assintótica dos Algoritmos 16](#_Toc69704099)

[5. Tempo dos Testes 19](#_Toc69704100)

[6. Quantidade de Testes de Chaves e Movimentação de Registros 23](#_Toc69704101)

[7.Conclusão e Melhor Algoritmo 36](#_Toc69704102)

[8. Referências 37](#_Toc69704103)

# 1.Introdução

Esse Projeto tem como base estudar os vários algoritmos de ordenação presentes no cenário da computação dentre eles estão o Bubble Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort. Iremos testar seus tempos de execução levando em conta quantas testes ocorreram e quantas mudanças aconteceram para que o vetor fosse ordenado. Iremos usar vetores em quatro formas diferentes I Ordenados, II Quase Ordenado, III Aleatório e IV Inversamente Ordenado para vetores de tamanho 10, 100,1000,10.000,100.000 e 1.000.000 de números.

# 2.Apresentação aos algoritmos

2.1 Bubble Sort

A bolha como é mais conhecido, é um algoritmo de ordenação mais básico que pode ser implementado, ele funciona por meio de percorrer uma lista várias vezes até que todo o vetor esteja ordenado, comparando dois itens I e I+1, e trocando caso estejam em ordens erradas, Ele para de percorrer a lista caso não ocorra nenhuma mudança na mesma.

2.2 Selection Sort

O Algoritmo Selection sort é a evolução do bolha, a cada passagem no vetor ele realiza apenas uma única troca, para que isso ocorra a cada passagem é escolhido o menor valor da mesma e ele é colocado na sua posição correta ao fim da passagem, por exemplo 0 é o menor valor logo ele é o primeiro a ser ordenado o segundo a ir para a ordem é o número 1 e assim sucessivamente.

2.3 Insertion Sort

A classificação por inserção é um dos algoritmos de classificação mais simples, pois classifica um único elemento em uma instância específica. Não é o melhor algoritmo de classificação em termos de desempenho, mas é ligeiramente mais eficiente do que a classificação por seleção e classificação por bolha em cenários práticos. Vamos considerar o exemplo de cartões para entender melhor a lógica por trás da classificação por inserção. Suponha que temos um conjunto de cartas em nossa mão, de modo que queremos organizá-las em ordem crescente. Para classificar esses cartões, temos várias maneiras intuitivas. Uma coisa que podemos fazer é, inicialmente, segurar todas as cartas em nossa mão esquerda, e podemos começar a pegar cartas uma após a outra na mão esquerda, seguido pela construção de um arranjo ordenado na mão direita. Supondo que a primeira carta já esteja classificada, selecionaremos a próxima carta não classificada. Se o cartão não classificado for maior do que o cartão selecionado, vamos simplesmente colocá-lo no lado direito, caso contrário, no lado esquerdo. Em qualquer estágio durante todo o processo, a mão esquerda não será classificada e a direita será classificada. Da mesma forma, classificaremos o restante das cartas não classificadas, colocando-as na posição correta. A cada iteração, o algoritmo de inserção coloca um elemento não classificado em seu lugar certo.

2.4 Merge Sort

O algoritmo do merge sort funciona com a premissa de dividir para conquistar e junto com o Quick sort é um dos algoritmos que possuem o melhor tempo de execução, dessa forma o vetor é dividido em partes menores (na metade) até que não possa mais ser divido (isso ocorre quando há apenas um número no vetor) após isso as partes menores vão sendo ordenadas da maneira correta até que todo o vetor esteja ordenado a melhor vantagem desse algoritmo está na sua estabilidade.

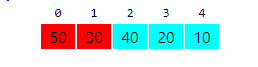
2.5 Quick Sort

O algoritmo Quick Sort assim como o Merge também funciona com a mentalidade do dividir para conquistar, só que diferentemente do merge a divisão é onde ocorre todo o trabalho de ordenação.

# 3. Funcionamento dos Algoritmos

3.1 Bubble Sort

Algoritmo: Percorre várias vezes o vetor de maneira sequencial (passos). Em cada passo, compara cada elemento no vetor com o seu sucessor (p[i] com p[i+1]) e troca o conteúdo das posições em análise, caso não estejam na ordem desejada.  
Observe a figura:  
O elemento da posição 0 (valor 50) é comparado com o elemento da posição 1 (valor 30).



Como o objetivo é ordenar crescentemente, os conteúdos dos elementos das posições 0 e 1 devem ser trocados.



Em seguida serão comparados os conteúdos dos elementos das posições 1 e 2:



Troca:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 40 | 50 | 20 | 10 |

Elementos das posições 2 e 3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 40 | 50 | 20 | 10 |

Troca:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 40 | 20 | 50 | 10 |

Elementos das posições 3 e 4:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 40 | 20 | 50 | 10 |

Troca:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 40 | 20 | 10 | 50 |

Apesar do vetor não estar ordenado ainda, observe que o maior elemento ficou na última posição:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 40 | 20 | 10 | 50 |

O processo recomeça, porém, ocorrerá entre as posições 0 e 3 (o elemento da posição 4 já está ordenado).

Elementos das posições 0 e 1.:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 40 | 20 | 10 | 50 |

Não troca:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 40 | 20 | 10 | 50 |

Elementos das posições 1 e 2.:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 40 | 20 | 10 | 50 |

Troca:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 20 | 40 | 10 | 50 |

Elementos das posições 2 e 3.:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 20 | 40 | 10 | 50 |

Troca:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 20 | 10 | 40 | 50 |

E o processo recomeça:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 30 | 20 | 10 | 40 | 50 |

Troca:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 20 | 30 | 10 | 40 | 50 |

Elementos das posições 1 e 2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 20 | 30 | 10 | 40 | 50 |

Troca:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 20 | 10 | 30 | 40 | 50 |

Recomeça... Elementos das posições 0 e 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 20 | 10 | 30 | 40 | 50 |

Troca:

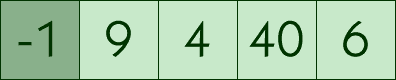
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 10 | 20 | 30 | 40 | 50 |

Vetor Ordenado!

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 10 | 20 | 30 | 40 | 50 |

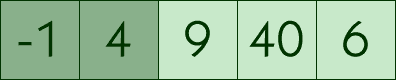
3.2 Selection Sort

Como exemplo, ordenaremos o vetor [40, 9, 4, -1, 6]. Vamos considerar o 0 (zero) como o primeiro índice do vetor.

Na primeira iteração, procuramos o menor valor entre os índices 0 e 4 (índice da última posição). O menor valor é o -1, que está no índice 3. Então, trocamos os valores dos índices 0 e 3:

Como o valor do índice 0 já está na posição correta, então não precisamos fazer mais nada nesse índice. Ou seja, nos resta ordenar o sub-vetor que vai do índice 1 até o índice 4.

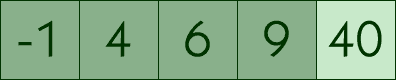
Na segunda iteração, procuramos o menor valor entre os índices 1 e 4. O menor valor é o 4, que está no índice 2. Então, trocamos os valores dos índices 1 e 2:



Na terceira iteração, procuramos o menor valor entre os índices 2 e 4. O menor valor é o 6, que está no índice 4. Então, trocamos os valores dos índices 2 e 4:



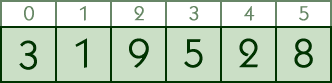
Na quarta iteração, procuramos o menor valor entre os índices 3 e 4. O menor valor é o 9, que está no índice 4. Então, trocamos os valores dos índices 3 e 4:



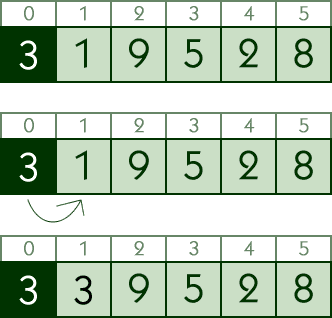
A cada iteração, um elemento é colocado na posição correta. Você pode, alternativamente, fazer a ordenação começando pela última posição do vetor, ou seja, começamos colocando o maior elemento na última posição, depois o segundo maior na penúltima posição e continuamos até chegarmos no início do vetor.

3.3 Insertion Sort

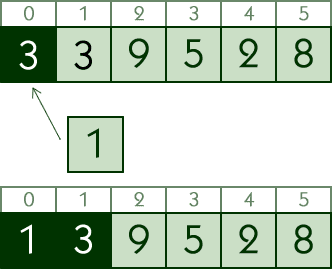
Para exemplificar o uso do algoritmo, ordenaremos o seguinte vetor



Inicialmente, temos i = 1, j = 0 e elemento = 1. Queremos inserir elemento na parte ordenado à esquerda. Como 3 > 1, então deslocamos o 3 para direita e decrementamos j: j = 0 - 1 = -1

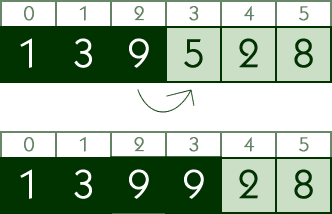


Como atingimos o início do vetor, então inserimos 1 em j + 1 = 0

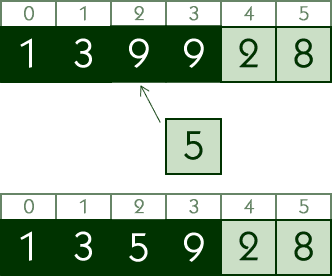


Agora, i = 2, j = 1 e elemento = 9. Como 3 < 9, então o elemento já está na posição correta (na verdade, o algoritmo insere o elemento na própria posição onde ele está).

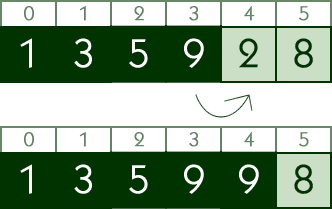
Na terceira iteração, i = 3, j = 2 e elemento = 5. Como 9 > 5, então deslocamos o 9 para direita e decrementamos j: j = 2 - 1 = 1



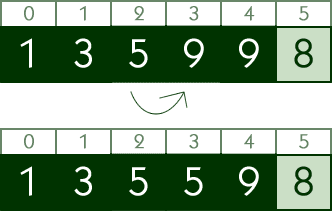
Como 3 < 5, então inserimos 5 em j + 1 = 2



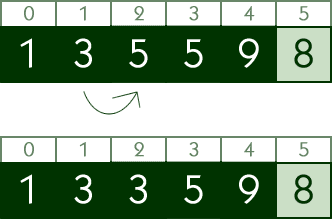
Na quarta iteração, i = 4, j = 3 e elemento = 2. Como 9 > 2, então deslocamos o 9 para direita e decrementamos j: j = 3 - 1 = 2



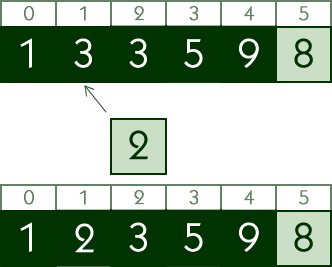
Como 5 > 2, então deslocamos o 5 para direita e decrementamos j: j = 2 - 1 = 1



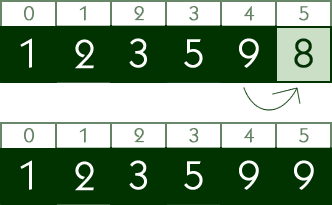
Como 3 > 2, então deslocamos o 3 para direita e decrementamos j: j = 1 - 1 = 0



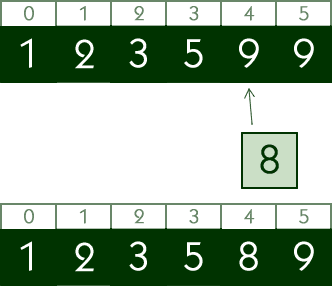
Como 1 < 2, então inserimos 2 em j + 1 = 1



Na quinta e última iteração, i = 5, j = 4 e elemento = 8. Como 9 > 8, então deslocamos o 9 para direita e decrementamos j: j = 4 - 1 = 3



Como 5 < 8, então inserimos 8 em j + 1 = 4



Com isso, finalizamos a ordenação.

3.4 Merge Sort

Como exemplo, ordenaremos o vetor [5, 2, 7, 6, 2, 1, 0, 3, 9, 4]. Inicialmente, dividimos o vetor em dois subvetores, cada um com metade dos elementos do vetor original.



Reaplicamos o método aos dois subvetores



De novo,



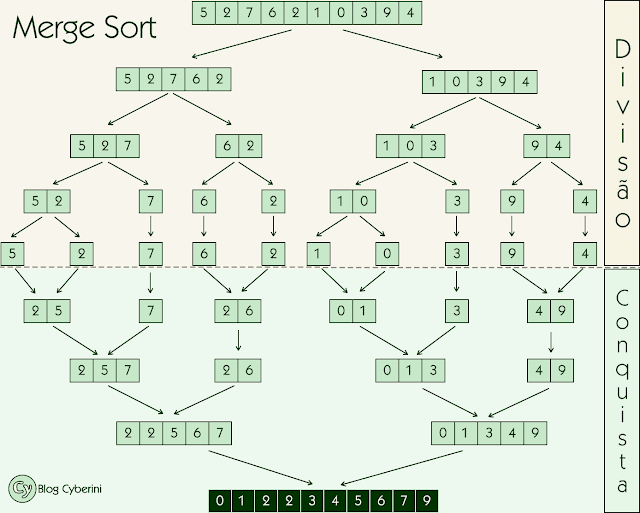
Mais uma vez, pois ainda não alcançamos o caso base em alguns subvetores



Finalmente, fazemos a fusão dos subvetores

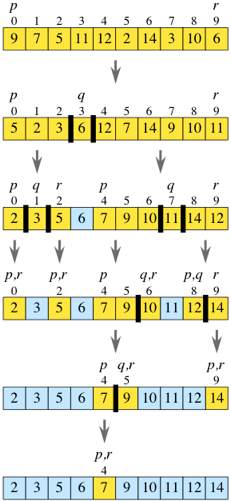


O diagrama a seguir ilustra todas as etapas



No exemplo, várias etapas foram realizadas simultaneamente para não estender o trabalho. Entretanto, na realidade, o Merge Sort ordena os subvetores um por vez.

3.5 Quick Sort



Outro Exemplo:

Para o array values= [3,8,7,10,0,23,2,1,77,7], temos que pivot=3. Vamos iterar no array identificando os elementos menores ou iguais a ele. O primeiro identificado é o valor 0.

values = [3, 8, 7, 10, 0, 23, 2, 1, 77, 7]

Nosso trabalho agora é colocar o valor 0 à frente do pivot. Então, trocamos esse valor com o valor 8 (imediatamente à frente de 3). Note, no estado parcial, que 0 ficou à frente de 3 e 8 assumiu o índice de 0.

values = [3, 0, 7, 10, 8, 23, 2, 1, 77, 7]

Acabou? Não. O próximo elemento menor ou igual ao pivot (3) é 2.

values = [3, 0, 7, 10, 8, 23, 2, 1, 77, 7]

Temos que trazer 2 para a frente de 3. Vamos fazer isso trocando este valor com o valor 7. Veja, no estado parcial, que agora os valores 0 e 2 estão à frente de 3.

values = [3, 0, 2, 10, 8, 23, 7, 1, 77, 7]

Acabou? Não. O próximo elemento menor ou igual ao pivot (3) é 1.

values = [3, 0, 2, 10, 8, 23, 7, 1, 77, 7]

Temos que trazer 1 para a frente de 3. Vamos fazer isso trocando este valor com o valor 10. Veja, no estado parcial, que agora os valores 0, 2 e 1 estão à frente de 3.

values = [3, 0, 2, 1, 8, 23, 7, 10, 77, 7]

E agora? Agora não há mais elementos menores ou iguais ao pivot para serem identificados. Todos os elementos menores ou iguais (0, 2 e 1) estão imediatamente à frente dele. Então, basta trocarmos o pivot (3) com o último deles (1).

values = [1, 0, 2, 3, 8, 23, 7, 10, 77, 7]

Feito. Agora 3 está em seu lugar, com todos os elementos menores ou iguais à sua esquerda e os elementos maiores à direita.

# 4. Análise Assintótica dos Algoritmos

4.1 Bubble Sort

No algoritmo em que utilizamos para os testes, temos o custo em (N+2)2 ou seja ele é O(n2)agora vamos analisar os casos.

Melhor caso:

Esse algoritmo no melhor caso tem a complexidade em O(n) isto ocorre quando o vetor já está completamente ordenado e só passará uma única vez por ele.

Caso Médio:

Neste caso o Algoritmo será O(n2) pois ele poderá no aleatório ter que passar várias vezes para que o vetor seja ordenado.

Pior Caso:

O pior caso desse algoritmo será O(n2) isto ocorre quando temos um vetor inversamente ordenado, então teremos que fazer várias modificações para que o vetor seja ordenado.

4.2 Selection Sort

Neste Algoritmo em que utilizamos para os testes, o custo é de ou seja ele é O(n2).

Melhor caso:

Infelizmente como esse algoritmo tem sempre que passar n2 vezes mesmo no melhor caso onde ele não precisa realizar nenhuma troca de chaves ele ainda será O(n2)

Caso Médio:

Como foi dito antes mesmo que a lista esteja aleatória ele ainda irá iterar n2 vezes no vetor fazendo assim que seja O(n2)

Pior Caso:

Mesmo com o vetor completamente ao contrário seu custo se manterá em O(n2)

4.3 Insertion Sort

O Insertion sort é um caso à parte se a complexidade dele no melhor caso é apenas n-1, porém no seu pior caso o custo se torna ou seja ele se torna O(n2).

Melhor caso:

No melhor caso é quando a lista está completamente ordenada, ou seja, ele não precisa fazer nenhuma troca ele é então O(n).

Caso Médio:

No seu caso médio ele é O(n2) pois podem aparecer números em maneiras confusas.

Pior Caso:

O seu Pior caso vem quando uma lista está completamente ordenada, porém do lado oposto ele então se torna O(n2).

4.4 Merge Sort

No merge como se trata de um método recursivo temos o custo dele em

2 T+O(n) (esse O(n) vem do método intercala). Utilizando o Método mestre temos que o Merge tem complexidade O(N\*logN).

Melhor caso:

O merge é um dos melhores algoritmos pois seu funcionamento é rápido e consistente no seu melhor caso o custo será O (N\* LogN).

Caso Médio:

Como ele funciona da mesma forma seu custo também se mantém estável ele também será O (N\* LogN).

Pior Caso:

O Mesmo no vetor decrescente o custo se matem o mesmo O (N\* LogN).

4.5 Quick Sort

O Quick mesmo sendo recursivo possui uma grande peculiaridade pois quem irá determinar se ele estará ou não no seu pior caso é o pivô que é utilizado se o pivô for capaz de dividir a lista na metade seu custo é utilizando o método mestre isso se torna O(N \* Log N ) , porém se pivô não dividir de forma eficiente seu vetor o custo se torna T(n) = T(n-1) + O(n) utilizando o método da arvore de recursão isso se torna O(n2)

Melhor caso:

Como foi dito o melhor caso é quando o pivô consegue dividir de forma eficiente a lista isto faz com que o custo seja O (N \* Log N).

Caso Médio:

Mesmo que o Pivô não consiga dividir muito bem o vetor se custo ainda se matem em O (N \* Log N).

Pior Caso:

No seu Pior caso é quando o vetor não divide o vetor de uma boa forma e isso faz com que seu funcionamento fique muito ruim em comparação com o melhor caso e caso médio, seu custo vai para O(n2).

# 5. Tempo dos Testes

5.1 Vetor Ordenado

Tempo do Vetor Ordenado em Milissegundos

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Vetor 10 | Vetor 100 | Vetor 1000 | Vetor 10.000 | Vetor 100.000 | Vetor 1.000.000 |
| Insertion Sort | 1ms | 1ms | 1ms | 1ms | 4ms | 10ms |
| Selection Sort | 1ms | 1ms | 6ms | 18ms | 1045ms | 211.841ms |
| BubbleSort | 1ms | 1ms | 1ms | 1ms | 3ms | 7ms |
| MergeSort | 1ms | 1ms | 1ms | 2ms | 14ms | 81ms |
| Quick Sort | 1ms | 1ms | 1ms | 1ms | 10ms | 49ms |

5.2 Vetor Inversamente Ordenado

Tempo do Vetor Inverso em Milissegundos

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Vetor 10 | Vetor 100 | Vetor 1000 | Vetor 10.000 | Vetor 100.000 | Vetor 1.000.000 |
| Insertion Sort | 1ms | 1ms | 11ms | 117ms | 12.072ms | 1.122.881ms |
| Selection Sort | 1ms | 1ms | 13ms | 89ms | 7176ms | 841.121ms |
| BubbleSort | 1ms | 1ms | 15ms | 160ms | 16325ms | 1.509.721ms |
| MergeSort | 1ms | 1ms | 1ms | 3ms | 19ms | 85ms |
| Quick Sort | 1ms | 1ms | 1ms | 2ms | 24ms | 95ms |

5.3 Vetor Aleatório

Tempo do Vetor Aleatório em Milissegundos

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Vetor 10 | Vetor 100 | Vetor 1000 | Vetor 10.000 | Vetor 100.000 | Vetor 1.000.000 |
| Insertion Sort | 1ms | 1ms | 9ms | 71ms | 6.325ms | 546.206ms |
| Selection Sort | 1ms | 1ms | 9ms | 51ms | 3816ms | 397.523ms |
| BubbleSort | 1ms | 2ms | 13ms | 177ms | 19.628ms | 2.182.283ms |
| MergeSort | 1ms | 1ms | 1ms | 3ms | 26ms | 223ms |
| Quick Sort | 1ms | 1ms | 1ms | 5Ms | 27ms | 182ms |

5.4 Vetor Quase Ordenado

Tempo do Vetor Quase Ordenado em Milissegundos

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Vetor 10 | Vetor 100 | Vetor 1000 | Vetor 10.000 | Vetor 100.000 | Vetor 1.000.000 |
| Insertion Sort | 1ms | 1ms | 9ms | 68ms | 5.626ms | 624.364ms |
| Selection Sort | 1ms | 1ms | 9ms | 49ms | 3.499ms | 290.839ms |
| BubbleSort | 1ms | 1ms | 12ms | 88ms | 6.153ms | 620.839ms |
| MergeSort | 1ms | 1ms | 1ms | 2ms | 23ms | 122ms |
| Quick Sort | 1ms | 1ms | 1ms | 3ms | 14ms | 79ms |

# 6. Quantidade de Testes de Chaves e Movimentação de Registros

6.1 Vetor de 10 Elementos

6.2 Vetor de 100 Elementos

6.3 Vetor de 1.000 Elementos

6.4 Vetor de 10.000 Elementos

6.5 Vetor de 100.000 Elementos

6.6 Vetor de 1.000.000 de Elementos

# 7.Conclusão e Melhor Algoritmo

A elaboração desse trabalho me deu novas perspectivas no estudo dos algoritmos, me ajudando a compreender que diferentes algoritmos podem ser benéficos ou maléficos em diferentes cenários. O maior desafio na elaboração desse projeto com certeza estaria na hora de analisar os diferentes tipos de resultados para cada tipo de vetor ao todo foram mais de 120 testes (tendo em vista que testes aleatórios foram refeitos para a melhor precisão do projeto).

O algoritmo que apresentou o melhor desempenho foi o Quick Sort, mas vale a pena salientar que o algoritmo Quick depende muito do seu pivô para que seu resultado seja o melhor possível enquanto que o Merge apresentou sempre uma estabilidade invejável.

# 8. Referências

EDUARDO, “Embaralhando o Conteúdo de Vetores em Java”, DevMedia. Disponível em <https://www.devmedia.com.br/embaralhando-o-conteudo-de-vetores-em-java/26382> Acesso em 29/03/2021

FILIPE, Henrique “Selection Sort”, Blog Cyberini. Disponível em [https://www.blogcyberini.com/2018/06/selection-sort.html#:~:text=Sabemos%20que%2C%20num%20vetor%20ordenado,segunda%20posição%20e%20assim%20sucessivamente.&text=Como%20exemplo%2C%20ordenaremos%20o%20vetor,%2C%20-1%2C%206%5D%20.Acesso em 29/03/2021](https://www.blogcyberini.com/2018/06/selection-sort.html#:~:text=Sabemos%20que%2C%20num%20vetor%20ordenado,segunda%20posição%20e%20assim%20sucessivamente.&text=Como%20exemplo%2C%20ordenaremos%20o%20vetor,%2C%20-1%2C%206%5D%20.Acesso em 29/03/2021 )

Acesso em :28/03/2021

FILIPE, Henrique “Merge Sort”, Blog Cyberini. Disponível em <https://www.blogcyberini.com/2018/07/merge-sort.html> Acesso em :28/03/2021

FILIPE, Henrique “Insertion Sort”, Blog Cyberini. Disponível em <<https://www.blogcyberini.com/2018/06/insertion-sort.html> > Acesso em :28/03/2021

FILIPE, Henrique “Quick Sort”, Blog Cyberini. Disponível em <https://www.blogcyberini.com/2018/08/quicksort-analise-e-implementacoes.html> Acesso em :28/03/2021

BRUNET, João “Ordenação por comparação Quick Sort”, GitHub. Disponível em <https://joaoarthurbm.github.io/eda/posts/quick-sort/> Acesso em 29/03/2021