

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

**Projeto de Análise de Algoritmos**

**Análise Experimental de Algoritmos**

**Aluno: Luan Miranda Bessa**

**Matricula: 201815020037**

**Professor: José Gomes Lopes Filho**

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

Sumário

[Introdução 5](#_Toc69855255)

[1. Algoritmo: 6](#_Toc69855256)

[1.1 Insertion Sort 6](#_Toc69855257)

[1.1.1 Análise de complexidade: 6](#_Toc69855258)

[1.1.2 Complexidade de tempo: 6](#_Toc69855259)

[1.1.3 Complexidade do Espaço 6](#_Toc69855260)

[**1.2 Bubble Sort** 7](#_Toc69855261)

[1.2.1 Análise de complexidade: 7](#_Toc69855262)

[1.2.2 Complexidades de tempo: 7](#_Toc69855263)

[1.2.3 Complexidades de espaço: 7](#_Toc69855264)

[**1.3 Merge Sort** 8](#_Toc69855265)

[1.3.1 Análise de complexidade: 8](#_Toc69855266)

[8](#_Toc69855267)

[1.2.2 Complexidades de tempo: 8](#_Toc69855268)

[1.3.3 Complexidades de espaço: 8](#_Toc69855269)

[**1.4 Quick Sort** 9](#_Toc69855270)

[1.4.1 Análise de complexidade: 9](#_Toc69855271)

[1.4.2 Complexidades de tempo: 9](#_Toc69855272)

[1.4.2 Complexidades de espaço: 9](#_Toc69855273)

[**1.5 Selection Sort** 10](#_Toc69855274)

[1.5.1 Análise de complexidade: 10](#_Toc69855275)

[1.5.2 Complexidades de tempo: 10](#_Toc69855276)

[1.5.3 Complexidades de espaço: 10](#_Toc69855277)

[2. Implementação: 11](#_Toc69855278)

[2.1.1 Código do Insertion Sort: 11](#_Toc69855279)

[2.1.2 Ilustração do funcionamento: 11](#_Toc69855282)

[2.2.1 Código do Bubble Sort: 12](#_Toc69855283) [16](#_Toc69855293)

[3. Listagem de testes executados: 16](#_Toc69855294)

[Conclusão: 22](#_Toc69855295)

[Bibliografia: 22](#_Toc69855296)

# Introdução

Este experimento consiste em fazer análise assintótica e de desempenho dos algoritmos de ordenação abordado em diferentes cenários com ArraysList<Integer> de modo ordenado, inverso, aleatório, quase ordenado, para encontrar o melhor entre eles no pior caso, entre os candidatos estão: Insertion Sort, Bubble Sort, Merge Sort, Quick Sort, Selection Sort, todos eles foram testados com mesmo computador para não haver divergências de dados, usando a linguagem Java para o desenvolvimento.

Os testes foram realizados com vetor de tamanho, 10, 100, 1000, 10.000, 100.000, 1.000.000. No decorre deste artigo mostraremos com detalhes o desempenho obtido de cada algoritmo, com três métricas especificas, número de movimentação de registros, número de trocas de chaves e tempo total gasto na ordenação do vetor.

Insertion Sort, é um dos algoritmos de classificação mais simples, pois classifica um único elemento em uma instância específica. Não é o melhor algoritmo de ordenação em termos de desempenho, mas é ligeiramente mais eficiente do que a Selection Sort e Bubble Sort em cenários práticos. É uma técnica de ordenação intuitiva.

Bubble Sort, também conhecido como bolha. É um algoritmo de ordenação simples, funciona percorrendo repetidamente a lista a ser ordenada, comparando dois itens por vez e trocando-os se estiverem na ordem errada. A passagem pela lista é duplicada até que nenhuma troca seja desejada, o que significa que a lista está ordenada.

Merge Sort, é um algoritmo de ordenação que se enquadra na categoria da técnica de [divisão e conquista](https://www.javatpoint.com/divide-and-conquer-introduction). É uma das melhores técnicas de ordenação que constrói com sucesso um algoritmo recursivo. Nesta técnica, dividimos um problema em duas partes e os resolvemos individualmente. Depois de encontrar a solução de cada parte, nós as mesclamos de volta para representar a solução do problema principal.

Quick Sort, é outro algoritmo do tipo divisão e conquista, onde divide os elementos do vetor em duas subpartes direita e esquerda, um elemento no meio é escolhido como pivô com base nesse elemento procura cada elemento na subparte esquerda seja menor ou igual a ele e cada elemento na subparte direita seja maior ou igual ao pivô. Um dos piores senários seria escolher um péssimo pivô, onde aumentaria a complexidade do algoritmo.

Selection Sort, uma versão aprimorada do método de ordenação bolha, fazendo apenas uma única troca para cada passagem no vetor. Para fazer isso, escolheram um valor mínimo no inicio e quando passar pelo vetor e encontrar outro valor que seja menor que o mínimo esse valor passa a ser nosso novo mínimo, isso se repete ate o fim do primeiro loop, encontrando assim nosso menor valor. Após isso vamos passar no vetor a partir n-1 e encontrar o segundo mínimo, fazendo assim ate que o vetor estiver ordenado.

# 1. Algoritmo:

## 1.1 Insertion Sort

## 1.1.1 Análise de complexidade:

**Entrada:** Dados n elementos de entrada.

**Lógica:** se tivermos **n** elementos, então, na primeira passagem, ele fará **n-1** comparações; na segunda passagem, ele fará **n-2** ; na terceira passagem, fará **n-3** e assim por diante. Assim, o número total de comparações é;

**Resultado;**

**(n-1) + (n-2) + (n-3) + (n-4) + ...... + 1**

Portanto, o algoritmo de ordenação por inserção abrange uma complexidade de tempo de **O (n 2 )**.

## 1.1.2 Complexidade de tempo:

* **Complexidade do melhor caso:** O algoritmo de ordenação de inserção tem uma complexidade de tempo de melhor caso de **O(n)** para um vetor já ordenado, apenas o loop externo está executando n vezes e o loop interno não é executado.
* **Complexidade do caso médio:** O algoritmo de ordenação por inserção é **O (n2)**, que ocorre quando os elementos existentes estão em ordem aleatória, ou seja, nem na ordem crescente nem na ordem decrescente.
* **Complexidade do pior caso:** A complexidade de tempo de pior caso também é **O (n2)**, ocorre quando ordenamos um vetor na ordem inversa. Neste algoritmo, cada elemento individual é comparado com o resto dos elementos, devido ao qual n-1 comparações são feitas para cada elemento.

O algoritmo de ordenação por inserção é altamente recomendado, especialmente quando alguns elementos são deixados para ordenação ou no caso de o vetor conter poucos elementos.

### 1.1.3 Complexidade do Espaço

O Insertion Sort tem complexidade de espaço de **O(1)** devido ao uso de uma variável extra.

**1.2 Bubble Sort**

## 1.2.1 Análise de complexidade:

**O bolha** começa com o primeiro índice que se torna um elemento bolha. Em seguida, ele é comparado com o segundo elemento do vetor, se o elemento bolha for maior que o segundo elemento, e feito a movimentação de registro, se não for o elemento bolha passa a ser o próximo elemento.

Após a troca se houver, o segundo elemento se tornará o elemento bolha. Agora vamos comparar o segundo elemento com o terceiro, como fizemos na etapa anterior, e trocá-los, se necessário. O mesmo processo é seguido até o último elemento.

Portanto, o algoritmo de classificação de bolhas abrange uma complexidade de tempo de **O (n2)**.

### 1.2.2 Complexidades de tempo:

* **Melhor caso de complexidade**: O algoritmo de ordenação bolha tem um melhor caso de complexidade de tempo de **O (n)** para o vetor já ordenado por estar modificado para isso na sua melhor implementação.
* **Complexidade de caso médio**: A complexidade de tempo no caso médio para o algoritmo é **O(n 2)**, que acontece quando o vetor esta aleatório ou seja nem crescente nem decrescente.
* **Complexidade de pior caso**: A complexidade de tempo no pior caso também é **O(n2)** , que ocorre quando o vetor está na ordem decrescente ou inversa.

### 1.2.3 Complexidades de espaço:

O bolha tem complexidade de espaço de **O(1)** porque necessita de algum espaço de memória extra para a variável temporária para troca.

**1.3 Merge Sort**

## 1.3.1 Análise de complexidade:

Onde T (n) o tempo total gasto pelo algoritmo Merge Sort.

* A ordenação de duas metades levará no máximo.
* Quando mesclamos as listas ordenadas, obtemos uma comparação total de n-1 porque o último elemento restante precisará ser copiado para baixo na lista combinada e não haverá comparação.

Assim, a fórmula relacional será

Mas ignoramos '-1' porque o elemento levará algum tempo para ser copiado nas listas do merge.

Então,

### Complexidades de tempo:

* **Complexidade de melhor caso:** O algoritmo tem uma complexidade de tempo de melhor caso de **O(n\*log n)** para a vetor já ordenado.
* **Complexidade de caso médio:** A complexidade de tempo no caso médio para o algoritmo de ordenação é **O(n\*log n)**, que acontece quando o vetor estar na ordem aleatória.
* **Complexidade de pior caso:** A complexidade de tempo no pior caso também é **O (n\*log n)**, que ocorre quando o vetor estar na ordem inversa.

### 1.3.3 Complexidades de espaço:

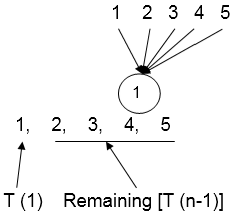
A complexidade do espaço do Merge Sort é **O(n)**.

**1.4 Quick Sort**

## 1.4.1 Análise de complexidade:

É algoritmo de ordenação muito usado que faz **(n.log n)** comparações em caso médio para ordenação de um vetor de n elementos. Este algoritmo segue a abordagem de dividir e conquistar. O algoritmo processa o vetor da seguinte maneira, define o primeiro índice do vetor para a variável esquerda e o elemento do meio como pivô. Defina o último índice do vetor para a variável direita. isto é, à esquerda = 0, pivô = [(n+n-1)/2], direita = n - 1, onde n é o tamanho do vetor.

**Análise no pior caso** T (n) = T ( 1 ) + T (n- 1 ) + n

**T (1)** é o tempo gasto pelo elemento pivô.

**T (n-1)** é o tempo gasto pelo elemento restante, exceto pelo elemento pivô.

**N:** o número de comparações necessárias para identificar a posição exata de cada elemento, se compararmos o pivô do primeiro elemento com outro, haverá 5 comparações.

### 1.4.2 Complexidades de tempo:

* + - **Complexidade de melhor caso:** Em qualquer ordenação, o melhor caso ocorre quando as partições estão balanceadas, com complexidade de **O(n\*logn).**
    - **Complexidade de melhor médio:** Geralmente, assumimos o meio elemento da lista como o elemento pivô. Em um caso médio, o número de chances de obter um elemento pivô é igual ao número de itens. **O (n\*logn).**
    - **Complexidade de pior caso:** é quando escolhemos péssimo pivô ou o primeiro ou o ultimo elemento podendo chegar ate **O (n²)** neste caso, mais o algoritmo testado foi implementado escolhendo o pivô do meio tendo assim **(n\*log(n)).**

### 1.4.2 Complexidades de espaço:

A complexidade de espaço do **Quick Sort** e de **T(1) + T(n-1),** porque T(1) e a escolha do elemento como pivô, e T(n-1) as comparações de n-1 elementos.

**1.5 Selection Sort**

## 1.5.1 Análise de complexidade:

Na ordenação de seleção, em primeiro lugar, definimos o elemento inicial como um **mínimo**. Agora vamos comparar o mínimo com o segundo elemento. Se o segundo elemento acabar sendo menor do que o mínimo, vamos trocá-los.

Caso contrário, se o segundo elemento for maior do que o mínimo, que é o nosso primeiro elemento, não fará nada e passaremos para o terceiro elemento e então o compararemos com o mínimo. Vamos repetir esse processo até chegarmos ao último elemento. Após a conclusão de cada iteração, notaremos que nosso mínimo atingiu o início da lista não ordenada.

**Entrada:** Dados **n** elementos de entrada.

**Saída:** número de etapas incorridas para ordenar uma lista.

**Lógica:** se tivermos n elementos, então, na primeira passagem, ele fará **n-1** comparações; na segunda passagem, ele fará **n-2**; na terceira passagem, fará **n-3** e assim por diante. Assim, o número total de comparações pode ser encontrado por:

### 1.5.2 Complexidades de tempo:

* **Complexidade de melhor caso:** O algoritmo de ordenação de seleção tem uma complexidade de tempo de melhor caso de **O (n2)** para um vetor já ordenado.
* **Complexidade de caso médio:** A complexidade de tempo do caso médio para o algoritmo de ordenação de seleção é **O(n 2)** , em que os elementos existentes estão em ordem aleatória, ou seja, nem em ordem crescente nem decrescente.
* **Complexidade de** **pior caso**: A complexidade de tempo de pior caso também é **O(n2)** , que ocorre quando ordenamos um vetor de ordem inversa para uma ordem ordenada.

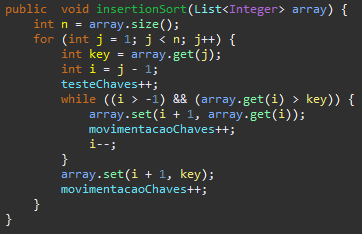
### 1.5.3 Complexidades de espaço:

A complexidade de espaço é de **O(1)** porque necessita de algum espaço de memória extra para a variável temporária para troca.

# Implementação:

### 2.1.1 Código do Insertion Sort:

O código do Insertion Sort é bem simples implementar o primeiro loop percorre a lista e o segunda loop roda enquanto a condição e verdadeira e realiza as inserções, testa se **a0<a1**, se for insere o a1 na primeira posição, senão continua o loop testando **a0< a1< a2< a3< a4<....n** sempre inserido o menor elemento na primeira posição, objetivo e testar mais e trocar menos.



### Entrada do vetor = {41, 22, 63, 14, 55, 36}

### Saída do vetor = {14, 22, 36, 41, 55, 63}

Figura 1: mostra implementação do método InsertionSort.

### 2.1.2 Ilustração do funcionamento:

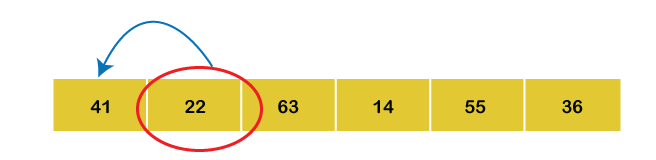


Figura 2: 1-Passo compara o Elemento a0 com a1

Figura 3: 2-Passo troca os elementos

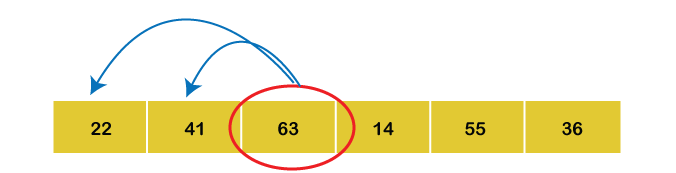


Figura 4: 3-Passo comparar o Elemento a2 <a1 ou a2<a0 senão for mantei no vetor.

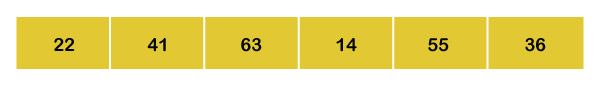


Figura 5: 4-Passo o elemento a2 e mantido por não ser menor que os das esquerdas.

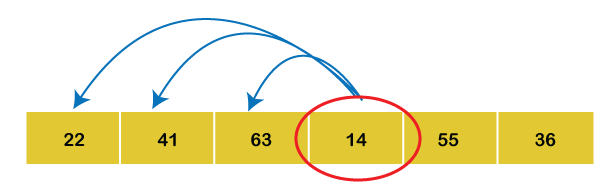


Figura 6: 5-Passo comparar o Elemento a3 <a2 & a2<a1 & a3<a0, se for menor faz a troca, senão for mantei no vetor.

Figura 7: 6-Troca dos elementos.

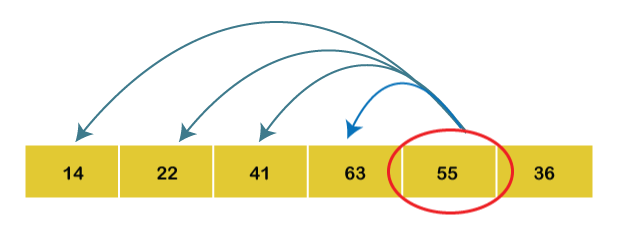


Figura 8: 7-Passo comparar o Elemento a4 <a3 , a2, a1, a0, se for menor faz a troca, senão for mantei no vetor.

Figura 9: 8-Passo troca dos elementos.

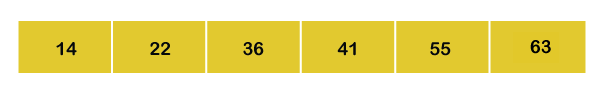
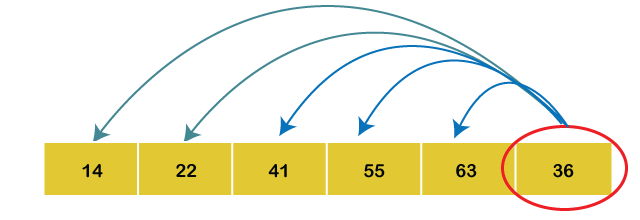


Figura 8: 7-Passo comparar o Elemento a5 <a4, a3, a2, a1, a0, se for menor faz a troca, senão for mantei no vetor.

Figura 9: 8-Passo troca dos elementos.

### 2.2.1 Código do Bubble Sort:

O método bolha é um algoritmo de troca onde realiza trocas entre os elementos do vetor sempre em pares, no inicio testa o elemento **a0 < a1** se for menor e ele troca senão, continua percorrer o vetor sempre testando em pares elementos, **a1 < a2, a2 < a3, a3 < a4.....n,** o ponto de parada e quando a ultima loop verificar que não tem mais elemento menores que o seu sucessor o algoritmo entende que o vetor esta ordenado.

public void bubbleSort(List<Integer> arr) {

int n = arr.size();

int temp = 0;

for (int i = 0; i < n; i++) {

for (int j = 1; j < (n - i); j++) {

if (arr.get(j - 1) > arr.get(j)) {

temp = arr.get(j - 1);

arr.set(j-1, arr.get(j));

arr.set(j, temp);

}

}

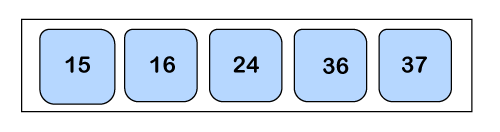
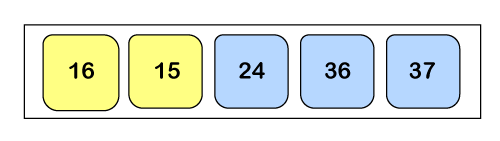
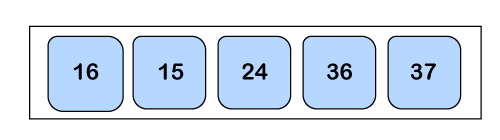
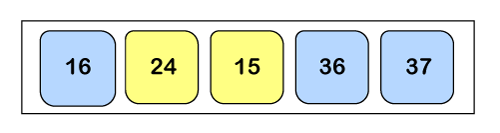
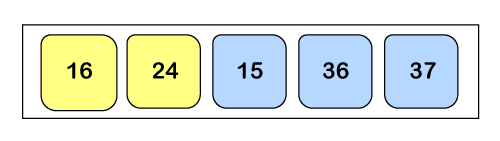
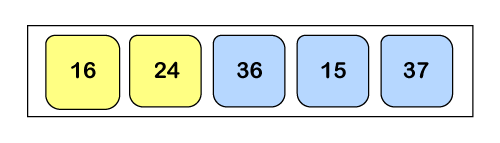
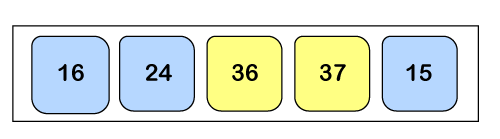
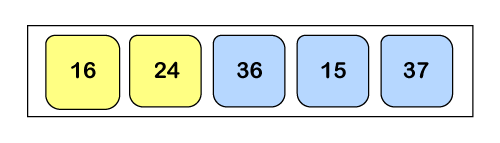
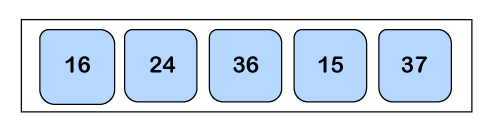
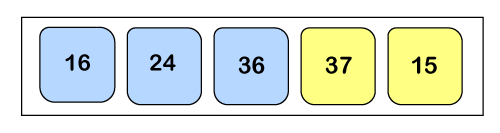
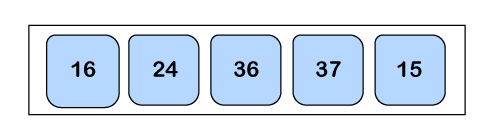
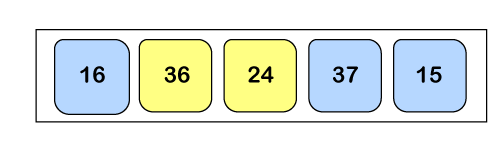
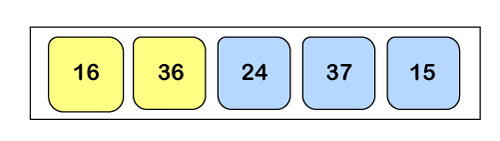
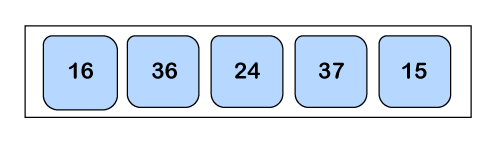
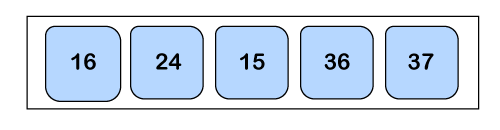
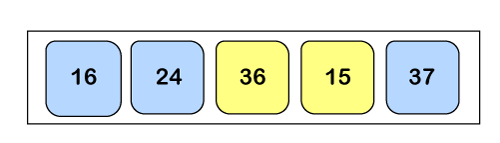
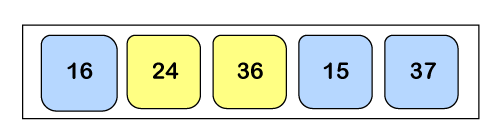
}

}

### Entrada do vetor = {16, 36, 24, 37, 15}

### Saída do vetor = {15, 16, 24, 36, 37}

**2.2.2 Ilustração do funcionamento:**



**1-Passo percorre o vetor 2-Passo testa o elemento a0<a1 3-Passo testa a1<a2 se for troca 4-Passo resultado da troca**

**5-Passo testa a2<a3 6-Passo testa a3<a4 se for troca 7- Passo resultado da troca 8-Passo testa a1<a2 se for troca**

**9-Passo testa a0<a1 10-Passo testa a1<a2 se for troca 11- Passo testa a3<a4 se for troca 12- Passo resultado da troca troca**

**13-Passo testa a0<a1 14-Passo testa a1<a2 se for troca 15 - Passo resultado da troca 16- Passo testa a0<a1**

**Resultado final: Movimentação de registro: 05 Teste de chaves: 10**

**16-ultima vez percorre o vetor testando para ver se esta ordenado**

**2.3.1 Código do Merge Sort:**

É um algoritmo de divisão e conquista que vai dividido o vetor a ate chegar no seu caso base que um elemento a0, depois mescla o elemento com o vetor de cima e testa se o elemento a0<a1 depois troca se for menor, depois mescla com o de cima e assim sucessivamente.

private void split(List<Integer> arr, int beg, int mid, int end) {

int l = mid - beg + 1;

// posição final

int r = end - mid;

// vetor da esquerda

int leftArray[] = new int[l];

// vetor da direita

int rightArray[] = new int[r];

//for andando para o lado esquerdo e sendo carregado ate i seja maior l

for (int i = 0; i < l; ++i) {

leftArray[i] = arr.get(beg + i);

}

// for andando para o lado direito e sendo carregado ate j seja maior r

for (int j = 0; j < r; ++j) {

rightArray[j] = arr.get(mid + 1 + j);

}

int i = 0, j = 0;

// k recebendo o inicio

int k = beg;

// laço enquanto é i<l e j<r

while (i < l && j < r) {

testeChave++;

if (leftArray[i] <= rightArray[j]) {

// se o lado esquerdo for menor que o lado direito fazer a movimentação

arr.set(k, leftArray[i]);

movimentacaoChaves++;

i++;

} else {

// senão o lado direito fazer a movimentação

arr.set(k, rightArray[j]);

movimentacaoChaves++;

j++;

}

k++;

}

while (i < l) {

arr.set(k, leftArray[i]);

movimentacaoChaves++;

i++;

k++;

}

while (j < r) {

arr.set(k, rightArray[j]);

movimentacaoChaves++;

j++;

k++;

}

}

public void merge(List<Integer> arr, int beg, int end) {

if (beg < end) {

int mid = (beg + end) / 2;

merge(arr, beg, mid);

merge(arr, mid + 1, end);

split(arr, beg, mid, end);

}

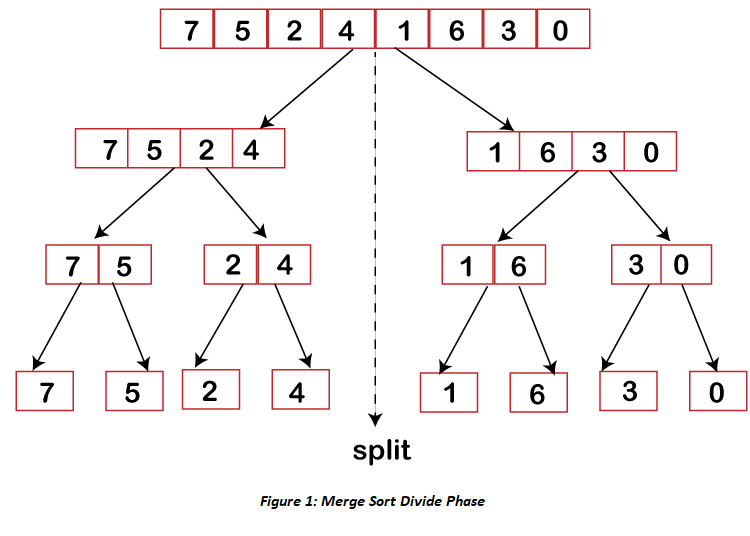
**PSEUDO CODIGO-MERGE SORT**

1. 1. Se p **< r**
2. 2. Então q → (p + r) / 2
3. 3. MERGE-SORT (A, p, q)
4. 4. MERGE-SORT (A, q + 1, r)
5. 5. FUSÃO (A, p, q, r)

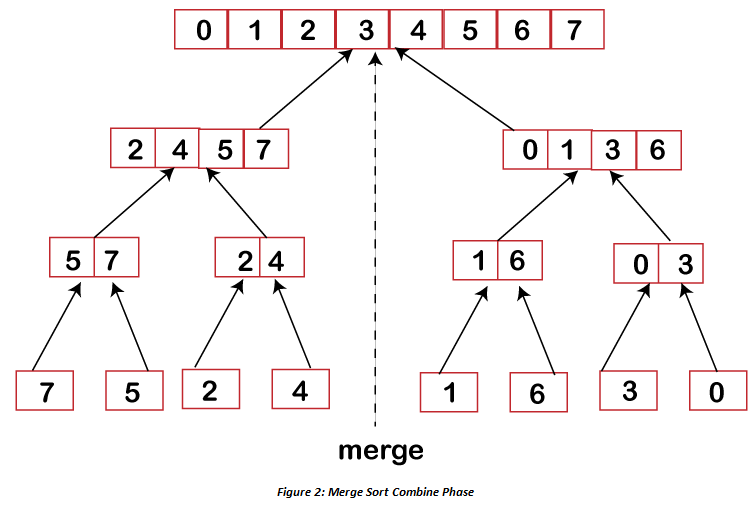
### Entrada do vetor = {7, 5, 2, 4, 1, 6, 3, 0}

### Saída do vetor = {0, 1, 2, 3, 4, 5, 6, 7}

**2.3.2 Ilustração do funcionamento:**



1-Passo Como você pode ver na imagem acima, o algoritmo de ordenação MergeSort, divide recursivamente o vetor em metades até que a condição básica seja satisfeita, onde ficamos com apenas 1 elemento no vetor. E então, o método de merge seleciona as subpartes ordenada e as mescla de volta para ordena o vetor inteiro.



2-Passo o algoritmo recursivo depende de um caso base, bem como de sua capacidade de mesclar de volta os resultados derivados dos casos base. Quando chega no caso base o algoritmo volta mesclando e testando se a0<a1...1 troca e junta com o vetor de cima.

**2.4.1 Código do Quick Sort:**

É algoritmo de divisão e conquista, onde e escolhido um pivô a partir de uma calculo matemático que é pivô=(n0 + n-1)/2, como base de teste se o elemento **n0<pivô**, se for ele para o loop e começa um lado direito se encontrar um elemento do lado direito menor que pivô ele para, depois testa se o elemento da direita e menor que da esquerda se for ele troca, assim sucessivamente, **a1<pivô a2<pivô a3<pivô....n**. / **b1>pivô b2>pivô b3>pivô....n**. **/ troca a<=b**

public void quickSort(List<Integer> arr, int esquerda, int direita) {

int esq = esquerda;

int dir = direita;

int pivo = (arr.get(esq) + arr.get(dir)) / 2;

int troca;

while (esq <= dir) {

while (arr.get(esq) < pivo) {

esq = esq + 1;

testeChave++;

}

while (arr.get(dir)> pivo) {

dir = dir - 1;

testeChave++;

}

if (esq <= dir) {

troca = arr.get(esq);

arr.set(esq, arr.get(dir));

arr.set(dir, troca);

esq = esq + 1;

dir = dir - 1;

}

}

if (dir > esquerda) {

quickSort(arr, esquerda, dir);

}if (esq < direita) {

quickSort(arr, esq, direita);

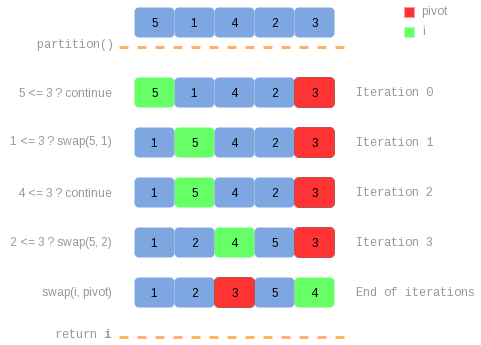
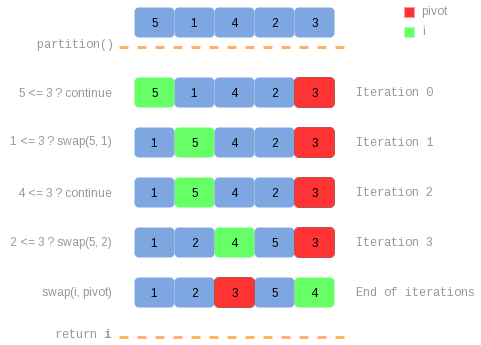
}

}

### Entrada do vetor = {5, 1, 4, 2, 3}

### Saída do vetor = {1, 2, 3, 4, 5}

**2.3.2 Ilustração do funcionamento:**



1 2 3 4 5

**2.4.1 Código do Selection Sort:**

O Selection Sort é um algoritmo do tipo troca por seleção, no código abaixo mostra como pode ser feita a implementação do selection sort de maneira iterativa um elemento a0 escolhido como mínimo e comparado com o a1, essas comparações se repete até achar um valor que menor que o mínimo e assim esse elemento passa a ser o novo mínimo como mostra na **figura 5**.

public void selectionSort(List<Integer> lista) {

for (int i = 0; i < lista.size() - 1; i++) {

int index = i;

for (int j = i + 1; j < lista.size(); j++) {

if (lista.get(j) < lista.get(index)) {

index = j;

}

}

int menor = lista.get(index);

lista.set(index, lista.get(i));

lista.set(i, menor);

}

}

PSEUDO CODIGO

1. 1 . k ← comprimento [A]
2. 2 . **para** j ← 1  a n- 1
3. 3 . menor ← j
4. 4 . **para** I ← j +  1  para k
5. 5 . **se** A [i] <A [menor]
6. 6 . então o menor ← i
7. 7 . troca (A [j], A[menor])

### Entrada do vetor = {20, 12, 10, 15, 2}

### Saída do vetor = {2, 12, 10, 15, 20}

# Selection Sort Algorithm

# 

Figura 5 mostra como e feita a troca por seleção

# Listagem de testes executados:

* + 1. ***Análise gráfica do Insertio Sort*:**

**Análise:** Com base no gráfico, gerados a partir dos dados coletados nos testes, notamos que o insertio sort tem um crescimento quadrático na medida em que o vetor aumenta de tamanho, também aumenta complexidade do algoritmo assim aumentando o seu tempo de execução.

“Se você tivesse que generalizar para todos os casos de ordenação por inserção, você teria de dizer que ela leva o tempo *O*(*n*2), Você não pode dizer que ela roda no tempo Θ(*n*2), em todos os casos, já que o melhor caso roda em  Θ(*n*). E você não pode dizer que ela roda em Θ(*n*), n, em todos os casos, porque o pior tempo de execução é Θ(*n*2) […]”. ([CORMEN](http://www.cs.dartmouth.edu/~thc/); [BALKCOM](http://www.cs.dartmouth.edu/~devin/), ciência da computação da Universidade de, 2015).

* + 1. ***Análise gráfica do*** ***Bubble Sort*:**

**Análise:** Com base no gráfico, gerados a partir dos dados coletados nos testes, notamos que o BubbleSort tem um crescimento quadrático na medida que o vetor é trocado, a complexidade do algoritmo se torna O(n²) no caso inverso e aleatório.

“[…] Para o desempenho do Bubble Sort, tem-se: no pior caso e no médio caso O(n²), ou seja, uma complexidade quadrática, já no melhor caso O(n) a complexidade é linear”. (FOLADOR; NETO; JORGE, 3.pg, 2014).

* + 1. ***Análise gráfica do*** ***Merge Sort*:**

**Análise:** Com base no gráfico, gerados a partir dos dados coletados nos testes, notamos que o Merge Sort tem um ótimo desempenho em reorganizar os dados em diferentes instancias, como mostrado nos gráficos acima, sendo quase um valor constante a partir da entra que recebe um dos melhores algoritmos dos testes.

“[…] Para o merge sort comum, sem variações, a complexidade de tempo no pior caso, caso médio e melhor caso é O(n log n)”. (FOLADOR; NETO; JORGE, 3.pg, 2014).

* + 1. ***Análise gráfica do*** ***Quick Sort*:**

**Análise:** Com base no gráfico, gerados a partir dos dados coletados nos testes, vimos no gráfico a eficácia comprovando do algoritmo quick sort em diferentes senários, mostra sempre um menor tempo na execução como também a uma menor troca de chaves, executando sempre teste antes de trocar as chaves e com isso ganhando desempenho na execução. Neste experimento foi considerado o melhor algoritmo na ordenação independente do tipo do vetor testado.

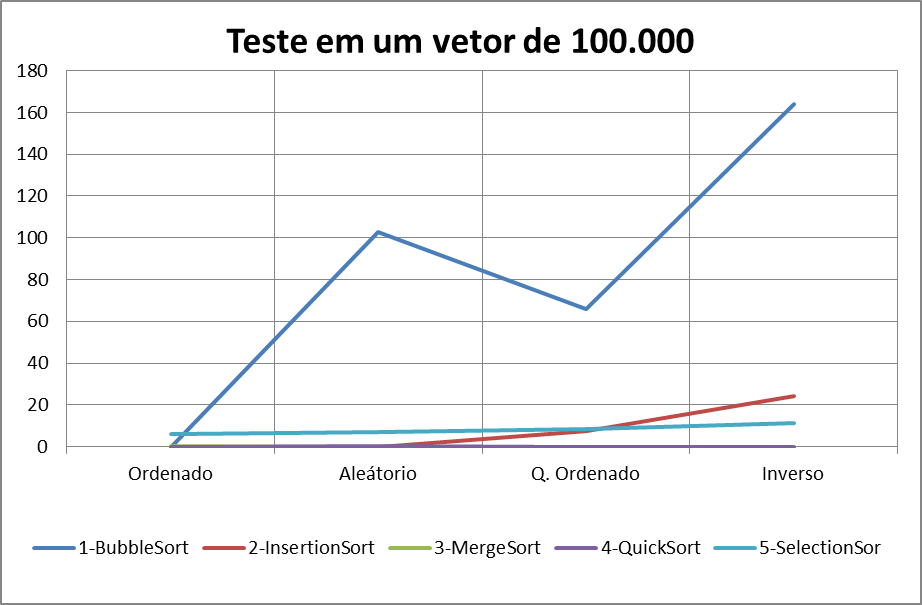
“O método de ordenação quicksort é considerado um dos métodos mais rápidos. Consiste em selecionar primeiramente um dos elementos do conjunto a ordenar para ser o elemento pivô, o qual é um elemento já ordenado, em seguida faz uma subdivisão do conjunto inicial em dois subconjuntos […]”. (XVI SEMINARIO, 1.pg, 2011).

* + 1. ***Análise gráfica do*** ***Seletion Sort*:**

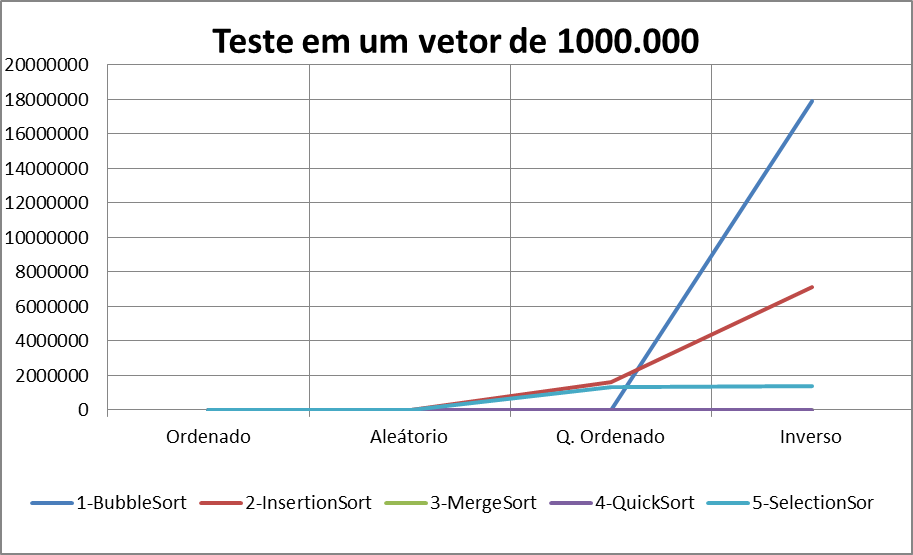
**Análise:** Com base no gráfico, gerados a partir dos dados coletados nos testes, fica evidente que o Seletion Sort tem um péssimo desempenho em reorganizar os dados, em senários aonde a entrada superar cem mil elementos o seu crescimento chega a ser quadrático na medida em que o vetor é trocado, a complexidade do algoritmo se torna O(n²).

* + 1. **Análise gráfica do** **Tempo de execução:**

A análise será feita apenas nos vetores de tamanho cem mil e um milhão de elementos, pois são vetores onde tem melhor apontamento dos dados do tempo de execução de cada algoritmos.



**Figura 10:** Gráfico do tempo execução em segundos comparando vetores divergentes



**Figura 11:** Gráfico do tempo execução em segundos comparando vetores divergentes

# Conclusão:

Os testes comprovam que Quick Sort e o Merge Sort são excelentes algoritmos de ordenação nos diversos senários colocados em prática neste experimento computacional, ambos levam enorme vantagem no melhor e no pior caso contra seus adversários, buble sort, insertion sort e seletion sort, mesmo com grandes entradas.

Neste experimento ficou comprovado que os vetores que os algoritmos tiveram uma dificuldade de ordenar foram o quase ordenado e o inverso, mostrado na **figura 10** e **figura 11** e comprovado na analise gráfica mostrada pelos gráficos acima, que a partir desse ponto com vetores de tamanho de cem mil e um milhão a linha gráfica tem um acedência maior, ficando nítido que os piores algoritmos são: buble sort, insertion sort e seletion sort.

Encontramos dificuldades na alocação dos dados, como seria armazenados e comparados para assim obter um resultado satisfatório, já na implementação dos algoritmos foi saber quando deve fazer a contagem da movimentação de chaves ou a troca delas, mais no demais vou uma ótima experiência em realizar os testes e comprovar realmente a eficácia dos algoritmos vencedores.

Os dados obtidos neste experimento se encontram organizados em planilhas neste link <https://docs.google.com/spreadsheets/d/1sfZDGwr35vOoQIqUUaTsiqJHh-TI0WQmixqclXb9398/edit?usp=sharing>

O programa que realizou os teste pode ser baixado por esse link

# Bibliografia:

[CC-BY-NC-SA](http://creativecommons.org/licenses/by-nc-sa/4.0/).(2015) [**Ciência da Computação da Universidade de Dartmouth**](http://www.cs.dartmouth.edu/), [Thomas Cormen](http://www.cs.dartmouth.edu/~thc/) e [*Devin Balkcom*](Devin%20Balkcom)*,* Fonte: <https://pt.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort>

UPF.(2014) JOÃO FOLADOR, LÁZARO NETO, DAVID JORGE, **Aplicativo para análise comparativa do comportamento de algoritmos de ordenação**, Fonte: <http://seer.upf.br/index.php/rbca/article/view/3792/2935>

(2011)**XVI SEMINARIO UNIVERSITARIO**, Fonte: <https://home.unicruz.edu.br/seminario/anais/anais-2011/agrarias/M%C3%83%E2%80%B0TODOS%20DE%20ORDENA%C3%83%E2%80%A1%C3%83%C6%92O-%20UMA%20AN%C3%83%C2%81LISE%20QUANTITATIVA%20PARA%20O%20PIOR%20CASO.pdf>

<https://docplayer.com.br/73508456-Algoritmos-de-ordenacao.html>

<https://joaoarthurbm.github.io/eda/posts/quick-sort/>

<https://www.programiz.com/dsa/selection-sort>

<https://panda.ime.usp.br/pythonds/static/pythonds_pt/05-OrdenacaoBusca/OBubbleSort.html>

<https://www.interviewbit.com/tutorial/merge-sort-algorithm/>

<https://www.geeksforgeeks.org/insertion-sort/>