Avalição de desempenho entre diferentes métodos de Ordenação

Tiago Vieira Paulin

Estrutura de Dados – Pontifícia Universidade Católica do Paraná (PUCPR) – Curitiba, Paraná – Brazil

tiago.paulin@pucpr.edu.br

Abstract. This report aims to propose a comparative analysis of different sorting methods. Through tests that will take into account the execution time, number of swaps, and iterations performed, the analysis will be carried out between the sorting methods QuickSort, MergeSort, and BubbleSort to assess their performance under varying numbers of elements. This will enable us to conclude which method performs better under specific circumstances

Resumo. Esse relatório tem como finalidade propor uma análise comparativa entre diferentes métodos de ordenação. Por meio de testes que levarão em conta o tempo de execução, número de trocas e iterações realizadas, a análise será feita entre os métodos de ordenação QuickSort, MergeSort e BubbleSort para averiguar o desempenho deles em diferentes com determinado número de elementos, para que dessa forma possamos concluir qual método se sai melhor em qual caso.

1. Introdução

Primeiramente no relatório será feita uma uma breve explicação de como foi feita a implementação da classe Ordenação em Java bem como os métodos de ordenação escolhidos para o trabalho, com prints e explicação do código completo. Após isso será feita a análise comparativa, exibindo como foram feitos os testes, os resultados da execução do código e gráficos baseado nos resultados obtidos. Posteriormente será feita a conclusão do trabalho, que tem a finalidade de refletir sobre os resultados obtidos.

2. Implementação da classe Ordenação

Algoritmos de ordenação tem a finalidade de ordenar um determinado conjunto de elementos de acordo com a situação desejada, seja crescente, decrescente, ordem alfabética etc. Uma vez que um conjunto de elementos está ordenado seguindo um padrão conhecido, tarefas como encontrar um elemento pode se torna mais simples e rápida, uma vez que a organização destes elementos segue um padrão conhecido. Alguns fatores que podem influenciar no desempenho dos métodos de ordenação é a forma que foi implementado, tamanho do conjunto de elementos, disposição dos elementos da entrada (desordenados, parcialmente ordenados, etc.) e a configuração da máquina em que o algoritmo está sendo executado.

Figura 1. Implementação dos atributos e Construtor da classe

A classe Ordenação possui 2 atributos, entre eles, o atributo iterações que contabiliza o número de iterações que determinado método de ordenação realizou para ordenar os elementos, e o atributo trocas que contabiliza as trocas realizadas por determinado método de ordenação ao ordenar um conjunto de elementos. No método construtor, quando o Objeto Ordenação é instanciado, inicialmente o valor dos atributos são inicializados como zero.

2.1. BubbleSort

O método de ordenação BubbleSort é um dos métodos de ordenação mais simples de se implementar, seu funcionamento consiste em percorrer o conjunto de elementos e realizar constantes comparações entre o elemento e seu sucessor, caso o sucessor seja menor que que o elemento, eles são trocados de posição. Dessa forma o processo se repete até que o conjunto de elementos esteja ordenado em ordem crescente.

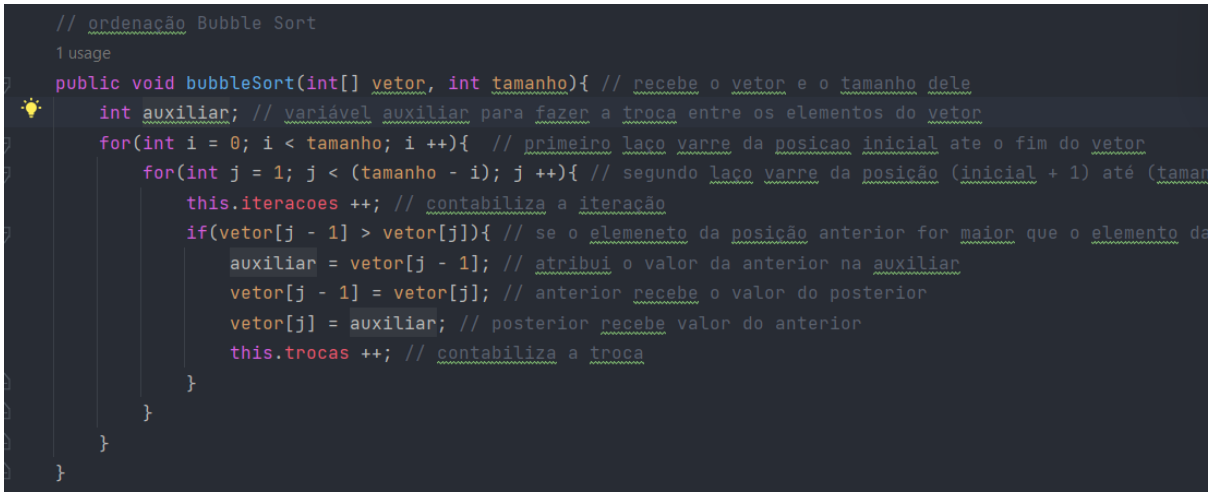


Figura 2. Implementação do método de ordenação bubbleSort

2.2. MergeSort

O método de MergeSort consiste em realizar a junção de dois vetores ordenados para gerar um novo vetor. Levando em conta que no trabalho em questão os conjutos de elementos na entrada estão completamente desordenados, método em questão fará constantes divisões

no conjunto de elementos até que o vetor seja subdividido em vetores menores, após isso os elementos são ordenados e os fragmentos do vetor vão ser rearranjados em novos vetores até que o vetor esteja completamente ordenado.

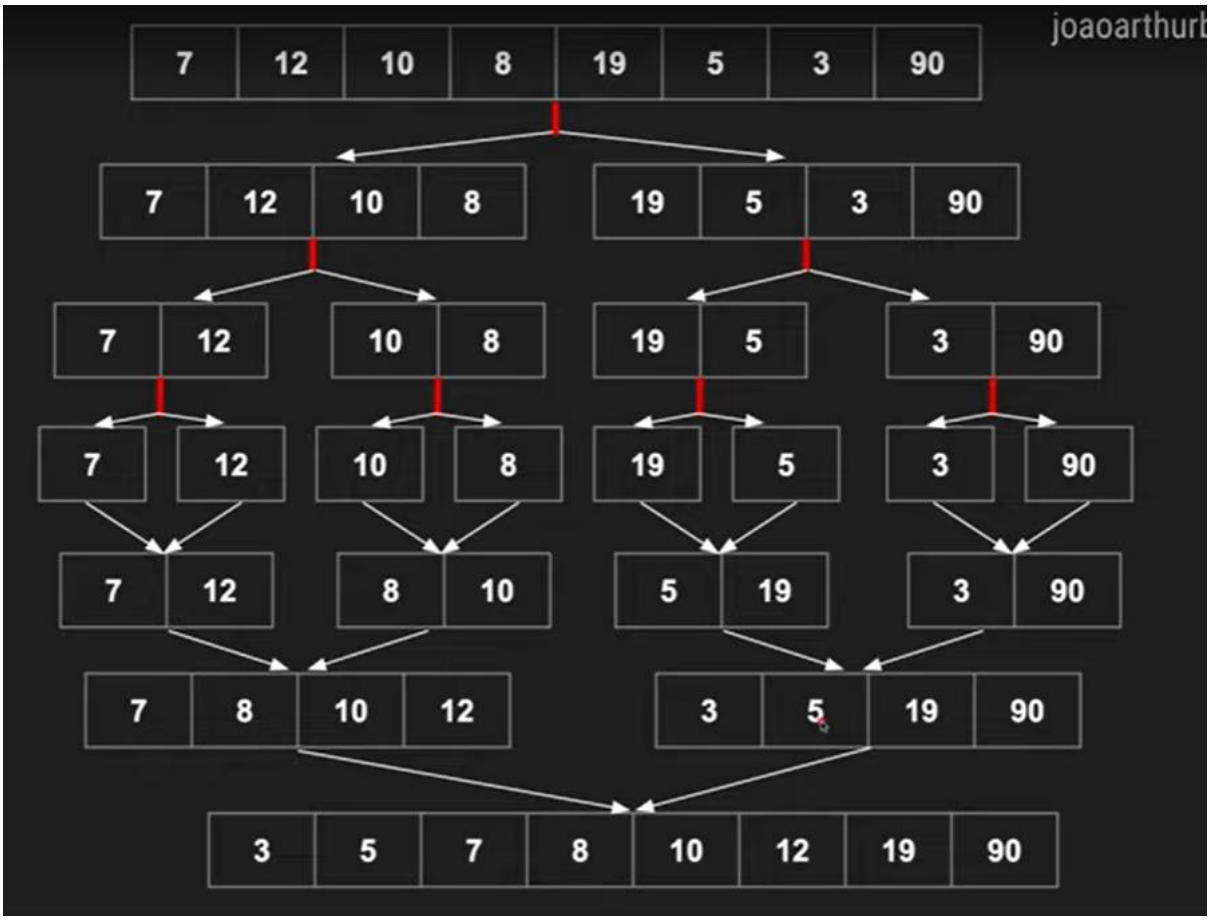
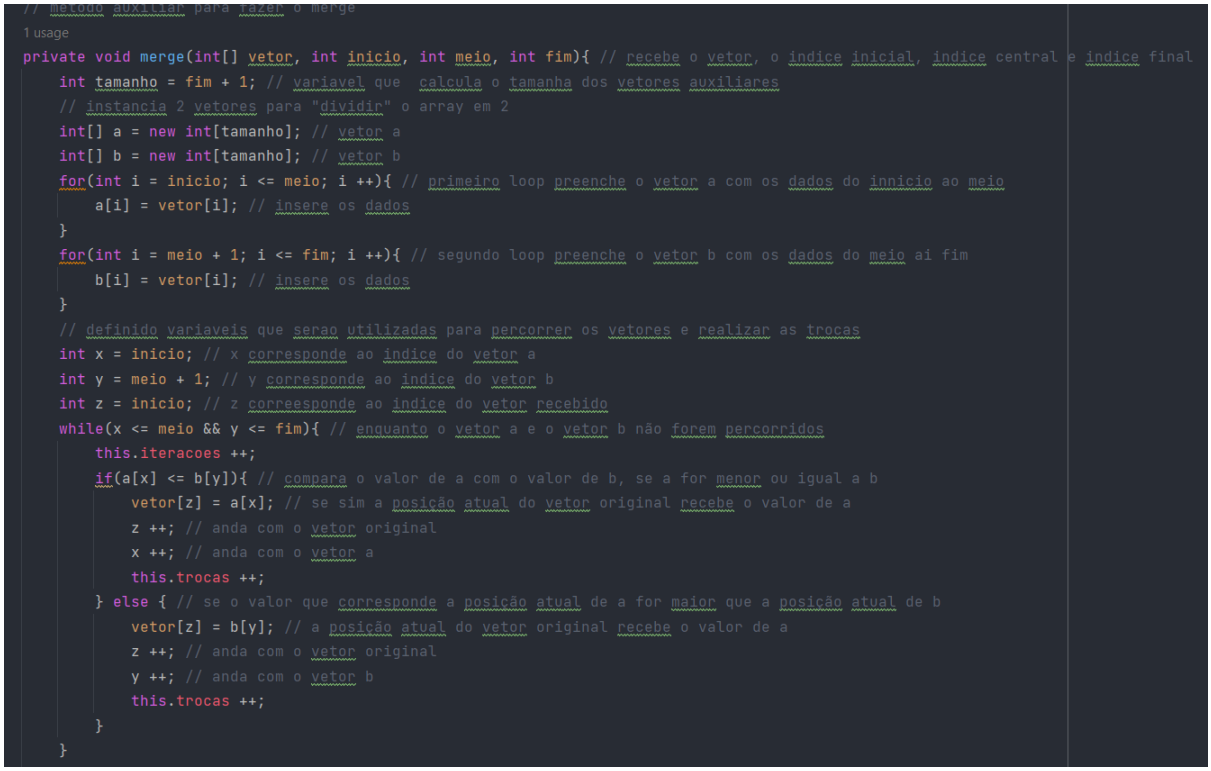


Figura 3. Representação do funcionamento do MergeSort

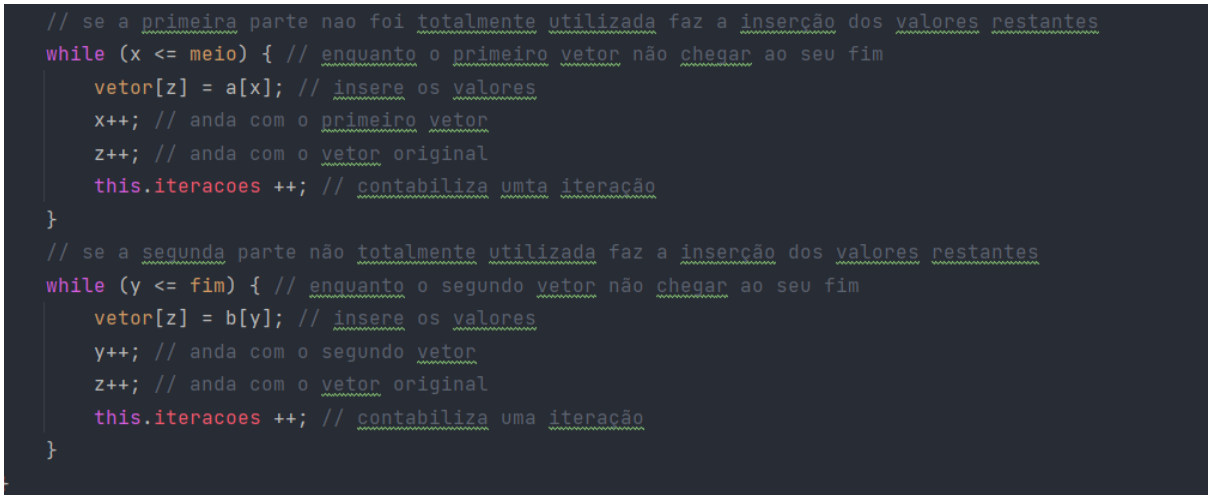
```java
// ordenação Merge Sort
3 usages
public void mergeSort(int[] vetor, int inicio, int fim){ // recebe o vetor, o indice inical e o indice final
    if(inicio >= fim){ // critério de parada da recursividade, se o indice inicial for maior ou igual ao indice final
        return; // a recursividade é interrompida
    }
    int meio = (int) ((inicio + fim) / 2); // variável que armazena o indice central do vetor, que se dá pelos indice inicial + final dividido por dois
    mergeSort(vetor, inicio, meio); // chamada recursiva para o vetor do inicio ao meio
    mergeSort(vetor,  inicio: meio + 1, fim); // chamada recursiva para o vetor do meio + 1 ao final
    merge(vetor, inicio, meio, fim); // metodo auxiliar que realiza o merge entre os vetores
}
```

Figura 4. Implementação do método mergeSort



```
// método auxiliar para fazer o merge
1 usage
private void merge(int[] vetor, int inicio, int meio, int fim){ // recebe o vetor, o indice inicial, indice central e indice final
    int tamanho = fim + 1; // variavel que  calcula o tamanha dos vetores auxiliares
    // instancia 2 vetores para "dividir" o array em 2
    int[] a = new int[tamanho]; // vetor a
    int[] b = new int[tamanho]; // vetor b
    for(int i = inicio; i <= meio; i ++){ // primeiro loop preenche o vetor a com os dados do innicio ao meio
        a[i] = vetor[i]; // insere os dados
    }
    for(int i = meio + 1; i <= fim; i ++){ // segundo loop preenche o vetor b com os dados do meio ai fim
        b[i] = vetor[i]; // insere os dados
    }
    // definido variaveis que serao utilizadas para percorrer os vetores e realizar as trocas
    int x = inicio; // x corresponde ao indice do vetor a
    int y = meio + 1; // y corresponde ao indice do vetor b
    int z = inicio; // z correesponde ao indice do vetor recebido
    while(x <= meio && y <= fim){ // enquanto o vetor a e o vetor b não forem percorridos
        this.iteracoes ++;
        if(a[x] <= b[y]){ // compara o valor de a com o valor de b, se a for menor ou igual a b
            vetor[z] = a[x]; // se sim a posição atual do vetor original recebe o valor de a
            z ++; // anda com o vetor original
            x ++; // anda com o vetor a
            this.trocas ++;
        } else { // se o valor que corresponde a posição atual de a for maior que a posição atual de b
            vetor[z] = b[y]; // a posição atual do vetor original recebe o valor de a
            z ++; // anda com o vetor original
            y ++; // anda com o vetor b
            this.trocas ++;
        }
    }
```

Figura 5. Implementação do método auxiliar merge



```
    // se a primeira parte nao foi totalmente utilizada faz a inserção dos valores restantes
    while (x <= meio) { // enquanto o primeiro vetor não chegar ao seu fim
        vetor[z] = a[x]; // insere os valores
        x++; // anda com o primeiro vetor
        z++; // anda com o vetor original
        this.iteracoes ++; // contabiliza umta iteração
    }
    // se a segunda parte não totalmente utilizada faz a inserção dos valores restantes
    while (y <= fim) { // enquanto o segundo vetor não chegar ao seu fim
        vetor[z] = b[y]; // insere os valores
        y++; // anda com o segundo vetor
        z++; // anda com o vetor original
        this.iteracoes ++; // contabiliza uma iteração
    }
```

Figura 6. Continuação da implementação do método auxiliar merge

O método mergeSort recebe o conjunto de elementos, seu início e seu fim, após isso é calculado o meio do vetor e são feitas chamadas recursivas para fazer o mergeSort do início ao meio do conjunto e do meio ao fim do conjunto, particionando o vetor até que o critério de parada seja atingido. O método merge por sua vez é responsável pela ordenação em si, após receber o conjunto, o mesmo é dividido em dois vetores auxiliares, onde basicamente os dois são percorridos do início ao fim e a cada posição o elemento do vetor a é comparado com o vetor b, o que tiver o menor valor será jogado em um terceiro vetor e ao fim o conjunto de dados recebido estará em ordem crescente

2.3. QuickSort

O método QuickSort consiste na ordenação de elementos em ordem crescente por meio do rearranjo dos elementos e isso ocorre por meio da escolha de um determinado elemento do conjunto, denominado como pivô, com o pivô definido o conjunto de elementos e dividido em dois grupos, o de elementos maiores que o pivô e o de elementos menores que o pivô.

```java
// ordenação Quick Sort
3 usages
public void quickSort(int[] vetor, int inicio, int fim){ // recebe o vetor, seu indice inicial e seu indice final
    if(inicio >= fim){ // critério de parada da chamada recursiva, se o indice inicial for maior ou igual ao indice final
        return; // interrompe a recursividade
    }
    int pivo = particiona(vetor, inicio, fim); // variável que recebe o indice do pivô por meio do método particiona
    quickSort(vetor, inicio,  fim: pivo - 1); // chamada recursiva para o vetor do inicio ao anterior ao pivo (esquerda do vetor)
    quickSort(vetor,  inicio: pivo + 1, fim); // chamada recursiva para o vetor a partindo do pivô até o fim (direita do vetor)
}
```

Figura 7. Implementação do método quickSort

```java
// metodo auxiliar para encontrar a posicao do pivo
1 usage
private int particiona(int[] vetor, int inicio, int fim){ // recebe o vetor, indice inicial e indice final
    int a = vetor[inicio]; // atributo a que recebe a primeira posição do vetor recebido
    int i = inicio + 1; // atributo que recebe a posição do vetor que irá crescer
    int j = fim; // atributo que recebe a posição do vetor que irá decrescer
    while (i <= j) { // enquanto o valor de 1 for menor ou igual ao valor de j
        while (i <= j && vetor[i] <= a) { // enquanto o valor da posição i do vetor for menor ou igual ao valor de a
            i++; // o i irá crescer
            this.iteracoes ++; // contabiliza uma iteração
        }
        while (i <= j && vetor[j] > a) { // enquanto o valor da posição j do vetor for maior que o valor de a
            j--; // o j irá decrescer
            this.iteracoes ++; // contabiliza uma iteração
        }
        if (i < j) { // ao final dos laços, se o valor de i for menor que o valor de j os valores correspondentes a essas posições serão trocados no vetor
            int aux = vetor[i]; // atributo auxiliar recebe o valor da posição i
            vetor[i] = vetor[j]; // posição i recebe o valor da posição j
            vetor[j] = aux; // posição j recebe o valor do atributo auxiliar (que corresponde ao valor da posição i)
            this.trocas ++; // contabiliza uma troca
        }
    }
    // ao final do primeiro loop a posição inicial do vetor recebe o valor da posição j que corresponde ao pivo e j recebe o valor que estava no inicio
    int aux = vetor[inicio]; // atributo auxiliar recebe o valor da posição inicial do vetor
    vetor[inicio] = vetor[j]; // posição inicial  recebe o valor da posição j
    vetor[j] = aux; // posição j recebe o valor da auxiliar (que corresponde ao valor da posição inicial do vetor)
    this.trocas ++; // contabiliza uma troca
    return j; // retorna a posição do pivo para o atributo pivo no metodo q qickSort
}
```

Figura 8. Implementação do método auxiliar particiona

Na implementação, o método quickSort recebe o grupo de elementos, o início e o fim após isso, é definido o índice do pivô, que é retornado pelo método particiona, o método particiona recebe o vetor com os elementos e pega sua posição inicial que será o pivô e com mais dois atributos auxiliares o vetor é percorrido de ambos os lados, no loop um índice vai crescendo a partir do início e o outro vai decrescendo a partir do fim, isso enquanto estão satisfazendo aa condição baseada no pivô, se ao fim dessa etapa o índice i for menor que o índice j será realizada a troca de posições e assim sucessivamente, ao final do loop principal, a o pivô inicial é trocada com a posição j e o índice de j é retornado para o método quickSort, para que dessa forma sejam feitas as chamadas recursivas para os subvetores até que o conjunto de elementos esteja ordenado.

```
// metodos get
3 usages
public int getIteracoes() { return iteracoes; }
3 usages
public int getTrocas() { return trocas; }
```

Figura 9. Implementação dos métodos get

Os métodos get foram implementados para que ao final de cada ordenação os números de iterações e trocas realizadas no processo possa ser consultado no terminal para o feedback dos testes.

3. Execução dos Testes

A execução dos testes será feita da seguinte forma, primeiro será instanciado 3 objetos do tipo Ordenacao, uma para cada método, em seguida, é solicitado o tamanho dos vetores para o testador.

```
// criando os objetospara cada ordenação
Ordenacao ordenarBubbleSort = new Ordenacao();
Ordenacao ordenarMergeSort = new Ordenacao();
Ordenacao ordenarQuickSort = new Ordenacao();
// criando objeto para gerar números aleatórios
Random random = new Random();
// definindo o tamanho dos vetores
Scanner scanner = new Scanner(System.in);
System.out.println("=====================================");
System.out.println("     Insira o tamanho dos vetores     ");
System.out.println("=====================================");
System.out.println(" ");
System.out.print("Digite um número: ");
int tamanho = scanner.nextInt();
scanner.close();
```

Figura 10. Menu para determinar o tamanho dos vetores

Em seguida, serão criados 4 vetores, onde 3 deles serão utilizados cada um para um método de ordenação e um quarto vetor chamado vetorReset que será populado com números aleatórios de 1 a 100, o vetor reset vai armazenar os valores desordenados e será responsável por possibilitar que os métodos de ordenação sejam executados ao menos 5 vezes, pois ao fim de cada ordenação o vetor ordenado será igualado ao vetorReset, tendo seus valores desordenados novamente.

```java
// criando os vetores e armazenando os valores de acordo com o tamanho recebido
int[] vetor1 = new int[tamanho];
int[] vetor2 = new int[tamanho];
int[] vetor3 = new int[tamanho];
int[] vetorReset = new int[tamanho];
int valor;
for(int i = 0; i < tamanho; i ++){
    valor = random.nextInt( bound: 100) + 1;
    vetorReset[i] = valor;
}
```

Figura 11. Criando vetores para teste

```java
// ordenando BubbleSort
inicioOrdenacao = System.nanoTime();
while(contador < 5){
    vetor1 = vetorReset;
    ordenarBubbleSort.bubbleSort(vetor1, tamanho);
    contador ++;
}
fimOrdenacao = System.nanoTime();
long tempoBubbleSort = (fimOrdenacao - inicioOrdenacao) / 5;
contador = 0;

// printando resultados no terminal
System.out.println("=====   BUBBLE SORT   =====");
System.out.println(" ");
System.out.println("Vetor ordenado com BubbleSort:");
for(int i = 0; i < tamanho; i ++){
    if(i == tamanho - 1){
        System.out.println(vetor1[i]);
    } else {
        System.out.print(vetor1[i] + " -> ");
    }
}
System.out.println(" ");
System.out.println("Média de tempo para realizar ordenação: " + tempoBubbleSort + "ns");
System.out.println("Iterações realizadas por rodada: " + (ordenarBubbleSort.getIteracoes() / 5));
System.out.println("Trocas realizadas por rodada: " + (ordenarBubbleSort.getTrocas() / 5));
System.out.println("\n");
```

Figura 12. Execução do teste BubbleSort

```java
//ordenando MergeSort
inicioOrdenacao = System.nanoTime();
while(contador < 5){
    vetor2 = vetorReset;
    ordenarMergeSort.mergeSort(vetor2, inicio: 0, fim: tamanho - 1);
    contador ++;
}
fimOrdenacao = System.nanoTime();
long tempoMergeSort = (fimOrdenacao - inicioOrdenacao) / 5;
contador = 0;

// printando resultados no terminal
System.out.println("=====   MERGE SORT   =====");
System.out.println(" ");
System.out.println("Vetor ordenado com MergeSort:");
for(int i = 0; i < tamanho; i ++){
    if(i == tamanho - 1){
        System.out.println(vetor2[i]);
    } else {
        System.out.print(vetor2[i] + " -> ");
    }
}
System.out.println(" ");
System.out.println("Média de tempo para realizar ordenação: " + tempoMergeSort + "ns");
System.out.println("Iterações realizadas por rodada: " + (ordenarMergeSort.getIteracoes() / 5));
System.out.println("Trocas realizadas por rodada: " + (ordenarMergeSort.getTrocas() / 5));
System.out.println("\n");
```

Figura 13. Execução do teste MergeSort

```java
// odenando QuickSort
inicioOrdenacao = System.nanoTime();
while(contador < 5){
    vetor3 = vetorReset;
    ordenarQuickSort.quickSort(vetor3, inicio: 0, fim: tamanho - 1);
    contador ++;
}
fimOrdenacao = System.nanoTime();
long tempoQuickSort = (fimOrdenacao - inicioOrdenacao) / 5;
contador = 0;

// printando resultados no terminal
System.out.println("=====   QUICK SORT   =====");
System.out.println(" ");
System.out.println("Vetor ordenado com QuickSort:");
for(int i = 0; i < tamanho; i ++){
    if(i == tamanho - 1){
        System.out.println(vetor3[i]);
    } else {
        System.out.print(vetor3[i] + " -> ");
    }
}
System.out.println(" ");
System.out.println("Média de tempo para realizar ordenação: " + tempoQuickSort + "ns");
System.out.println("Iterações realizadas por rodada: " + (ordenarQuickSort.getIteracoes() / 5));
System.out.println("Trocas realizadas por rodada: " + (ordenarQuickSort.getTrocas() / 5));
System.out.println("\n");
```

Figura 14. Execução do teste QuickSort

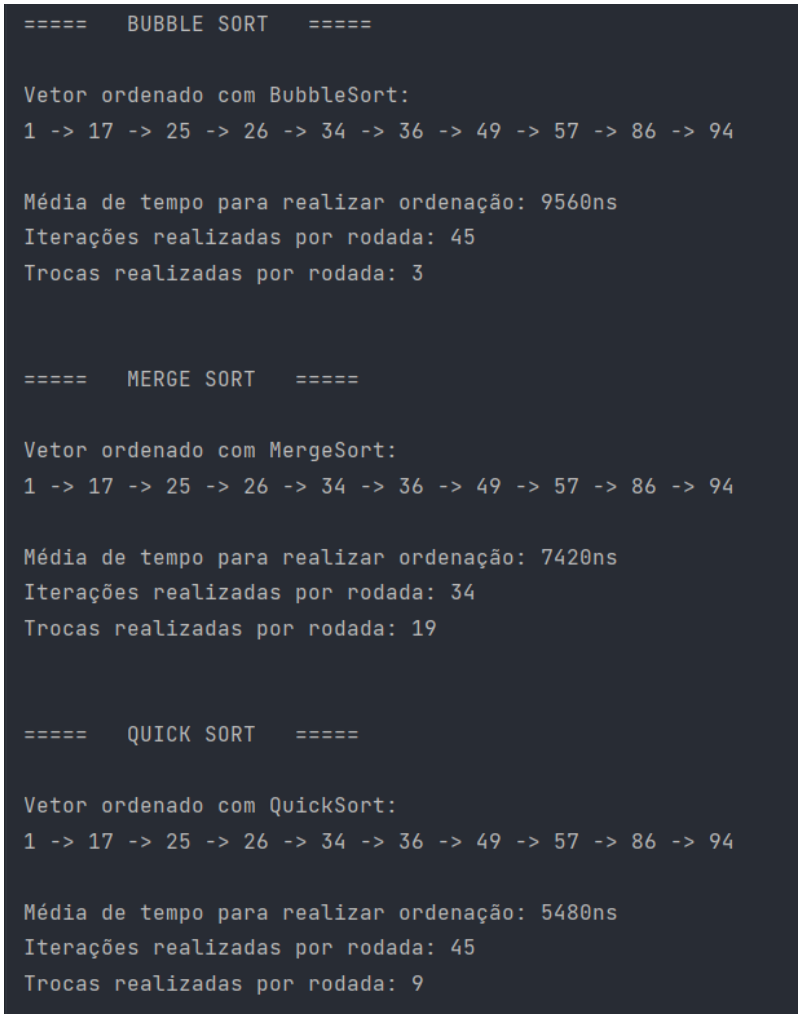


Figura 15. Resultado de execução com conjuntos de 10 elementos

Os testes funcionarão da seguinte forma, com o vetor reset criado os vetores 1, 2 e 3 serão igualados ao vetor reset e ordenados 5 vezes por cada método de ordenação implementado, o processo de execução será cronometrado em nanossegundos e será feita uma média de tempo decorrido para cada ordenação, após isso, os resultados daquele método serão exibidos no terminal, tanto a média de tempo para execução quanto o número de iterações de trocas realizadas no processo. Para a análise, será feita a ordenação utilizando cada método com conjuntos de elementos de 50, 500, 1000, 5000 e 10000 elementos, os dados de cada execução serão coletados e armazenados em tabelas para que a análise possa ser feita.

4. Resultados

BUBBLESORT			
N° ELEMENTOS	TEMPO (NS)	N° ITERAÇÕES	N° TROCAS
50	67100	1225	104
500	913760	124750	12424
1000	3138340	499500	49236
5000	12585820	12497500	1234909
10000	56852080	49995000	4937987

Figura 16. Dados coletados com BubbleSort

MERGESORT			
N° ELEMENTOS	TEMPO (NS)	N° ITERAÇÕES	N° TROCAS
50	96360	286	153
500	1164180	4488	2272
1000	5510220	9976	5044
5000	52850640	61808	32004
10000	190369640	133616	69008

Figura 17. Dados coletados com MergeSort

QUICKSORT			
N° ELEMENTOS	TEMPO (NS)	N° ITERAÇÕES	N° TROCAS
50	78420	1095	44
500	279440	24887	404
1000	464000	53302	900
5000	1914360	373094	4900
10000	4754900	992315	9900

Figura 18. Dados coletados com QuickSort

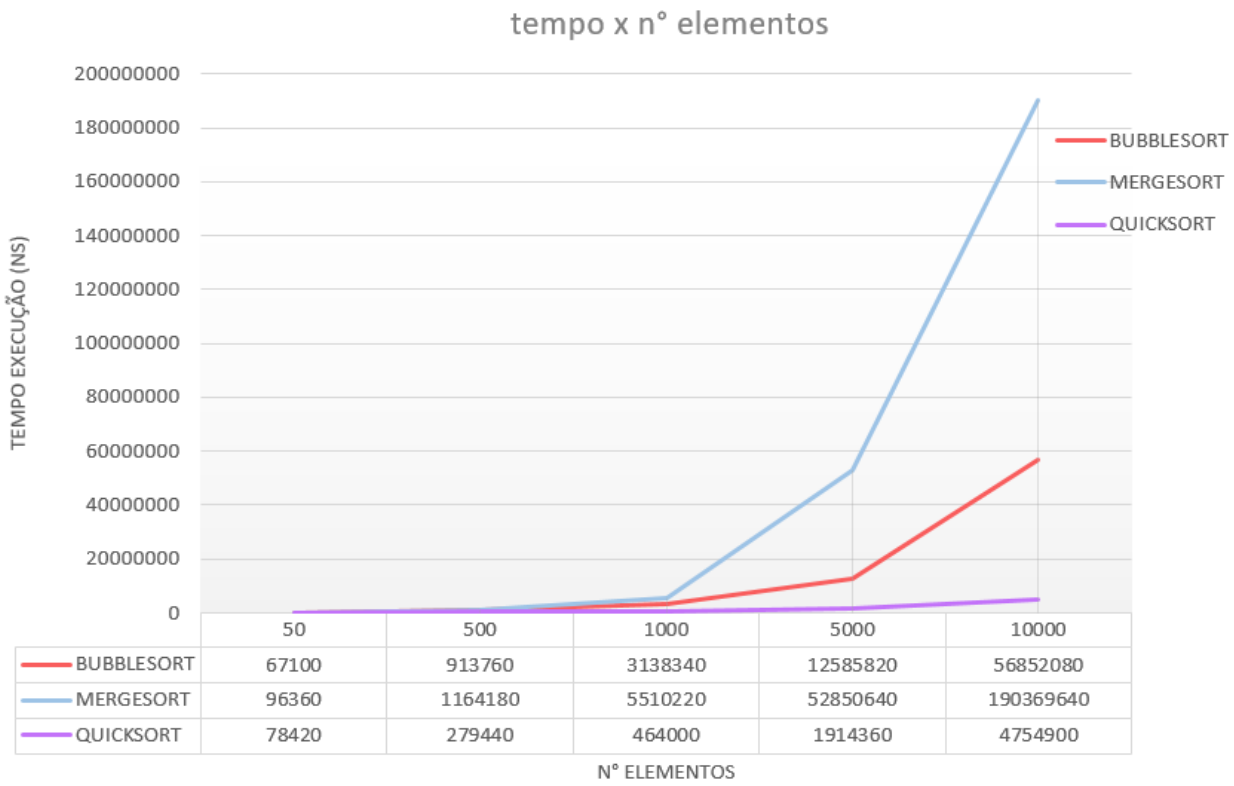


Figura 19. Gráfico tempo de execução em função do número de elementos

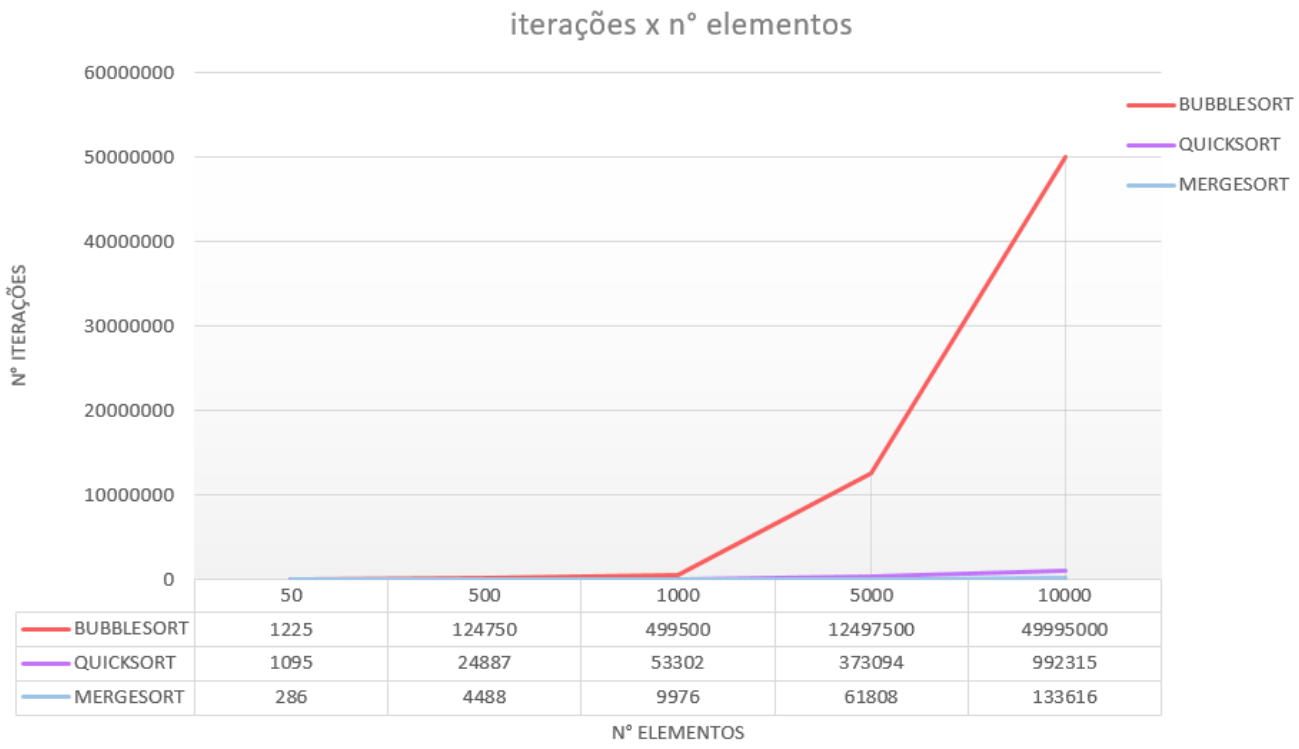


Figura 20. Gráfico número de iterações em função do número de elementos

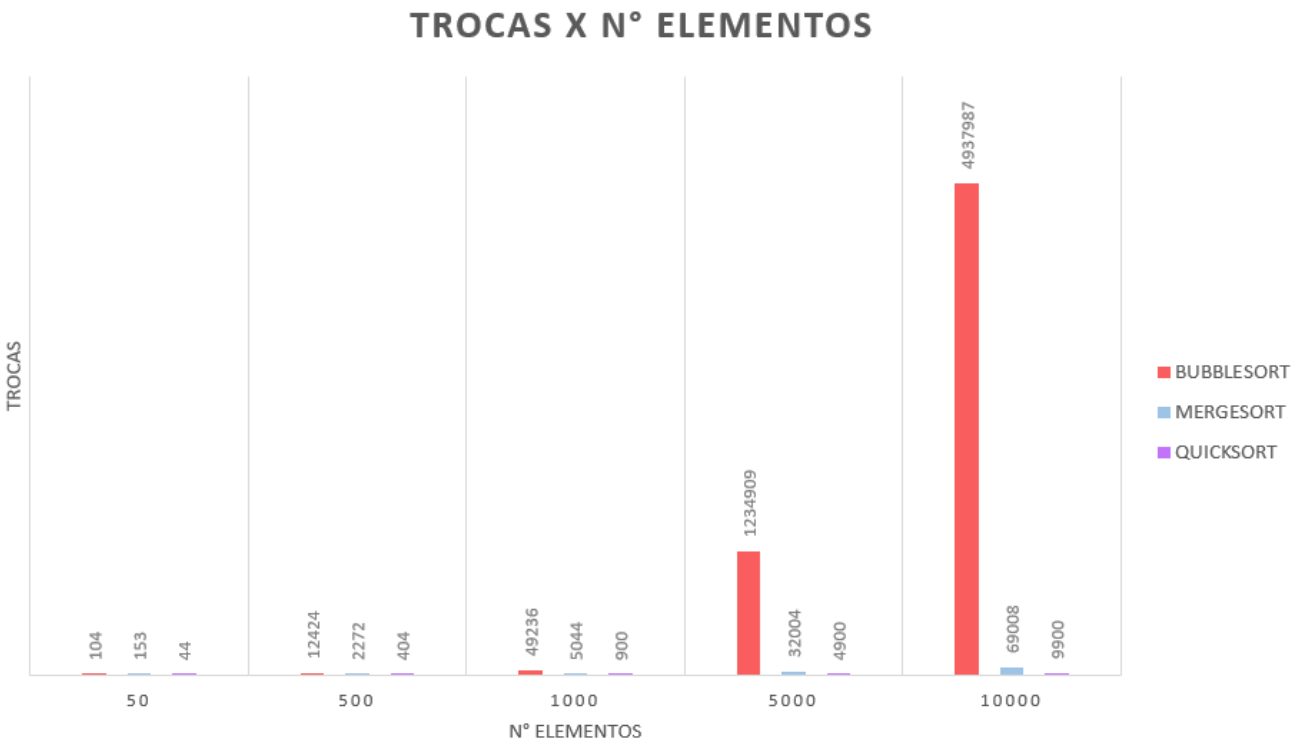


Figura 21. Gráfico número de trocas em função do número de elementos

5. Conclusão

Primeiramente, vale ressaltar que os resultados coletados são baseados nos testes realizados nesta implementação.
Baseado nos resultados obtidos é possível concluir que sem dúvidas o método de ordenação com o melhor desempenho, o primeiro indício disso é observável no gráfico de tempo de execução em função do número de elementos, dessa forma é possível determinar a notação big O de cada método por meio de seu comportamento no gráfico. Dessa forma é possível observar que o método QuickSort possui notação O(n log n) devido ao seu comportamento que se mantém quase constante conforme os elementos aumentam, ou seja, o QuickSort teve o seu melhor desempenho possível. Em contrapartida, o método MergeSort teve o pior desempenho possível em questão de tempo de execução, isso se deve a forma como o método foi implementado que poderia ser feito de forma mais otimizada. Porém em questão de número de número de iterações e número de trocas os resultados estão coerentes com o método. O método BubbleSort teve um desempenho intermediário comparado com os outros 2 métodos na questão de tempo de execução, apresentando assim um comportamento O(n^2). Da mesma forma que na análise de tempo de execução, os dados de iterações e trocas realizadas em cada operação estão coerentes com os resultados, onde o BubbleSort apresenta uma quantidade de iterações e trocas realizadas bem maior em comparação aos demais. Nesse sentido o QuickSort apresentou os melhores resultados.

Algoritmo	Tempo		
	Melhor	Médio	Pior
Merge sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$

Figura 22. Tabela Big O Notation para cada método implementado

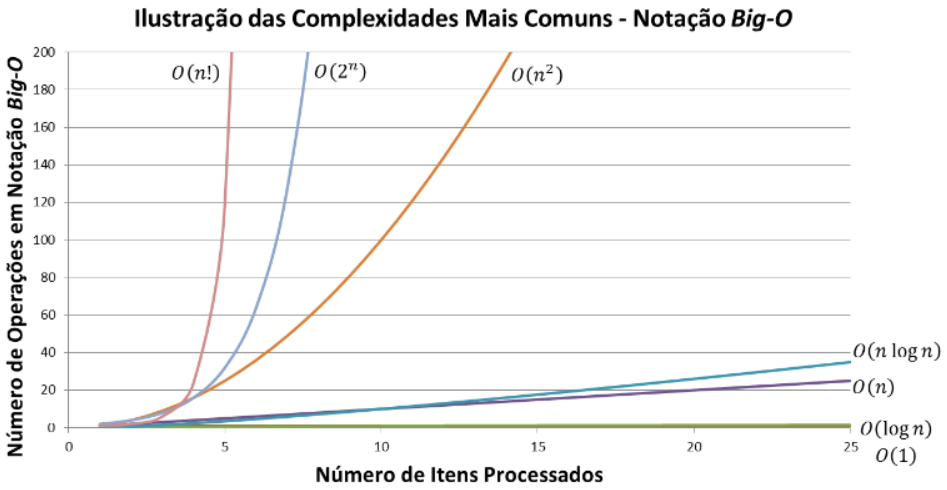


Figura 23. Gráfico de comportamento Big O Notation

6. Referências

João Arthur B. M. "Merge Sort - Algoritmo de Ordenação."
Disponível em: https://joaoarthurbm.github.io/eda/posts/merge-sort/

Wikipedia. "Quicksort."
Disponível em: https://pt.wikipedia.org/wiki/Quicksort

Stack Overflow em Português. "Definição da Notação Big O."
Disponível em: https://pt.stackoverflow.com/questions/56836/defini%C3%A7%C3%A3o-da-nota%C3%A7%C3%A3o-big-o

Medium. "O que é Big O Notation."
Disponível em: https://medium.com/linkapi-solutions/o-que-%C3%A9-big-o-notation-32f171e4a045

Wikipedia. "Merge Sort."
Disponível em: https://pt.wikipedia.org/wiki/Merge_sort

Wikipedia. "Bubble Sort."
Disponível em: https://pt.wikipedia.org/wiki/Bubble_sort

Materiais auxiliares disponibilizados pelo professor no Canvas.