TDE03 - ORDENAÇÕES
Análise de Desempenho

Gabriel Felipe Jess Meira

[1]Engenharia de Software - Pontíficia Universidade Católica do Paraná (PUCPR)

meira.gabriel@tecpuc.com.br

Abstract. *This report aims to provide data regarding performance of ordinations in vectors with different sizes, using the same data. In order to discover the best method for sorting vectors.*
The sorting methods that were chosen to carry out this TDE were: BubbleSort, Merge Sort and Quick Sort.

Resumo. *Este relatório visa disponibilizar dados referentes ao desempenho de ordenações em vetores com diversos tamanhos diferentes, utilizando os mesmos dados. Afim de descobrir qual o melhor método para ordenação de vetores.*
Os métodos de ordenação que foram escolhidos para a realização deste TDE, foram: BubbleSort, Merge Sort e Quick Sort.

1. Ordenações

A utilização de vetores ordenados para buscas em conjuntos de dados desempenha um papel fundamental na eficiência e na organização de algoritmos de busca. Com a ordem estabelecida, algoritmos como a busca binária tornam-se altamente eficientes, reduzindo drasticamente o tempo de busca, especialmente em conjuntos de dados extensos. A facilidade de implementação, embora possa exigir algoritmos mais complexos para inserção e remoção de elementos, é notável, simplificando a manutenção contínua dos dados.

Além disso, a capacidade de manter a ordem dos dados ao inserir ou remover elementos é crucial, garantindo a consistência dos conjuntos de dados ao longo do tempo. Em uma ampla gama de cenários, a utilização de vetores ordenados é essencial para sistemas de gerenciamento de bancos de dados e algoritmos de processamento de grandes conjuntos de dados, assegurando a responsividade e o desempenho adequado. A simplicidade dos algoritmos de busca em vetores ordenados também reduz a complexidade geral do código, contribuindo para uma melhor compreensão e manutenção a longo prazo.

2. Código

O látex desformatou o código, por isso será possível acessar o código pelo seguinte link no github: https://github.com/gabrielfjmeira/TDE03Ordenacao.

Também há prints das implementações neste relatório.

3. Ordenação - BubbleSort

O Bubble Sort é um algoritmo de classificação simples e fundamental, usado para ordenar uma lista de elementos. Neste método, o algoritmo percorre a lista várias vezes,

comparando elementos adjacentes e trocando-os se estiverem na ordem errada. Durante cada iteração, o algoritmo movimenta o maior elemento não classificado para o final da lista, levando gradualmente à formação da lista ordenada. No entanto, devido à sua natureza de comparar e trocar elementos adjacentes repetidamente, o Bubble Sort pode ser ineficiente em grandes conjuntos de dados, resultando em uma complexidade de tempo no pior caso de $O(n^2)$.

Embora seja fácil de entender e implementar, o Bubble Sort é geralmente evitado em conjuntos de dados extensos devido à sua performance mais lenta em comparação com algoritmos de classificação mais eficientes, como o Merge Sort e o Quick Sort.

4. Implementação - BubbleSort

```
//Função que ordena o vetor pelo método de bubble sort
1 usage
public static void bubbleSort(){
    //Definição de variáveis
    int vetor[] = Main.vetorBubbleSort;
    int tamanhoVetor = Main.tamanho;
    int aux = 0;

    //Execução do bubblesort
    for (int i = 0; i < tamanhoVetor; i++){
        //Incrementa a iteração
        Main.iteracoesBS ++;
        for (int j = 1; j < tamanhoVetor; j++){
            //Incrementa a iteração
            Main.iteracoesBS ++;
            if (vetor[j-1] > vetor[j]){
                //Troca os elementos de posição
                aux = vetor[j-1];
                vetor[j-1] = vetor[j];
                vetor[j] = aux;
                //Incrementa a troca
                Main.trocasBS ++;
            }
        }
    }
}
```

5. Ordenação - Merge Sort

O Merge Sort é um algoritmo de classificação eficiente e de divisão, usado para ordenar listas ou vetores. Este método funciona dividindo repetidamente a lista não ordenada em metades menores, até que restem sublistas de um único elemento. Em seguida, ele mescla essas sublistas de volta em ordem crescente, combinando-as de forma ordenada.

A eficiência do Merge Sort decorre de sua abordagem recursiva, que garante a ordenação confiável de grandes conjuntos de dados com uma complexidade de tempo de O(n log n), onde n é o número de elementos na lista. Embora a implementação do Merge Sort seja mais complexa do que a de algoritmos de classificação mais simples, sua eficiência em grandes conjuntos de dados faz dele uma escolha popular em muitas aplicações de programação e ciência da computação.

6. Implementação - Merge Sort

```java
//Função que ordena o vetor pelo método de merge sort
3 usages
public static void mergeSort(int []vetor, int []auxiliar, int inicio, int fim){
    //Verifica se foi alcaçado o limite de divisão
    if (inicio < fim){
        //Incrementa a iteração
        Main.iteracoesMS ++;
        //Define o meio
        int meio = (int) (inicio + fim) / 2;
        //Chama o merge sort da primeira metade
        mergeSort(vetor, auxiliar, inicio, meio);
        //Chama o merge sort da segunda metade
        mergeSort(vetor, auxiliar, inicio: meio+1, fim);
        //Combina as metades
        merge(vetor, auxiliar, inicio, meio, fim);
    }
}
```

```java
//Função que une as metades do vetor
1 usage
private static void merge(int []vetor, int []auxiliar, int inicio, int meio, int fim){
    //Popula vetor auxiliar
    for (int i = inicio; i <= fim; i++){
        auxiliar[i] = vetor[i];
    }

    //Indice da esquerda
    int esquerda = inicio;

    //Indice da direita
    int direita = meio + 1;

    //Percorre o vetor para comparar e ordenar
    for (int i = inicio; i <= fim; i++){

        //Incrementa a iteração
        Main.iteracoesMS++;

        //Se o meu vetor da esquerda ultrapassar o meio, significa que o mesmo acabou, logo só tem elementos a direita para serem ordenados
        if (esquerda > meio) {
            vetor[i] = auxiliar[direita++];
            //Incrementa a troca
            Main.trocasMS++;
        //Se o meu vetor da direita ultrapassar o fim, significa que o mesmo acabou, logo só tem elementos a esquerda para serem ordenados
        }else if(direita > fim){
            vetor[i] = auxiliar[esquerda++];
            //Incrementa a troca
            Main.trocasMS++;
        //Se o elemeto da esquerda for menor q o da direita, o vetor principal recebe o auxiliar da esquerda
        }else if (auxiliar[esquerda] < auxiliar[direita]){
            vetor[i] = auxiliar[esquerda++];
            //Incrementa a troca
            Main.trocasMS++;
        //Se o elemeto da direita for menor q o da esquerda, o vetor principal recebe o auxiliar da direita
        }else {
            vetor[i] = auxiliar[direita++];
            //Incrementa a troca
            Main.trocasMS++;
        }
    }
}
```

7. Ordenação - Quick Sort

O Quick Sort é um algoritmo de classificação eficiente e amplamente utilizado que se baseia no conceito de divisão e conquista. Ele opera selecionando um elemento como pivô e particionando a lista original em duas sub-listas, uma contendo elementos menores que o pivô e outra contendo elementos maiores. Em seguida, o algoritmo ordena recursivamente essas sub-listas.

A eficiência do Quick Sort provém de sua rápida velocidade de execução em conjuntos de dados grandes, com uma complexidade de tempo médio de O(n log n), mas em seu pior caso pode chegar a $O(n^2)$.

Apesar da possibilidade de comportamento de pior caso, o Quick Sort é valorizado pela sua eficiência em geral e é amplamente aplicado em uma variedade de cenários de programação e engenharia de software.

8. Implementação - Quick Sort

```
//Função que ordena o vetor pelo método de quick sort
3 usages
public static void quickSort(int []vetor, int esquerda, int direita){
    //Enquanto os elementos estiverem desordenados
    if (esquerda < direita){

        //Incrementa a iteração
        Main.iteracoesQS++;

        //Obtém o pivô a partir do método da partição
        int pivo = particao(vetor, esquerda, direita);

        //Ordena a esquerda do pivô
        quickSort(vetor, esquerda, direita: pivo-1);

        //Ordena a direita do pivô
        quickSort(vetor, esquerda: pivo+1, direita);
    }
}
```

```java
//Método que auxilia na execução do quick sort
1 usage
private static int particao(int []vetor, int esquerda, int direita){
    //Definição de variáveis para auxiliar a execução
    int i = esquerda + 1;
    int j = direita;

    //Definição do pivô
    int pivo = vetor[esquerda];

    //Reposicionamento de elementos usando quick sort
    while (i <= j){

        //Incrementa a iteração
        Main.iteracoesQS++;

        //Busca elemento maior que o pivô
        if (vetor[i] <= pivo){
            i++;
        //Busca elemento menor que o pivô
        }else if(vetor[j] > pivo){
            j--;
        //Verifica se o i é menor ou igual ao j
        }else if(i <= j){
            //Realiza a troca de posições
            int aux = vetor[i];
            vetor[i] = vetor[j];
            vetor[j] = aux;
            i++;
            j--;
            //Incrementa a troca
            Main.trocasQS++;
        }
    }

    //Realizo a troca do pivô pois o elemento i trocou de posição com o elemento j
    int aux = vetor[esquerda];
    vetor[esquerda] = vetor[j];
    vetor[j] = aux;
    //Incrementa a troca
    Main.trocasQS++;
    return j;
}
```

9. Desempenho Durante a Ordenação - Coleta de Dados

O desempenho foi medido executando a ordenação dos vetores com 5 tamanhos diferentes e com 5 seeds diferentes, assim, foi possível coletar o tempo da execução da ordenação, o número de iterações e o número de trocas.

Tabelas de coletas de dados:

	ORDENAÇÃO DOS VETORES				
	NÚMERO DE ELEMENTOS				
	50	500	1.000	5.000	10.000
BubbleSort	83	4.106	5.827	48.620	180.713
	88	5.388	6.251	43.052	176.185
	156	5.392	6.162	43.471	175.313
	88	4.341	7.293	43.477	178.849
	87	4.731	7.304	43.548	174.966
Merge Sort	36	347	769	1.524	2.794
	38	369	671	1.651	2.169
	37	380	741	1.550	2.831
	36	354	843	1.195	2.021
	35	362	715	1.354	2.285
Quick Sort	22	187	449	1.060	2.043
	20	192	404	1.108	2.106
	20	191	415	983	1.873
	20	235	532	959	1.679
	28	245	504	1.234	1.551
	TEMPO (µs)				

	ORDENAÇÃO DOS VETORES				
	NÚMERO DE ELEMENTOS				
	50	500	1.000	5.000	10.000
BubbleSort	2.500	250.000	1.000.000	25.000.000	100.000.000
	2.500	250.000	1.000.000	25.000.000	100.000.000
	2.500	250.000	1.000.000	25.000.000	100.000.000
	2.500	250.000	1.000.000	25.000.000	100.000.000
	2.500	250.000	1.000.000	25.000.000	100.000.000
Merge Sort	335	4987	10.975	66.807	143.615
	335	4.987	10.975	66.807	143.615
	335	4.987	10.975	66.807	143.615
	335	4.987	10.975	66.807	143.615
	335	4.987	10.975	66807	143.615
Quick Sort	289	4.265	10.621	66.170	129.135
	223	4.280	9.723	63.822	143.922
	233	4.568	9.546	58.752	130.664
	244	4.420	9.216	59.235	134.635
	268	3.986	9.066	59.836	139.081
	ITERAÇÕES				

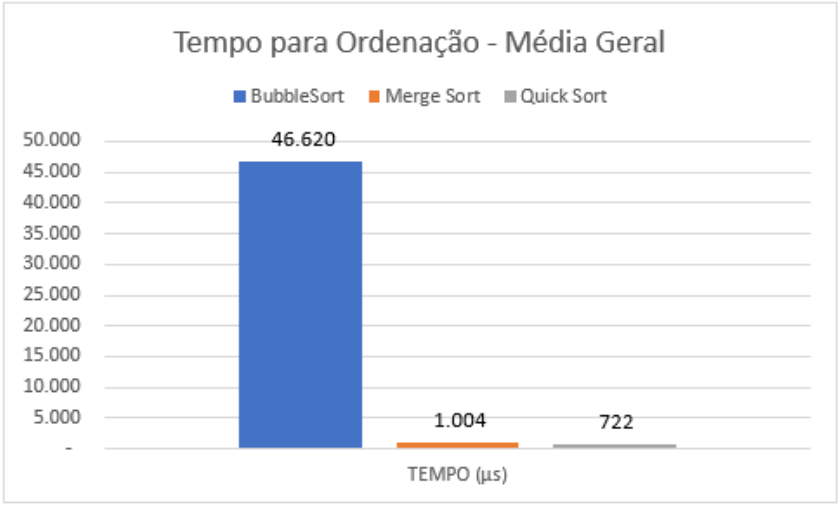
	ORDENAÇÃO DOS VETORES				
	NÚMERO DE ELEMENTOS				
	50	500	1.000	5.000	10.000
BubbleSort	483	60.108	254.207	6.145.432	24.751.955
	668	65.891	240.474	6.188.695	24.960.010
	591	59.728	256.867	6.323.365	25.010.696
	682	62.738	240.209	6.180.545	24.833.514
	628	61.606	255.388	6.137.809	25.097.200
Merge Sort	286	4.488	9.976	61.808	133.616
	286	4.488	9.976	61.808	133.616
	286	4.488	9.976	61.808	133.616
	286	4.488	9.976	61.808	133.616
	286	4.488	9.976	61.808	133.616
Quick Sort	66	1.084	2.398	14.350	31.632
	71	1.066	2.319	14.680	31.332
	75	1.066	2.405	14.763	31.443
	70	1.070	2.397	14.754	31.537
	71	1.086	2.413	14.619	31.309
	TROCAS				

10. Desempenho Durante a Ordenação - Conclusão

Para melhor visualização dos dados foi elaborada a média geral de cada critério coletado e gerado um gráfico do mesmo.

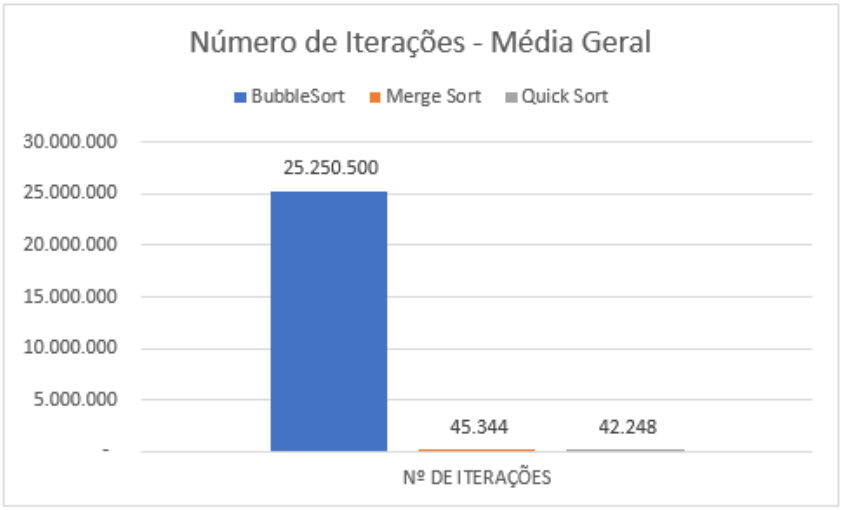
Refente ao tempo(µs):

	MÉDIA DE TEMPO NAS ORDENAÇÕES:						
	NÚMERO DE ELEMENTOS						
	50	500	1.000	5.000	10.000	MÉDIA GERAL	
BubbleSort	100	4.792	6.567	44.434	177.205	46.620	µs
Merge Sort	36	362	748	1.455	2.420	1.004	µs
Quick Sort	22	210	461	1.069	1.850	722	µs



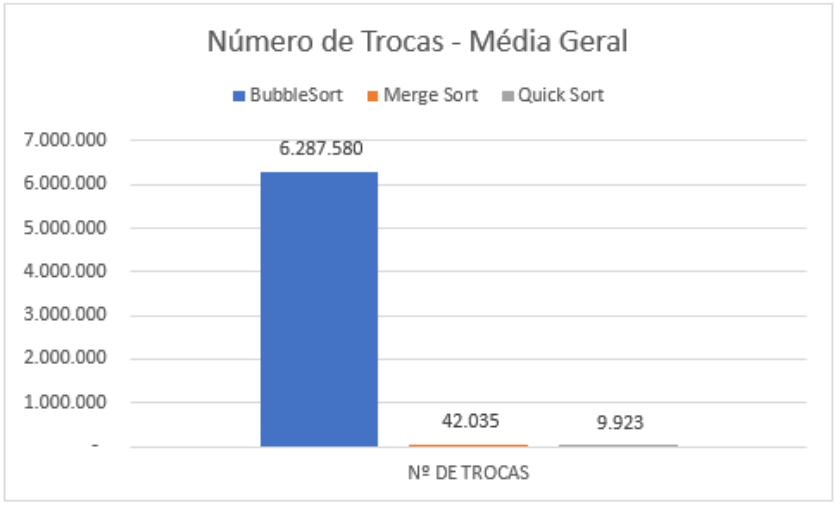
Refente a Iterações:

	MÉDIA DE ITERAÇÕES NAS ORDENAÇÕES:					
	NÚMERO DE ELEMENTOS					
	50	500	1.000	5.000	10.000	MÉDIA GERAL
BubbleSort	2.500	250.000	1.000.000	25.000.000	100.000.000	25.250.500
Merge Sort	335	4.987	10.975	66.807	143.615	45.344
Quick Sort	251	4.304	9.634	61.563	135.487	42.248



Refente a Trocas:

	MÉDIA DE TROCAS NAS ORDENAÇÕES:					
	NÚMERO DE ELEMENTOS					
	50	500	1.000	5.000	10.000	MÉDIA GERAL
BubbleSort	610	62.014	249.429	6.195.169	24.930.675	6.287.580
Merge Sort	286	4.488	9.976	61.808	133.616	42.035
Quick Sort	71	1.074	2.386	14.633	31.451	9.923



O algoritmo de ordenação Quick Sort supera o Merge Sort e o Bubble Sort devido à sua abordagem de divisão e conquista eficiente. Ao dividir a lista em subconjuntos menores, o Quick Sort minimiza a necessidade de comparações com iterações e trocas, reduzindo a complexidade para O(n log n).

Em contraste, o Merge Sort, embora também eficaz na ordenação, requer

operações adicionais de mesclagem de listas ordenadas, aumentando sua complexidade para O(n log n). Porém é menos eficiente que o quick sort devido a possuir mais iterações e trocas.

Enquanto isso, o Bubble Sort, embora seja fácil de implementar ele é menos eficiente devido à necessidade de inúmeras trocas e iterações, tornando-se mais lento por causa de sua complexidade quadrática O(n^2).

References

SEABRA, G. Bubble Sort: o que é e como usar? Exemplos práticos! Disponível em: ¡https://blog.betrybe.com/tecnologia/bubble-sort-tudo-sobre/¿. Acesso em: 3 nov. 2023.

Merge Sort - data structure and algorithms tutorials. Disponível em: ¡https://www.geeksforgeeks.org/merge-sort/¿. Acesso em: 3 nov. 2023.

DO CÓDIGO, C. Como fazer Merge Sort em Java - Canal do Código. Disponível em: ¡https://www.youtube.com/watch?v=yj8igr9DjeY¿. Acesso em: 3 nov. 2023.

QuickSort - data structure and algorithm tutorials. Disponível em: ¡https://www.geeksforgeeks.org/quick-sort/¿. Acesso em: 3 nov. 2023.

DO CÓDIGO, Canal. Quicksort simples e sem complicações - Canal do Código. Disponível em: ¡https://www.youtube.com/watch?v=PrR3nfq9wSYt=562s¿. Acesso em: 3 nov. 2023.

Materiais disponibilizados no Canvas da matéria de Resolução de Problemas Estruturados em Computação.