UNIVERSIDADE FEDERAL DE ITAJUBÁ

CIÊNCIA DA COMPUTAÇÃO

DESENVOLVIMENTO DE BENCHMARK PARA ALGORITMOS DE ORDENAÇÃO

Breno, João Vitor Ribeiro
Juliana, Lucas Luiz e Tiago Vilela

Trabalho da Disciplina
Algoritmos e Estrutura de Dados II

INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO
UNIFEI - Itajubá

2024

UNIVERSIDADE FEDERAL DE ITAJUBÁ
INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Desenvolvimento de Benchmark para algoritmos de ordenação

Breno, João Vitor Ribeiro, Juliana, Lucas Luiz e Tiago Vilela, ,

breno.hiroyuki@unifei.edu.br, d2021031942@unifei.edu.br, d2019002587@unifei.edu.br, d2022008662@unifei.edu.br, d2022000969@unifei.edu.br, ,

1 Introdução

Um algoritmo é um procedimento computacional definido, sendo a entrada um valor ou um conjunto de valores inserido a esse procedimento, e a saída o valor ou conjunto de dados retornável após as operações realizadas. Um algoritmo de ordenação é uma operação fundamental em ciência da computação e que se encontra uma grande variedade de tipos desses algoritmos (Cormen et al., 2022). O melhor algoritmo de ordenação a se utilizar depende do número de itens a serem inseridos, se ele já se encontra quase ordenado, ordenado, desordenado ou de forma aleatória.

Uma forma de analisar a eficiência dos algoritmos de ordenação, e que vai ser utilizado neste trabalho, é a avaliação analítica e que consiste em obter uma estimativa do esforço de computação em termos de uma taxa de crescimento do tempo de execução em função do tamanho da entrada Souza (2024a).

Sendo assim, este trabalho tem por objetivo comparar os algoritmos de ordenação: Insertion Sort, Selection Sort, MergeSort e QuickSort com o algoritmo que será o alvo de nosso estudo: Merge Sort In Place, por meio da avaliação analítica. Com a finalidade de descobrir se o algoritmo alvo é mais eficiente ou não e em quais determinadas situações dadas aos tipos de valores de entrada.

2 Conceitualização

Nessa seção, serão abordados os conceitos de algoritmos de ordenação destacados na introdução, envolvendo a lógica de cada um, assim como a implementação deles.

2.1 Algoritmo de Inserção

O algoritmo de inserção é um ótimo algoritmo para ordenar um número pequeno de elementos (Cormen et al., 2022). Tem por finalidade dividir o vetor de entrada em duas partes, com a parte direita contendo a parte desordenada do vetor e a parte esquerda a parte ordenada do vetor. O primeiro elemento do vetor começa com a parte ordenada do vetor, e o elemento da posição 1 é a posição inicial do marcador na primeira posição da parte desordenada do vetor. Por meio de looping, a ideia é mover o marcador para a sua posição correta na parte ordenada do vetor, assim como ele é incrementado depois para a próxima verificação até que o vetor esteja totalmente ordenado. A ideia do pseudocódigo pode ser analisada na figura abaixo:

Algorithm 1 Ordenação por Inserção

```
procedure INSERCAO(V, tamVet)                    ▷ V é um vetor numérico
    for (marcador = 1; marcador < tamVet) do
        pos ← marcador − 1
        aux ← V[marcador]

        ...Inserindo V[marcador] na parte ordenada...
        while (pos ≥ 0)E(aux < V[pos]) do
            V[pos + 1] ← V[pos]
            pos ← pos − 1
        end while
        V[pos + 1] ← aux

    end for
end procedure
```

Figura 1: Representação do pseudocódigo de inserção.

A complexidade assintótica de um algoritmo de ordenação é analisada através da função de complexidade com a finalidade de analisar o comportamento do custo quando os valores de entrada crescem.

A complexidade de tempo do algoritmo de inserção é O(n2), o seu melhor caso se encontra quando os valores de entrada já estão ordenados e a complexidade passa a ser O(n). O caso médio se encontra quando os valores de entrada estão ordenados de forma aleatoriamente, e sua complexidade assintótica é O(n2). E para o pior caso, isto é, quando os valores de entrada estão em ordem inversa, o algoritmo tem a complexidade O(n2) (GeeksforGeeks, 2024a). Além disso, o algoritmo é estável, ou seja, a ordem relativa dos elementos com chaves iguais é preservada após a ordenação (Souza, 2024b)

2.2 Algoritmo Seleção

O algoritmo de Seleção, assim como o algoritmo de Inserção, é um algoritmo melhor para ordenar pequenos conjuntos de dados de entrada. A sua lógica consiste em dividir o vetor em duas partes, sendo à direita a parte desordenada do vetor e a parte esquerda, a parte ordenada do vetor. O primeiro elemento do vetor consiste na posição ordenada e onde se encontra o início do marcador, em sua posição 0. Através dos processos de looping, a ideia é encontrar o menor elemento do conjunto desordenado e substituí-lo na posição do marcador. E posteriormente o marcador avança uma posição, e segue a mesma lógica até o fim da varredura do vetor, a procura do

menor elemento até o último maior ser alocado em sua posição correta. O seu pseudocódigo pode ser visto através da figura a seguir:

Algorithm 1 Ordenação por Seleção

```
procedure SELECAO(V, tamVet)                    ▷ V é um vetor numérico
    marcador ← 0
    menor ← 0
    while (marcador < tamVet − 1) do
        menor ← índice do menor elemento da parte desordenada do vetor
        if (vet[menor] < vet[marcador]) then
            troque vet[marcador] com vet[menor]
        end if
        marcador ← marcador + 1
    end while
end procedure
```

Figura 2: Representação do pseudocódigo de seleção.

A complexidade de tempo do algoritmo seleção é O(n2), e para valores de entrada ordenado, aleatório, ou ordenado de forma decrescente o algoritmo apresenta a mesma complexidade O(n2) em todos os casos (GeeksforGeeks, 2024b) Assim, apesar do algoritmo não apresentar uma eficiência em termos de desempenho, em contrapartida, ele apresenta uma maior simplicidade de implementação do algoritmo e que determinados contextos podem ser vantajosos. Já em relação a sua estabilidade, o algoritmo é instável, ou seja, não mantém a ordem dos elementos com chaves iguais.

2.3 Algoritmo MergeSort

O algoritmo MergeSort assim como o próximo a ser retratado, QuickSort, são algoritmos com estrutura recursiva, que seguem a abordagem de dividir e conquistar. Esse método segue uma abordagem em dividir o problema em vários subproblemas que são semelhantes ao problema, mas menores em tamanho, e depois resolve esses subproblemas recursivamente e os combinando até obter uma solução para o problema original (Cormen et al., 2022).

Por meio dessa abordagem, o algoritmo MergeSort é composto em duas fases: divisão e junção. Na fase de divisão o vetor original é dividido em dois outros de tamanhos menores, recursivamente, até obter vetores de tamanho 1. E na etapa de junção, os elementos do vetor divididos são ordenados em um vetor auxiliar e que depois é copiado para o vetor original (Souza, 2024c). Na figura abaixo é demonstrado o pseudocódigo deste algoritmo:

Algorithm 1 MergeSort

```
procedure MERGESORT(V, inicio, fim)          ▷ inicio e fim sao indices do vetor
    meio ← ⌊(inicio+fim)/2⌋
    if inicio < fim then
        MergeSort(V, inicio, meio)
        MergeSort(V, meio+1, fim)
        Merge(V, inicio, meio, fim)
    end if
end procedure

procedure MERGE(V, inicio, meio, fim)
    v1 ← V[inicio, meio]
    v2 ← V[meio+1, fim]
    vAux                                     ▷ vetor auxiliar com tamanho igual a (fim-inicio)+1
    while v1.size > 0 E v2.size > 0 do
        Compara os elementos de v1 e v2 e ordena-os no vetor auxiliar
    end while
    Copia o resto de v1 ou o resto de v2 para o vetor auxiliar
    Copia o vetor auxiliar para o vetor original
end procedure
```

Figura 3: Representação do pseudocódigo de MergeSort.

A complexidade assintótica do algoritmo MergeSort é de O(nlog2n) e é considerado um algoritmo ótimo, uma vez que, ele sempre possui a mesma complexidade independente da estrutura do vetor, ou seja, se encontra ordenado, desordenado ou ordenado decrescente. No entanto, por precisar de alocar um espaço de memória para o vetor auxiliar, e que deste copia os valores para o vetor original, há uma complexidade de espaço de memória maior, sendo assim, quanto maior o vetor original mais espaço ele ocupa na memória principal para fazer a ordenação. E com relação a sua estabilidade, o algoritmo é instável, ele não mantém a ordem dos elementos com chaves iguais.

2.4 Algoritmo QuickSort

O algoritmo QuickSort também utiliza a estratégia de Dividir para Conquistar, e a sua lógica consiste em escolher um elemento pivô e dividir o vetor desordenado em duas partes, a esquerda com elementos menores que o pivô, e a direita com elementos maiores do que o pivô. Ao final de cada passada o algoritmo garante que o pivô se encontra na sua posição ordenada (Souza, 2024d). A seguir é ilustrado o pseudocódigo deste algoritmo:

Algorithm 1 QuickSort

```
procedure QUICKSORT(V, inicio, fim)          ▷ inicio e fim sao indices do vetor
    if inicio < fim then
        pivo ← Particiona(vet, inicio, fim)
        QuickSort(V, inicio, pivo-1)
        QuickSort(V, pivo+1, fim)
    end if
end procedure

procedure PARTICIONA(V, inicio, fim)
    pivo ← V[inicio]
    pos ← inicio                             ▷ guarda a posicao final do pivo no vetor
    for (i = inicio + 1; i <= fim; i++) do
        if (V[i] < pivo) then
            pos ← pos + 1
            if (i ≠ pos) then
                troca V[i] com V[pos]
            end if
        end if
    end for
    troca V[inicio] com V[pos]
    Retorna pos
end procedure
```

Figura 4: Representação do pseudocódigo de QuickSort.

O QuickSort é um algoritmo de ordenação eficiente, com complexidade média O(nlog n). Ele é rápido na maioria dos casos, devido à sua estratégia de dividir e conquistar, no entanto, seu pior caso ocorre quando o vetor está ordenado ou ordenado de forma decrescente. Isso acontece devido à escolha inadequada do pivô, levando a um comportamento de particionamento ineficiente e resultando em uma complexidade quadrática O(n2) (Souza, 2024d). E assim como o MergeSort e o Seleção é um algoritmo de instabilidade.

2.5 Algoritmo MergeSort InPlace

O algoritmo MergeSort in place é uma otimização do algoritmo MergeSort tradicional em relação a complexidade de espaço de memória (GeeksforGeeks, 2024c). Como foi visto, o mergeSort por apresentar um vetor auxiliar para ordenar e deste é copiado para o vetor original, há um gasto de memória para a inserção do vetor auxiliar. Dessa forma, a ideia do

mergeSort in place é realizar a ordenação no próprio vetor e mantendo a complexidade assintótica de O(nlog2n).

Sendo assim, como um algoritmo alvo de pesquisa e inserção ao trabalho da equipe, ele também apresenta a característica de dividir para conquistar, em que a função MergeSortInPlace divide o vetor em vetores menores para assim, a função MergeInPlace os ordenar e juntar no vetor de origem. Outras características observadas desse algoritmo, ele também tem uma complexidade assintótica de tempo O(nlog2n) em todos os casos, e a sua complexidade de armazenamento é O(1), ao contrário do MergeSort tradicional, uma vez que, a sua otimização é evitar criar vetor temporária para a ordenação. Outra informação observada, é que o algoritmo realizado pela equipe é estável, sendo assim, a ordem dos elementos iguais não é alterada. O algoritmo pode ser analisado a seguir:

```
void MergeInPlace(int *vetor, int inicio, int meio, int fim){
    int i = inicio, j = meio + 1;
    while (i <= meio && j <= fim)
    {
        if (vetor[i] <= vetor[j])
        {
            i++;
        }
        else{
            int temp = vetor[j];

            for (int k = j; k > i; k--){
                vetor[k] = vetor[k - 1];
            }
            vetor[i] = temp;
            i++;
            j++;
            meio++;
        }
    }
}

void MergeSortInPlace(int *v, int inicio, int fim){
    if (inicio < fim)
    {
        int meio = inicio + (fim - inicio) / 2;

        MergeSortInPlace(v, inicio, meio);
        MergeSortInPlace(v, meio + 1, fim);

        if (v[meio] <= v[meio + 1])
        {
            return;
        }
        MergeInPlace(v, inicio, meio, fim);
    }
}
```

A lógica do MergeSortInPlace permanece parecida com a do MergeSort Tradicional, com a exceção em que ele verifica antes de fazer o merge se o vetor na posição do meio é menor do que o vetor na posição meio + 1, e se for verdadeira é considerado que essas posições estão ordenadas e não será preciso ser ordenadas. Outra diferença encontrada entre os MergeSort é que o MergeSortInPlace não há a chamada do vetor de uma unidade, e sim um subconjunto de unidades de até de dois elementos, isso é para otimizar e analisar a troca entre as posições do vetor. Já com relação ao MergeInPlace, é verificado na função se a posição i do vetor é menor ou igual a posição j do vetor, se verdadeira, não ocorre trocas dos elementos, assim como, mesmo se for igual os elementos de mesma chave, não haverá trocas e i é incrementado e verificado pela função while. Se a sentença for falsa, a variável temp irá obter o valor do vetor na posição i, e será comparado com os elementos do subvetor e ser trocado pela posição em que deve estar nesse subvetor, em seguida as variáveis i, j e meio são incrementadas e verificadas pela função while.

3 Metodologia

A metodologia deste trabalho consiste em comparar os algoritmos discutidos quanto a sua resposta de tempo para tamanhos de entradas de: 1000, 10000, 100000, 10000000 e 100000000. Os vetores de entrada apresentam o comportamento de valores ordenados, quase ordenados, ordenados decrescente e aleatório. Além disso, também será comparado a quantidade de trocas e comparações feitas por esses algoritmos com esses respectivos tamanhos de entrada. Apenas nos casos de conjunto aleatórios e quase ordenado o resultado do teste será uma média de 5 repetições com conjuntos diferentes. Assim, se espera analisar qual o comportamento do algoritmo alvo em relação aos demais, se por apresentar uma ordenação própria no vetor ele se é melhor ou pior em relação aos outros. E com uma ressalva que, todos os testes foram feitos na mesma máquina para que não houvesse discrepância de tempo de uma máquina a outra.

4 Resultados e Discussão

Nessa parte ocorre a discussão dos benchmarks realizados com os algoritmos vistos neste trabalho. Primordialmente, foi foram feitos quadros variando o tamanho da entrada entre: 1000, 10000, 100000, 10000000 e 100000000, e com tipos de entrada de: aleatório, ordem crescente, ordem decrescente e quase ordenado para os algoritmos: Inserção, Seleção, MergeSort, QuickSort e MergeSort In Place.

Na tabela de inserção, é visto que, para valores aleatório e com o aumento de quantidade de valores da entrada para 1000000 ele apresenta um tempo de 546 segundos, em compensação em ordem crescente o tempo é muito pequeno de 0,004 segundos, em ordem decrescente foi de 1091 segundo e quase ordenado foi de 48 segundos. A explicação a esse comportamento, é que o algoritmo de inserção é um algoritmo bom para valores já ordenados e de um conjunto de entrada pequeno, e por isso é visto o tempo de ordenação o menor de todos. Já o seu pior caso é decrescente e como visto é o que apresenta o maior tempo para ordenar. Os valores 0 encontrados na tabela são devido a faixa de casas escolhidas pelo grupo, pois o tempo

de ordenação foi bem pequeno. E aos valores de conjunto de dados de entrada de 1000000, como a inserção é um algoritmo ruim para grandes valores, o tempo de execução foi extenso a ponto de ter levados horas e a equipe optou em não anotar no quadro.

Teste	Tamanho de Entrada	Aleatório (média de 5 repetições) segundos	Ordem crescente segundos	Ordem decrescente segundos	Quase ordenado (média de 5 repetições) segundos
1	1000	0,0006	0,0000	0,0000	0,0004
2	10000	0,057	0,0000	0,1110	0,0420
3	100000	5,4422	0,0000	11,0450	1,4632
4	1000000	546,0992	0,0000	1091,1350	48,2004
5	10000000	-	-	-	-

Figura 5: Tabela de benchmarck do inserção

Em relação ao benchmark do algoritmo seleção é visto que, nos cinco tipos de caso de entrada o tempo gasto para ordenar foram parecidos, e isso é devido o algoritmo apresentar uma complexidade O(n2) em todos os casos, uma vez que, o método utiliza comparação para encontrar o menor elemento e ordená-lo na posição do vetor de ordenação. O tempo gasto pelo algoritmo é mostrado a seguir:

Teste	Tamanho de Entrada	Aleatório (média de 5 repetições) segundos	Ordem crescente segundos	Ordem decrescente segundos	Quase ordenado (média de 5 repetições) segundos
1	1000	0,001	0,002	0,001	0,0008
2	10000	0,0984	0,095	0,099	0,0974
3	100000	9,3878	9,459	9,712	9,381
4	1000000	941,211	940,175	968,163	940,7112
5	10000000	-	-	-	-

Figura 6: Tabela de benchmarck do seleção

O mergeSort por sua vez, apresentou um comportamento de tempo bem parecido entre os vetores de ordem crescente e decrescente, assim como entre os vetores aleatório e quase ordenado. Isso é devido que, no primeiro caso, a função de mesclagem apresenta um custo menor do que em relação ao vetor aleatório e quase ordenado, e por isso o tempo é menor em relação a esses dois. A tabela abaixo ilustra essa situação:

Teste	Tamanho de Entrada	Aleatório (média de 5 repetições) segundos	Ordem crescente segundos	Ordem decrescente segundos	Quase ordenado (média de 5 repetições) segundos
1	1000	0,0032	0,0090	0,0000	0,0066
2	10000	0,2376	0,0390	0,0240	0,0688
3	100000	1,0286	0,3570	0,3290	1,0276
4	1000000	13,8446	3,7280	3,7320	12,338
5	10000000	-	-	-	-

Figura 7: Tabela de benchmarck do mergesort

Os resultados obtidos do Quicksort mostram que, em ordem crescente e decrescente, os valores de tempo foram menores do que os valores de tempo de vetores em ordem aleatória e de quase ordenados. E isso é devido a implementação realizada pela equipe que envolve o quicksort com o pivô escolhido aleatoriamente, e dessa forma, ele evita que o pior caso seja quando o pivô escolhido é a primeira posição e apresenta um comportamento de árvore de recursão quando o vetor se encontra ordenado ou quase ordenado. A tabela a seguir mostra o benchmark para esse algoritmo:

Teste	Tamanho de Entrada	Aleatório (média de 5 repetições) segundos	Ordem crescente segundos	Ordem decrescente segundos	Quase ordenado (média de 5 repetições) segundos
1	1000	0.0000	0,0000	0,0000	0,0088
2	10000	0,0344	0,0020	0,0000	0,0688
3	100000	0,4622	0,0420	0,0510	0,3266
4	1000000	4,9248	0,5190	0,5480	3,9379
5	10000000	-	-	-	-

Figura 8: Tabela de benchmarck do quicksort

Quanto o MergeSort In Place, é possível verificar que o vetor de entrada em ordem crescente obteve um tempo menor e eficiente em relação aos demais. E isso é devido a comparação do if dentro da função mergesortinplace, que se o vetor já estiver ordenado ele não entra no mergeinplace para fazer trocas dos elementos. Já com relação ao vetor aleatório e de ordem decrescente ele obteve valores muito altos de tempo em comparação ao de ordem crescente e de quase ordenado, e isto é devido a entrada da função mergeinplace para ordenar subvetores que são divididos recursivamente, e comparar cada elemento do subvetor secundário com o primeiro para colocar na posição correta. A tabela de tempo do mergesort in place pode ser visto na tabela a seguir:

Teste	Tamanho de Entrada	Aleatório (média de 5 repetições) segundos	Ordem crescente segundos	Ordem decrescente segundos	Quase ordenado (média de 5 repetições) segundos
1	1000	0,0052	0,0000	0,0020	0,0172
2	10000	0,2574	0,0000	0,1560	0,2012
3	100000	22,1706	0,0000	14,5700	59,576
4	1000000	2194,7294	0,0030	1458,5040	3,9379
5	10000000	-	-	-	-

Figura 9: Tabela de benchmarck do mergesort in place

De acordo com os valores na tabela, o algoritmo de Inserção teve um desempenho razoável para ruim. Em valores aleatórios sua execução não foi das melhores, demorando muito para calcular a média entre as cinco iterações, tanto para valores pequenos quanto para valores muito grandes. O código demorou bastante para calcular um milhão de valores. No pior caso, quando o vetor está organizado em ordem decrescente, a notação do algoritmo é $O(n^2)$, o que resultou em um tempo muito grande para ordenar o vetor. Porém, os valores ordenados em ordem crescente tiveram um dos melhores desempenhos de todos os algoritmos. Como ele tem notação O(n) no melhor caso, seu tempo saiu muito melhor do que os outros algoritmos.

O algoritmo de Seleção foi um dos piores em desempenho. Durante a execução, foi o algoritmo mais demorado. Em todos os tipos de vetores ordenados, o Seleção teve um tempo de ordenação constante em relação às entradas. Por exemplo, tanto nos valores aleatórios quanto nos valores ordenados crescentemente, para um milhão de valores a média de tempo foi quase igual. Isso se dá por conta do algoritmo realizar a mesma quantidade de comparações independente da entrada, além do mesmo ter uma notação de $O(n^2)$ para qualquer tipo de entrada.

Para o MergeSort tradicional, o grupo obteve um dos melhores algoritmos de ordenação. Por conta da sua recursividade e notação O(n log2 n) para todos os tipos de casos, em sua grande maioria, o tempo para ordenação foi bem baixo, até mesmo para valores muito grandes. Quando os valores estavam tanto ordenados crescentemente quanto decrescentemente, o tempo se manteve quase inalterado. Para os valores

aleatórios e quase ordenados, o desempenho do algoritmo se destacou como um dos melhores.

O QuickSort também foi um dos melhores algoritmos durante os testes. Como o Quick possui seu melhor e pior caso atrelados ao pivô escolhido, neste código o pivô foi definido como o meio do vetor. Assim, era possível que o vetor fosse dividido de uma forma que sempre resultasse no melhor caso, sendo o algoritmo com melhor desempenho dentre os demais. A execução do Quick foi muito rápida independente da entrada. Se compararmos os valores do tempo de ordenação com os demais códigos, em geral, foram os menores valores, perdendo somente para o Inserção e o MergeSort In-Place na ordem Crescente.

O algoritmo merge in place tem uma eficiência notável quando se trata de um vetor ordenado em ordem crescente, pois realiza em um tempo muito curto sobre qualquer outro algoritmo. Entretanto, se tratando de outros tipos de entradas, seu desempenho foi um dos piores, demorando muito em sua execução. Com valores quase ordenados, o algoritmo teve um desempenho aceitável e não demorou muito, mas para valores aleatórios e ordenados decrescentemente, seu desempenho foi pior do que os outros algoritmos não recursivos.

A comparação entre os algoritmos com relação ao tipo de entrada: ordenado, decrescente, aleatorio e quase ordenado, assim como o número de trocas e o número de comparações, podem ser vistos nas figuras adiantes, respectivamente:

Algoritmo	Tipo	Tamanho de Entrada	Tempo para ordenação	Número de trocas	Número de comparações
Inserção	Ordem Crescente	1000	0,0000	999	0
		10000	0,0000	9999	0
		100000	0,0000	99999	0
		1000000	0,0040	999999	0
Seleção	Ordem Crescente	1000	0,0020	0	499500
		10000	0,0950	0	49995000
		100000	9,4590	0	704982704
		1000000	940,1750	0	1783293664
MergeSort	Ordem Crescente	1000	0,0090	9976	5044
		10000	0,0390	133616	69008
		100000	0,3570	1668928	853904
		1000000	3,7280	19951424	10066432
QuickSort	Ordem Crescente	1000	0,0000	511	7987
		10000	0,0020	5904	113631
		100000	0,04200	65535	1468946
		1000000	0,5190	524287	17951445
MergeSort In place	Ordem Crescente	1000	0.0000	0	0
		10000	0.0000	0	0
		100000	0.0000	0	0
		1000000	0,003	0	0

Figura 10: Tabela de comparação dos algoritmos em relação ao vetor de entrada crescente

Algoritmo	Tipo	Tamanho de Entrada	Tempo para ordenação	Número de trocas	Número de comparações
Inserção	Ordem Decrescente	1000	0	999	499500
		10000	0,111	9999	49995000
		100000	11,045	99999	704982704
		1000000	1091,135	999999	1783293664
Seleção	Ordem Decrescente	1000	0,001	500	499500
		10000	0,099	5000	49995000
		100000	9,712	50000	704982704
		1000000	968,163	500000	1783293664
MergeSort	Ordem Decrescente	1000	0	9976	4932
		10000	0,024	133616	64608
		100000	0,329	1668928	815024
		1000000	3,732	19951424	9884992
QuickSort	Ordem Decrescente	1000	0	1010	6996
		10000	0	10904	103644
		100000	0,051	115534	1368962
		1000000	0,548	1024286	16951464
MergeSort In place	Ordem Decrescente	1000	0,002	4932	4932
		10000	0,156	64608	64608
		100000	14,57	815024	815024
		1000000	1458,504	9884992	9884992

Figura 11: Tabela de comparação dos algoritmos em relação ao vetor de entrada decrescente

Algoritmo	Tipo	Tamanho de Entrada	Tempo para ordenação	Número de trocas	Número de comparações
Inserção	Aleatório	1000	0,0006	999	249.108
		10000	0,057	9999	25020723
		100000	5,4422	99999	1795837223
		1000000	546,0992	999999	897757166
Seleção	Aleatório	1000	0,001	992	499500
		10000	0,0984	9989	49995000
		100000	9,3878	99984	704982704
		1000000	941,211	999957	1783293664
MergeSort	Aleatório	1000	0,0032	29928	26145
		10000	0,2376	400848	361300
		100000	1,0286	5006784	4609234
		1000000	13,8446	59854272	56022366
QuickSort	Aleatório	1000	0	7670	22959
		10000	0,0344	101800	314863
		100000	0,4622	1314234	3661366
		1000000	4,9248	17199288	39654929
MergeSort In place	Aleatório	1000	0	12998	25129
		10000	0,2574	177564	349580
		100000	22,1706	2279812	4486386
		1000000	2194,7294	27837668	54966563

Figura 12: Tabela de comparação dos algoritmos em relação ao vetor de entrada aleatório

Algoritmo	Tipo	Tamanho de Entrada	Tempo para ordenação	Número de trocas	Número de comparações
Inserção	Ordem Decrescente	1000	0,0004	999	195640
		10000	0,042	9999	19068639
		100000	1,4632	99999	672242270
		1000000	48,2004	999999	682073233
Seleção	Ordem Decrescente	1000	0,0008	498	499500
		10000	0,0974	4999	49995000
		100000	9,381	49990	704982704
		1000000	940,7112	499990	1783293664
MergeSort	Ordem Decrescente	1000	0,0066	29928	23491
		10000	0,0688	400848	319716
		100000	1,0276	5006784	3502848
		1000000	12,338	59854272	41005850
QuickSort	Ordem Decrescente	1000	0,0088	5907	22522
		10000	0,0068	73623	325025
		100000	0,3266	677957	4386536
		1000000	3,9379	7305289	52047685
MergeSort In place	Ordem Decrescente	1000	0,0172	9655	22016
		10000	0,2012	128792	301377
		100000	5,9576	1060421	2085717
		1000000	250,6651	12021079	24628712

Figura 13: Tabela de comparação dos algoritmos em relação ao vetor de entrada quase ordenado

5 Conclusão

Após a análise dos respectivos algoritmos no decorrer do presente trabalho, a equipe pode concluir em relação ao algoritmo alvo de estudo. Ele apresentou o pior cenário de ordenação para dados de entrada aleatório e de ordem de decrescente, e quando o vetor já está ordenado ele foi melhor que o inserção, assim sendo, ele apresentou o melhor cenário a este caso, como também ele apresentou uma boa eficiência ao valores de entrada quase ordenados, mas ficou após o algoritmo de inserção. Assim sendo, apesar do algoritmo ser uma otimização do MergeSort tradicional, ele tem a vantagem na complexidade de armazenamento, mas em tempo de execução ele demonstrou não ser tão eficiente, assim esse algoritmo para uma melhor aplicação depende do cenário em que for utilizado.

Anexos

Links para o repositório do trabalho no GitHub:

- GitHub:

  `https://github.com/Tiago-htm/MergeSortInPlace-/tree/main/codigo`

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms.* MIT press.

GeeksforGeeks (2024a). Insertion sort. Disponível em: https://www.geeksforgeeks.org/insertion-sort/.

GeeksforGeeks (2024b). Insertion sort. Disponível em: https://www.geeksforgeeks.org/selection-sort/?ref=shm.

GeeksforGeeks (2024c). Insertion sort. Disponível em: https://www.geeksforgeeks.org/in-place-merge-sort/.

Souza, V. (2024a). Algoritmos e estrutura de dadosii: Aula 2 - complexidade.

Souza, V. (2024b). Algoritmos e estrutura de dadosii: Aula 5 - inserção.

Souza, V. (2024c). Algoritmos e estrutura de dadosii: Aula 6 - mergesort.

Souza, V. (2024d). Algoritmos e estrutura de dadosii: Aula 7 - quicksort.