Análise de Desempenho de Métodos de Ordenação: Um Estudo Empírico Utilizando Experimentos e Gráficos

Dayan Freitas Alves[1]

[1]Centro de Ciências Exatas e Tecnológicas - Universidade Federal do Acre (UFAC)
CEP 69.9280-900 - Rio Branco - AC - Brasil

dayan.alves@sou.ufac.br

Abstract. *This article presents a comparative analysis of sorting methods, emphasizing the performance of each algorithm concerning the execution time across different sizes of datasets. The sorting methods are described using pseudocode and implemented in the C programming language. The experiments cover a wide range of vector sizes, providing a comprehensive understanding of the algorithms' behavior. The results are showcased through individual graphs for each method, along with comparative graphs to facilitate data interpretation.*

Resumo. *Este artigo apresenta uma análise comparativa de métodos de ordenação, destacando o desempenho de cada algoritmo em relação ao tempo de execução em diferentes tamanhos de conjuntos de dados. Os métodos de ordenação são descritos através de pseudocódigos e implementados em linguagem C. Os experimentos abrangem uma ampla gama de tamanhos de vetores, proporcionando uma compreensão abrangente do comportamento dos algoritmos. Os resultados são apresentados através de gráficos individuais para cada método, bem como gráficos comparativos para facilitar a interpretação dos dados.*

1. Introducão

A ordenação de dados é uma operação fundamental em ciência da computação, desempenhando um papel crucial em diversas aplicações. A necessidade de organizar dados de maneira eficiente é evidente em algoritmos de busca, manipulação de grandes conjuntos de dados e otimização de processos computacionais. A falta de uma ordenação eficaz pode resultar em ineficiência computacional, prejudicando o desempenho de sistemas e aplicativos.

Dada a variedade de métodos de ordenação disponíveis, é essencial entender o desempenho relativo de cada algoritmo. A análise de desempenho não apenas auxilia na escolha do método mais apropriado para uma determinada tarefa, mas também contribui para o desenvolvimento de algoritmos mais eficientes e adaptáveis a diferentes cenários. A compreensão dos trade-offs entre eficiência temporal e requisitos de espaço é fundamental para otimizar a implementação de sistemas.

2. Ambiente e Configuração Experimental

Na implementação dos métodos de ordenação, a linguagem escolhida foi o C, a natureza eficiente e próxima ao hardware do C o torna uma escolha natural para algoritmos que demandam alto desempenho, como os métodos de ordenação em foco.. O ambiente de execução utilizou um PC robusto, equipado com um Intel Core i9-13980HX, 16GB de

RAM, GeForce RTX 4060 e um SSD de 512GB, operando sob o Windows 11. Em alguns casos o computador ficou operando durante semanas.

Para o desenvolvimento, optou-se pelo Visual Studio Code (VSCode), com uma extensão de compilador de C, para facilitar o processo devido sua interface amigável. Visando otimizar o desempenho, foram desativados todos os processos não essenciais, utilizando o máximo da máquina durante os experimentos.

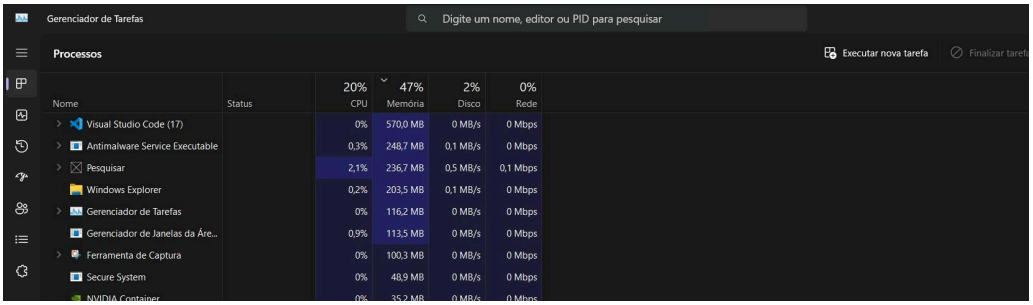


Figura 1. Gerenciador de Tarefas.

Essa configuração estratégica busca garantir a consistência e a confiabilidade dos resultados, oferecendo um ambiente controlado para a execução dos experimentos.

Além disso, para a metodologia, foram gerados vetores de tamanhos variados, indo de 10 a 1 milhão, preenchidos com números aleatórios. Essa abordagem permite medir o tempo de execução dos métodos de ordenação em diferentes cenários, abrangendo desde conjuntos de dados pequenos até grandes volumes, proporcionando uma análise abrangente do desempenho dos algoritmos em escalas diversas.

3. Métodos e Tempo

É de crucial importância que se avalie, para cada método de ordenação, sua complexidade, eficiência em relação a arrays grandes e sua estabilidade. Cabe destacar que um algoritmo de ordenação é considerado estável quando consegue preservar a ordem de registros com chaves iguais. Em outras palavras, se os registros aparecem na sequência ordenada, eles devem manter a mesma ordem em que estavam na sequência inicial. Essa análise prévia possibilita uma melhor previsão e avaliação do desempenho dos métodos. Ao considerar a complexidade, a capacidade de lidar com grandes conjuntos de dados e a estabilidade, é possível realizar uma análise mais precisa e embasada, identificando quais métodos se destacam em diferentes contextos e proporcionam resultados mais eficientes. A estrutura principal em C é crucial para executar os métodos de ordenação e medir o tempo em diferentes tamanhos de vetores.

```c
int main() {
    setlocale(LC_NUMERIC, "C");
    // Seed para números aleatórios
    srand(time(NULL));
    // Cria um array de números aleatórios
    int sizes[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000,
    6000, 7000, 8000, 9000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000, 200000, 300000, 400000, 500000,
    600000, 700000, 800000, 900000, 1000000};
    int* arr = NULL;
    int size = 0;
    clock_t start, end;
    double cpu_time_used;
    for (size_t i = 0; i < sizeof(sizes) / sizeof(sizes[0]); i++) {
        size = sizes[i];
        arr = malloc(size * sizeof(int));
        for (int j = 0; j < size; j++) {
            arr[j] = rand() % 100;  // Números aleatórios entre 0 e 99
        }
        // Bubble Sort
        // Bubble Sort Recursivo
        // Selection Sort
        // Selection Sort Recursivo
        // Insertion Sort
        // Insertion Sort Recursivo
        // Quick Sort
        // Merge Sort
        // Heap Sort
        // Libera a memória alocada para o array
        free(arr);
    }
    return 0;
}
```

Figura 2. Main, do código C.

Os resultados são convertidos em um arquivo CSV. Posteriormente, para análise e visualização, utiliza-se um Notebook Python com a biblioteca pandas para manipulação de dados e matplotlib.pyplot para a criação de gráficos.Para realizar essas etapas, a plataforma Kaggle foi utilizada.

```python
import pandas as pd
import matplotlib.pyplot as plt

# Leia o arquivo CSV
dados = pd.read_csv('/kaggle/input/yooooo/Planilha sem ttulo - Pgina1.csv')


# Crie um gráfico para cada algoritmo
for algoritmo in dados['Algoritmo'].unique():
    # Filtrar os dados para o algoritmo atual
    dados_algoritmo = dados[dados['Algoritmo'] == algoritmo]

    # Crie o gráfico
    plt.figure(figsize=(10, 6))
    plt.plot(dados_algoritmo['Tamanho do vetor'], dados_algoritmo['Tempo de execucao'], label=algoritmo)
    plt.xlabel('Tamanho do vetor')
    plt.ylabel('Tempo de execução (ms)')
    plt.title(f'Gráfico de {algoritmo}')
    plt.legend()
    plt.show()
```

Figura 3. Código Python para gerar os gráficos.

3.1. Bubble Sort

O Bubble Sort, também conhecido como Ordenação por Bolha, opera comparando pares de elementos adjacentes e realizando trocas quando necessário, com o objetivo de alcançar uma ordenação crescente. Esse processo de comparação e troca é repetido iterativamente até que a lista esteja completamente ordenada. Uma característica visualmente marcante desse algoritmo é a maneira como os elementos maiores "borbulham" para o final da lista.Este método de ordenação é caracterizado pela sua estabilidade, onde registros com chaves iguais mantêm a mesma ordem relativa. No entanto, apesar dessa estabilidade, o Bubble Sort apresenta algumas limitações notáveis. Sua versão recursiva, embora conceitualmente interessante, é notavelmente mais lenta que a versão iterativa. Além disso, o algoritmo demonstra ineficiência quando aplicado a arrays de grande porte.

```c
// Método de ordenação: Bubble Sort
// Pseudocódigo:
// Função BubbleSort(lista):
//     n = comprimento da lista
//     para i de 0 até n-1, exclusivo:
//         para j de 0 até n-i-1, exclusivo:
//             se lista[j] > lista[j+1]:
//                 trocar lista[j] e lista[j+1]
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // swap(&arr[j], &arr[j + 1]);
                // Troca os elementos se estiverem fora de ordem
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Figura 4. Pseudocódigo e Função Bubble Sort em C.

No que diz respeito à complexidade computacional, o desempenho do Bubble Sort varia em diferentes cenários. No melhor caso, quando a lista já está ordenada, resultando em um desempenho otimizado, a complexidade é de ordem N. No caso médio, o Bubble Sort requer um número significativo de comparações e trocas, com uma complexidade de ordem N ao quadrado. No pior caso, que ocorre quando a lista está inversamente ordenada, demandando o máximo de operações de troca, a complexidade também é de ordem N ao quadrado. Embora o Bubble Sort seja uma abordagem inicial e compreensível para a ordenação, suas limitações em termos de eficiência tornam-no menos adequado para situações que envolvem grandes conjuntos de dados.

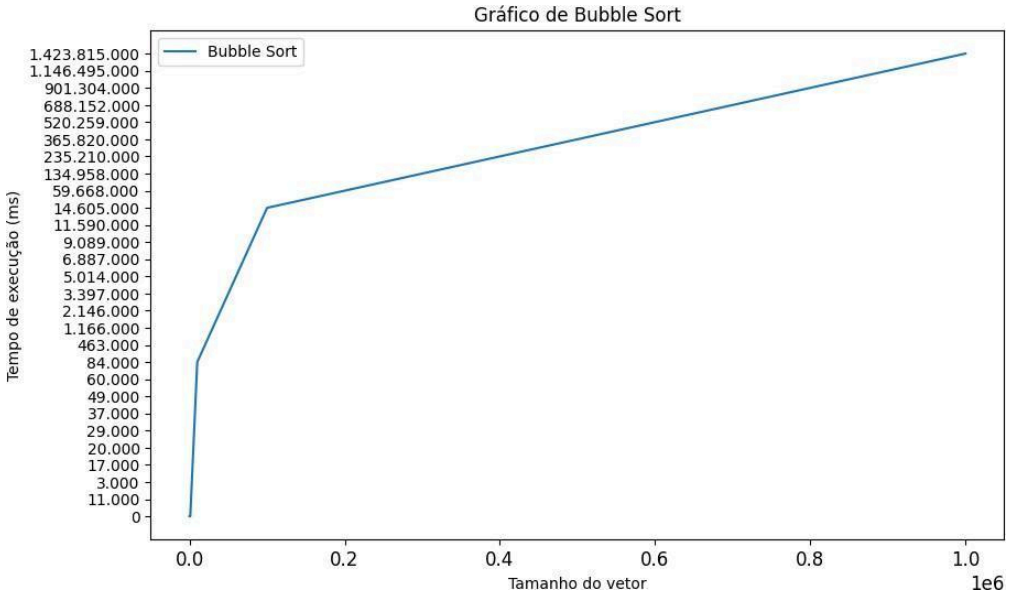


Figura 5. Grafico do Bubble Sort.

Vale ressaltar o uso do Bubble Sort recursivo para essa métrica. Uma observação relevante é que, ao empregá-lo, identifica-se limitações que se manifestam especialmente ao alcançar vetores de tamanho 40000. Antes de concluir a execução do

programa, ocorre um possível término devido ao Bubble Sort recursivo atingir o limite de profundidade de recursão.

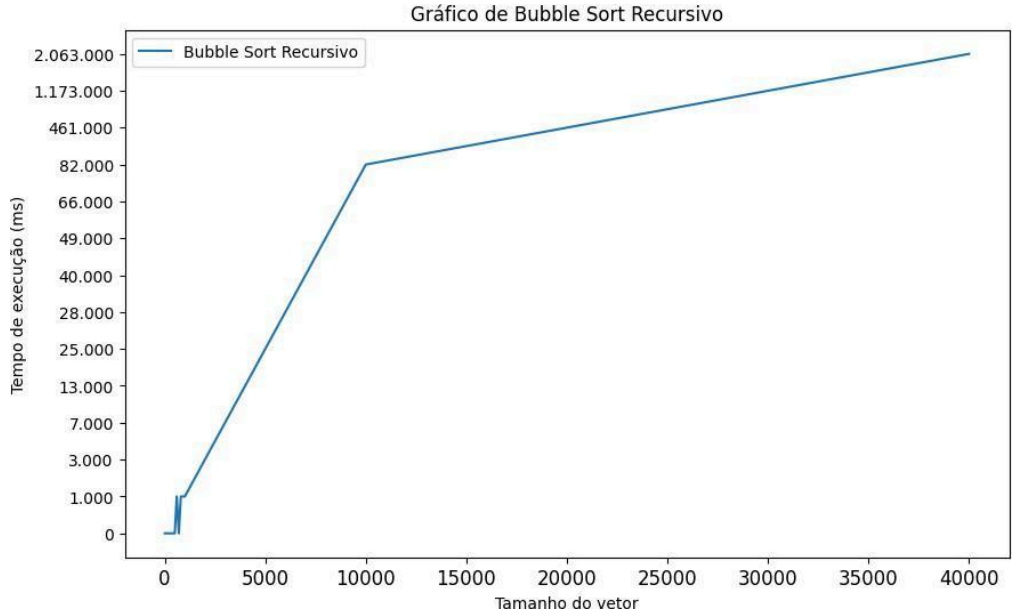


Figura 6. Grafico do Bubble Sort Recursivo.

Cada chamada recursiva adiciona uma nova camada à pilha de chamadas do programa. No entanto, a quantidade de espaço disponível na pilha é restrita. Assim, se o programa tentar realizar mais chamadas recursivas do que o espaço na pilha permite, ocorrerá um erro de estouro de pilha. É importante observar que o Bubble Sort não é comumente implementado de forma recursiva devido a essa limitação. Em contrapartida, a versão iterativa do Bubble Sort não enfrenta esse problema, uma vez que não utiliza a pilha de chamadas da mesma maneira.

3.2. Selection Sort

Selection Sort ou Ordenação por Seleção é um algoritmo que, a cada passo, busca o menor elemento na parte não ordenada da lista e o coloca na seguinte posição da parte ordenada. Esse processo é repetido iterativamente até que a lista esteja completamente ordenada. Diferentemente do Bubble Sort, o Selection Sort não é considerado um algoritmo estável, pois a ordem relativa de chaves iguais pode ser alterada durante a ordenação.

A versão recursiva do Selection Sort, embora conceitualmente interessante, tende a ser mais lenta quando comparada à sua contraparte iterativa. Além disso, o Selection Sort demonstra ineficiência quando aplicado a arrays de grande porte, sendo superado por algoritmos mais avançados em termos de desempenho.

```c
// Pseudocódigo:
// Função SelectionSort(lista):
//     Para cada elemento i do início até o penúltimo da lista:
//         índice_minimo = i
//         Para cada elemento j de i+1 até o final da lista:
//             Se lista[j] < lista[índice_minimo]:
//                 índice_minimo = j

//         Se índice_minimo diferente de i:
//             Trocar lista[i] com lista[índice_minimo]

//     Retornar lista
// Método de ordenação: Selection Sort
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Supõe que o índice atual é o mínimo
        int minIndex = i;
        // Encontra o índice do elemento mínimo na parte não ordenada
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // swap(&arr[i], &arr[minIndex]);
        // Troca o elemento mínimo encontrado com o primeiro elemento
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}
```

Figura 7. Pseudocódigo e Função Selection Sort em C.

No que diz respeito à complexidade computacional, o desempenho do Selection Sort varia em diferentes cenários. No melhor caso, quando a lista já está ordenada, resultando em um desempenho otimizado, a complexidade é de ordem N. No caso médio, o Selection Sort requer um número significativo de comparações e trocas, com uma complexidade de ordem N ao quadrado. No pior caso, que ocorre quando a lista está inversamente ordenada, demandando o máximo de operações de troca, a complexidade também é de ordem N ao quadrado.

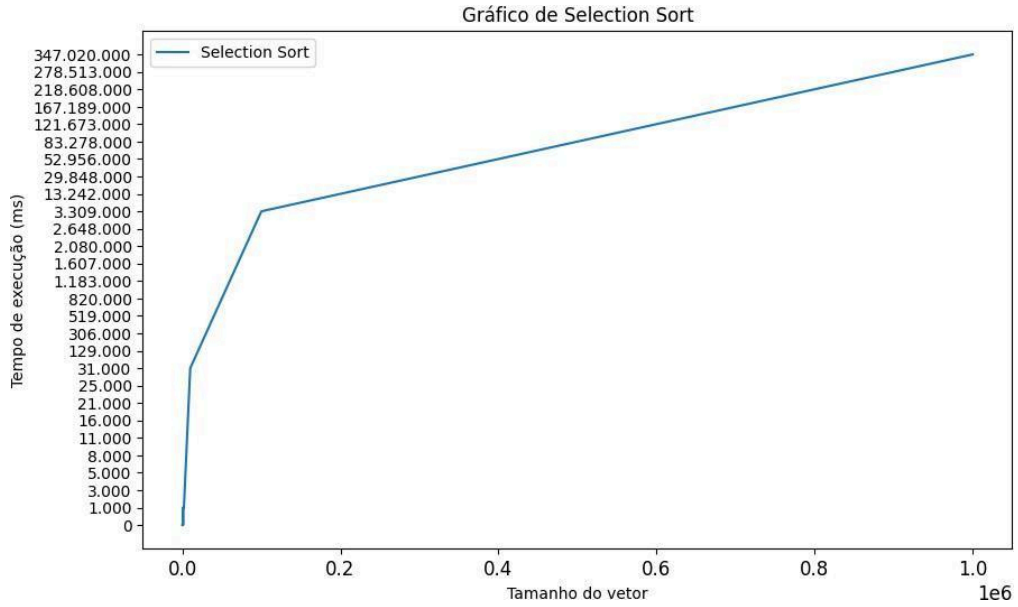


Figura 8. Grafico do Selection Sort.

Assim como o Bubble Sort, o Selection Sort pode atingir o limite de profundidade de recursão quando implementado de forma recursiva. Cada chamada

recursiva adiciona uma nova camada à pilha de chamadas do programa. Se o programa tentar fazer mais chamadas recursivas do que o espaço na pilha permite, ocorrerá um erro de estouro de pilha. O Selection Sort, assim como o Bubble Sort, não é geralmente implementado de forma recursiva por causa desse problema. A versão iterativa do Selection Sort não tem essa limitação, já que não utiliza a pilha de chamadas da mesma maneira.

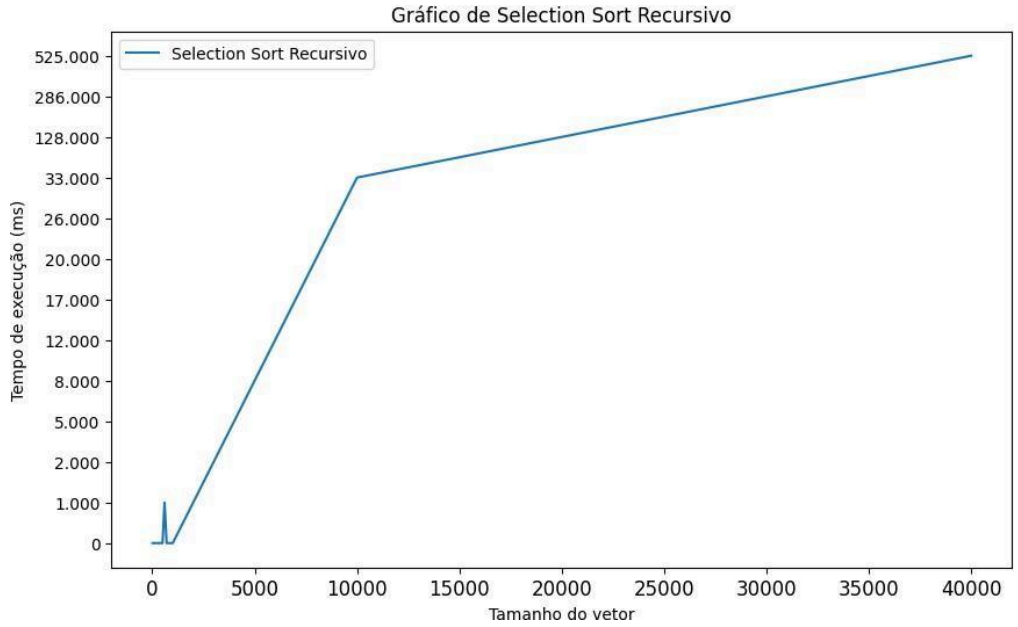


Figura 9. Grafico do Selection Sort Recursivo.

3.3. Insertion Sort

O algoritmo de ordenação por inserção, conhecido como Insertion Sort, adota uma abordagem peculiar para ordenar elementos em um array. A cada iteração, percorre o array e insere cada elemento na posição correta da parte ordenada, assemelhando-se ao processo de organizar cartas de baralho.

Diferentemente do Bubble Sort e do Selection Sort, o Insertion Sort é considerado um algoritmo estável, preservando a ordem relativa de elementos iguais. No entanto, assim como seus predecessores, o Insertion Sort apresenta desvantagens notáveis. A versão recursiva do Insertion Sort, embora conceitualmente interessante, tende a ser mais lenta quando comparada à sua contraparte iterativa. Além disso, o algoritmo revela ineficiência ao lidar com arrays de grande porte, sendo superado por algoritmos mais avançados em termos de desempenho.

```
// Método de ordenação: Insertion Sort
// Pseudocódigo:
// Função InsertionSort(lista):
//     Para cada elemento i de 1 até o final da lista:
//         valor_atual = lista[i]
//         j = i - 1

//         Enquanto j >= 0 e lista[j] > valor_atual:
//             lista[j + 1] = lista[j]
//             j = j - 1

//         lista[j + 1] = valor_atual

//     Retornar lista

void insertionSort(int arr[], int n) {
    int key, j;
    for (int i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Figura 10. Pseudocódigo e Função Insertion Sort.

Quanto à complexidade computacional, o desempenho do Insertion Sort é caracterizado no melhor caso, ordem de N, este cenário ocorre quando o array já está parcialmente ordenado, resultando em um desempenho otimizado. Caso médio ordem de N ao quadrado, no cenário médio, o Insertion Sort requer um número significativo de comparações e deslocamentos para ordenar o array. No pior caso, ordem de N ao quadrado, este cenário ocorre quando o array está inversamente ordenado, demandando o máximo de operações de comparação e deslocamento possível.

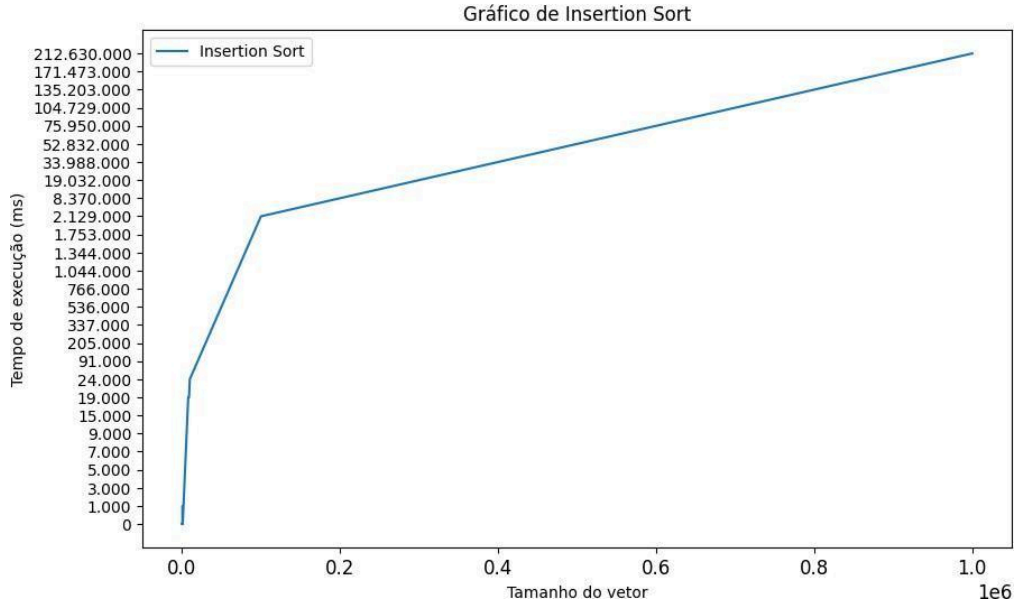


Figura 11. Grafico do Insertion Sort.

O algoritmo Insertion Sort, quando implementado de forma recursiva, enfrenta um desafio semelhante aos algoritmos Bubble Sort e Selection Sort na questão do limite de profundidade de recursão. Cada chamada recursiva acrescenta uma nova camada à pilha de chamadas do programa. Se o programa tentar realizar um número excessivo de chamadas recursivas, ultrapassando a capacidade disponível na pilha, isso resultará em um erro de estouro de pilha.

Em virtude dessa limitação, o Insertion Sort recursivo, assim como suas contrapartes Bubble Sort e Selection Sort, não é comumente implementado de forma recursiva. Em vez disso, a versão iterativa do Insertion Sort é preferida, pois ela não utiliza a pilha de chamadas da mesma maneira, evitando assim problemas relacionados ao limite de profundidade de recursão. Essa abordagem iterativa oferece uma alternativa mais eficiente e segura para ordenar elementos.

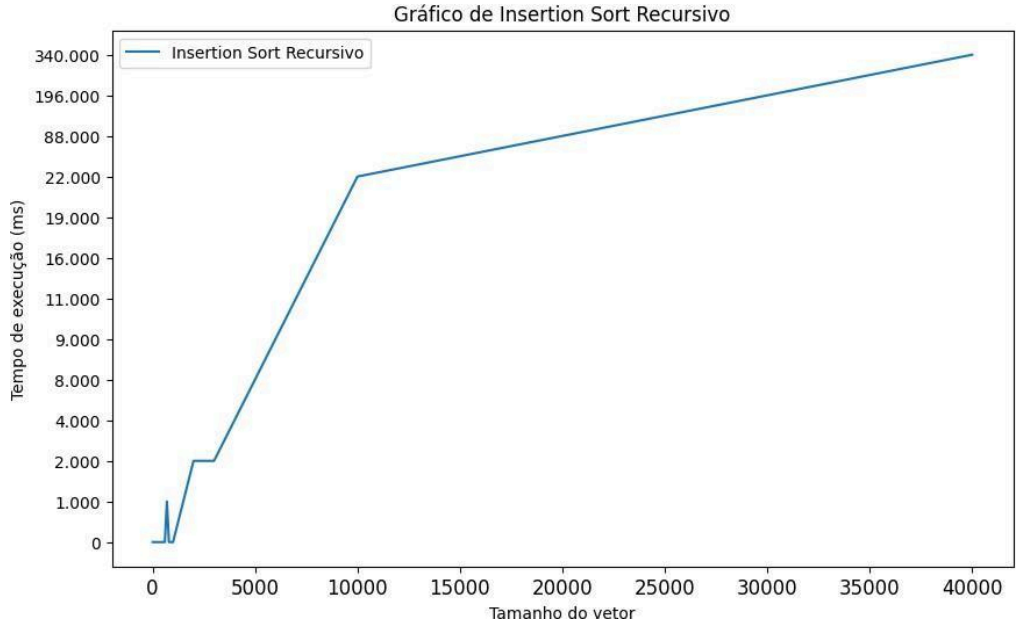


Figura 12. Grafico do Insertion Sort Recursivo.

3.4. Quick Sort

O algoritmo conhecido como Quick Sort, ou Ordenação Rápida, é uma técnica de ordenação altamente eficaz, notável por sua abordagem eficiente baseada em divisão e conquista. Este método de ordenação opera selecionando um elemento como pivô e, em seguida, rearranja a array, colocando os elementos menores antes do pivô e os maiores após ele. Esse processo é aplicado de forma recursiva às sub-listas geradas, seguindo o princípio de divisão e conquista.

```
// Pseudocódigo:
// Função quickSort(array, início, fim)
//     Se início < fim
//         Pivô = particionar(array, início, fim)
//         quickSort(array, início, Pivô - 1)
//         quickSort(array, Pivô + 1, fim)

// Função particionar(array, início, fim)
//     Pivô = array[fim]
//     ÍndicePivô = início

//     Para i de início até fim - 1
//         Se array[i] <= Pivô
//             Trocar array[i] com array[ÍndicePivô]
//             Incrementar ÍndicePivô

//     Trocar array[ÍndicePivô] com array[fim]

//     Retornar ÍndicePivô

// Função para encontrar a posição correta do pivô no array
// Nesse caso, o pivô é o primeiro elemento
int partition(int arr[], int low, int high) {
    int pivot = arr[low];
    int i = low + 1;

    for (int j = low + 1; j <= high; j++) {
        if (arr[j] < pivot) {
            // swap(&arr[i], &arr[j]);
            arr[i] = arr[i] + arr[j];
            arr[j] = arr[i] - arr[j];
            arr[i] = arr[i] - arr[j];
            i++;
        }
    }

    // swap(&arr[low], &arr[i - 1]);
    // Troca o pivô com o elemento na posição correta
    arr[low] = arr[i - 1];
    arr[i - 1] = pivot;
    return i - 1;
}

// Função principal do Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Encontra a posição do pivô
        int pi = partition(arr, low, high);

        // Recursivamente ordena os elementos antes e depois do pivô
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Figura 13. Pseudocódigo e Função Quick Sort.

Em relação à estabilidade, é importante notar que o Quick Sort não é um algoritmo estável. Isso significa que a ordem relativa de elementos iguais pode não ser preservada após a conclusão da ordenação.Quando se trata da eficiência do Quick Sort em arrays grandes, embora geralmente seja um método bastante eficaz, pode apresentar desempenho subótimo em cenários onde as arrays são muito extensas.

Um desafio significativo associado ao Quick Sort reside na escolha apropriada do pivô. A decisão sobre qual elemento será designado como pivô pode ter um impacto considerável no desempenho global do algoritmo. No contexto da análise de complexidade, o algoritmo Quick Sort revela diferentes desempenhos em cenários diversos. No Cenário Ótimo, onde a seleção estratégica do pivô é consistente, a complexidade do Quick Sort é de Ordem N log N. Este cenário ideal ocorre quando a escolha do pivô é sempre feita de maneira eficiente, resultando em um desempenho otimizado. No Cenário Médio, que representa situações típicas e comuns, o Quick Sort continua a exibir uma eficiência notável, mantendo uma complexidade de Ordem N log N. Este caso reflete a performance esperada na maioria das situações práticas. No Pior

Cenário, que é considerado raro e ocorre quando a escolha do pivô não é otimizada, a complexidade do Quick Sort pode atingir Ordem N ao quadrado. Este cenário adverso destaca uma situação em que a estratégia de escolha do pivô resulta em sub-listas extremamente desbalanceadas, prejudicando o desempenho geral do algoritmo.

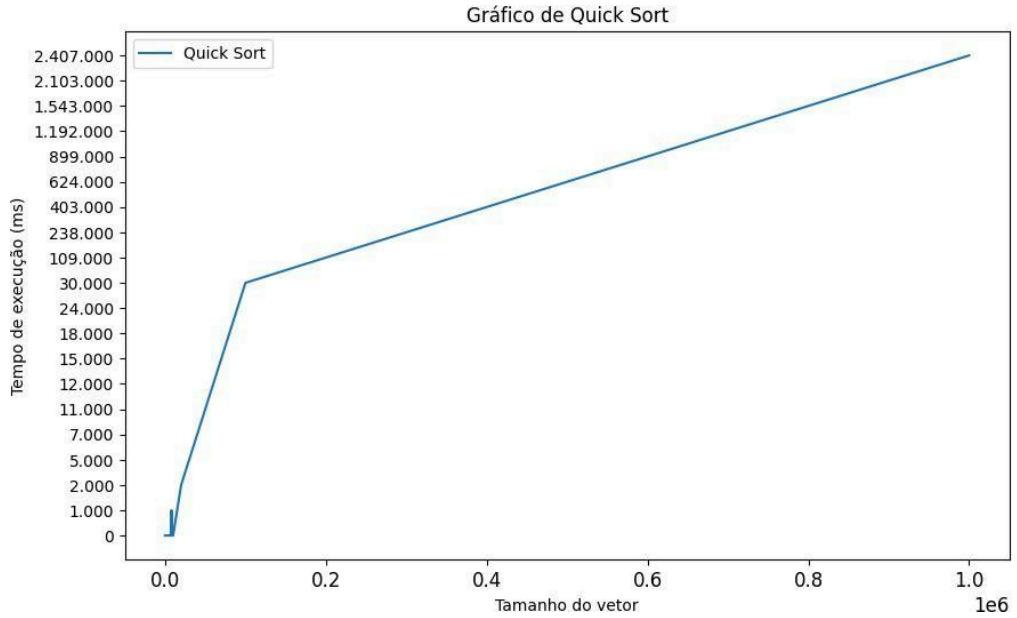
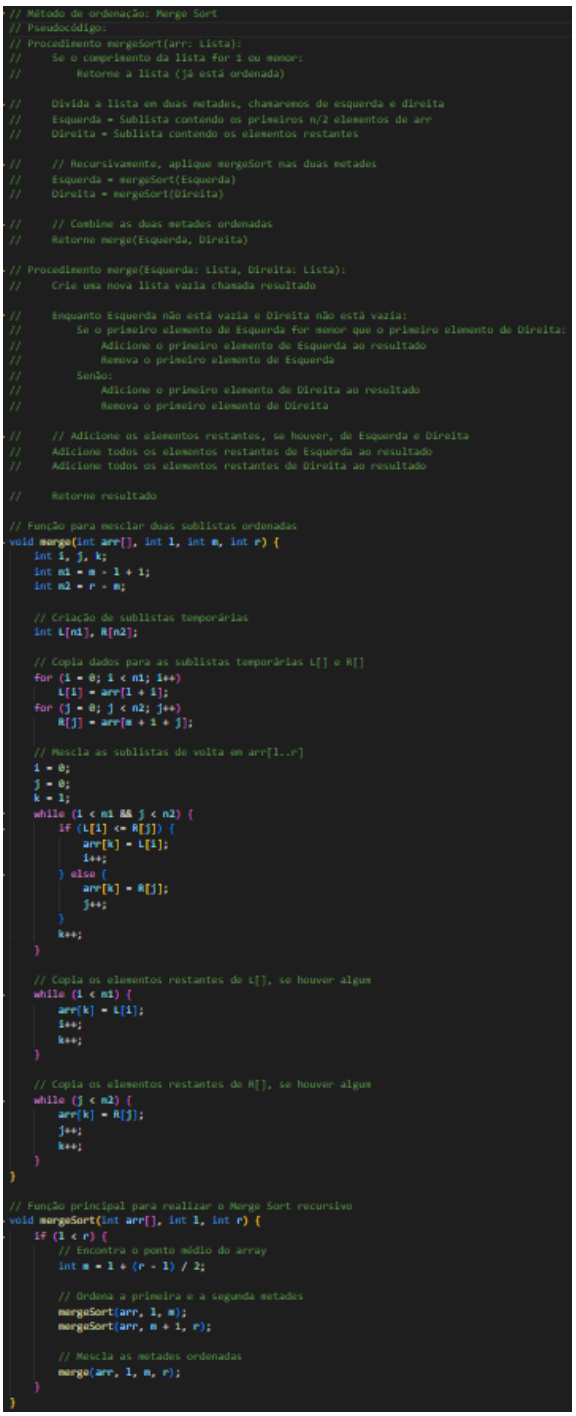


Figura 14. Grafico do Quick Sort.

3.5. Merge Sort

No contexto da ordenação de conjuntos de dados, o Merge Sort emerge como um algoritmo distinto, caracterizado por sua abordagem eficiente e estável. Este algoritmo fundamenta-se no princípio de dividir e conquistar, uma estratégia que se desdobra em diversas etapas intricadas.A estabilidade do Merge Sort é um dos seus atributos notáveis. Essa característica implica que o algoritmo preserva a ordem relativa dos elementos que compartilham chaves iguais durante o processo de ordenação. Essa propriedade é valiosa em situações em que manter a consistência na sequência ordenada é essencial.

A implementação do Merge Sort recorre à recursividade, um mecanismo que se revela fundamental para sua eficácia. O algoritmo divide o conjunto original em subconjuntos cada vez menores, persistindo nesse processo até que cada subconjunto contenha apenas um elemento. Posteriormente, realiza a fusão ordenada desses subconjuntos, construindo assim o array final. Uma consideração importante acerca do Merge Sort é sua necessidade de utilizar um vetor auxiliar durante o processo de intercalação. Embora essa prática contribua para sua eficiência, é vital ter ciência de que pode acarretar um aumento no consumo de memória.



```
// Método de ordenação: Merge Sort
// Pseudocódigo:
// Procedimento mergeSort(arr: Lista):
//     Se o comprimento da lista for 1 ou menor:
//         Retorne a lista (já está ordenada)

//     Divida a lista em duas metades, chamaremos de esquerda e direita
//     Esquerda = Sublista contendo os primeiros n/2 elementos de arr
//     Direita = Sublista contendo os elementos restantes

//     // Recursivamente, aplique mergeSort nas duas metades
//     Esquerda = mergeSort(Esquerda)
//     Direita = mergeSort(Direita)

//     // Combine as duas metades ordenadas
//     Retorne merge(Esquerda, Direita)

// Procedimento merge(Esquerda: Lista, Direita: Lista):
//     Crie uma nova lista vazia chamada resultado

//     Enquanto Esquerda não está vazia e Direita não está vazia:
//         Se o primeiro elemento de Esquerda for menor que o primeiro elemento de Direita:
//             Adicione o primeiro elemento de Esquerda ao resultado
//             Remova o primeiro elemento de Esquerda
//         Senão:
//             Adicione o primeiro elemento de Direita ao resultado
//             Remova o primeiro elemento de Direita

//     // Adicione os elementos restantes, se houver, de Esquerda e Direita
//     Adicione todos os elementos restantes de Esquerda ao resultado
//     Adicione todos os elementos restantes de Direita ao resultado

//     Retorne resultado

// Função para mesclar duas sublistas ordenadas
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    // Criação de sublistas temporárias
    int L[n1], R[n2];

    // Copia dados para as sublistas temporárias L[] e R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Mescla as sublistas de volta em arr[l..r]
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copia os elementos restantes de L[], se houver algum
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copia os elementos restantes de R[], se houver algum
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Função principal para realizar o Merge Sort recursivo
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Encontra o ponto médio do array
        int m = l + (r - l) / 2;

        // Ordena a primeira e a segunda metades
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Mescla as metades ordenadas
        merge(arr, l, m, r);
    }
}
```

Figura 15. Pseudocódigo e Função Merge Sort.

Na análise da complexidade computacional, o Merge Sort se destaca em diversas circunstâncias, apresentando diferentes cenários de desempenho. No Melhor Caso, observa-se que o algoritmo atinge sua máxima eficiência quando as fases de divisão inicial e subsequente fusão são executadas de maneira otimizada. Nesse contexto, a complexidade computacional assume a forma de O(N log N), evidenciando uma notável eficiência que se traduz em um desempenho superior. No Caso Médio, em situações típicas, o algoritmo mantém um desempenho eficaz, proporcionando uma complexidade

que é proporcional a N log N. Essa eficiência em situações cotidianas torna o Merge Sort uma escolha confiável e eficaz para uma variedade de conjuntos de dados. No Pior Caso, mesmo em cenários desafiadores, o Merge Sort exibe uma eficiência notável, garantindo uma ordenação eficaz. Nesses contextos mais adversos, a complexidade computacional também se mantém em O(N log N), evidenciando a robustez do algoritmo mesmo diante de desafios significativos.

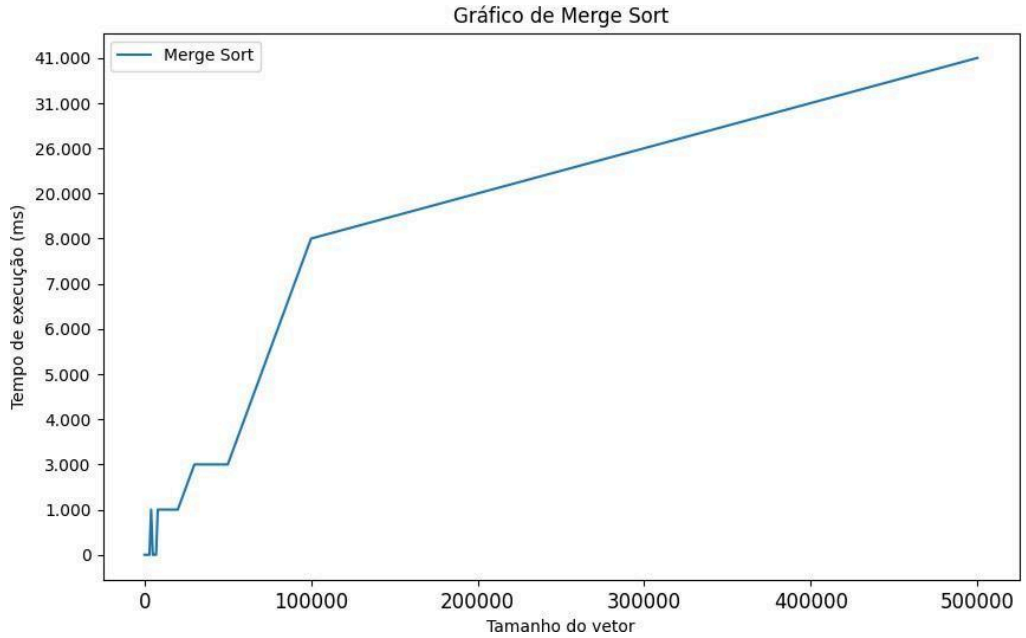


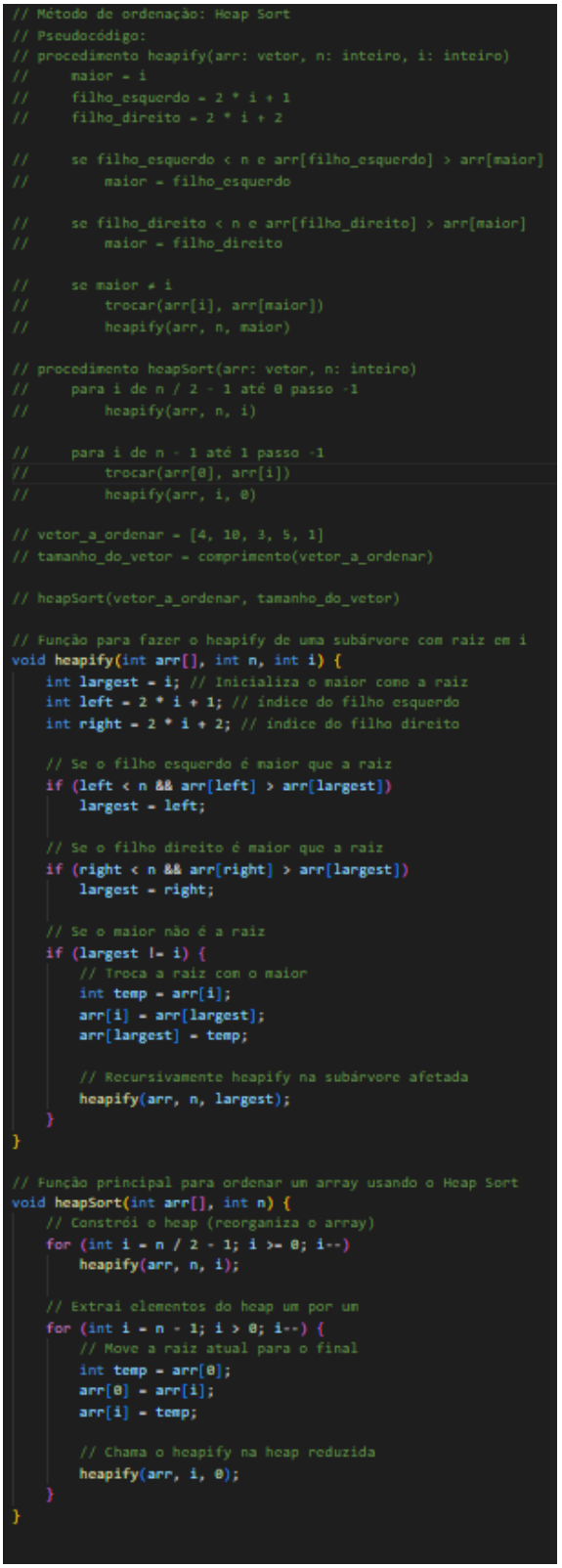
Figura 16. Grafico do Merge Sort.

O Merge Sort, embora eficaz para grandes conjuntos de dados, apresenta desafios. Sua abordagem não "in-place" demanda mais memória, causando problemas em ambientes restritos. A recursividade pode levar a estouros de pilha com arrays grandes, e para conjuntos pequenos, o Insertion Sort pode ter melhor desempenho. O Merge Sort foi até 500.000, antes de parar a execução.

3.5. Heap Sort

Heap Sort é um algoritmo de ordenação que se destaca por sua eficácia e utiliza uma estrutura de dados conhecida como heap. Essa estrutura é uma árvore binária especial em que o valor de cada nó é, no máximo (para um Max Heap) ou no mínimo (para um Min Heap), o valor de seus filhos. Esta característica fundamental contribui para a eficiência do algoritmo, tornando-o uma escolha sólida em diversas situações.

Diferentemente de algoritmos estáveis, Heap Sort não preserva a ordem relativa de elementos com chaves iguais, tornando-o instável. A versão iterativa do algoritmo é notavelmente mais rápida quando comparada à sua contraparte recursiva, proporcionando uma alternativa eficiente para situações em que a performance é uma prioridade. Entretanto, é crucial observar as desvantagens do Heap Sort. Sua natureza não estável e a exigência de um vetor auxiliar durante a ordenação podem impactar a escolha do algoritmo dependendo das características específicas do problema.



```
// Método de ordenação: Heap Sort
// Pseudocódigo:
// procedimento heapify(arr: vetor, n: inteiro, i: inteiro)
//     maior = i
//     filho_esquerdo = 2 * i + 1
//     filho_direito = 2 * i + 2

//     se filho_esquerdo < n e arr[filho_esquerdo] > arr[maior]
//         maior = filho_esquerdo

//     se filho_direito < n e arr[filho_direito] > arr[maior]
//         maior = filho_direito

//     se maior ≠ i
//         trocar(arr[i], arr[maior])
//         heapify(arr, n, maior)

// procedimento heapSort(arr: vetor, n: inteiro)
//     para i de n / 2 - 1 até 0 passo -1
//         heapify(arr, n, i)

//     para i de n - 1 até 1 passo -1
//         trocar(arr[0], arr[i])
//         heapify(arr, i, 0)

// vetor_a_ordenar = [4, 10, 3, 5, 1]
// tamanho_do_vetor = comprimento(vetor_a_ordenar)

// heapSort(vetor_a_ordenar, tamanho_do_vetor)

// Função para fazer o heapify de uma subárvore com raiz em i
void heapify(int arr[], int n, int i) {
    int largest = i; // Inicializa o maior como a raiz
    int left = 2 * i + 1; // índice do filho esquerdo
    int right = 2 * i + 2; // índice do filho direito

    // Se o filho esquerdo é maior que a raiz
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // Se o filho direito é maior que a raiz
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // Se o maior não é a raiz
    if (largest != i) {
        // Troca a raiz com o maior
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursivamente heapify na subárvore afetada
        heapify(arr, n, largest);
    }
}

// Função principal para ordenar um array usando o Heap Sort
void heapSort(int arr[], int n) {
    // Constrói o heap (reorganiza o array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extrai elementos do heap um por um
    for (int i = n - 1; i > 0; i--) {
        // Move a raiz atual para o final
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Chama o heapify na heap reduzida
        heapify(arr, i, 0);
    }
}
```

Figura 17. Pseudocódigo e Função Heap Sort.

Em termos de complexidade, Heap Sort exibe um desempenho consistente em diferentes cenários. Seu melhor caso, caso médio e pior caso têm complexidades de O(N

log N), destacando a previsibilidade e eficiência do algoritmo em diferentes contextos de aplicação.

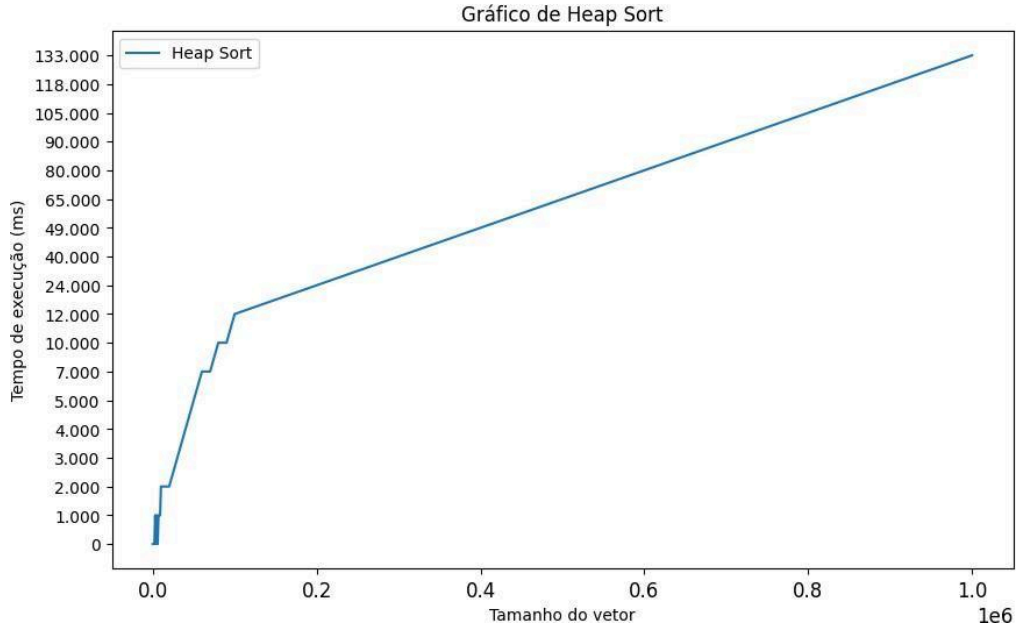


Figura 18. Grafico do Heap Sort.

4. Conclusão

Ao analisar o desempenho médio dos algoritmos de ordenação, destaca-se que o Merge Sort demonstrou consistentemente ser a opção mais eficiente em relação ao tempo médio de execução. Em contraste, o Insertion Sort apresentou um tempo médio mais elevado, indicando menor eficiência para conjuntos de dados de maior magnitude. Essa tendência é visível no gráfico, onde o Merge Sort exibiu um desempenho mais ágil, enquanto o Insertion Sort mostrou-se menos eficiente à medida que o tamanho do conjunto de dados aumentou.

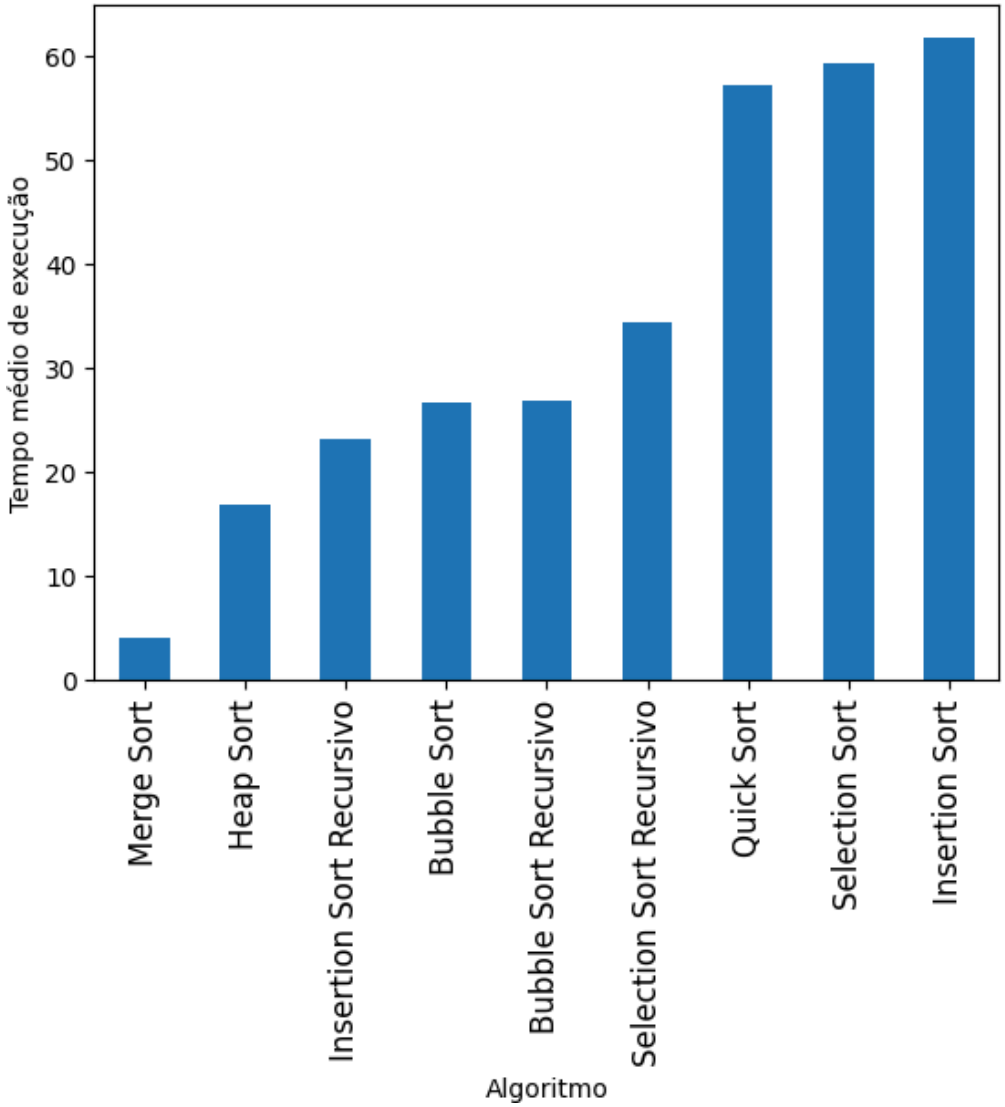


Figura 19. Gráfico do Tempo Médio.

Considerando a complexidade computacional, o Merge Sort manteve sua eficiência tanto no melhor quanto no pior caso, com uma complexidade de tempo de O(N log N). Essa característica reforça a robustez do algoritmo em lidar efetivamente com conjuntos de dados de variados tamanhos.

Porém, ao enfrentar a tarefa de ordenação, a escolha do algoritmo deve levar em consideração não apenas o tempo médio de execução, mas também as características específicas do problema, os requisitos de memória e o tamanho dos conjuntos de dados. O Merge Sort se destaca como uma escolha globalmente eficiente, embora contextos particulares possam demandar a consideração de algoritmos alternativos, no caso o Mege Sort não conseguiu ultrapassar vetores com mais de 5 mil de tamanho.

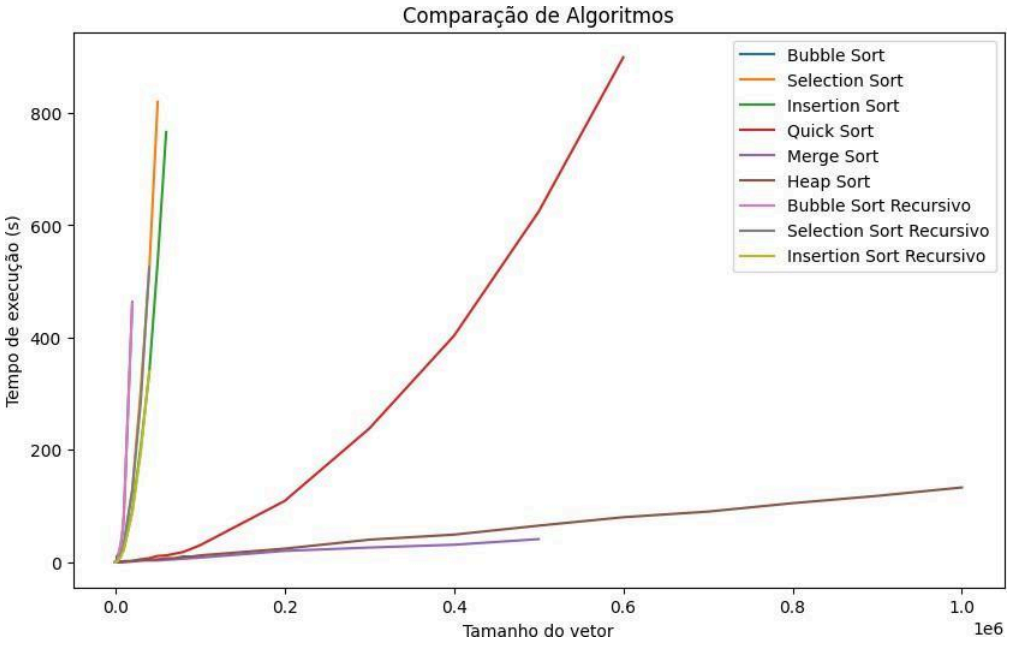


Figura 20. Comparação de Algoritmos.

Referências

Dayan F.A. (2024). Material usado na atividade. Disponivel em: https://github.com/DayanFA/Sistemas-de-Informacao-UFAC/tree/main/Estrutura%20de%20Dados/Atividade%202