UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE ENSINO SUPERIOR DO SERIDÓ
DEPARTAMENTO DE COMPUTAÇÃO E TECNOLOGIA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Trabalho Avaliativo para disciplina de Estrutura de Dados.

Felipe Augusto Araújo da Cunha

Caicó - RN
Julho de 2024

Felipe Augusto Araújo da Cunha

Trabalho Avaliativo para disciplina de Estrutura de Dados.

Trabalho apresentado a disciplina Estrutura de Dados do Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção da nota da avaliação I.

Orientador(a): Prof. Dr. João Paulo de Souza Medeiros.

Caicó - RN
Julho de 2024

RESUMO

O trabalho apresentado se baseia na implementação dos algoritmos solicitados pelo professor João Paulo. Este documento inclui os códigos dos algoritmos, juntamente com comentários e explicações detalhadas sobre cada um deles. Além disso, são apresentados os gráficos gerados para cada algoritmo, seguidos por uma comparação e análise dos tempos de execução.

Palavras-chave: Questões, Limites, Cálculo, Resoluções.

ABSTRACT

The work presented is based on the implementation of the algorithms requested by Professor João Paulo. This document includes the codes of the algorithms, along with detailed comments and explanations about each of them. In addition, the graphs generated for each algorithm are presented, followed by a comparison and analysis of the execution times.

Keywords: Questions, Limits, Calculation, Resolutions.

SUMÁRIO

1 INTRODUÇÃO 5

2 RESOLUÇÃO DOS CÓDIGOS 6

2.1 Selection Sort 6

2.1.1 Código do Selection Sort: 7

2.2 Distribution Sort 8

2.2.1 Código do Distribution Sort: 9

2.3 Merge Sort 10

2.3.1 Código do Merge Sort: 11

2.4 Quick Sort - Melhor Caso 12

2.4.1 Código do Quick Sort - Melhor Caso: 12

2.5 Quick Sort - Pior Caso 14

2.5.1 Código do Quick Sort - Pior Caso: 14

2.6 Quick Sort - Caso Médio 16

2.6.1 Código do Quick Sort - Caso Médio: 16

2.7 Comparação dos Casos do Quick Sort 18

2.7.1 Análise Comparativa 18

2.8 Insertion Sort - Melhor Caso 19

2.8.1 código do Insertion Sort: 20

2.9 Insertion Sort - Pior Caso 21

2.9.1 Código do Insertion Sort - Pior Caso: 21

2.10 Insertion Sort - Caso Médio 22

2.10.1 Código do Insertion Sort - Caso Médio 23

3 CONCLUSÃO 25

1 INTRODUÇÃO

Este trabalho tem como objetivo analisar e comparar diferentes algoritmos de ordenação, especificamente: Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Distribution Sort. Os códigos dos algoritmos foram desenvolvidos utilizando a linguagem de programação C, escolhida por sua eficiência e controle de baixo nível, o que facilita a análise de desempenho.

Para a visualização dos resultados, foram gerados gráficos utilizando a ferramenta Gnuplot. Esses gráficos mostram as estimativas práticas dos tempos de execução de cada algoritmo. No entanto, vale ressaltar que os gráficos não apresentaram a melhor formatação possível devido a limitações ao compilar e executar no sistema operacional Windows.

Além da implementação dos algoritmos e geração dos gráficos, este trabalho também inclui uma análise analítica do tempo de execução de cada algoritmo, bem como uma comparação de desempenho em termos de tempo e memória. Por fim, um relatório detalhado foi preparado, consolidando todas as observações e conclusões obtidas durante o estudo.

2 RESOLUÇÃO DOS CÓDIGOS

2.1 SELECTION SORT

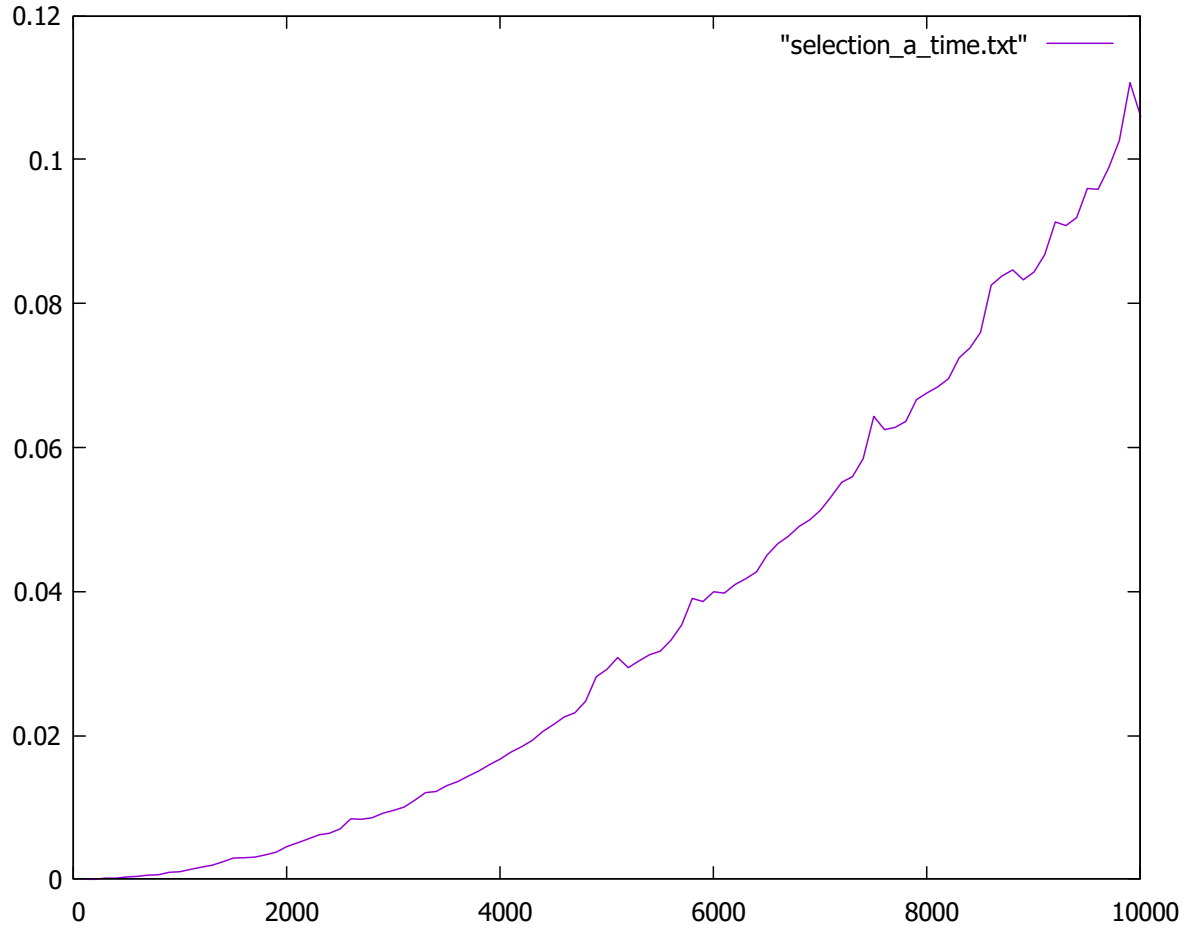

Figura 1 – Gráfico de tempo de execução para Selection Sort

ANÁLISE ANALÍTICA

O Selection Sort é um algoritmo simples de ordenação por comparação que funciona selecionando repetidamente o menor elemento da lista não ordenada e movendo-o para sua posição final. Este algoritmo possui uma complexidade de tempo de $O(n^2)$ no pior caso, onde n é o número de elementos na lista.

No código do Selection Sort abaixo, podemos observar sua estrutura básica:

2.1.1 CÓDIGO DO SELECTION SORT:

```
1. void selectionSort(int arr[], int n) {
2.     int i, j, minIndex, temp;
3.     for (i = 0; i < n - 1; i++) {
4.         minIndex = i;
5.         for (j = i + 1; j < n; j++) {
6.             if (arr[j] < arr[minIndex])
7.                 minIndex = j;
8.         }
9.         if (minIndex != i) {
10.            temp = arr[i];
11.            arr[i] = arr[minIndex];
12.            arr[minIndex] = temp;
13.        }
14.    }
15. }
```

Neste código:

- O laço externo percorre a lista da esquerda para a direita, considerando cada elemento como o potencial menor.
- O laço interno busca o menor elemento não ordenado à direita do índice atual.
- Se um elemento menor é encontrado, ele é trocado com o elemento na posição atual do laço externo.

O Selection Sort é estável e in-place, o que significa que não requer espaço extra e preserva a ordem relativa de elementos iguais. No entanto, sua eficiência é limitada para grandes conjuntos de dados devido à sua complexidade quadrática.

2.2 DISTRIBUTION SORT

O Distribution Sort, também conhecido como Bucket Sort, é um algoritmo de ordenação que divide os elementos do array em grupos chamados "buckets". Cada bucket é então ordenado separadamente, e os buckets ordenados são combinados para obter o array ordenado final.

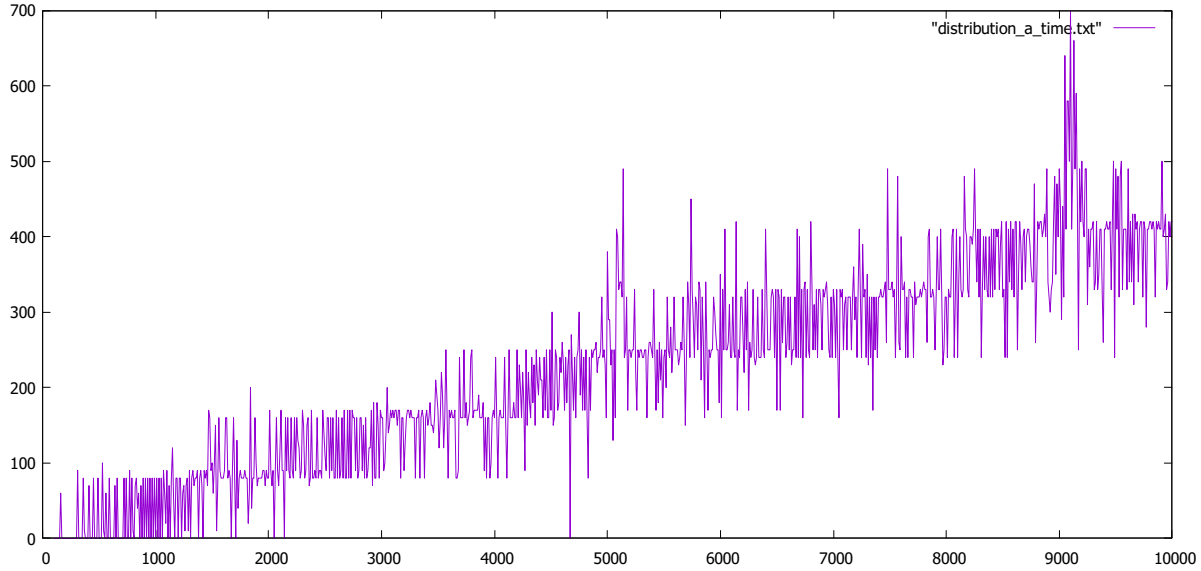

Figura 2 – Gráfico de tempo de execução para Distribution Sort

ANÁLISE ANALÍTICA

O Distribution Sort opera da seguinte maneira:

- **Identificação do Valor Máximo:** Primeiramente, o algoritmo itera pelo array para identificar o valor máximo presente.
- **Divisão em Buckets:** Com base no valor máximo encontrado, o array é dividido em vários buckets (ou recipientes). Cada bucket abrange um intervalo específico de valores.
- **Ordenação dos Buckets:** Cada bucket é ordenado individualmente. Isso pode ser feito usando um algoritmo de ordenação simples, como o Insertion Sort, devido à sua eficiência em pequenos conjuntos de dados.
- **Concatenação dos Buckets:** Por fim, os buckets ordenados são concatenados de volta na ordem correta para formar o array ordenado final.
- **Complexidade de Tempo:** O Distribution Sort possui uma complexidade de tempo média de $O(n + k)$, onde n é o número de elementos e k é o número de "buckets". Isso o torna ideal para casos onde os dados estão distribuídos uniformemente.

- **Eficiência de Espaço:** Requer espaço adicional dependendo do número de "buckets"utilizados, mas pode ser eficientemente implementado in-place com uso mínimo de memória adicional além de estruturas de controle.

2.2.1 CÓDIGO DO DISTRIBUTION SORT:

```
1. void distributionSort(int array[], int tamanho) {
2.     int max = array[0];
3.     for (int i = 1; i < tamanho; i++) {
4.         if (array[i] > max) {
5.             max = array[i];
6.         }
7.     }
8.     // Implementar a lógica de divisão em buckets e ordenação aqui
9. }
```

O Distribution Sort é uma escolha eficiente para ordenar grandes volumes de dados uniformemente distribuídos, aproveitando o princípio de divisão e conquista adaptado para distribuições específicas de valores.

2.3 MERGE SORT

O Merge Sort é um algoritmo de ordenação eficiente que utiliza o conceito de dividir para conquistar. Ele divide o array em duas metades, ordena cada metade recursivamente e depois combina as duas metades ordenadas para obter o array completamente ordenado.

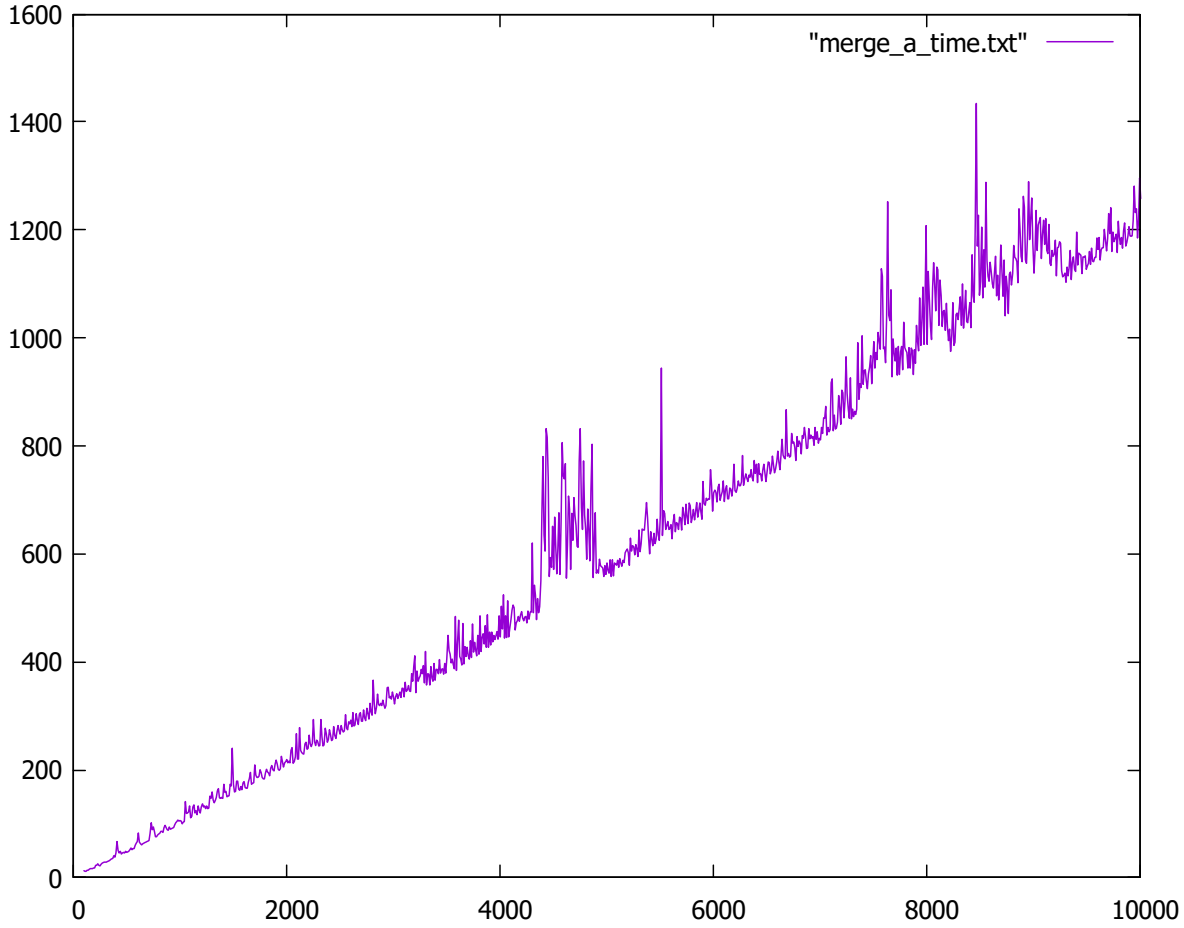

Figura 3 – Gráfico de tempo de execução para Merge Sort

ANÁLISE ANALÍTICA

O Merge Sort opera da seguinte maneira:

- **Divisão do Array:** Primeiramente, o array é dividido ao meio repetidamente até que cada subarray tenha um único elemento.
- **Ordenação Recursiva:** Cada par de subarrays é ordenado recursivamente utilizando o Merge Sort.
- **Combinação (Merge):** Os subarrays ordenados são mesclados (merged) de volta em ordem crescente. Isso é feito de forma eficiente, garantindo que o array resultante esteja ordenado.

- **Complexidade de Tempo:** O Merge Sort possui uma complexidade de tempo $O(n \log n)$, onde n é o número de elementos. Isso o torna ideal para grandes conjuntos de dados.

- **Eficiência de Espaço:** Requer espaço adicional para armazenar os subarrays durante o processo de ordenação, mas pode ser implementado de forma estável (stable sort) e com uso mínimo de memória adicional além das estruturas de controle.

2.3.1 CÓDIGO DO MERGE SORT:

```
1. void mergeSort(int vetor[], int comeco, int fim) {
2.     if (comeco < fim) {
3.         int meio = (fim + comeco) / 2;
4.
5.         mergeSort(vetor, comeco, meio);
6.         mergeSort(vetor, meio + 1, fim);
7.         merge(vetor, comeco, meio, fim);
8.     }
9. }
```

O Merge Sort é amplamente utilizado devido à sua eficiência e estabilidade na ordenação de grandes conjuntos de dados. Ele é particularmente eficaz quando é necessário garantir a ordem dos elementos e quando o desempenho em termos de tempo é crucial.

2.4 QUICK SORT - MELHOR CASO

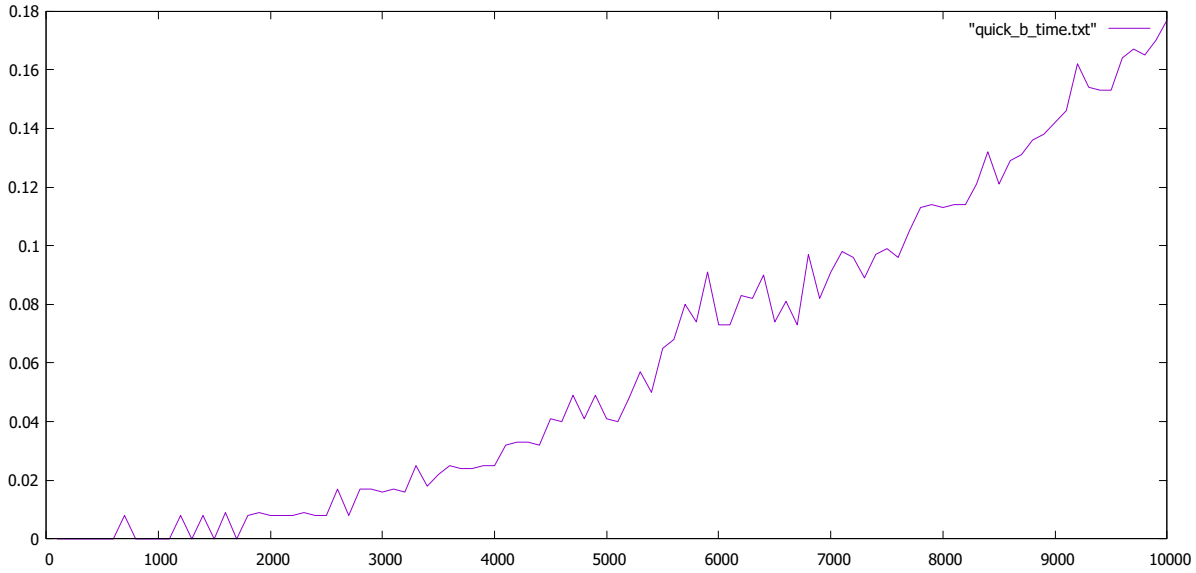

Figura 4 – Gráfico de tempo de execução para Quick Sort no melhor caso

ANÁLISE ANALÍTICA

O Quick Sort é um algoritmo de ordenação que utiliza a estratégia de dividir para conquistar. No melhor caso, o desempenho do Quick Sort se aproxima de $O(n \log n)$, onde n é o número de elementos na lista a ser ordenada. Isso ocorre quando o pivô escolhido divide o array de forma balanceada, dividindo-o aproximadamente pela metade a cada recursão.

No código abaixo, temos a implementação das funções auxiliares ‘swap‘ e ‘partition‘ necessárias para o Quick Sort:

2.4.1 CÓDIGO DO QUICK SORT - MELHOR CASO:

```
1. void swap(int* a, int* b) {
2.     int t = *a;
3.     *a = *b;
4.     *b = t;
5. }
6.
7. int partition(int arr[], int low, int high) {
8.     int pivot = arr[high];
9.     int i = (low - 1);
10.
```

```
11.     for (int j = low; j <= high - 1; j++) {
12.         if (arr[j] <= pivot) {
13.             i++;
14.             swap(&arr[i], &arr[j]);
15.         }
16.     }
17.     swap(&arr[i + 1], &arr[high]);
18.     return (i + 1);
19. }
```

Detalhamento da Análise

No melhor caso do Quick Sort:

- **Escolha Balanceada do Pivô:** O desempenho ideal do Quick Sort depende crucialmente da escolha do pivô. No melhor cenário, o pivô é escolhido de forma a dividir o array em duas partes aproximadamente iguais a cada passo recursivo. Isso minimiza o número de comparações e trocas necessárias para ordenar o array.
- **Complexidade $O(n \log n)$:** Devido à escolha balanceada do pivô, o Quick Sort alcança uma complexidade de tempo médio de $O(n \log n)$. Isso é muito eficiente para grandes conjuntos de dados e é uma das razões pelas quais o Quick Sort é amplamente utilizado em aplicações práticas.
- **Eficiência de Espaço:** O Quick Sort é in-place, o que significa que não requer espaço adicional significativo além da pilha de recursão. Isso o torna eficiente em termos de uso de memória comparado a algoritmos como o Merge Sort, que requer espaço adicional para fusão.

No entanto, é importante notar que o desempenho do Quick Sort pode degradar no pior caso ($O(n^2)$) se o pivô não dividir o array de forma balanceada. Em cenários práticos, estratégias para melhorar a escolha do pivô, como a escolha do pivô mediano ou aleatório, são frequentemente utilizadas para mitigar isso.

2.5 QUICK SORT - PIOR CASO

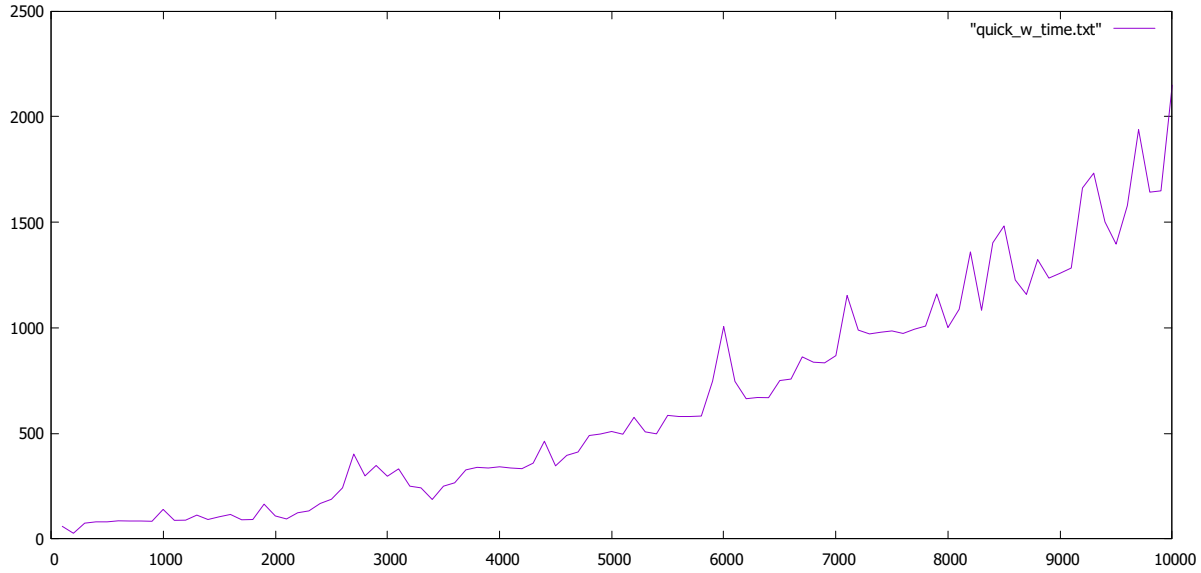

Figura 5 – Gráfico de tempo de execução para Quick Sort no pior caso

ANÁLISE ANALÍTICA

No pior caso, o desempenho do Quick Sort pode degradar para uma complexidade de $O(n^2)$, onde n é o número de elementos na lista. Isso ocorre quando o pivô escolhido não divide o array de forma balanceada, resultando em partições muito desiguais. Um exemplo típico é quando o pivô é escolhido como o maior ou o menor elemento do array em cada iteração.

No código abaixo, temos a implementação da função ‘partition‘ para o pior caso do Quick Sort:

2.5.1 CÓDIGO DO QUICK SORT - PIOR CASO:

```
1. int partition(int arr[], int low, int high) {
2.     int pivot = arr[high];
3.     int i = (low - 1);
4.
5.     for (int j = low; j <= high - 1; j++) {
6.         if (arr[j] < pivot) {
7.             i++;
8.             swap(&arr[i], &arr[j]);
9.         }
10.    }
```

```
11.     swap(&arr[i + 1], &arr[high]);
12.     return (i + 1);
13. }
```

Detalhamento da Análise

No pior caso do Quick Sort:

- **Escolha Desbalanceada do Pivô:** Se o pivô é sempre escolhido como o maior ou o menor elemento do array, o Quick Sort pode degenerar para um desempenho quadrático ($O(n^2)$). Isso ocorre porque a partição não divide o array de forma equilibrada, levando a muitas iterações desnecessárias.
- **Complexidade $O(n^2)$:** Quando o pivô não divide o array de forma balanceada, cada partição pode deixar apenas um elemento à esquerda ou à direita, resultando em n partições para um array de tamanho n. Isso resulta em uma complexidade de tempo quadrática no pior caso.
- **Impacto na Eficiência:** O desempenho do Quick Sort no pior caso pode ser severamente afetado em cenários onde os dados estão quase ou completamente ordenados, já que a escolha inadequada do pivô pode não aproveitar a estratégia de dividir para conquistar de forma eficiente.

Para mitigar o pior caso do Quick Sort, estratégias como a escolha do pivô mediano ou aleatório podem ser adotadas para tentar equilibrar melhor as partições. Essas estratégias ajudam a reduzir a probabilidade de ocorrência de partições desbalanceadas, melhorando assim o desempenho geral do algoritmo.

2.6 QUICK SORT - CASO MÉDIO

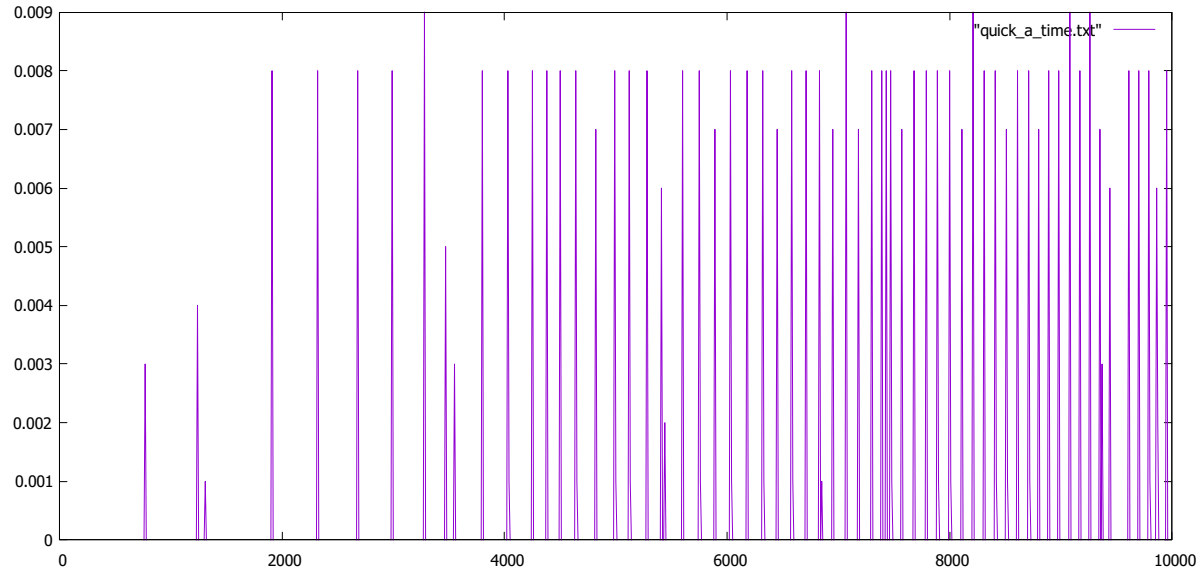

Figura 6 – Gráfico de tempo de execução para Quick Sort no caso médio

ANÁLISE ANALÍTICA

No caso médio, o Quick Sort possui um desempenho médio de $O(n \log n)$, onde n é o número de elementos na lista a ser ordenada. Este desempenho é alcançado quando o pivô divide o array de forma balanceada na maioria das partições.

No código abaixo, temos a implementação das funções ‘trocar‘, ‘particionar‘ e ‘quickSort‘ para o caso médio do Quick Sort:

2.6.1 CÓDIGO DO QUICK SORT - CASO MÉDIO:

```
1. void trocar(int* a, int* b) {
2.     int t = *a;
3.     *a = *b;
4.     *b = t;
5. }
6.
7. int particionar(int arr[], int baixo, int alto) {
8.     int pivo = arr[alto];
9.     int i = (baixo - 1);
10.
11.     for (int j = baixo; j < alto; j++) {
12.         if (arr[j] <= pivo) {
```

```
13.             i++;
14.             trocar(&arr[i], &arr[j]);
15.         }
16.     }
17.     trocar(&arr[i + 1], &arr[alto]);
18.     return (i + 1);
19. }
20.
21. void quickSort(int arr[], int baixo, int alto) {
22.     if (baixo < alto) {
23.         int pi = particionar(arr, baixo, alto);
24.         quickSort(arr, baixo, pi - 1);
25.         quickSort(arr, pi + 1, alto);
26.     }
27. }
```

Detalhamento da Análise

No caso médio do Quick Sort:

- **Divisão Balanceada do Pivô:** O desempenho médio $O(n \log n)$ é alcançado quando, em média, o pivô divide o array em duas partes quase iguais em cada iteração recursiva. Isso minimiza o número de comparações e trocas necessárias para ordenar o array.
- **Complexidade de Tempo:** A complexidade de tempo média $O(n \log n)$ do Quick Sort é ideal para ordenação eficiente de grandes conjuntos de dados. Ele supera muitos outros algoritmos de ordenação devido à sua eficiência na estratégia de dividir para conquistar.
- **Eficiência de Espaço:** O Quick Sort opera in-place, o que significa que não requer espaço adicional significativo além da pilha de recursão. Isso o torna eficiente em termos de uso de memória comparado a algoritmos como o Merge Sort.

O Quick Sort no caso médio representa a situação mais comum de desempenho eficiente, onde a estratégia de escolha do pivô resulta em partições balanceadas na maioria dos casos. Essa eficiência torna-o uma escolha popular para aplicações práticas de ordenação.

2.7 COMPARAÇÃO DOS CASOS DO QUICK SORT

Nesta seção, serão comparados os casos do Quick Sort: melhor caso, caso médio e pior caso. O gráfico abaixo mostra o tempo de execução em relação ao tamanho do vetor para cada caso.

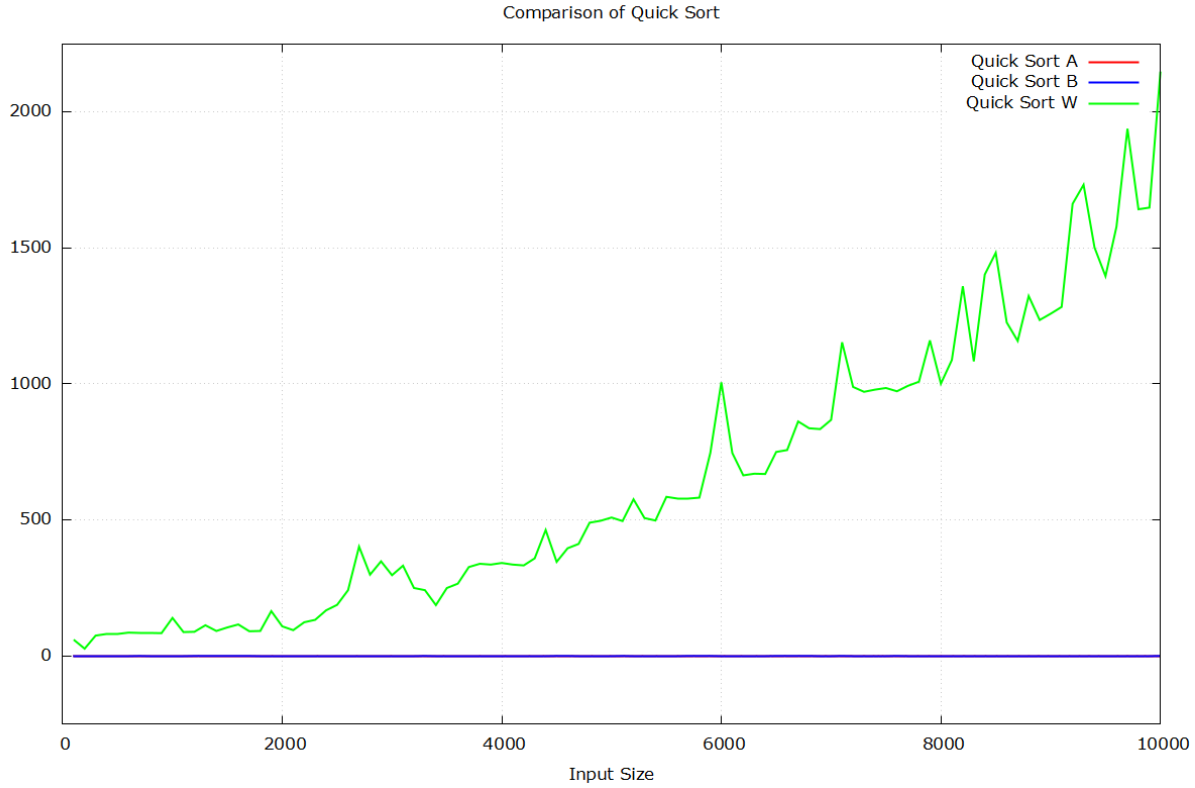

Figura 7 – Comparação de desempenho entre os casos do Quick Sort

2.7.1 ANÁLISE COMPARATIVA

O Quick Sort demonstra comportamentos distintos dependendo do caso:

- **Melhor Caso:** O melhor caso ocorre quando o pivô selecionado divide o array em duas partes quase iguais a cada iteração. Isso resulta em partições balanceadas, formando uma recursão em forma de árvore balanceada. O tempo de execução no melhor caso é tipicamente $O(n \log n)$, onde n é o número de elementos.

- **Caso Médio:** No caso médio, o Quick Sort tem um desempenho similar ao melhor caso na maioria das vezes. Ele divide o array em partições aproximadamente iguais, embora possam ocorrer variações ocasionais. O tempo de execução no caso médio também é $O(n \log n)$.

- **Pior Caso:** O pior caso ocorre quando o pivô escolhido divide o array de maneira desigual em cada iteração, por exemplo, sempre escolhendo o maior ou o menor

elemento como pivô. Isso resulta em partições muito desequilibradas, transformando a recursão em uma estrutura semelhante a uma lista encadeada. O tempo de execução no pior caso pode atingir $O(n^2)$, tornando o algoritmo significativamente mais lento conforme o número de elementos aumenta.

A análise dos casos do Quick Sort destaca a importância da escolha do pivô e suas consequências diretas no desempenho do algoritmo. Enquanto o melhor e o caso médio são geralmente eficientes para a maioria dos conjuntos de dados, o pior caso pode levar a um desempenho muito inferior, sendo crucial considerar estratégias para evitar este cenário em aplicações práticas.

Em resumo, o Quick Sort é um algoritmo poderoso quando implementado corretamente, oferecendo desempenho eficiente na maioria dos casos, mas requer atenção especial para evitar degradações significativas de desempenho no pior caso.

2.8 INSERTION SORT - MELHOR CASO

Nesta seção, será apresentada a análise do melhor caso do Insertion Sort, seguida pelo gráfico correspondente ao tempo de execução em relação ao tamanho do vetor.

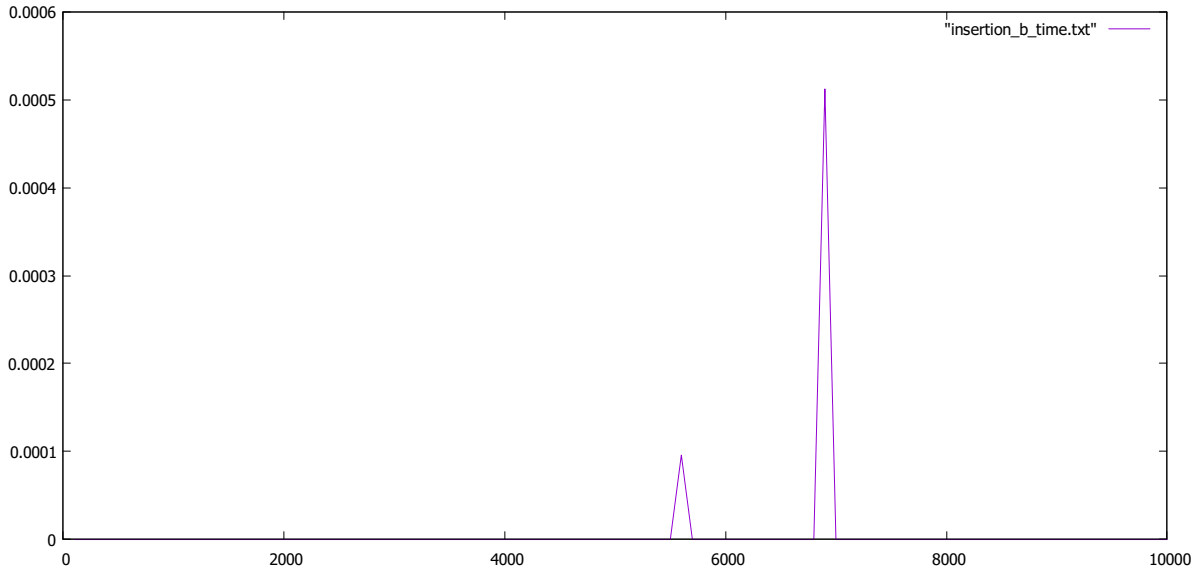

Figura 8 – Tempo de execução do Insertion Sort no melhor caso

ANÁLISE ANALÍTICA

No melhor caso, o Insertion Sort demonstra um desempenho otimizado quando o array está inicialmente ordenado ou quase ordenado. Graficamente, o algoritmo requer menos operações de comparação e troca de elementos, já que cada elemento novo é inserido na posição correta, deslocando-se apenas o necessário para manter a ordenação.

O tempo de execução no melhor caso é $O(n)$, onde n é o número de elementos no array. Isso ocorre porque, em cada iteração do loop externo, apenas uma comparação é necessária para determinar a posição correta do elemento atual em relação aos elementos já ordenados. Portanto, o Insertion Sort no melhor caso é ideal para conjuntos de dados que já estão parcialmente ou totalmente ordenados.

2.8.1 CÓDIGO DO INSERTION SORT:

```
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (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;
    }
}
```

Este código implementa o algoritmo Insertion Sort em C. Ele percorre o array a partir do segundo elemento ($i = 1$) até o último ($i = n - 1$), inserindo cada elemento na posição correta em relação aos elementos já ordenados à sua esquerda.

A eficiência do Insertion Sort no melhor caso o torna uma escolha viável para pequenos conjuntos de dados ou em situações onde a ordem parcial dos elementos é conhecida de antemão.

2.9 INSERTION SORT - PIOR CASO

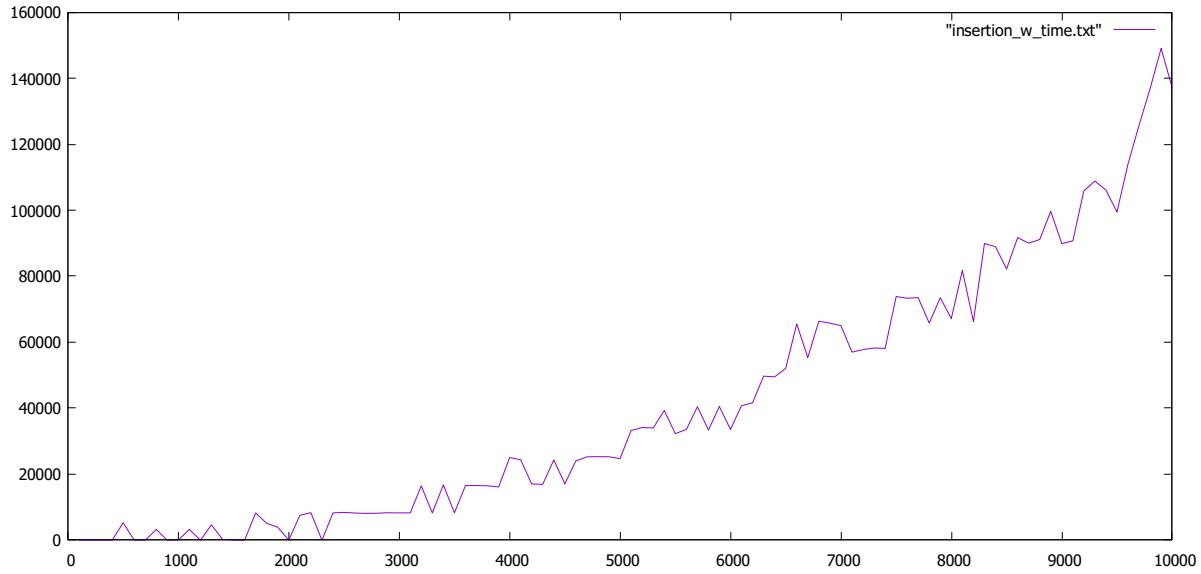

Figura 9 – Gráfico de tempo de execução para Insertion Sort no pior caso

ANÁLISE ANALÍTICA

No pior caso, o desempenho do Insertion Sort pode ser bastante impactado, especialmente quando o array está ordenado em ordem decrescente. Nesse cenário, cada elemento precisa ser movido para a posição correta no array ordenado, resultando em um número significativo de comparações e trocas.

No código abaixo, temos a implementação da função ‘insertionSort‘ para o pior caso do Insertion Sort, onde o array está ordenado em ordem decrescente:

2.9.1 CÓDIGO DO INSERTION SORT - PIOR CASO:

```
1. // Função para realizar o Insertion Sort em um vetor de tamanho "size"
2. void insertionSort(int arr[], int size) {
3.     for (int i = 1; i < size; i++) {
4.         int key = arr[i];
5.         int j = i - 1;
6.         while (j >= 0 && arr[j] > key) {
7.             arr[j + 1] = arr[j];
8.             j = j - 1;
9.         }
10.        arr[j + 1] = key;
11.    }
12. }
```

Detalhamento da Análise

No pior caso do Insertion Sort:

- **Array Ordenado em Ordem Decrescente:** Quando o array está ordenado de forma decrescente, cada elemento precisa ser movido para a sua posição correta no array ordenado. Isso resulta em muitas comparações e deslocamentos de elementos para trás.
- **Complexidade Quadrática:** O tempo de execução do Insertion Sort no pior caso é $O(n^2)$, onde n é o número de elementos no array. Isso ocorre porque, para cada elemento no array, pode ser necessário percorrer todos os elementos anteriores no pior cenário.
- **Impacto na Eficiência:** O desempenho do Insertion Sort no pior caso pode ser impraticável para grandes conjuntos de dados devido à sua complexidade quadrática. É menos eficiente em comparação com algoritmos como o Merge Sort ou Quick Sort no caso médio ou melhor.

Para melhorar o desempenho do Insertion Sort em cenários onde o array pode estar pré-ordenado de forma decrescente, estratégias alternativas de ordenação, como o Quick Sort ou o Merge Sort, podem ser preferíveis devido à sua complexidade de tempo mais eficiente.

2.10 INSERTION SORT - CASO MÉDIO

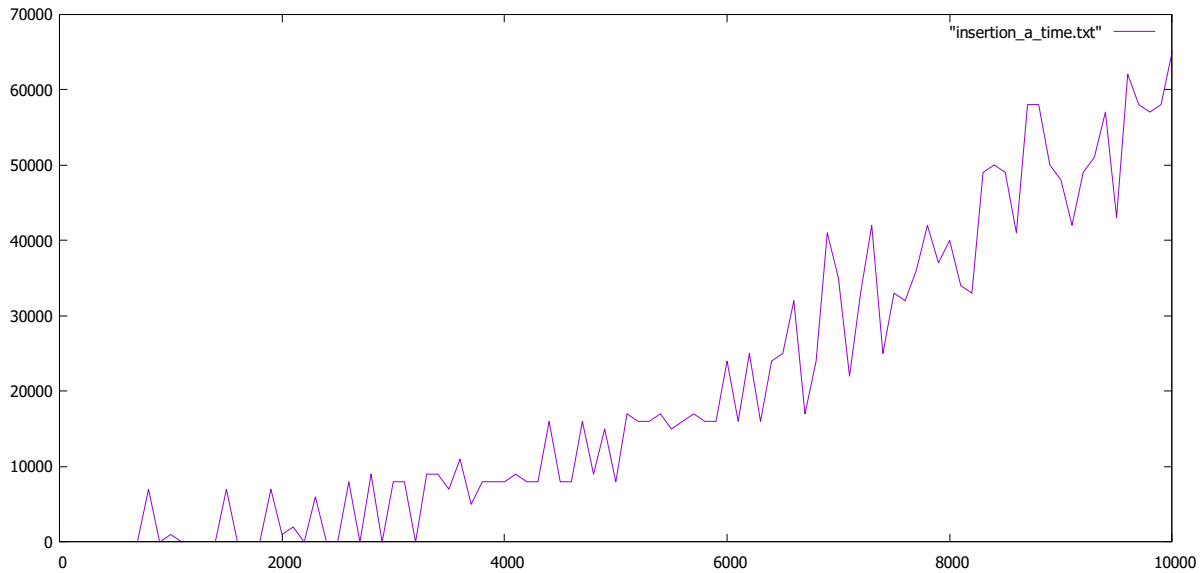

Figura 10 – Gráfico de tempo de execução para Insertion Sort no caso médio

ANÁLISE ANALÍTICA

No caso médio, o Insertion Sort demonstra um desempenho melhor em comparação com o pior caso, especialmente quando o array é aleatorizado. Nesse cenário, o tempo de execução do Insertion Sort é influenciado pelo número de inversões no array, ou seja, quantos elementos estão fora de ordem relativa.

No código abaixo, temos a implementação da função ‘insertionSort‘ para o caso médio do Insertion Sort, onde o array é aleatorizado:

2.10.1 CÓDIGO DO INSERTION SORT - CASO MÉDIO

```
1. // Função para realizar o Insertion Sort em um vetor de tamanho "size"
2. void insertionSort(int arr[], int size) {
3.     int i, key, j;
4.     for (i = 1; i < size; i++) {
5.         key = arr[i];
6.         j = i - 1;
7.
8.         while (j >= 0 && arr[j] > key) {
9.             arr[j + 1] = arr[j];
10.            j = j - 1;
11.        }
12.        arr[j + 1] = key;
13.    }
14. }
15. //...
16. //aleatorizando
17. for (int i = 0; i < size; i++) {
18.   arr[i] = rand();
19. }
20. //...
```

Detalhamento da Análise

No caso médio do Insertion Sort:

- **Array Aleatorizado:** Quando o array é aleatorizado, o tempo de execução do Insertion Sort é influenciado pelo número de inversões no array. Inversões referem-se ao número de pares de elementos que estão fora de ordem relativa.

- **Complexidade Quadrática:** Assim como no pior caso, o tempo de execução do Insertion Sort no caso médio é $O(n^2)$, onde n é o número de elementos no array. Isso ocorre porque o algoritmo pode precisar percorrer todos os elementos anteriores para inserir corretamente cada elemento no seu lugar.

- **Impacto na Eficiência:** Embora o caso médio seja melhor que o pior caso em termos de desempenho, o Insertion Sort ainda pode ser menos eficiente em comparação com algoritmos de ordenação como o Merge Sort ou Quick Sort no caso médio ou melhor.

No contexto de grandes conjuntos de dados, é importante considerar outras estratégias de ordenação que possam oferecer um desempenho mais consistente e eficiente, especialmente em cenários onde a aleatorização dos dados não é garantida.

3 CONCLUSÃO

Ao longo deste estudo, analisamos diversos algoritmos de ordenação, cada um com suas particularidades em relação ao desempenho em diferentes cenários. Abaixo, apresentamos um resumo das principais conclusões:

SELECTION SORT

O Selection Sort demonstrou ser um algoritmo simples e intuitivo, porém com um desempenho constante de $O(n^2)$ em todos os casos, o que o torna menos eficiente para grandes conjuntos de dados.

INSERTION SORT

O Insertion Sort, apesar de também possuir complexidade $O(n^2)$ no pior caso, mostra-se eficiente para pequenos conjuntos de dados ou em situações onde os elementos já estão quase ordenados, devido ao seu comportamento de inserção direta.

QUICK SORT

O Quick Sort mostrou-se altamente eficiente em média, com complexidade de tempo médio $O(n \log n)$. Seu desempenho é notável em comparação com outros algoritmos de ordenação, especialmente para grandes conjuntos de dados. No entanto, seu pior caso pode degradar para $O(n^2)$, tornando-o menos desejável em cenários onde a distribuição dos elementos é adversa.

MERGE SORT

O Merge Sort, com complexidade de tempo $O(n \log n)$ em todos os casos, demonstrou ser estável e eficiente para grandes volumes de dados, especialmente em situações onde a memória adicional para a mesclagem não é uma limitação.

DISTRIBUTION SORT

O Distribution Sort, utilizando buckets para distribuir os elementos, é eficiente quando os dados estão uniformemente distribuídos. No entanto, requer espaço adicional e pode não ser ideal para conjuntos de dados muito grandes ou com distribuições não uniformes.

COMPARAÇÃO E PREFERÊNCIAS

Para aplicações onde o desempenho médio é crucial e a distribuição dos dados não é conhecida antecipadamente, o Quick Sort e o Merge Sort são escolhas sólidas devido à sua complexidade média de $O(n \log n)$. Se a estabilidade é um requisito importante, o Merge Sort é preferível. Por outro lado, para pequenos conjuntos de dados ou situações onde a simplicidade e o espaço são críticos, o Insertion Sort pode ser mais adequado, especialmente se os dados já estão quase ordenados.

Em conclusão, a escolha do algoritmo de ordenação depende fortemente das características específicas do problema em mãos, como o tamanho do conjunto de dados, a distribuição dos elementos e as restrições de espaço e tempo. Cada algoritmo apresentado possui vantagens e desvantagens distintas, sendo essencial considerar esses fatores ao selecionar o mais adequado para uma aplicação específica.