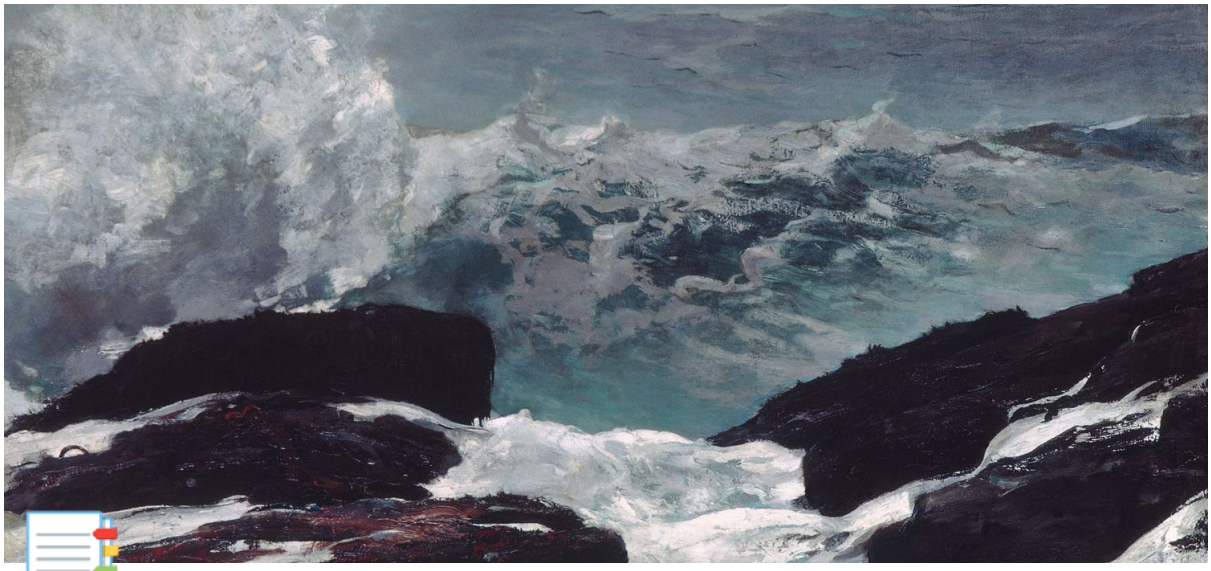

Relatório

Relatório do Trabalho Prático I - Projeto e Análise de Algoritmos - Universidade Federal de Ouro Preto

Integrantes

- Lucas de Araújo - 18.2.4049

- Fábio Henrique - 19.1.4128
- Lorrayne Cristine - 20.1.4009

Professor

- Dr. Anderson Almeida Ferreira

Resumo

Considerando a imensa quantidade de dados gerados diariamente em todo o mundo, os algoritmos de classificação desempenham um papel fundamental na simplificação da manipulação desses dados. Tais algoritmos são amplamente utilizados por programadores em tarefas tanto simples quanto complexas. Devido à alta demanda, existem várias abordagens lógicas distintas para a realização da tarefa de classificação. Portanto, é crucial compreender e estudar a eficiência desses algoritmos, a fim de selecionar o mais adequado para cada situação. Neste estudo, será realizada uma análise comparativa da eficiência de três algoritmos de classificação: **Insertion Sort**, **Merge Sort** e **Radix Sort**.

Pela comparação e testes estatísticos *t* pareado com 95% de confiança, pode-se concluir que o algoritmo Radix Sort possui o melhor desempenho geral entre os algoritmos selecionados e testados.

Introdução

Este trabalho consiste em implementar e analisar o desempenho de três algoritmos conhecidos de ordenação (Insertion Sort, Merge Sort e Radix Sort) a fim de melhor compreender o comportamento de cada algoritmo ao lidar com diferentes instâncias. Os algoritmos foram desenvolvidos na linguagem de programação C e cada algoritmo foi executado cerca vinte vezes para cada determinado tamanho de instância. Vale salientar que, cada uma das vinte execuções utilizam diferentes configurações, ou seja, os dados ordenados são diferentes em cada uma das vinte execuções e a mesma em cada respectivo algoritmo.

Para recolher uma quantidade significativa de dados a serem avaliados, foi gerado instâncias de seis tamanhos diferentes: 10, 100, 1.000, 10.000, 100.000, 1.000.000 utilizando a linguagem de programação Python em sua versão 3.10, totalizando 120 instâncias. Após todas as execuções, os dados foram coletados e armazenados em arquivos separados para que, posteriormente, utilizando novamente a linguagem Python, fossem gerados gráficos e tabelas para fornecer maior facilidade na interpretação dos resultados.

Todos os arquivos gerados para a realização deste trabalho estão presentes no arquivo enviado na plataforma Moodle e foram organizados em pastas mediante ao seu conteúdo. Segue a tabela de como foi configurado o respectivo projeto:

Diretório	Conteúdo
Src	Arquivos com implementação dos algoritmos em C
Input	Arquivos de entrada que serão lidos e ordenados
Output	Arquivos com os resultados de tempos dos algoritmos após ordenação de determinada instância
Utils	Multíplos *scripts* escritos na linguagem Python e Bash para gerar tabelas, gráficos e automatizar o processo de testes
Graphs	Gráficos gerados após coleta de resultados dos algoritmos
Tables	Tabelas geradas após coleta de resultado dos algoritmos

Insertion Sort

O Insertion Sort é um algoritmo de ordenação básico que se assemelha ao processo de ordenar um conjunto de cartas. Ao lidar com uma estrutura contendo vários elementos, o algoritmo busca o local ideal para inserir cada item, resultando em um subconjunto de elementos ordenados e outro não ordenado. A implementação do algoritmo é apresentada abaixo

```
void insertionSort(int *arr, int n)
{
    int i, key, j;

    for (i = 1; i < n; i++) // Custo 2 (por iteração)
    {

        key = arr[i]; // 1
        j = i - 1; // 1

        while (j >= 0 && arr[j] > key) // 3
        {
            arr[j + 1] = arr[j]; // 1
            j--; // 1
        }
```

```
        arr[j + 1] = key; // 1
    }
}
```

A análise de complexidade desta implementação do Insertion Sort segue:

- Loop Externo 2 operações: $O(n)$;
- Linhas dentro do loop externo, com exceção do while, executam: $3(n-1)$;
- Loop Interno (while) executa um total de $T_i + 1$ vezes, onde T_i é a execução das linhas presentes dentro do mesmo;
- Sendo assim, temos como total de execuções no loop interno:

$$\left(3 \cdot \sum_{i=1}^{n-1} T_i + 1\right) + \left(2 \cdot \sum_{i=1}^{n-1} T_i\right)$$

- Por fim, a complexidade do Insertion Sort é:

$$T(n) = 5n - 3 + \left(5 \cdot \sum_{i=1}^{n-1} T_i\right) + \left(3 \cdot \sum_{i=1}^{n-1} 1\right) = 8n - 6 + \left(5 \cdot \sum_{i=1}^{n-1} T_i\right)$$

$$T(n) = 8n - 6 + \left(5 \cdot \sum_{i=1}^{n-1} T_i\right)$$

- Complexidades:
  - Melhor Caso ($T_i = 0$):

$$T(n) = 8n - 6 + 0 = \Omega(n)$$

  - Caso Médio ($T_i = \frac{1}{i} * \sum_{j=0}^{i-1} j$):

$$T(n) = \frac{5n^2 + 17n - 14}{4} \rightarrow \theta(n^2)$$

  - Pior Caso ($T_i = i - 1$)

$$T(n) = \frac{5n^2 + n - 2}{2} \rightarrow O(n^2)$$

Merge Sort

O Merge Sort é um algoritmo de ordenação que utiliza o princípio de Dividir e Conquistar para atingir seu objetivo. Em cada chamada recursiva ou iteração, dependendo da implementação, a estrutura que contém os dados é dividida em duas partes e cada uma delas é resolvida separadamente. Após a resolução de todas as partes, elas são combinadas para resultar na estrutura ordenada. A implementaçaõ do algoritmo é apresentada a seguir:

```
void mergesort(int *v, int esq, int dir)
{
    if (esq < dir)
    {                                 // 1
        int meio = (esq + dir) / 2;  // 1
        mergesort(v, esq, meio);     // n/2
        mergesort(v, meio + 1, dir); // n/2
        merge(v, esq, meio, dir);    // 18n - 4 => O(n)
    }
}

void merge(int *v, int esq, int meio, int dir)
{
    int tamV1 = meio - esq + 1; // 1
    int tamV2 = dir - meio;     // 1

    int i; // 1
    int j; // 1
    int k; // 1

    int *V1 = malloc(tamV1 * sizeof(int)); // 1
    int *V2 = malloc(tamV2 * sizeof(int)); // 1

    // 3n - 2
    for (i = 0; i < tamV1; i++)  // 2 - n/2
        V1[i] = v[esq + i];      // 1 - (n/2 - 1)
    for (i = 0; i < tamV2; i++)  // 2 - n/2
        V2[i] = v[meio + 1 + i]; // 1 - (n/2 - 1)

    for (i = 0, j = 0, k = esq; k <= dir; k++) // (15n - 11)
    {
        if (i == tamV1)
        {
            v[k] = V2[j];
            j++;
        }
        else if (j == tamV2)
        {
            v[k] = V1[i];
            i++;
        }
        else if (V1[i] < V2[j])
        {
            v[k] = V1[i];
            i++;
```

```
        }
        else
        {
            v[k] = V2[j];
            j++;
        }
    }

    // 3n - 2 + 15n - 11 + 9
    // 18n - 4

    free(V1);
    free(V2);
}
```

A análise de complexidade desta implementação do Merge Sort segue:

- Funções
  - Merge: $O(18n - 4) = O(n)$
  - Mergesort: $T(n) = 2 * T(\frac{n}{2}) + O(n)$
- Utilizando Teorema Mestre, temos que:
  - $\log_2 2 = 1$
  - Expoente do custo local: $O(n^1)$
  - Por fim, a complexidade do Merge Sort é:

$$O(n * \log n)$$

- Tanto o pior, quanto melhor e caso médio do Merge Sort são: $O(n * \log n)$

Radix Sort

O Radix Sort é um algoritmo de classificação (ou ordenação) eficiente que opera nos dígitos individuais de um número ou nas posições dos caracteres em uma sequência. Ele se baseia no princípio de classificação estável, o que significa que a ordem relativa dos elementos com chaves idênticas é preservada durante todo o processo. A implementação do algoritmo é apresentada a seguir:

```
int findMax(int *arr, int size)
{
    int max = arr[0];
    for (int i = 1; i < size; i++) // n
        if (arr[i] > max)
            max = arr[i];
```

```
    return max;
}

void countingSort(int *arr, int size, int place)
{
    const int max = findMax(arr, size);
    int output[size];
    int count[max];

    for (int i = 0; i < max; ++i) // Custo: k + 1
        count[i] = 0;

    for (int i = 0; i < size; i++) // Custo: n + 1
        count[(arr[i] / place) % max]++;

    for (int i = 1; i < max; i++) // Custo: k
        count[i] += count[i - 1];

    for (int i = size - 1; i >= 0; i--) // Custo: n + 1
    {
        output[count[(arr[i] / place) % max] - 1] = arr[i];
        count[(arr[i] / place) % max]--;
    }

    for (int i = 0; i < size; i++) // Custo: n + 1
        arr[i] = output[i];
}

void radixsort(int *arr, int size)
{
    int max = findMax(arr, size);

    for (int place = 1; max / place > 0; place *= 10)
        countingSort(arr, size, place);
}
```

A análise de complexidade desta implementação do Radix Sort segue:

- Funções
  - findMax: $O(n)$
  - countingSort: $O(3n + 2k + 4) \rightarrow O(n + k)$
  - radixsort: $O(n) + P * O(n + k) \rightarrow P * O(n + k) \rightarrow O(n + k)$
- Para o melhor, pior e caso médio, teremos o Radix Sort como um algoritmo de ordem linear, ou seja, o Radix Sort possui complexidade $O(n)$

Análise dos Resultados

Como mencionado previamente, este trabalho teve como objetivo realizar a análise de desempenho dos três algoritmos que foram apresentados. Durante a realização dos testes, ao chegar em instâncias maiores que 1.00.000 o tempo de execução apresentou-se muito elevado para ser ordenado pelo algoritmo *Insertion Sort*, por conta disto, o tamanho máximo utilizado foi 1.000.000

Para cada execução, analisou-se o tempo necessário para cada algoritmo ordenar a estrtura com os dados fornecidos pela instância. Os dados coletados para cada um dos algoritmos estão representados nos gráficos no formato de **BoxPlot** a seguir onde o eixo *x* equivale ao tamanho da instância e o eixo *y* corresponde ao tempo gastos em segundos para completar a ordenação

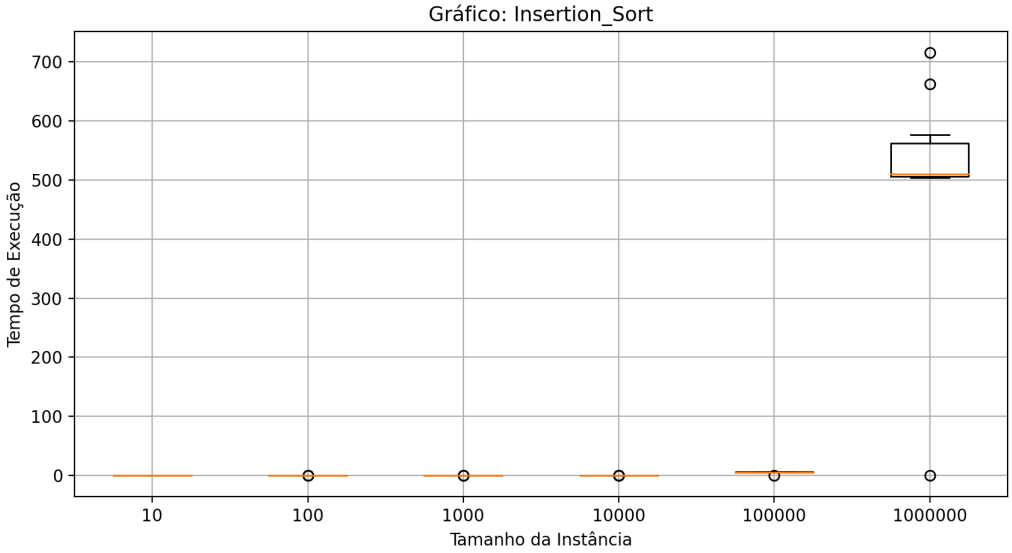

Gráfico do Insertion Sort

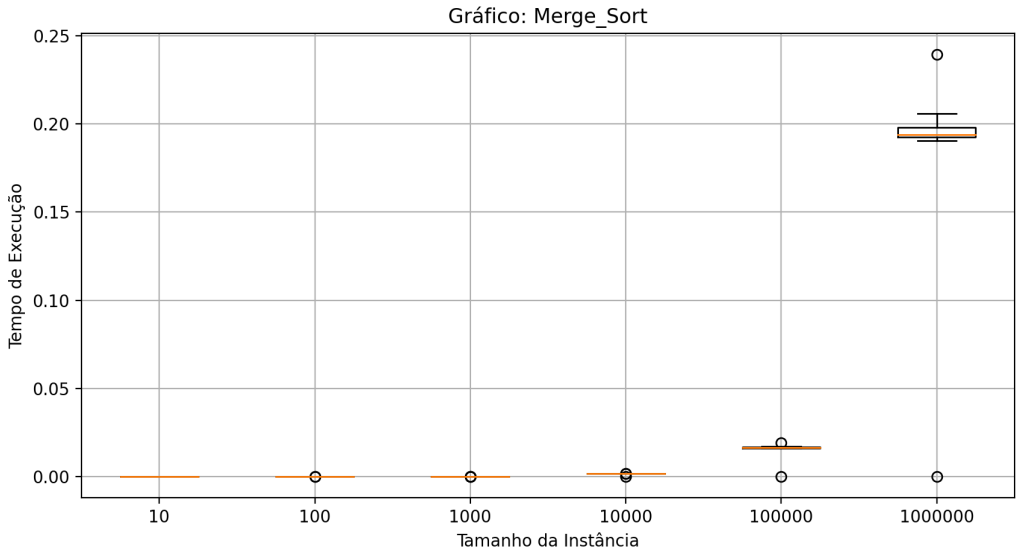

Gráfico do Merge Sort

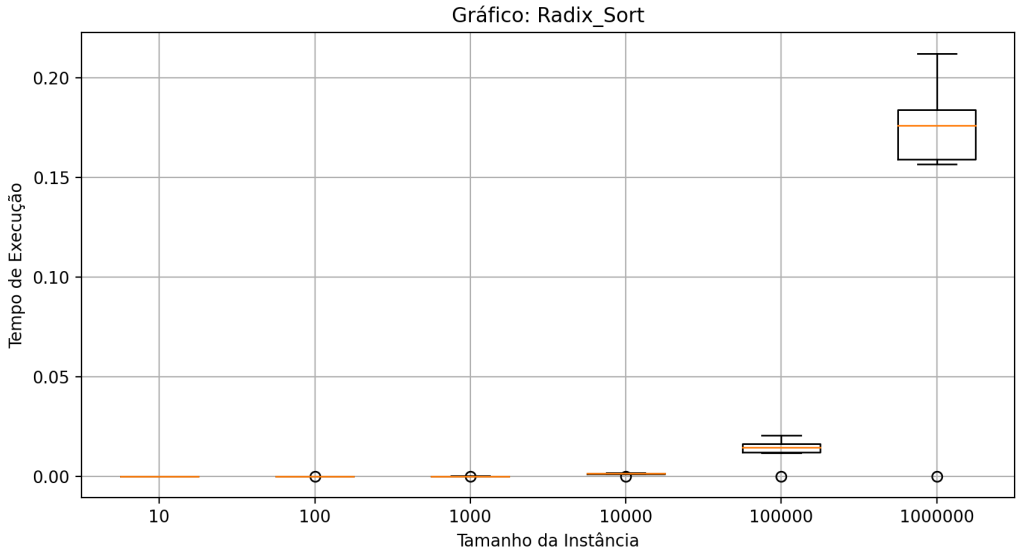

Gráfico do Radix Sort

Os gráficos a seguir, em formato de linhas, apresentão a comparação de desempenho entre os algoritmos

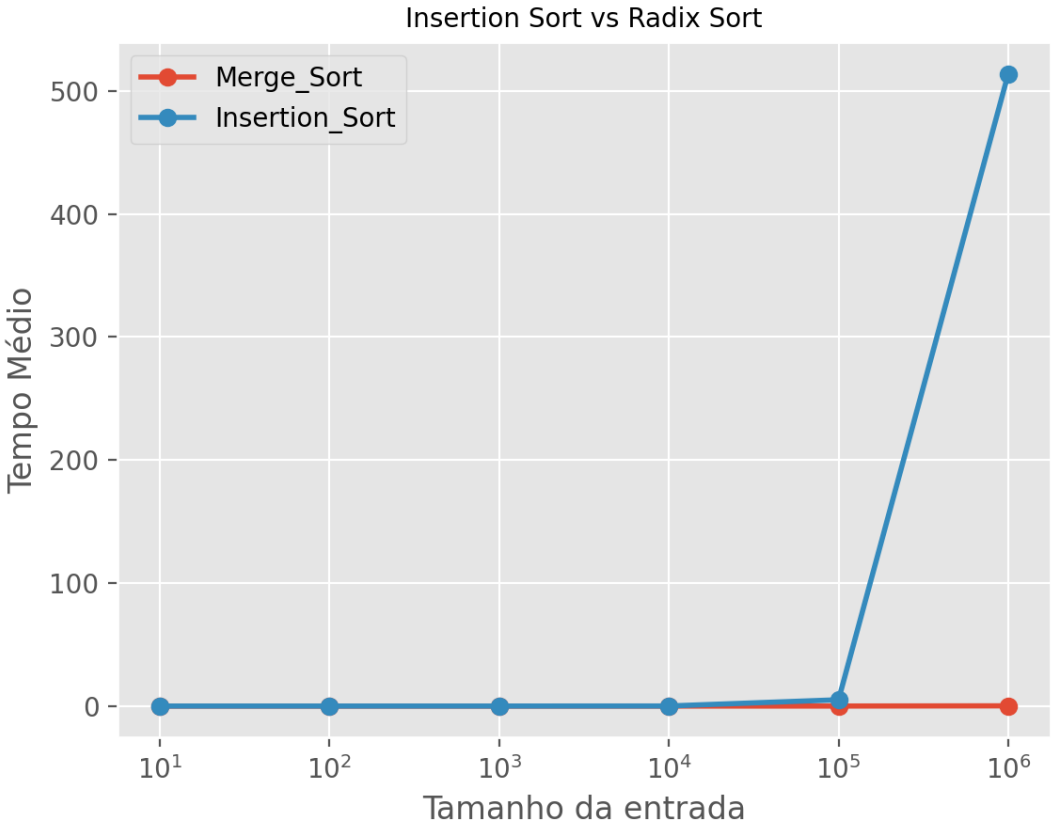

Gráfico comparativo entre Insertion Sort e Merge Sort

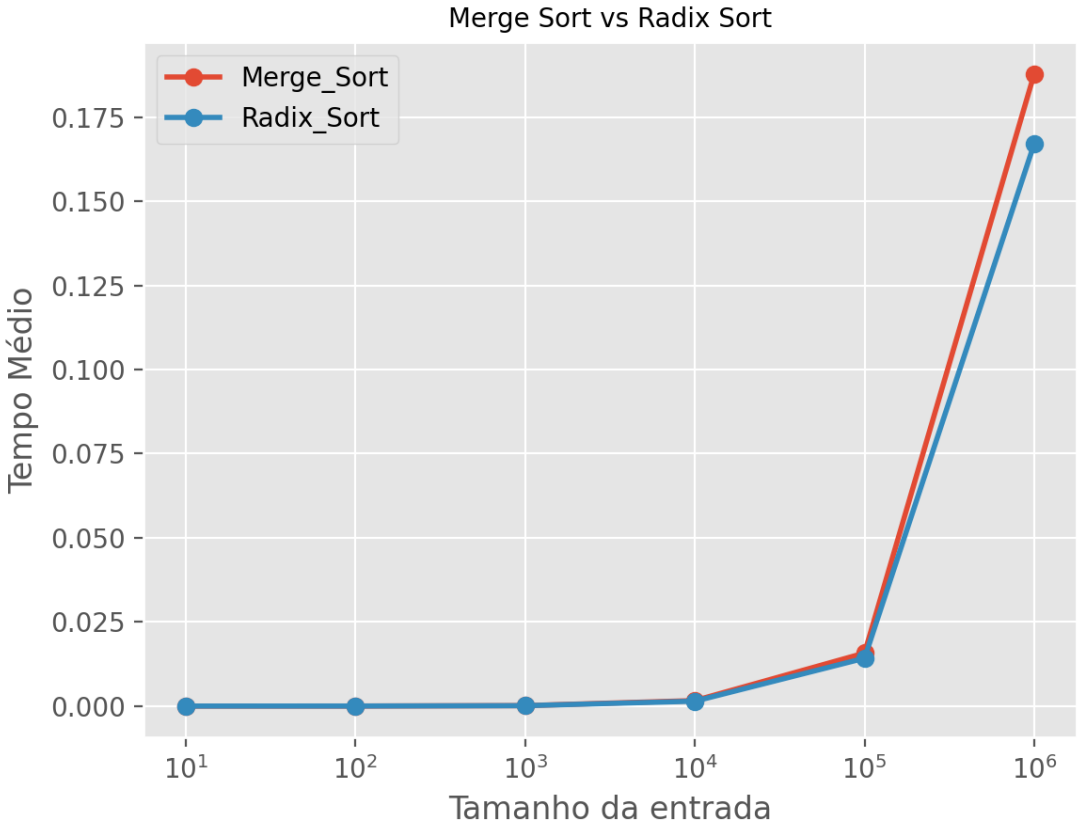

Gráfico comparativo entre Merge Sort e Radix Sort

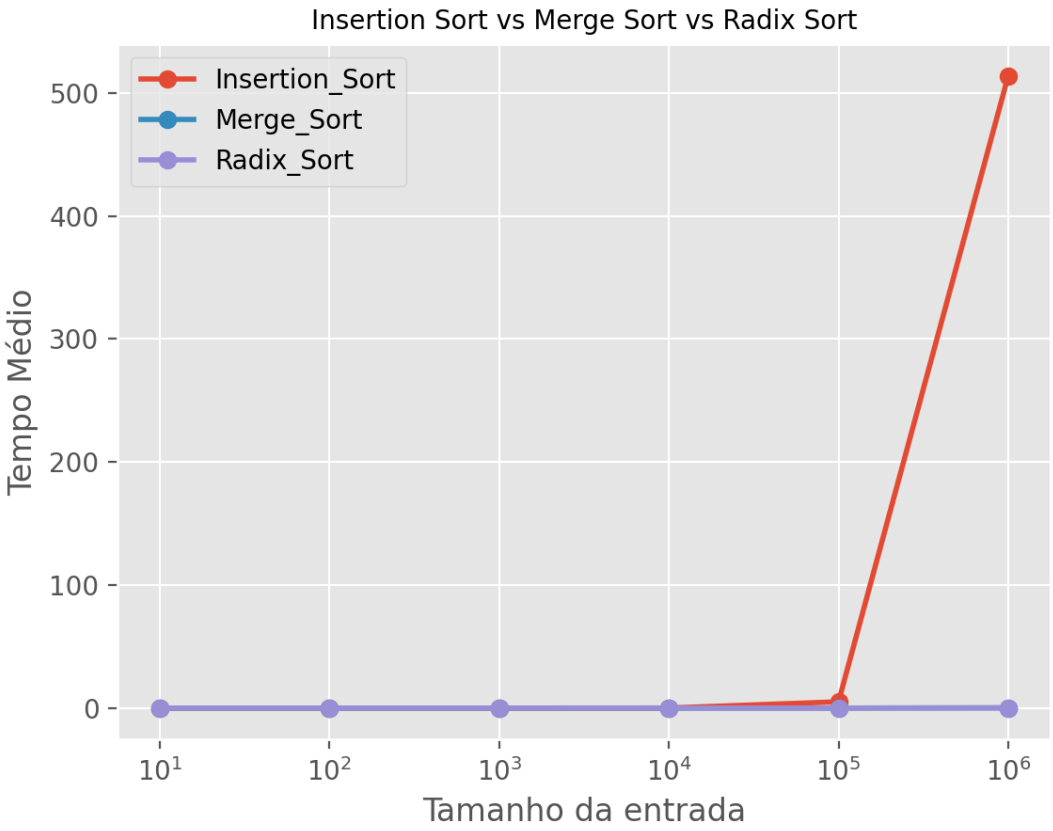

Gráfico comparativo entre Insertion Sort, Merge Sort e Radix Sort

Testes T Pareado

A fim de melhor comparar os algoritmos, em seguida foi analisado a tabela do intervalo de confiança dos métodos. Será apresentado na próxima image, uma tabela contendo o intervalo de confiança de cada algoritmo de forma separada para análise do tempo de execução para um dado tamanho. Para a execução deste Teste T Pareado, utilizou-se confiança de 95% junto a $t = 2.093$ e $\alpha = 0.05$

Instância	10	100	1000	10000	100000	1000000
Insertion Sort	(0.0000; 0.0000)	(0.0000; 0.0000)	(0.0006; 0.0008)	(0.0443; 0.0559)	(4.5289; 5.6972)	(451.2725; 576.4651)
Merge Sort	(0.0000; 0.0000)	(0.0000; 0.0000)	(0.0001; 0.0001)	(0.0015; 0.0018)	(0.0140; 0.0175)	(0.1665; 0.2091)
Radix Sort	(0.0000; 0.0000)	(0.0000; 0.0000)	(0.0001; 0.0001)	(0.0013; 0.0016)	(0.0122; 0.0162)	(0.1472; 0.1869)

Valores resultante do Teste-T Pareado com confiança 95% para cada algoritmo

Conclusões do Teste Estatístico

Podemos perceber pelo teste estatístico que:

- Ainda que para valores pequenos onde a instância é inferior a 1.000, o algoritmo Insertion Sort apresenta resultados similares ao dos outros algoritmos, porém, torna-se inviável quando valores altos de instância são utilizados
- Os algoritmos Merge Sort e Radix Sort são similares na maioria dos casos, porém o algoritmo Radix Sort, devido a sua complexidade linear, apresentou resultados levemente superiores a medida que grandes valores de instância foram utilizados

Conclusões Finais

Por meio de avaliações e estudos realizados, conclui-se que dentre os três algoritmos, o Insertion Sort apresenta o pior rendimento, uma observação que é corroborada por sua ordem de complexidade no pior cenário, que é $O(n^2)$. Tanto o MergeSort quanto o Radix Sort têm performances comparáveis, pois o tempo que eles necessitam para completar a organização permanece semelhante conforme o aumento do tamanho das instâncias. No entanto, o Radix Sort tem uma leve vantagem sobre o MergeSort, o que também é evidenciado por suas ordens de complexidade, que são $O(n)$ e $O(n \times \log n)$, respectivamente.

Contudo, para instâncias menores, a eficácia dos três algoritmos é quase idêntica, sendo que apenas a partir da instância de tamanho 10.000 que a diferença no tempo requerido para a ordenação entre os algoritmos se torna notável.

Bibliografia

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms. MIT press.
- Sedgewick, R., & Wayne, K. (2011). Algorithms, Part I [and II]. Addison-Wesley Professional.
- Skiena, S. S. (2008). The algorithm design manual. Springer Science & Business Media.
- Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008). Algorithms. McGraw-Hill Education.
- Cormen, T. H. (2013). Algorithms unlocked. MIT press.
- MELO, Diego. O que é Python? [Guia para iniciantes] – Tecnoblog. Disponível em: **https://tecnoblog.net/responde/o-que-e-python-guia-para-iniciantes/**. Acesso em: 9 jun. 2023.
- W3SCHOOLS. Matplotlib Pyplot. Disponível em: **https://www.w3schools.com/python/matplotlib_pyplot.asp**. Acesso em: 9 jun. 2023.