



# **Apresentação**

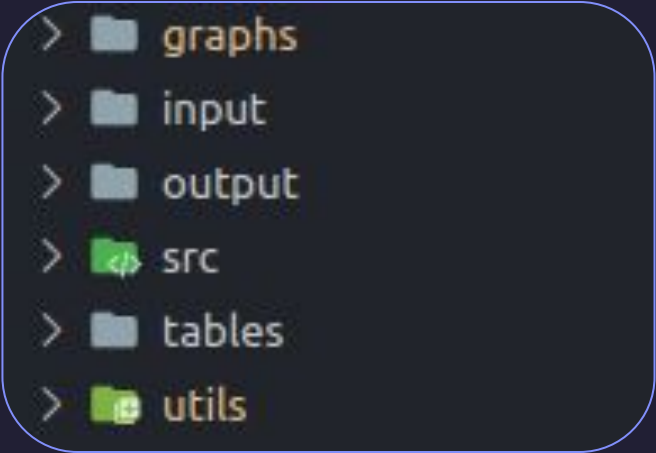
# **TRABALHO PRÁTICO I – PAA**







Lucas Araújo (18.2.4049) – Fábio Henrique (19.1.4128) – Lorryne Cristine (20.1.4009)

# Conteúdo

<b>Arquitetura do Projeto</b>	Estrutura do projeto
<b>Insertion Sort - Análise</b>	Código, ordem de complexidade e derivados
<b>Merge Sort - Análise</b>	Código, ordem de complexidade e derivados
<b>Radix Sort - Análise</b>	Código, ordem de complexidade e derivados
<b>Gráficos e Testes T</b>	Gráficos de cada algoritmo e comparações entre os mesmos
<b>Conclusões</b>	Considerações finais

# Arquitetura do Projeto



- >  **graphs**
- >  **input**
- >  **output**
- >  **src**
- >  **tables**
- >  **utils**

- **Graphs:** Gráficos finais;
- **Tables:** Tabelas finais;
- **Input:** Arquivos de entrada desordenados;
- **Output:** Arquivos com médias de tempo entre algoritmos;
- **Src:** Implementações dos algoritmos;
- **Utils:** Scripts auxiliares utilizados para facilitar a análise de resultados.

# Arquivo de Entrada x Arquivo de Saída

```
1 89973
2 992171
3 513518
4 858977
5 9819
6 771277
7 134799
8 902307
9 245214
10 342279
11 746818
12 642420
13 141195
14 24444
15 579903
16 767758
17 56004
18 375588
19 922528
20 948793
```

```
1 Insertion Sort
2
3 Instancia 1: 0.000000
4 Instancia 2: 662.993823
5 Instancia 3: 508.469191
6 Instancia 4: 504.844155
7 Instancia 5: 505.889091
8 Instancia 6: 562.695471
9 Instancia 7: 576.265676
10 Instancia 8: 573.041562
11 Instancia 9: 715.669969
12 Instancia 10: 562.355105
13 Instancia 11: 526.035569
14 Instancia 12: 518.497669
15 Instancia 13: 509.346076
16 Instancia 14: 508.489740
17 Instancia 15: 510.169351
18 Instancia 16: 513.445274
19 Instancia 17: 503.814139
20 Instancia 18: 505.601101
21 Instancia 19: 505.906720
22 Instancia 20: 503.846755
23
24 Tempo Médio: 513.868822
```

# Insertion Sort

# Insertion Sort - Código

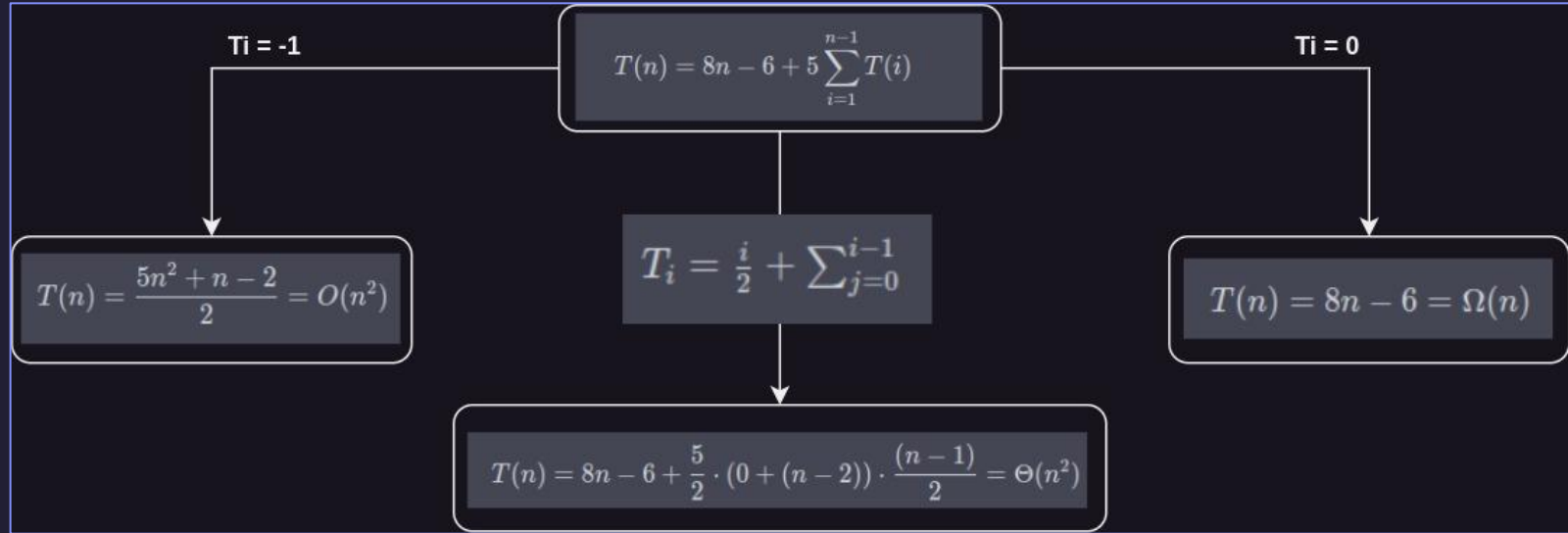
```
void insertionSort(int *arr, int n)
{
    int i, key, j;

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

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

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

# Insertion Sort - Complexidade





# Merge Sort

# Merge Sort - Código

```
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); //  $18n - 4 \Rightarrow O(n)$ 
    }
}
```

# Merge Sort - Código

```
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++) //  $(15n - 11) \dots$ 
```

```
        //  $3n - 2 + 15n - 11 + 9$ 
```

```
        //  $18n - 4$ 
```

```
    free(V1);
```

```
    free(V2);
```

```
}
```

```
    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
```

```
        {
```

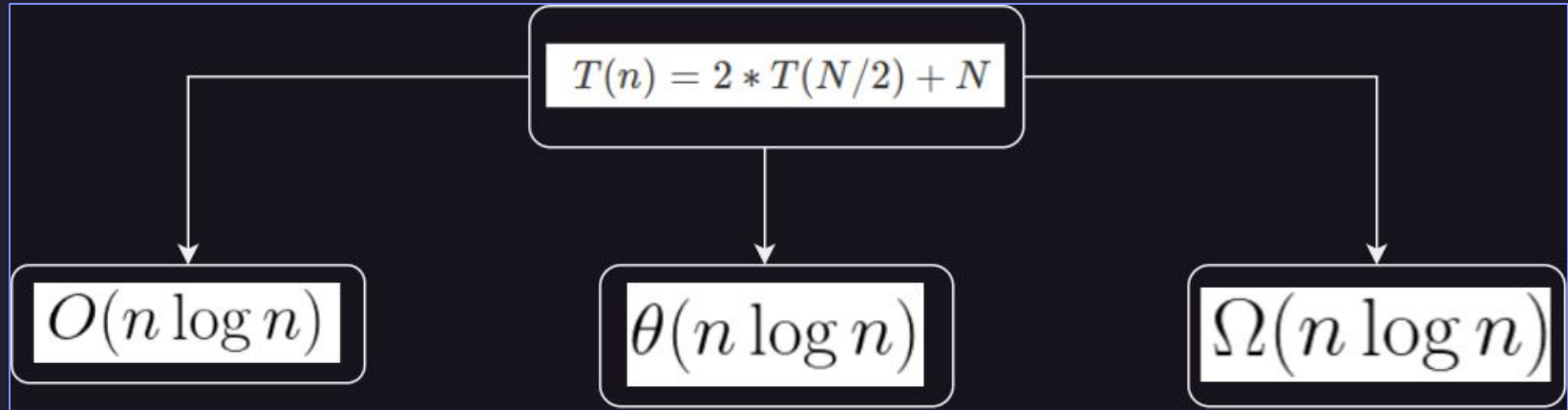
```
            v[k] = V2[j];
```

```
            j++;
```

```
        }
```

```
    }
```

## Merge Sort - Complexidade



# Radix Sort

# Radix Sort - Código

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

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

```
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];

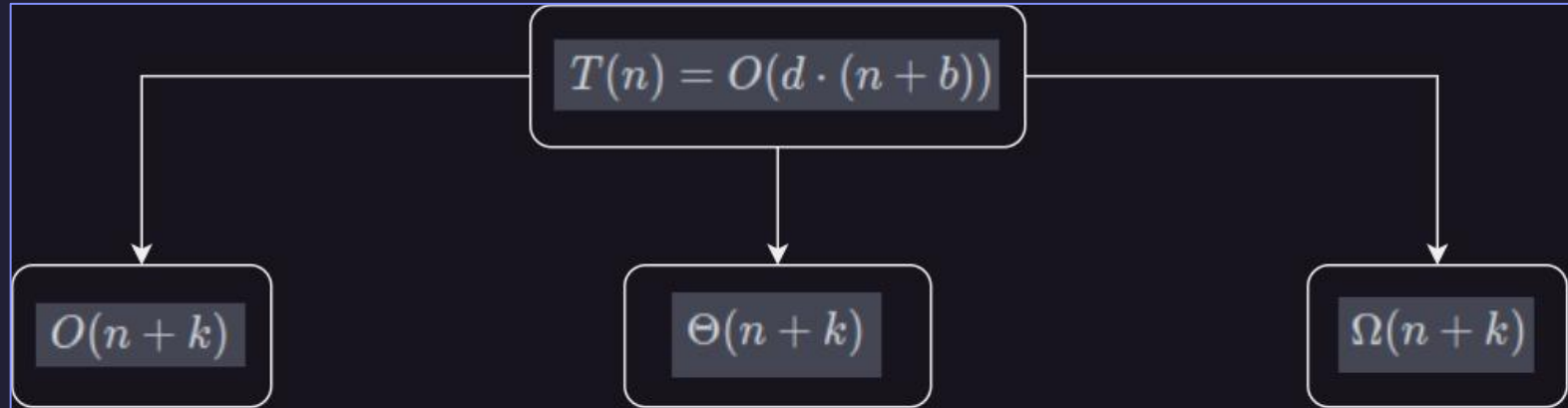
    for (int i = size - 1; i ≥ 0; i--) // Custo: n + 1
    {
        output[count[(arr[i] / place) % max] - 1] = arr[i];
        count[(arr[i] / place) % max]--;
    }

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

## Radix Sort - Complexidade

FindMax	CountingSort	RadixSort
$O(n)$	$O(3n + 2k + 4) = O(n + k)$	$O(n) + P * O(n + k) = O(n + k)$

## Radix Sort - Complexidade





# Gráficos e Testes T

Gráfico: Insertion\_Sort

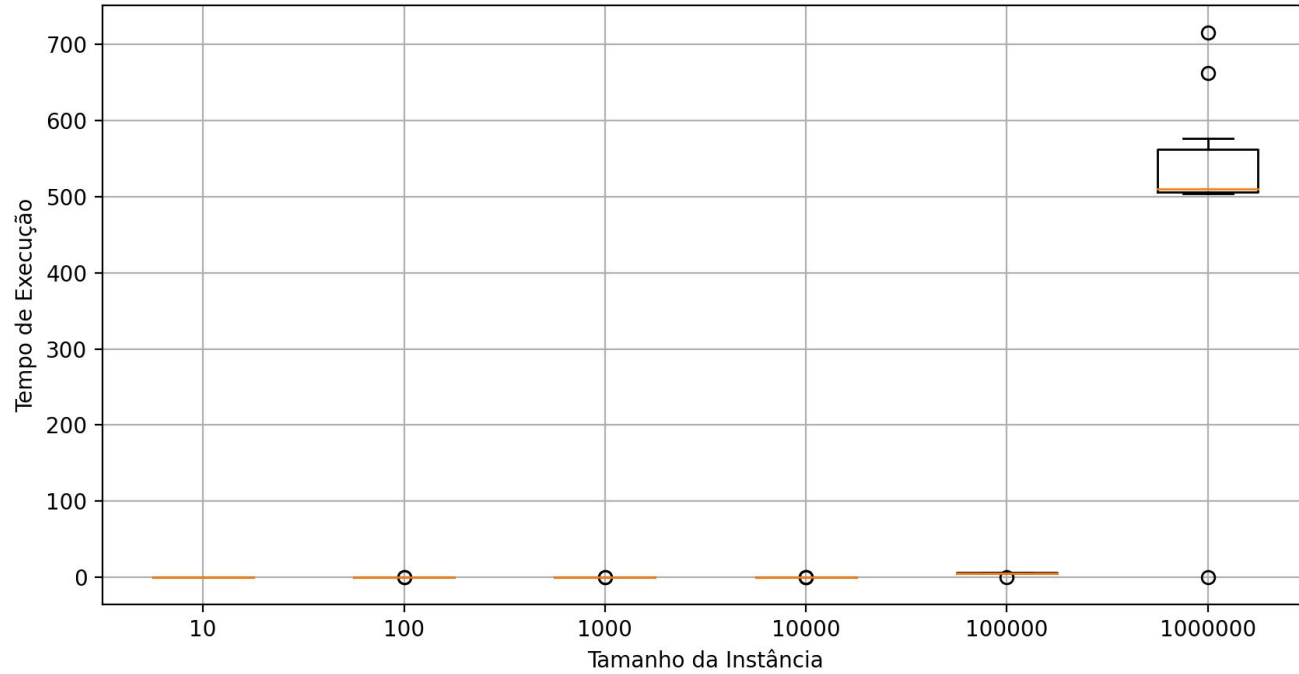


Gráfico: Merge\_Sort

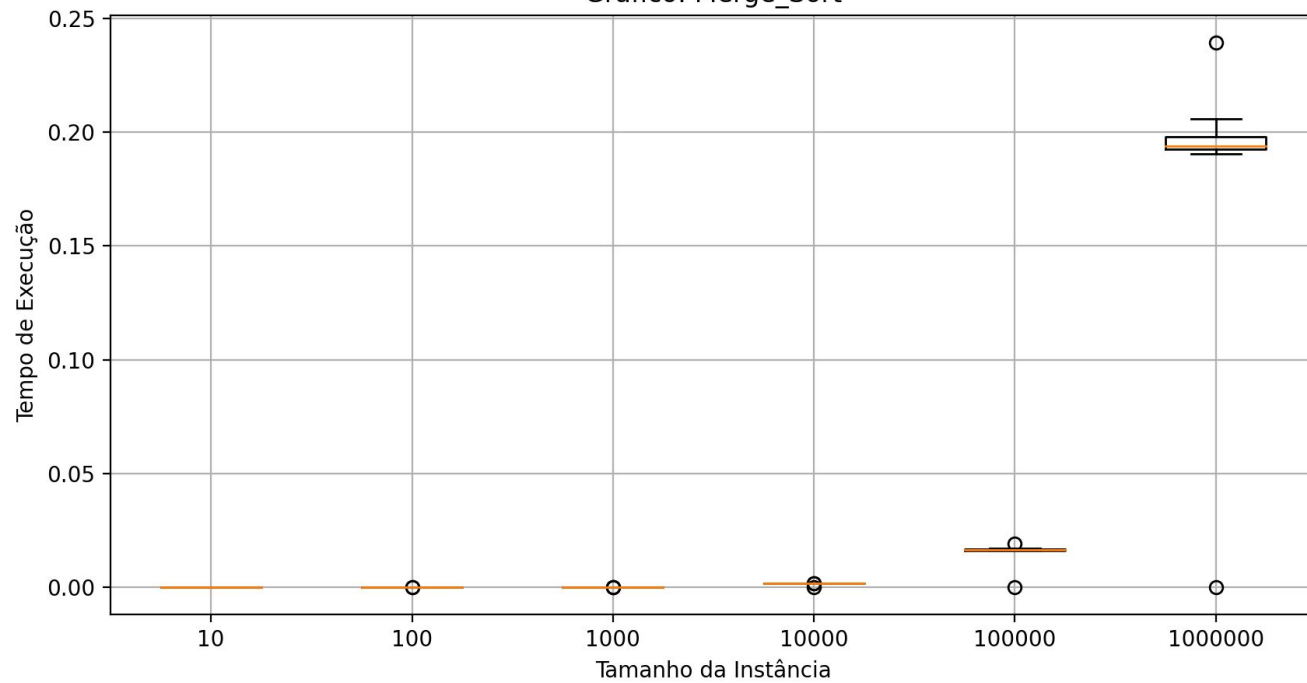
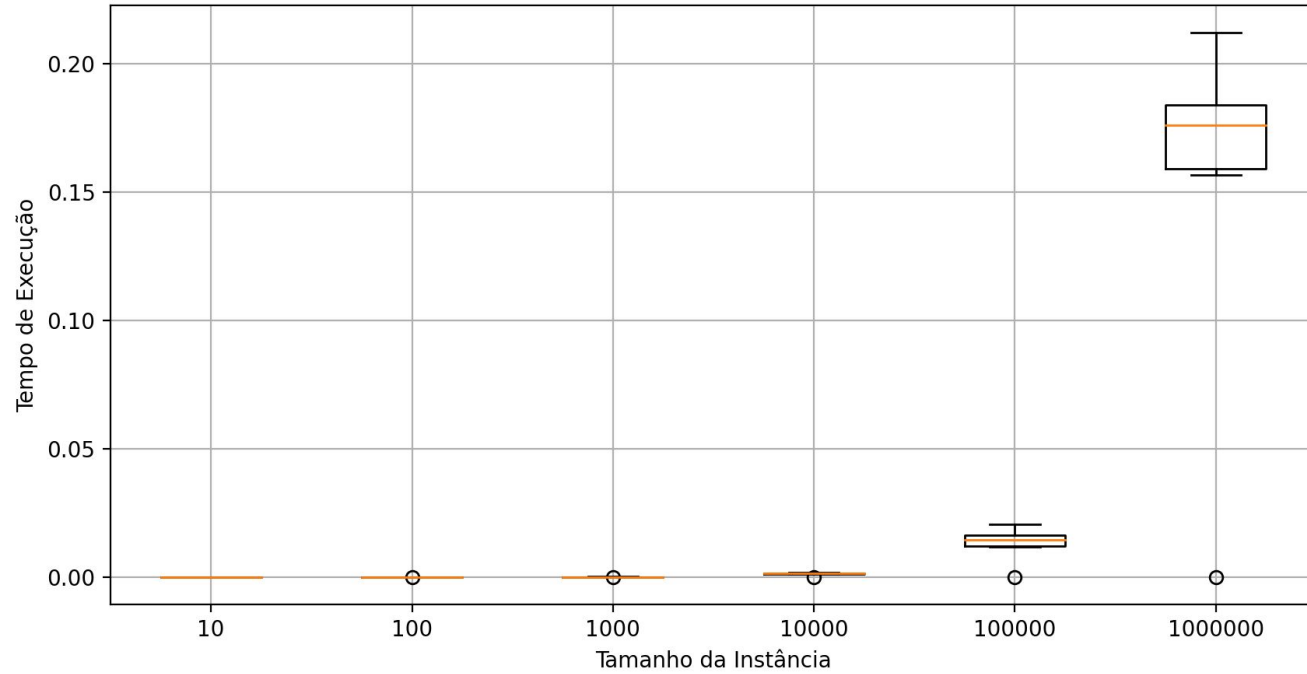
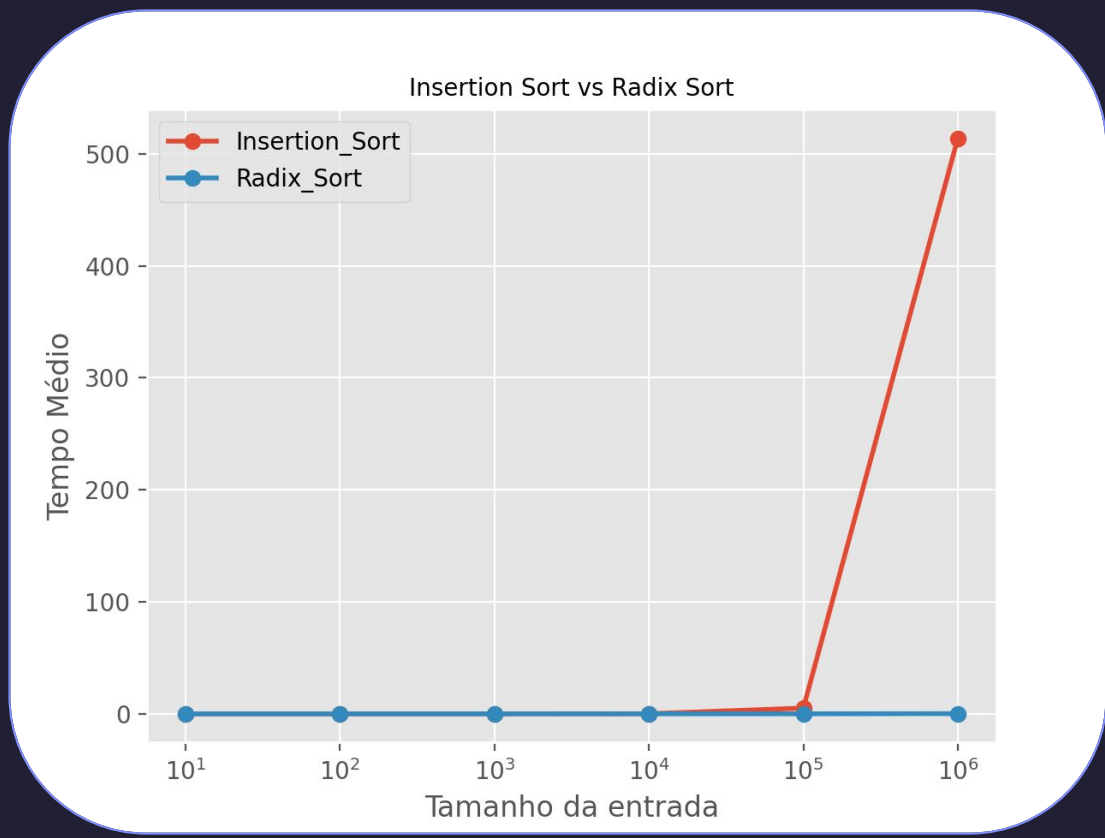
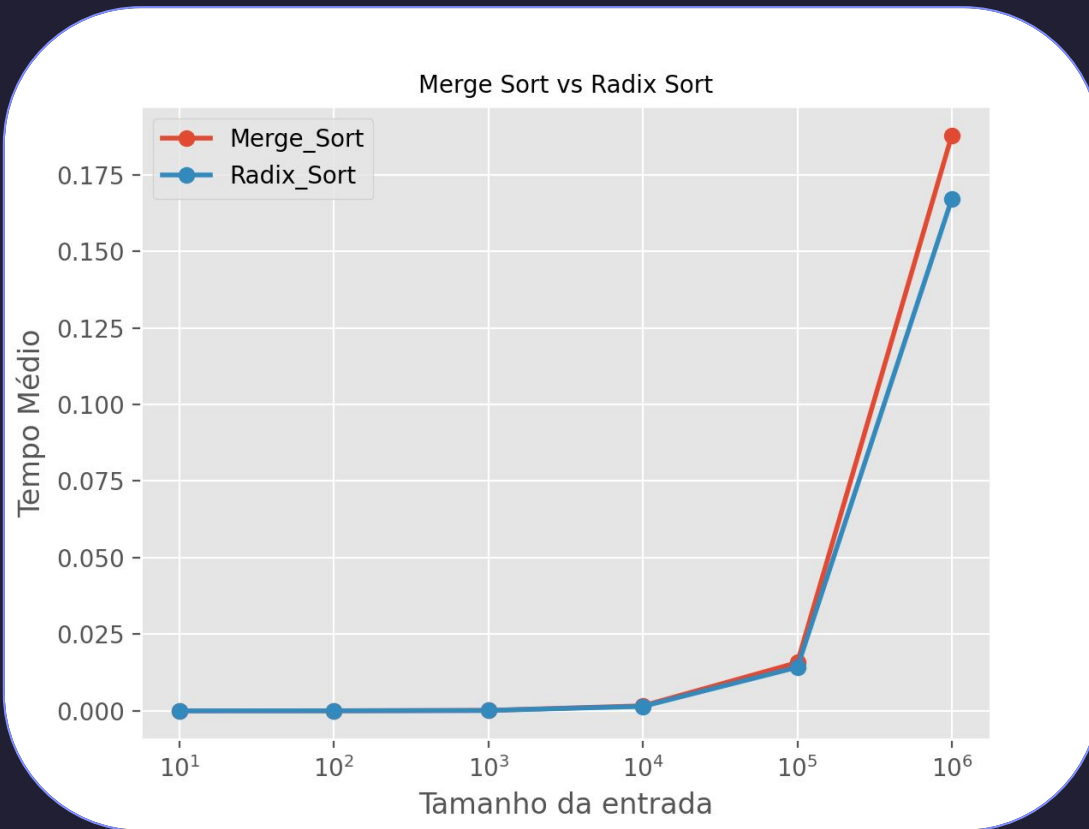
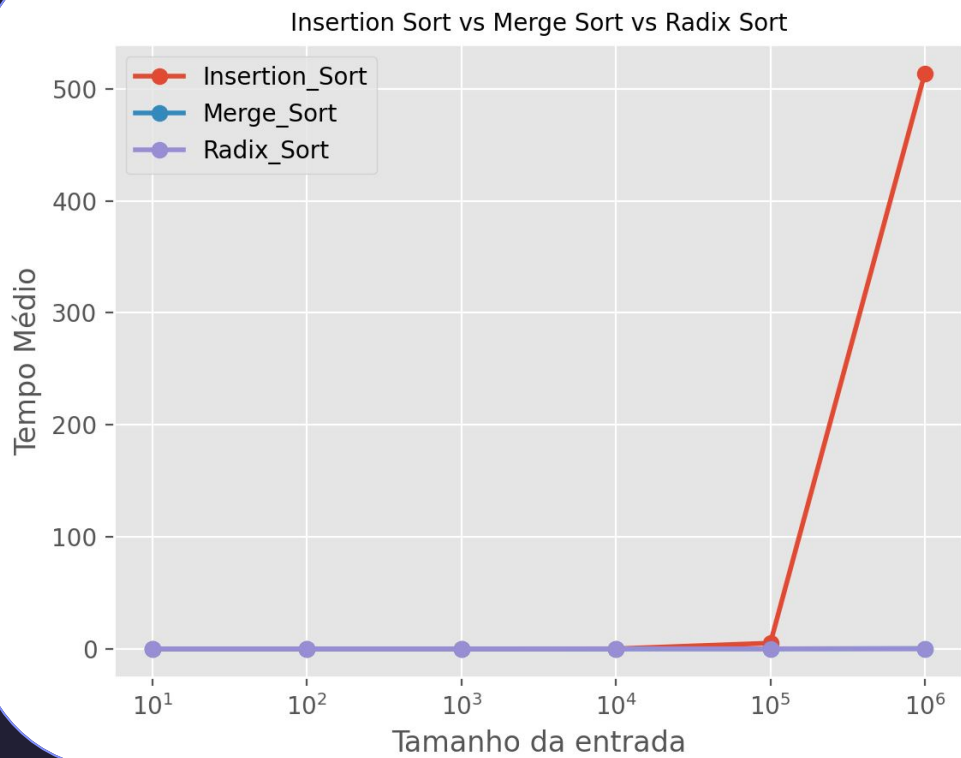


Gráfico: Radix\_Sort









# Testes T - Tabela

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)



# Conclusões

	Insertion Sort	Merge Sort	Radix Sort
Vantagem	Pequenas listas ou listas parcialmente ordenadas	Grandes listas de dados	Lidar com números inteiros e strings
Desvantagem	Grandes listas de dados	Maior uso de memória e não tão eficiente em listas pequenas	Eficiência diminui se a distribuição de dígitos não é uniforme ou se o intervalo de números é muito grande.

# 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.



**Obrigado(a) pela atenção!**