UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

PRIMEIRO TRABALHO PRÁTICO DA DISCIPLINA DE PROJETO E ANÁLISE DE ALGORITMOS (BCC241)

Vinicius Gabriel Angelozzi Verona de Resende
&
Carlos Eduardo Romaniello de Souza

Professor: Dr. ANDERSON ALMEIDA FERREIRA

Relatório referente ao primeiro trabalho prático da disciplina BCC241-11

Data: 27 de abril de 2022
Local: Ouro Preto – Minas Gerais – Brasil

Sumário

Resumo

Primeiro trabalho prático da disciplina de Projeto e Análise de Algoritmos (BCC241)

Devido a enorme quantidade de dados gerados diariamente ao redor do mundo, os algoritmos de ordenação são essenciais para facilitar a manipulação desses dados. Esses algoritmos são utilizados diariamente pelos programadores desde tarefas simples até as mais complexas e devido a essa alta demanda existem diversos algoritmos com lógicas diferentes que realizam a tarefa da ordenação. Por isso é necessário saber e estudar a eficiência desses algorimos para que se possa escolher o melhor algorimos para a situação. Nesse trabalho será analisada a eficiência de três algoritmos de ordenação, o *Insertion Sort*, *Merge Sort* e o *Radix Sort*.

1 Introdução

Este trabalho consiste em implementar e analisar o desempenho de três algoritmos de ordenação (*Insertion Sort*, *Merge Sort*, *Radix Sort*) a fim de melhor estudar o comportamento de cada algoritmo para variadas instâncias. Para isso, os algoritmos foram desenvolvidos na linguagem de programação *C* e cada um foi executado vinte vezes para cada tamanho de instância. Vale salientar que, cada uma das 20 execuções utilizam diferentes instâncias, ou seja, a ordenação dos dados de entrada é diferente em cada uma das 20 execuções e a mesma em cada algoritmo.

Para recolher uma grande quantidade de dados a serem avaliados, foram criadas instâncias de 6 tamanhos diferentes (*10, 100, 1.000, 10.000, 100.000, 1.000.000*) na linguagem de programação *Julia*[1], totalizando 120 instâncias. Após todas as execuções, os dados foram coletados e armazenados em arquivos separados para que posteriormente fossem utilizados para a geração de gráficos na linguagem de programação *JavaScript*[2] com o intuito de facilitar a interpretação dos dados coletados.

Todos os arquivos gerados para a realização deste trabalho estão disponíveis no *GitHub*[3] e foram separados em pastas de acordo com o seu conteúdo. O diretório *src* possui as implementações dos algoritmos de ordenação e o algoritmo gerador das instâncias. O diretório *data* possui os arquivos de input, output, o algoritmo gerador dos gráficos e os próprios gráficos. O diretório *bin* é o destino para todos os arquivos executáveis e compilados. No diretório *statistics*, existem os arquivos resultantes do Teste-T Pareado, assim como os códigos em *Julia* que realizam o teste estatístico. Por fim, o diretório *scripts* possui três arquivos em *shell-script* que executam todo o projeto e seus testes.

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 testados. Ademais, os resultados adquiridos são abordados em sessões futuras.

A fim de um melhor entendimento, este relatório se encontra subdividido em seções focadas a cada fase do projeto, desde o objetivo e organização dos dados de entrada à formação das soluções comparadas na seção de resultados. A distribuição esta feita da seguinte forma:

- Seção 2 encontra-se descrição e a análise do método *Insertion Sort*;
- Seção 3 encontra-se descrição e a análise do método *Merge Sort*;
- Seção 4 encontra-se descrição e a análise do método *Radix Sort*;

[1]Mais informações em: https://julialang.org/
[2]Mais informações em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript
[3]Código disponível em: https://github.com/CaduRomaniello/BCC241-TP1

- Seção 5 apresenta os resultados obtidos pelos algoritmos, e a análise destes;
- Seção 6 apresenta uma análise final mais abrangente acerca da pesquisa.

2 Insertion Sort

O *Insertion Sort* é um algoritmo de ordenação simples similar ao que fazemos ao ordernar um conjunto de cartas. Dado uma estrutura com vários elementos, para cada um dos itens será feito uma busca pelo local ideal de sua inserção, tendo então um subconjunto ordenado e não ordenado de elementos. Sendo assim, a implementação do algoritmo está representada a baixo:

```
1  void insertionSort(int* vet, int n) {
2      int aux;
3      int i, j;
4
5      for (i = 1; i < n; i++) {              // Custo 2 (Por iteração)
6          aux = vet[i];                      // Custo 1
7          j = i - 1;                         // Custo 1
8          while (j >= 0 && aux < vet[j]) {   // Custo 3
9              vet[j + 1] = vet[j];           // Custo 1
10             j--;                           // Custo 1
11         }
12         vet[j + 1] = aux;  // Custo 1
13     }
14 }
```

A análise da complexidade dessa implementação do *Insertion Sort* é a seguinte:

- Loop externo (*for*) executa 2 operações n vezes $\rightarrow O(2n)$;
- Linhas 6, 7, 12 totalizam $3(n-1)$ operações;
- Loop interno (*while*) executa um total $T_i + 1$ vezes, onde T_i é a execução das linhas $9 - 10$;
- Sendo assim, o bloco $8 - 11$ tem uma complexidade:
  $(3 * \sum_{i=1}^{n-1} T_i + 1) + (2 * \sum_{i=1}^{n-1} T_i)$
- Sendo assim temos que a função de complexidade da função insertionSort é:
  $T(n) = 5n - 3 + (5 \times \sum_{i=1}^{n-1} T_i) + (3 \times \sum_{i=1}^{n-1} 1)$
  $T(n) = 8n - 6 + (5 \times \sum_{i=1}^{n-1} T_i)$
- Melhor caso: $T_i = 0$
  $T(n) = 8n - 6 \rightarrow O(n)$

- Pior caso: $T_i = i - 1$
  $T(n) = \frac{5n^2+n-2}{2} \rightarrow O(n^2)$
- Caso médio: $T_i = \frac{1}{i} \times \sum_{j=0}^{i-1} j$
  $T(n) = \frac{5n^2+17n-14}{4} \rightarrow O(n^2)$
- Ordem de complexidade:
  Pior caso $\rightarrow O(n^2)$
  Caso médio $\rightarrow O(n^2)$
  Melhor médio $\rightarrow O(n)$

Sendo assim, como consideraremos o pior caso como fator de comparação entre os algoritmos, temos que o *Insertion Sort* é um algoritmo de ordenação de ordem quadrática $O(n^2)$.

3 Merge Sort

O *Merge Sort* é um algoritmo de ordenação que usa do princípio *Dividir e Conquistar* para alcançar 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 e após a resolução de todas as partes elas são combinadas resultando na estrutura ordenada. A implementação utilizada nesse trabalho foi:

```
1  void mergeSort(int* vet, int n) { mergeSort_ordena(vet, 0, n - 1); }
2
3  void mergeSort_ordena(int* vet, int esq, int dir) {
4      if (esq >= dir) {
5          return;
6      }
7
8      int meio = (esq + dir) / 2;
9      mergeSort_ordena(vet, esq, meio);
10     mergeSort_ordena(vet, meio + 1, dir);
11     mergeSort_intercala(vet, esq, meio, dir);
12 }
13
14 void mergeSort_intercala(int* vet, int esq, int meio, int dir) {
15     int i, j, k;
16     int a_tam = meio - esq + 1;
17     int b_tam = dir - meio;
18     int* a = (int*)malloc(sizeof(int) * a_tam);
19     int* b = (int*)malloc(sizeof(int) * b_tam);
20
21     for (i = 0; i < a_tam; i++) {
22         a[i] = vet[i + esq];
23     }
24     for (i = 0; i < b_tam; i++) {
25         b[i] = vet[i + meio + 1];
26     }
27
28     for (i = 0, j = 0, k = esq; k <= dir; k++) {
29         if (i == a_tam) {
30             vet[k] = b[j++];
31         } else if (j == b_tam) {
32             vet[k] = a[i++];
33         } else if (a[i] < b[j]) {
34             vet[k] = a[i++];
35         } else {
36             vet[k] = b[j++];
```

```
37          }
38      }
39
40      free(a);
41      free(b);
42  }
```

A analise de complexidade dessa implementação do *Merge Sort* é a seguinte:

- mergeSort_intercala $\rightarrow$ O(n);
- Função mergeSort_ordena $\rightarrow T(n) = 2 \times T(\frac{n}{2}) + O(n)$;
- Pelo Teorema Mestre simplificado temos que: $\log_2 2 = 1$. O expoente do custo local igual a $1 \rightarrow O(n^1) \therefore$ a complexidade é $O(n \times \log n)$;

Sendo assim, como consideraremos o pior caso como fator de comparação entre os algoritmos, temos que o *Merge Sort* é um algoritmo de ordenação de ordem logarítmica $O(n \times \log n)$.

4 Radix Sort

O *Radix Sort* é um algoritmo de ordenação rápido e estável que ordena uma estrutura com dados processando dígitos individuais de cada dado começando do dígito menos significativo e indo até o mais significativo. A implementação utilizada nesse trabalho foi:

```
int findMaximum(int* v, int tam) {
    int max = v[0];

    for (int i = 1; i < tam; i++) {
        if (v[i] > max) {
            max = v[i];
        }
    }

    return max;
}

void countingSort(int* content, int length, int expo) {
    int maximum = findMaximum(content, length);
    int* countingArray = calloc(maximum + 1, sizeof(int));
    int* sortedArray = calloc(length + 1, sizeof(int));

    for (int i = 0; i < length; i++) {  // Custo 2 * (n + 1)
        int idx = (content[i] / expo) % 10;
        countingArray[idx] += 1;
    }

    for (int i = 1; i <= maximum; i++) {  // Custo K
        countingArray[i] += countingArray[i - 1];
    }

    for (int i = maximum - 1; i >= 0; i--) {  // Custo K
        int idx = (content[i] / expo) % 10;
        int pos = countingArray[idx];

        sortedArray[pos - 1] = content[i];
        countingArray[idx] -= 1;
    }

    // Copy the sorted elements into original array
    for (int i = 0; i < length; i++) {  // Custo 2 * (n + 1)
        content[i] = sortedArray[i];
    }
```

```
39
40      free(countingArray);
41      free(sortedArray);
42  }
43
44  void radixSort(int* v, int length) {
45      int max = findMaximum(v, length);
46
47      for (int expo = 1; (max / expo) > 0; expo *= 10) {  // Custo P
48          countingSort(v, length, expo);
49      }
50  }
```

A análise de complexidade dessa implementação do *Radix Sort* é a seguinte:

- Função findMaximum $\rightarrow O(n)$;
- Função countingSort $\rightarrow O(2n + 2 + 2k) \rightarrow O(n + k)$;
- Função radixSort $\rightarrow O(n) + P \times O(n + k) \rightarrow P \times O(n + k) = O(n + k)$;
- Como ambos P e K são constantes, complexidade é $O(n)$;

Sendo assim, como consideraremos o pior caso como fator de comparação entre os algoritmos, temos que o *Radix Sort* é um algoritmo de ordenação de ordem linear $O(n)$.

5 Resultados

Como citado anteriormente, este trabalho consiste na análise do desempenho dos três algoritmos apresentados anteriormente. Os testes foram realizados com vinte instâncias para cada tamanho n (10, 100, 1.000, 10.000, 100.000, 1.000.000). Instâncias maiores que 1.000.000 estavam demorando um tempo elevado para serem ordenadas pelo algoritmo *Insertion Sort*, por isso o tamanho máximo utilizado foi 1.000.000.

Para cada execução foi analisado o tempo necessário para cada algoritmo ordenar a estrutura com os dados das instâncias. Vale ressaltar que as instâncias são iguais para todos os algoritmos o que fornece uma maior credibilidade para a comparação dos resultados. Os dados coletados para cada um dos algoritmos estão representados nos gráficos 5.1, 5.2, 5.3 onde o eixo x equivale ao tamanho da instãncia e o eixo y equivale ao tempo gasto em segundos para completar a ordenação.

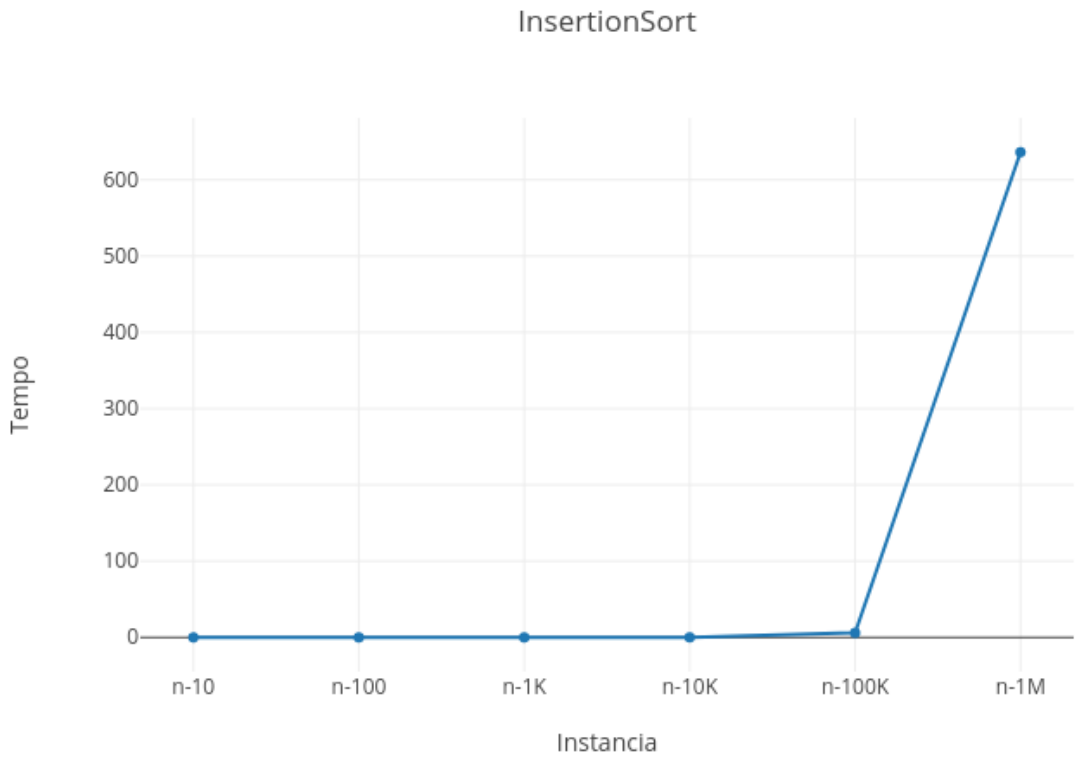

Figura 5.1: Insertion Sort

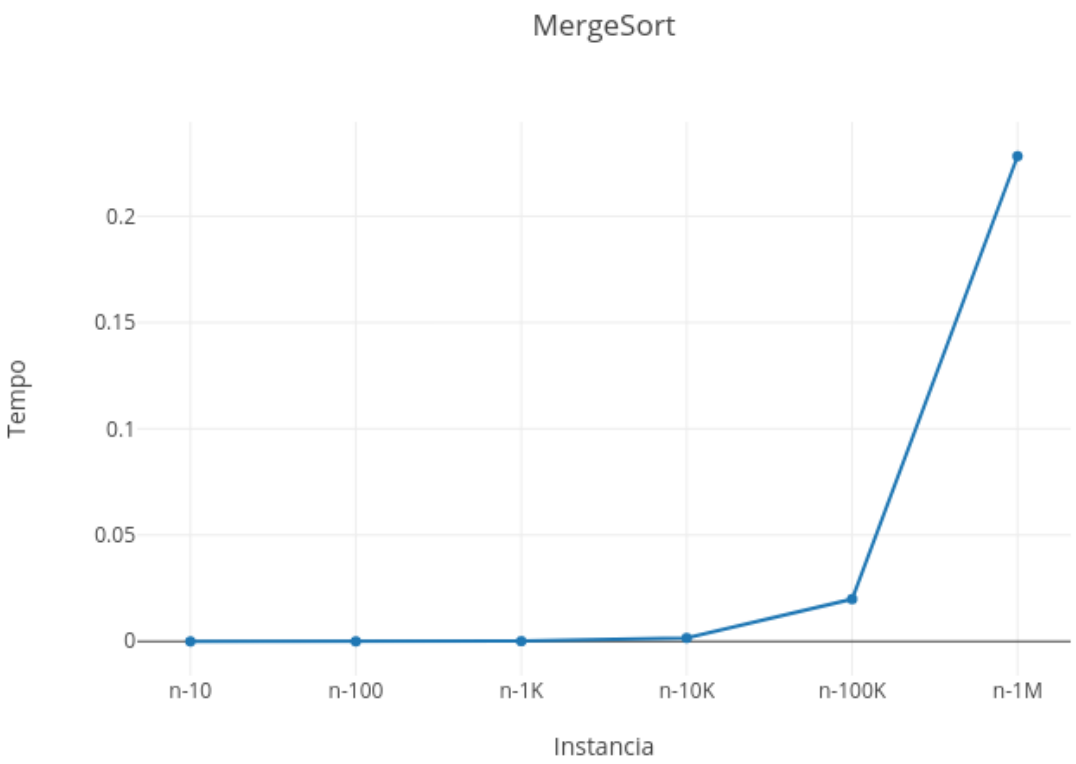

Figura 5.2: Merge Sort

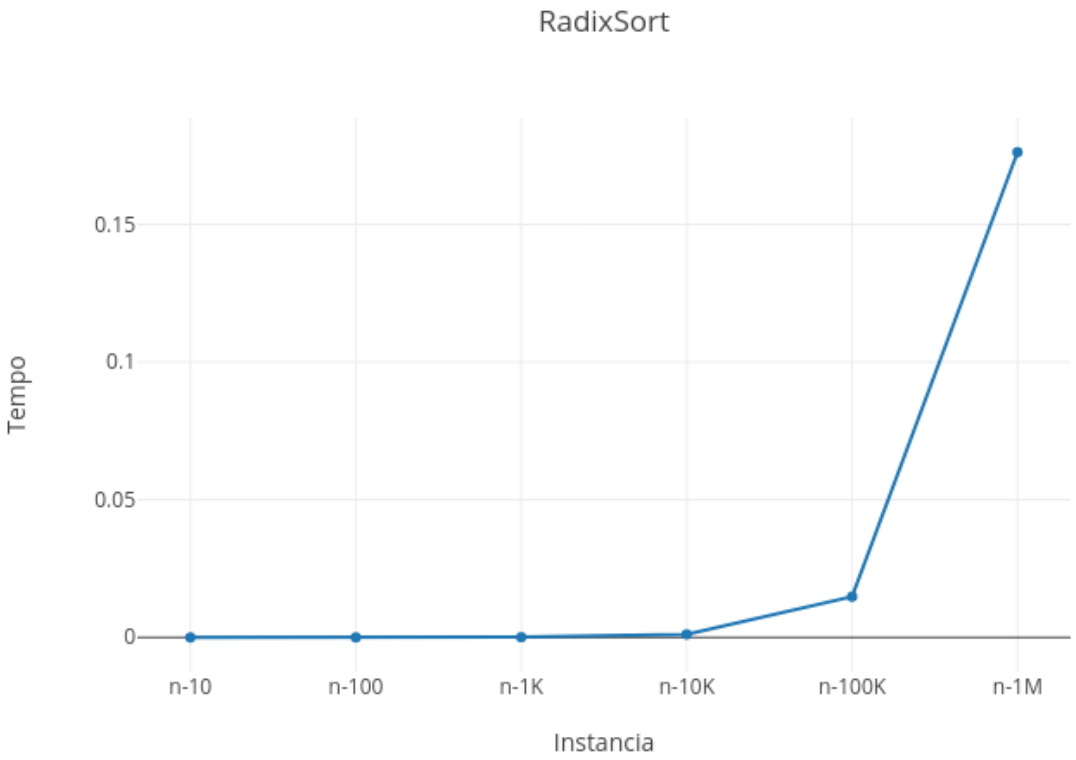

Figura 5.3: Radix Sort

Para facilitar a análise entre o desempenho dos algoritmos dois a dois, os dados dos gráficos 5.1, 5.2, 5.3 foram agrupados dois a dois nos gráficos 5.4, 5.5 e 5.6.

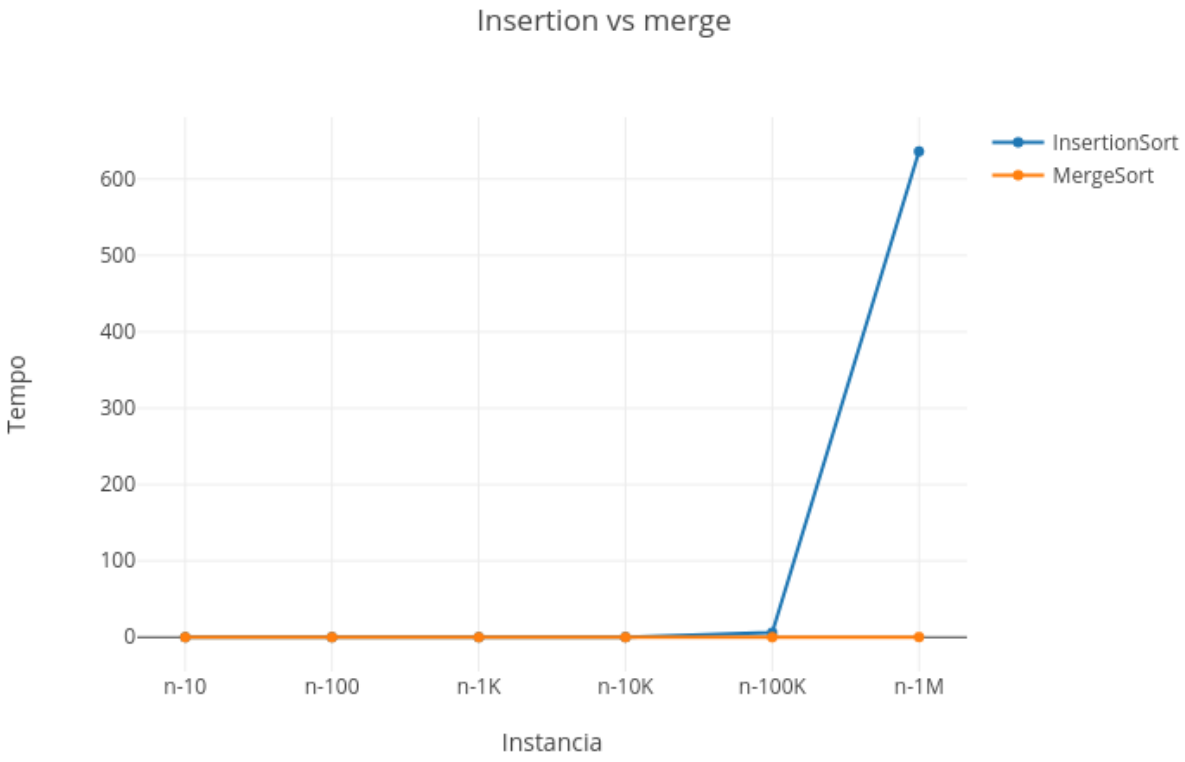

Figura 5.4: Insertion Sort x Merge Sort

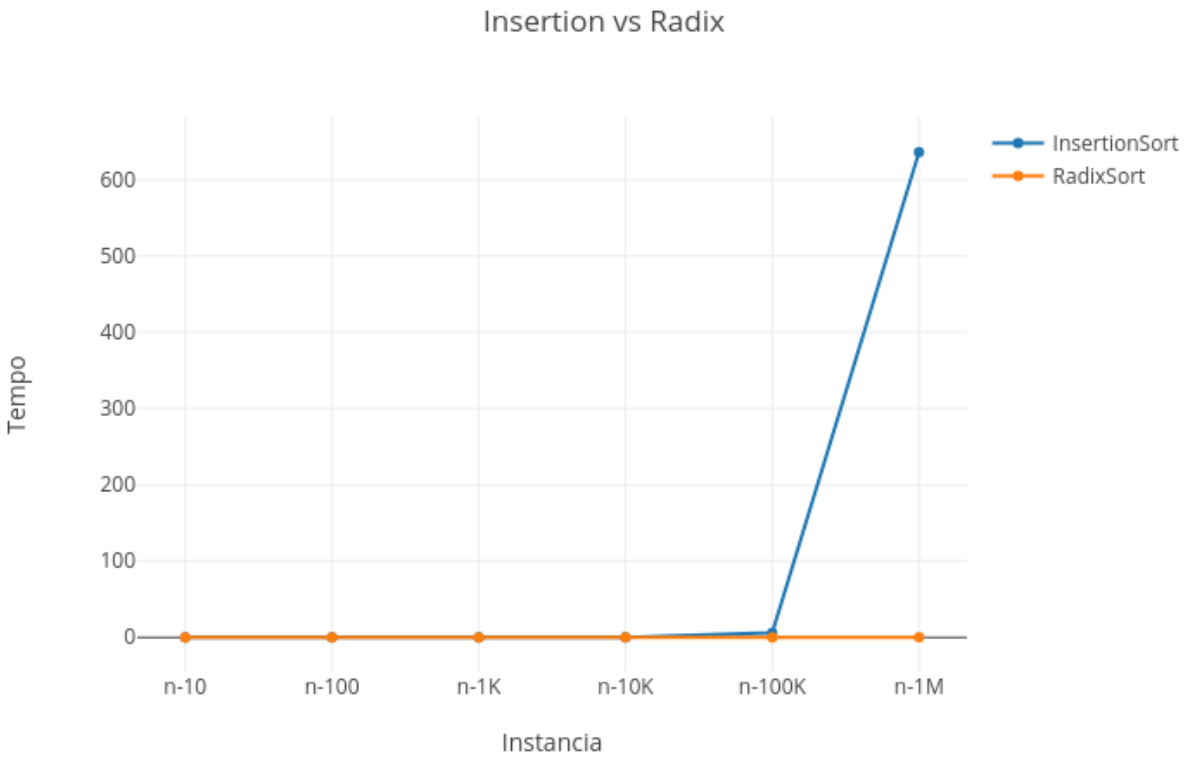

Figura 5.5: Insertion Sort x Radix Sort

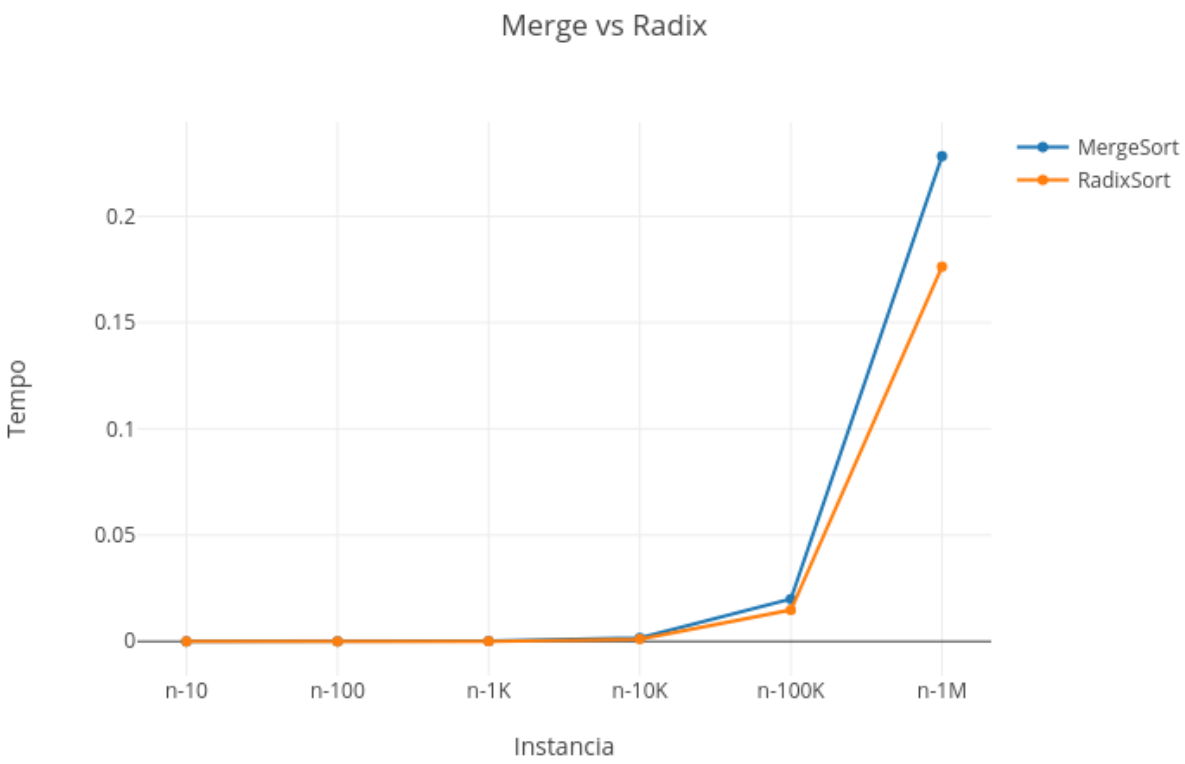

Figura 5.6: Merge Sort x Radix Sort

Por fim, para poder analisar os dados dos três algoritmos, o gráfico 5.7 foi gerado contendo os dados de todos eles. Em comparação com o *Insertion Sort*, os outros dois possuem um desempenho muito melhor e ambos são similares, o que dificulta essa percepção no gráfico 5.7. Essa semelhança no desempenho do *Merge Sort* e do *Radix Sort* está mais evidente no gráfico 5.6.

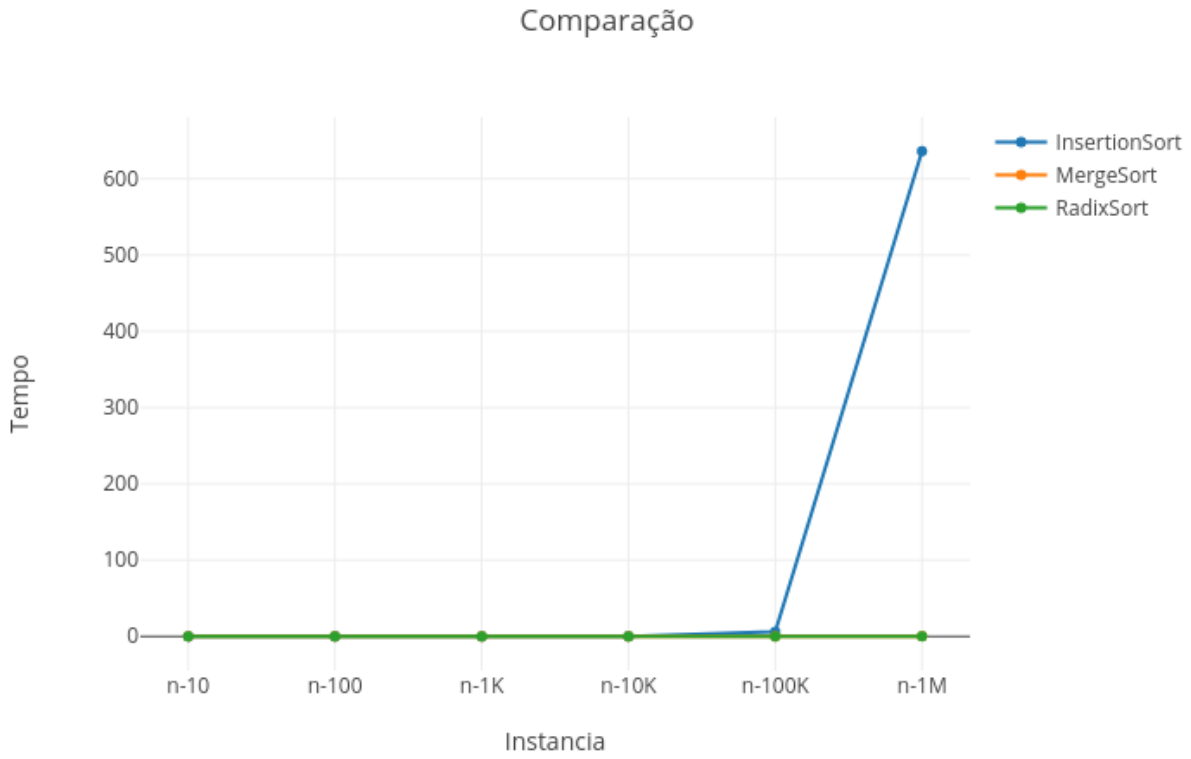

Figura 5.7: Insertion Sort x Merge Sort x Radix Sort

A fim de melhor comparar os algoritmos, em seguida analisa-se a tabela do intervalor de confiança dos métodos. Inicialmente, pela Tabela 5.1 apresenta-se o intervalo de confiança de cada algoritmo de forma separada para análise do tempo de execução para um dado tamanho. Na Tabela 5.2 apresenta-se o intervalo de confiança dos algoritmos combinados par a para a fim de identificar os melhores algoritmos (estatisticamente) em cada instância. Para ambos, foi utilizado o *Teste T Pareado* com confiança *95%*, $t = 2.093$ e $\alpha = 0.05$.

Tamanho (n)	***10***	***100***	***1000***	***10000***	***100000***	***1000000***
Insertion Sort	(0.0000; 0.0000)	(0.0000; 0.0000)	(0.0005; 0.0007)	(0.0534; 0.0667)	(5.3728; 6.7076)	(635.2858; 637.3242)
Merge Sort	(0.0000; 0.0000)	(0.0000; 0.0000)	(0.0001; 0.0001)	(0.0014; 0.0018)	(0.0177; 0.0222)	(0.2031; 0.2535)
Radix Sort	(0.0000; 0.0000)	(0.0000; 0.0000)	(0.0001; 0.0001)	(0.0010; 0.0012)	(0.0131; 0.0165)	(0.1560; 0.1965)

Tabela 5.1: Valores resultante do Teste-T Pareado com confiança *95%* para cada algoritmo

Tamanho (n)	***10***	***100***	***1000***	***10000***	***100000***	***1000000***
Insertion Sort & **Merge Sort**	(-0.0000; -0.0000)	(-0.0000; -0.0000)	(0.0004; 0.0005)	(0.0520; 0.0649)	(5.3551; 6.6855)	(635.0551; 637.0984)
Insertion Sort & **Radix Sort**	(-0.0000; 0.0000)	(-0.0000; -0.0000)	(0.0005; 0.0006)	(0.0525; 0.0654)	(5.3596; 6.6912)	(635.1090; 637.1485)
Merge Sort & **Radix Sort**	(-0.0000; 0.0000)	(-0.0000; -0.0000)	(0.0000; 0.0001)	(0.0004; 0.0006)	(0.0042; 0.0060)	(0.0433; 0.0607)

Tabela 5.2: Valores resultante do Teste-T Pareado com confiança *95%* para cada par de algoritmos

Pelo teste estatístico, percebe-se que:

- Ainda que para valores pequenos ($n \geq 1000$) o *Insertion Sort* apresenta resultados similares aos dos outros algoritmos, no entando para valores altos é evidente que ele se torna inviável comparado com os outros algoritmos.
- Pelas estatísticas, vemos que ambos *Merge Sort* e *Radix Sort* são similares na maioria dos casos, no entanto o *Radix Sort* obteve melhores resultados estatisticamente para grandes valores de instância, o que é esperado visto que sua complexidade é linear.

6 Conclusões

Através dos testes e das análises e realizadas pode-se concluir que dos três algoritmos, o *Insertion Sort* possui o pior desempenho o que é confirmado pela sua ordem de complexidade que no pior caso é de $O(n^2)$. O *MergeSort* e o *Radix Sort* possuem desempenhos semelhantes pois o tempo necessário para que eles completassem a ordenação se mantém próximo à medida que as instâncias aumentam de tamanho, porém o *Radix Sort* é ligeiramente melhor que o *MergeSort* o que também é visível através de suas ordens de complexidade que são $O(n)$ e $O(n \times \log n)$ respectivamente.

Apesar disso, para instâncias pequenas, o desempenho dos 3 algoritmos é praticamente igual uma vez que somente a partir da instância de tamanho 10.000 é que a diferença do tempo necessário para a ordenação entre os algoritmos se torna perceptível.

Referências Bibliográficas

Centre, M. L. S. Statistics: 1.1 paired t-tests. `https://www.lboro.ac.uk/media/media/schoolanddepartments/mlsc/downloads/1_1_Pairedttest.pdf`.

Ziviani, N. *Projetos de Algoritmos com implementações em Pascal e C*; Cengage Learning, 2010.

ASCENCIO, A. F. G.; ARAÚJO, G. S. d. Estruturas de Dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++. *São Paulo: Perarson Prentice Halt* **2010**, *3*.

Dasgupta, S.; Papadimitriou, C. H.; Vazirani, U. V. *Algorithms*; McGraw-Hill Higher Education New York, 2008.