Trabalho prático 1 - Avaliação empírica

Carlos Eduardo Gonzaga Romaniello de Souza - 19.1.4003
Vinícius Gabriel Angelozzi Verona de Resende - 19.1.4005

Tópicos

01 Insertion Sort

02 Merge Sort

03 Radix Sort

04 Resultados

05 Conclusão

06 Referências

01

Insertion Sort

Código

```
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 }
```

Análise assintótica

Pior caso: Ti = i - 1

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

Função de complexidade

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

Melhor caso: Ti = 0

$$T(n) = 8n - 6 \rightarrow O(n)$$

Caso médio

$$T_i = \frac{1}{i} \times \sum_{j=0}^{i-1} j$$

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

02

Merge Sort

Código

```c
void mergeSort(int* vet, int n) { mergeSort_ordena(vet, 0, n - 1); }

void mergeSort_ordena(int* vet, int esq, int dir) {
    if (esq >= dir) {
        return;
    }

    int meio = (esq + dir) / 2;
    mergeSort_ordena(vet, esq, meio);
    mergeSort_ordena(vet, meio + 1, dir);
    mergeSort_intercala(vet, esq, meio, dir);
}
```

Código

```
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  }
```

Análise assintótica

mergeSort_intercala

O(n)

mergeSort_ordena

$$T(n) = 2 \times T(\tfrac{n}{2}) + O(n)$$

Teorema Mestre

$$O(n \times \log n)$$

03

Radix Sort

Código

```c
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];
    }

    free(countingArray);
    free(sortedArray);
}

void radixSort(int* v, int length) {
    int max = findMaximum(v, length);

    for (int expo = 1; (max / expo) > 0; expo *= 10) {  // Custo P
        countingSort(v, length, expo);
    }
}
```

Análise assintótica

findMaximun $O(n)$

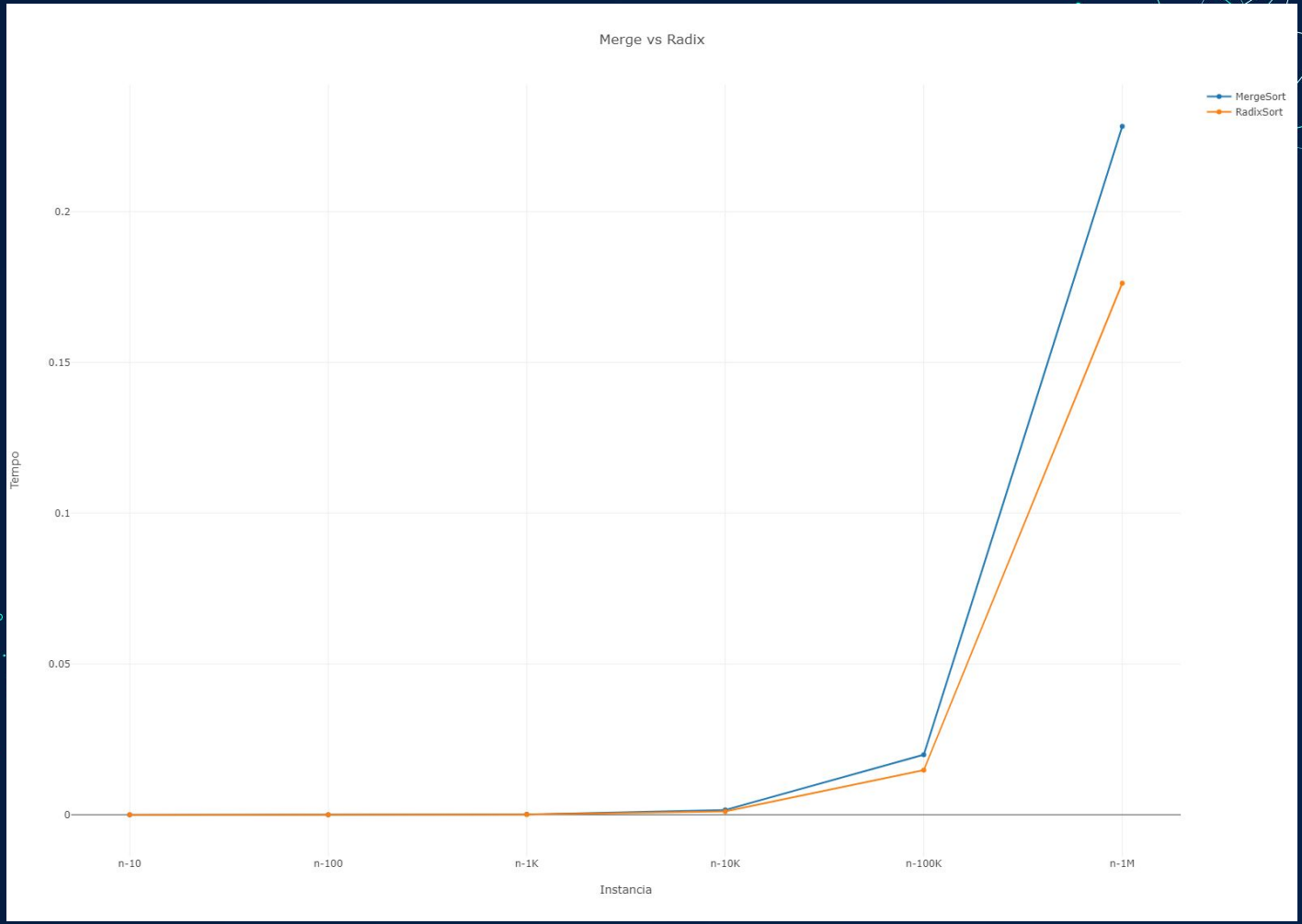
countingSort $O(2n + 2 + 2k) \rightarrow O(n + k)$

radixSort $O(n) + P \times O(n + k) \rightarrow P \times O(n + k) = O(n + k)$

complexidade Como P e K são constantes, a complexidade é O(n)

04

Resultados


Merge vs Radix
MergeSort
RadixSort
0.2
0.15
0.1
0.05
0
Tempo
n-10
n-100
n-1K
n-10K
n-100K
n-1M
Instancia

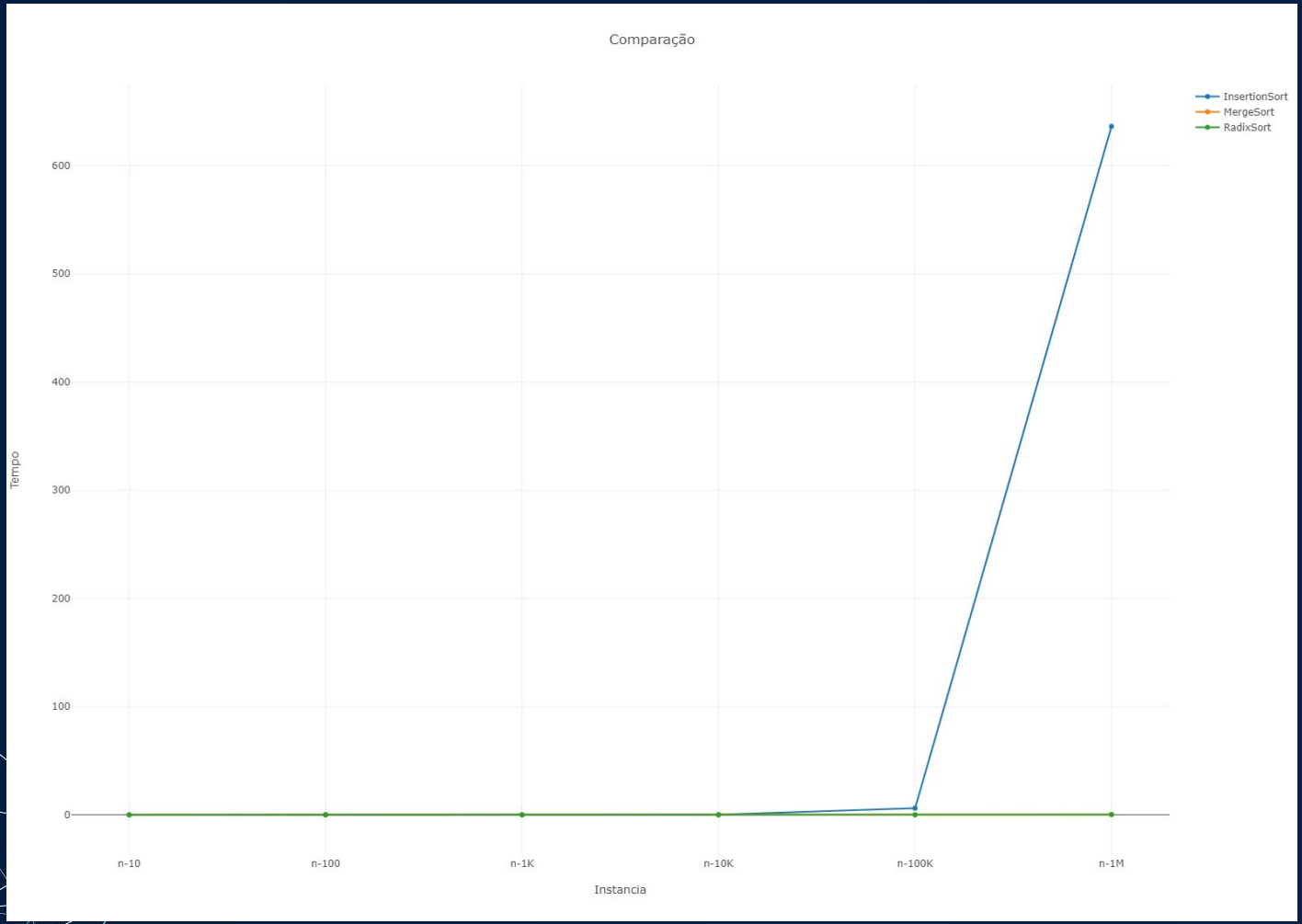

Comparação
InsertionSort
MergeSort
RadixSort
600
500
400
300
200
100
0
Tempo
n-10
n-100
n-1K
n-10K
n-100K
n-1M
Instancia

Tabela 1: Teste-T Pareado entre tempo de execução para cada algoritmos

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 2: Teste-T Pareado entre pares de algoritmos

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)

05
Conclusão

06

Referências

Bibliografia

- 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 Prentic Halt 2010, 3.
- Dasgupta, S.; Papadimitriou, C. H.; Vazirani, U. V. Algorithms; McGraw-Hill Higher Education New York, 2008.
- Centre, M. L. S. Statistics: 1.1 paired t-tests. Disponível em: https://www.lboro.ac.uk/media/media/schoolanddepartments/mlsc/downloads/1_1_Pairedttest.pdf.

Obrigado!

Dúvidas?


slidesgo