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

João Vitor Gonçalves da Silva
Sabrina Suellen da Silva

TRABALHO PRÁTICO
AVALIAÇÃO EMPÍRICA

Ouro Preto, MG
2021

João Vitor Gonçalves da Silva
Sabrina Suellen da Silva

TRABALHO PRÁTICO
AVALIAÇÃO EMPÍRICA

Resumo: *O trabalho em questão busca desenvolver a análise empírica dos algoritmos MergeSort, InsertionSort e RadixSort, estes sendo algoritmos de ordenação. Esta análise é feita com testes experimentais com 20 instâncias de tamanho* 10^n *sendo n variando de 1 a 20*

Ouro Preto, MG
2021

Sumário

1 Introdução

Algoritmos de organização são de extrema importância nos dias atuais. Organizar dados, seja de forma crescente ou decrescente, para se obter um sistema organizado e melhorar algumas algumas funções de pesquisa são uns dos pontos cruciais em sua utilização. Afinal, como diz a velha frase: "Tempo é dinheiro".

Entretanto, a operação de ordenar dados também é custosa, dependendo da quantidade de dados que se deseja organizar. Por isso, é importante que se escolha o algoritmo que seja eficiente para o sistema que ele irá ordenar, já que estes podem variar dependendo da forma em que o sistema está. Neste trabalho, realizamos experimentos de três famosos algoritmos de ordenação: MergeSort, InsertionSort e RadixSort.

De acordo com os testes, aquele que mais se destaca foi o MergeSort, que ordena com custo O(n logn), sendo n o número de elementos a serem ordenados. Em segundo lugar, temos o RadixSort, que obteve custo de tempo similar ao Merge, sendo sua complexidade de O(nk), mas falha em instâncias muito grandes. Por último, temos o InsertionSort, que é rápido para instâncias pequenas mas extremamente lento para outros casos, tendo o custo de O(n^2). Todos os algoritmos desse trabalho foram implementados em C e as instâncias foram geradas aleatoriamente durante a execução.

2 Análises de complexidades

A seguir será apresentado a análise e complexidades de cada um dos algoritmos desenvolvido no trabalho.

2.1 MergeSort - $O(nlogn)$

Esse algoritmo realiza a organização por meio da divisão e conquista. Essa técnica consiste em dividir o problema em pequenas instâncias em tamanhos mínimos e resolve-las separadamente. Após resolver todas as instâncias pequenas, ele junta os resultados. Gerando assim, várias soluções de tamanho [n/2/2...].

Sua complexidade pode ser expressada por $T(n) = 2T(n/2) + n$, mais formalmente tida como $0(nlogn)$.

```
#include "mergesort.hpp"
#include <iostream>

using namespace std;


// Ordena o vetor v [0..n -1]
void mergeSort (long long int* v , long long int n) {
  mergeSort_ordena (v , 0, n -1) ;
}

// Ordena o vetor v[ esq .. dir ]
void mergeSort_ordena (long long int *v, long long int esq, long long
    int dir) {
  if ( esq >= dir )
  return ;
  long long int meio = ( esq + dir ) / 2;
  mergeSort_ordena (v,esq,meio);
  mergeSort_ordena (v,meio+1,dir);
  mergeSort_intercala (v,esq,meio,dir);
}

// Intercala os vetores v[esq .. meio ] e v[ meio +1.. dir ]
void mergeSort_intercala (long long int* v ,long long int esq , long
    long int meio , long long int dir) {
  long long int i , j , k ;
  long long int a_tam = meio - esq +1;
  long long int b_tam = dir - meio ;
  long long int* a = new long long int [a_tam];
```

```
28   long long int* b = new long long int [b_tam];
29
30   for (i=0; i< a_tam ; i ++)
31     a[i] = v[i+ esq];
32     for (i = 0; i < b_tam ; i ++)
33       b[i] = v[i+meio+1];
34       for (i=0, j=0 , k=esq; k<=dir ; k++) {
35         if (i==a_tam)
36           v[k] = b[j++];
37         else if (j == b_tam)
38           v[k] = a[i++];
39         else if (a[i] < b[j])
40           v[k] = a[i++];
41         else
42           v[k] = b[j++];
43       }
44   delete a;
45   delete b;
46 }
```

2.2 InsertionSort - $O(n^2)$

Este algoritmo de ordenação realiza sua tarefa fazendo trocas com os elementos que estão a direita do elemento chave. O elemento chave é sempre iniciado com a segunda posição do vetor e comparado com a primeira posição, seguindo a ideia da primeira iteração temos que ele faz comparação do elemento i com seu antecessor i-1 até a posição final.

A função de complexidade do insertion sort pode ser representada como:

$$(n-1) + (n-2) + ... + 2 + 1 = n(n-1)/2 = n^2$$

sendo assim temos que a complexidade é $O(n^2)$. No melhor caso, que seria o array de entrada já ordenado, a complexidade é $O(n)$ visto que o algoritmo visita apenas uma vez cada posição do array.

```
1  void insertionsort ( long long int* vector, long long int n) {
2    int i,j;
3
4    for (i = n-2; i >= 0; i --) {
5      vector[n] = vector[i];
6      j = i + 1;
7
8      while (vector[n] > vector[j] ) {
9        vector[j - 1] = vector[j];
10       j ++;
11     }
```

```
12
13   vector[j - 1] = vector[n];
14  }
15 }
```

2.3 RadixSort - $O(nk)$

Esse algoritmo ordena baseado nos decimais dos números. Ele pega os decimais e os tem como chaves para cada número. Então, é ordenado baseado em cada chave processando os digitos como individuais. Sua complexidade é de $O(nk)$, sendo k o tamanho das chaves.

```
1
2  int getMax(long long int arr[], long long int n) {
3      int mx = arr[0];
4      int i;
5      for (i = 1; i < n; i++)
6          if (arr[i] > mx)
7              mx = arr[i];
8      return mx;
9  }
10
11 void countSort(long long int arr[], long long int n, long long int exp)
       {
12     int output[n]; // vetor retornado
13     int i, count[10] = { 0 };
14
15     // Armazena as ocorrencias
16     for (i = 0; i < n; i++)
17         count[(arr[i] / exp) % 10]++;
18
19     for (i = 1; i < 10; i++)
20         count[i] += count[i - 1];
21
22     // Constroi o vetor de saida
23     for (i = n - 1; i >= 0; i--) {
24         output[count[(arr[i] / exp) % 10] - 1] = arr[i];
25         count[(arr[i] / exp) % 10]--;
26     }
27
28     for (i = 0; i < n; i++)
29         arr[i] = output[i];
30 }
31
32 // Funcao principal chamada pelo main que ordena o vetor
33 void radixsort(long long int arr[], long long int n) {
34     int m = getMax(arr, n);
```

```
35
36      int exp;
37      for (exp = 1; m / exp > 0; exp *= 10)
38          countSort(arr, n, exp);
39  }
```

3 Avaliação Experimental

As instâncias utilizadas para a avaliação foram de 10^1 até 10^{10} para Merge Sort e de 10^1 à 10^6 para Insertion Sort e Radix Sort, as demais instâncias recomendadas para teste não foram executadas por serem valores extremamente altos ocorrendo erros e problemas nos hardwares usados para tal.

Com o MergeSort foi possível realizar teste de 10^1 até 10^{10}, em instâncias menores o tempo de ordenação foi 0, para instâncias maiores que 10^5 temos que o tempo aumentou porém de forma leve se comparado a outros algoritmos de ordenação, como o Insertion que veremos a seguir.

Merge Sort	
10^1	0 milissegundos
10^2	0 milissegundos
10^3	0 milissegundos
10^4	0 milissegundos
10^5	15 milissegundos
10^6	140 milissegundos
10^7	1656 milissegundos
10^8	29671 milissegundos
10^9	37312 milissegundos
10^{10}	615437 milissegundos

Com Insertion Sort foi possível testes de 10^1 até apenas 10^6, devido a complexidade de $O(n^2)$ se tornou invíavel a ordenação de instâncias muito grandes, pois quanto maior a instância maior seria o custo de armazenamento e comparação. Não sendo um algoritmo indicado para instâncias muito altas.

Insertion Sort	
10^1	0 milissegundos
10^2	0 milissegundos
10^3	15 milissegundos
10^4	93 milissegundos
10^5	11203 milissegundos
10^6	1132687 milissegundos

Foi possível com o Radix Sort executar testes com instancias de tamanho até 10^6, mesmo tendo um custou um pouco melhor que o Insertion Sort ele também não é recomendado para instâncias

muito grandes. Além de que para ser eficiente as entradas devem possuir uma configuração específica.

Radix Sort	
10^1	0 milissegundos
10^2	0 milissegundos
10^3	0 milissegundos
10^4	15 milissegundos
10^5	15 milissegundos
10^6	234 milissegundos

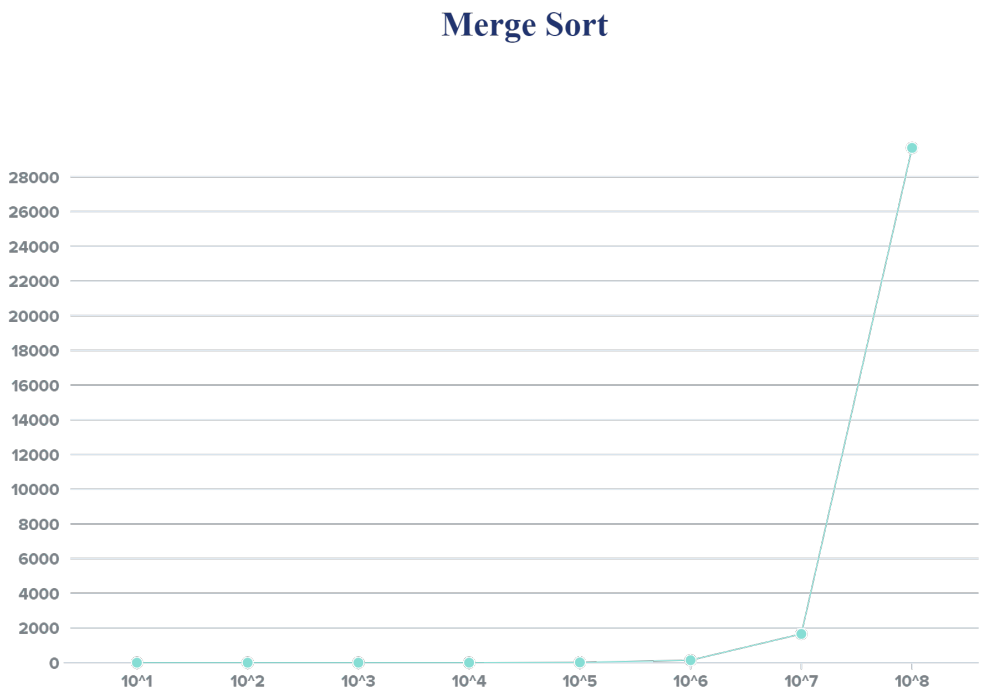

Figura 3.1 – Resultados do tempo de execução para cada instância no Merge Sort

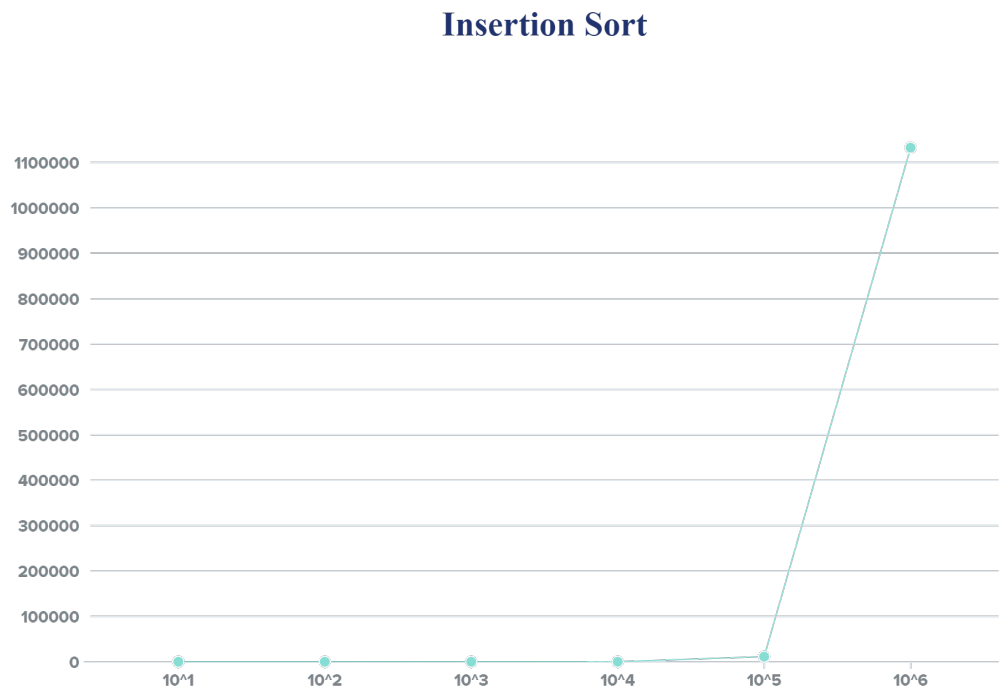

Figura 3.2 – Resultados do tempo de execução para cada instância no Insertion Sort

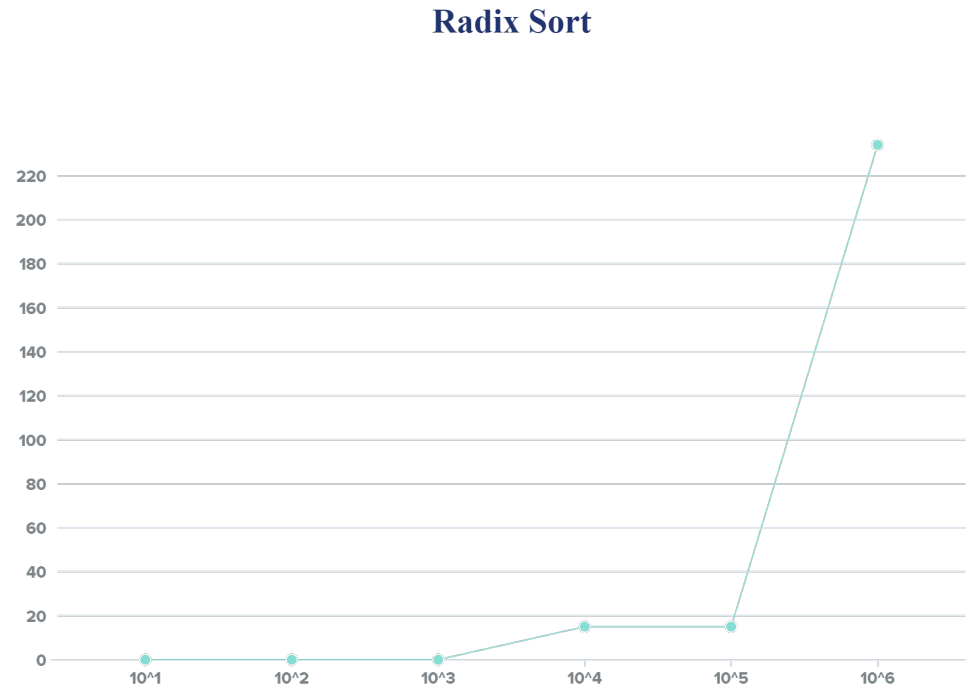

Figura 3.3 – Resultados do tempo de execução para cada instância no Radix Sort

3.1 Distribuição t

Foi realizada a estatística da distribuição T para analisar se os programas possuem uma diferença entre eles, com 95% de certeza. Para os cálculos, foi considerada apenas as 6 primeiras instâncias do MergeSort para igualar com os outros dois algorítmos.

Merge vs Insertion:

- Diferença das médias: -190,57005
- Intervalo obtido: [-192,5305; -188,6105]

Como o intervalo não inclui o zero, eles são diferentes com 95% de certeza.

Merge vs Radix:

- Diferença das médias: -0,018166
- Intervalo obtido: [-1,9781; 1,9418]

Como o intervalo inclui o zero, eles não possuem diferença para as primeiras 6 instâncias.

Radix vs Insertion:

- Diferença das médias: 190,6405
- Intervalo obtido: [188,6805; 192,6005]

Como o intervalo não inclui o zero, eles são diferentes com 95% de certeza.

4 Conclusão

Foi observado que cada algoritmo possui casos melhores de uso dependente da entrada que é recebida, através desse estudo é perceptível que o MergeSort é mais funcional quando é uma entrada de tamanho grande, apesar de sua difícil implementação. O InsertionSort deve ser utilizado quando se possui uma base pequena de dados, já que sua implementação é mais simples. O Radix sai melhor nos casos que envolvam cadeias de caracteres. Portanto, cada algoritmo possui seu caso em que se destaca e, para utilização do mesmo, deve haver uma análise do sistema para definir qual algoritmo deve ser utilizado para se obter melhores resultados.

5 Referências Bibliográficas

BRAGA, Rodrigo. Algoritmos de Ordenação: Insertion Sort. Medium, 2018. Disponível em: https://henriquebraga92.medium.com/algoritmos-de-ordena

ALVES, Gabriel. et al. Insertion Sort. Brilliant, 2021. Disponível em: https://brilliant.org/wiki/insertion/

BHOJASIA, Manish. C Program to Implement RadixSort. Sanfoundry, 2011. Disponível em: https://www.sanfoundry.com/c-program-implement-radix-sort/