Instituição: Universidade Federal do Rio Grande do Norte – UFRN
Unidade: Instituto Metrópole Digital – IMD
Curso: Bacharelado em Tecnologia da Informação – BTI
Disciplina: IMD0029 – Estrutura De Dados Básicas I – T01
Professor: Selan Rodrigues Dos Santos

Relatório técnico de análise de algoritmos de ordenação

Aluno

ÁLVARO FERREIRA PIRES DE PAIVA – 2016039162
alvarofepipa@gmail.com

2017

Natal - RN

Sumário

1 INTRODUÇÃO 3

2 METODOLOGIA 5

2.1 Informações técnicas 5

2.2 Algoritmos implementados 6

2.2.1 Insertion sort 6

2.2.2 Selection sort 8

2.2.3 Bubble sort 10

2.2.4 Shell sort 11

2.2.5 Quicksort 14

2.2.6 Merge sort 14

2.2.7 Radix sort (LSD) 14

2.2.8 Visão geral 14

2.3 Cenários 14

2.3.1 Arranjos com elementos aleatórios 14

2.3.2 Arranjos com elementos não decrescentes 14

2.3.3 Arranjos com elementos não crescentes 14

3 RESULTADOS 15

4 DISCUSSÃO 16

1 Introdução

Um algoritmo é uma sequência finita de passos/instruções, ordenadas de forma lógica, que permitem resolver um determinado problema ou conjunto de problemas de mesmo tipo. Quando tratamos de algoritmo no meio computacional, podemos dividir em 3 partes:

1. Entrada de dados;
2. Processamento;
3. Saída dos dados resultantes.

No mundo real, lidamos com diversos tipos de dados e um dos modos que possuímos de armazená-los é através de arranjos. Os arranjos são conjuntos/coleções de elementos de tal forma que esses elementos possam ser identificados por um índice ou chave.

Arranjo A de tamanho n:

$$A = \Big[a_1, \quad a_2, \quad a_3, \quad \ldots, \quad a_{n-1}, \quad a_n\Big]$$

Identificando elemento do arranjo:

$$A[1] = a_1$$

Em determinados casos, precisamos ordenar esses arranjos para facilitar o processamento realizado posteriormente. Esse problema de ordenação é chamado de **problema da ordenação de um arranjo sequencial**. Como entrada desse problema, temos um arranjo $[a_1, \ldots, a_n]$, com $n \in \mathbb{Z}$ e $n > 0$ e a saída é uma permutação $[a_{\pi 1}, \ldots, a_{\pi n}]$ no qual temos a garantia que $a_{\pi 1} \leq a_{\pi 2} \leq \cdots \leq a_{\pi n}$.

O presente relatório analisará um total de 7 algoritmos que resolvam o problema citado anteriormente, sendo eles:

1. Insertion sort;
2. Selection sort;
3. Bubble sort;
4. Shell sort;
5. Quick sort;

6. Merge sort;
7. Radix sort (LSD).

Esses algoritmos irão ser análisados em 3 situações:

1. Arranjos com elementos aleatórios;
2. Arranjos com elementos não decrescentes;
3. Arranjos com elementos não crescentes;

2 Metodologia

Esse capítulo constará, respectivamente, com as informações técnicas referentes aos experimentos (características do computador utilizado, sistema operacional, linguagem de programação adotada, etc), os algoritmos implementados (uma breve explicação de cada um e seus respectivos códigos) e uma explicação de cada cenário analisado nesse trabalho.

2.1 Informações técnicas

Para a realização do trabalho, foi utilizado um notebook com as seguintes características:

Fabricante	Acer
Modelo	Aspire 4739
Placa-mãe	HMA CP (versão 1.08)
Disco	Hitachi HTS54757 (750 GB)
RAM	8 GB
Processador	Intel Core i5-480M (2.67GHz)
Gráficos	Intel Ironlake Mobile
Sistema base	Ubuntu 16.04.2 LTS 64-bit

A linguagem de programação adotada foi **C++** (*C mais mais* ou *C plus plus*). C++ foi escolhido devido ser considerado uma linguagem poderosa para a resolução de problemas de baixo e alto nível, prezando pela performance rápida, pois cada recursos presente foi criteriosamente projetado para se usado onde performance for uma exigência crítica (**??**).

O compilador usado foi o `g++`, compilador integrante da `gcc`[1]. Para automatizar a compilação, fez-se uso de um arquivo Makefile[2].

O editor de texto usado para escrever os códigos usados no trabalho foi o **Sublime text**[3].

[1] Originalmente escrito como compilador para o sistema operacional GNU, a *GNU Compiler Collection* é um conjunto de compiladores de diversas linguagens (C, C++, Fortran, Ada, Go, etc) e é distribuído pela *Free Software Foundation* (FSF). Seu site oficial é: <https://gcc.gnu.org/>.

[2] O arquivo Makefile define regras de compilação que serão seguidas no projeto. Ele é interpretado pelo programa **make**. A página oficial é: <https://www.gnu.org/software/make/>.

[3] Site oficial: <https://www.sublimetext.com/>.

2.2 Algoritmos implementados

Nessa seção será descrito os algoritmos implementados e análisados, como também será posto seus respectivos códigos usados no projeto. Perceba que todas as funções possuem a mesma assinatura. Isso foi feito para facilitar na implementação dos códigos no projeto.

2.2.1 Insertion sort

O *Insertion sort* funciona da seguinte maneira: você tem como entrada um vetor de elementos, ele irá percorrer índice por índice e a cada iteração pega aquele elemento e o coloca na posição correta, realizando as trocas necessárias com os elementos anteriores para só depois avançar na iteração. Para entender melhor, veja a imagem a seguir:

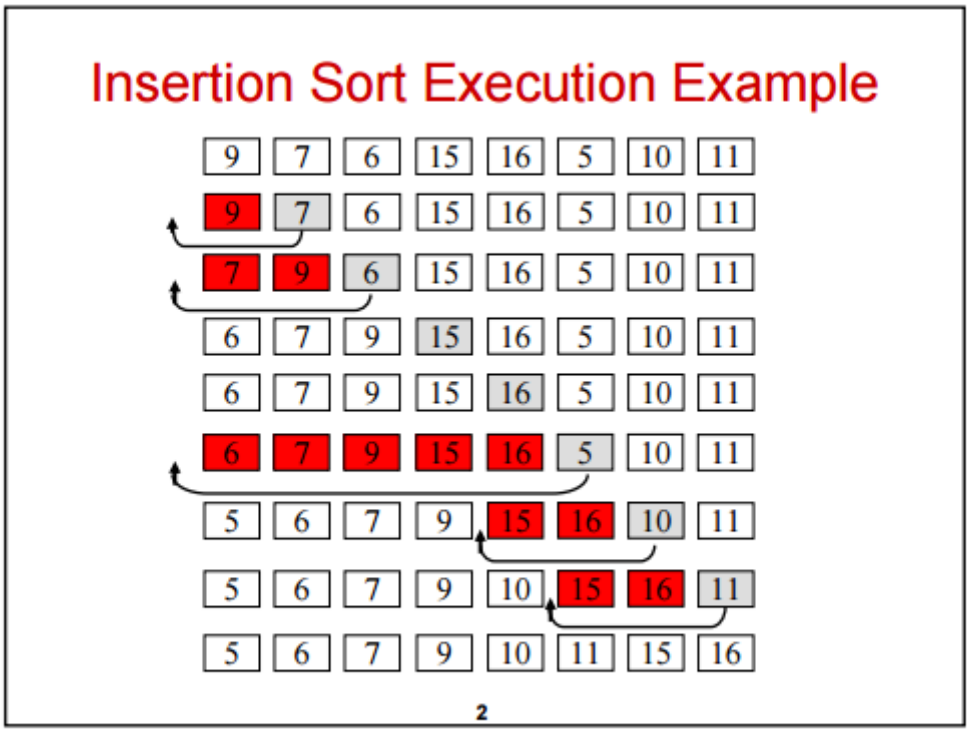

Figura 1 – Insertion Sort.

Imagem retirada de: <http://www.geeksforgeeks.org/insertion-sort/>.

Suas informações referentes a complexidade e estabilidade são:

Complexidade pior caso	$O(n^2)$
Complexidade caso médio	$O(n^2)$
Complexidade melhor caso	$O(n)$
Estabilidade	Estável

Tabela 1 – Complexidade e estabilidade do Insertion Sort.

No código a seguir, passamos como argumento o vetor que queremos ordenar. Ignoramos os outros dois parâmetros (`_left` e `_right`), pois eles foram colocados para apenas termos as mesmas assinaturas que as outras funções do projeto.

```
1  void insertion_sort ( vector<int> & _vetor, int _left = 0, int
       _right = 0 )
2  {
3          // Tamanho do vetor
4          int size = _vetor.size();
5          // Variavel auxiliar
6          int aux;
7
8          // Percorremos o vetor
9          for (int i = 0; i < size-1; i++)
10         {
11                 // Pegamos o indice da frente ao que estamos no
                       ciclo do 'for'
12                 int j = i+1;
13                 // Salvamos seu valor na variaavel auxiliar
14                 aux = _vetor[j];
15                 /*
16                 * Enquanto 'j' nao for o primeiro valor do vetor
17                 * e 'aux' for menor que o elemento anterior a 'j
                       '.
18                 */
19                 while ((j > 0) && (aux < _vetor[j-1]))
20                 {
21                         // Colocamos os elementos anteriores a 'j
                               ' no seu lugar
22                         _vetor[j] = _vetor[j-1];
23                         // Voltamos um indice no valor de 'j'
24                         j--;
25                 }
26                 // Colocamos o valor anteriormente salvo na
                       variavel auxiliar no seu devido lugar
27                 _vetor[j] = aux;
28         }
29 }
```

2.2.2 Selection sort

O *Selection sort* funciona da seguinte maneira: você tem como entrada um vetor de elementos, ele irá percorrer todo o vetor atrás do menor elemento. Após percorrido, irá trocar a posição do elemento de menor valor com a posição da iteração. Assim, a cada ciclo ele garante que os elementos da posição inicial até $i - 1$ (i = iteração) já estejam ordenados. Para entender melhor, veja a imagem a seguir:

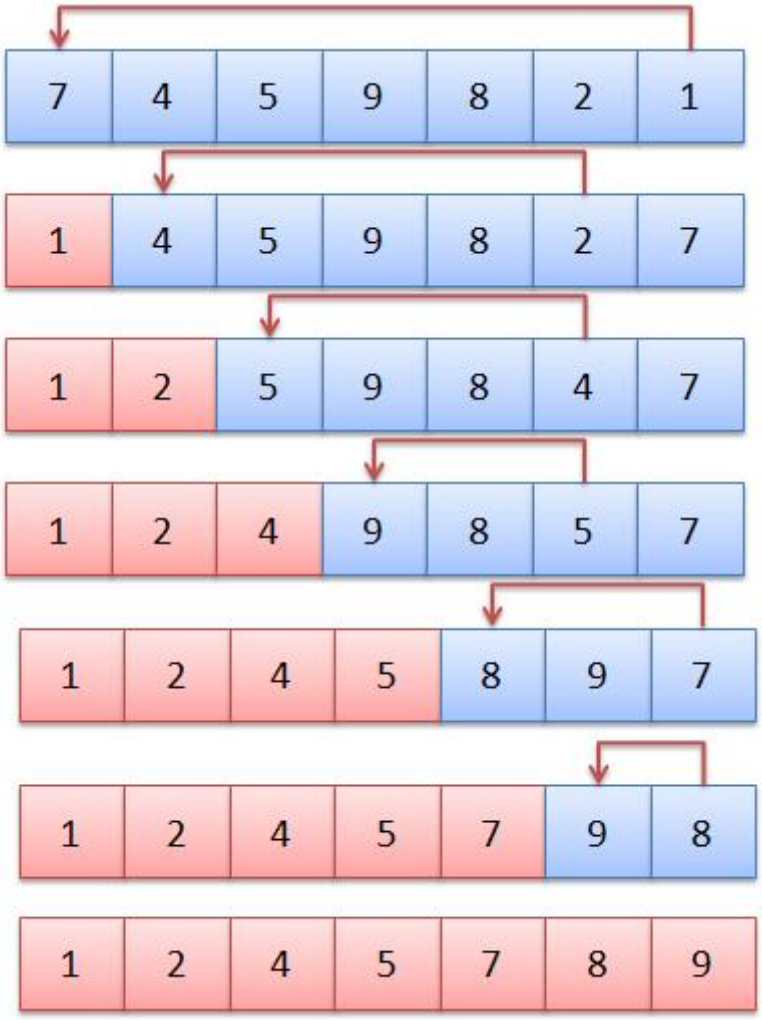

Figura 2 – Selection Sort.

Imagem retirada de:
<http://nerds-attack.blogspot.com.br/2012/09/estrutura-dados-selection-sort.html>.

Suas informações referentes a complexidade são:

Complexidade pior caso	$O(n^2)$
Complexidade caso médio	$O(n^2)$
Complexidade melhor caso	$O(n^2)$

Tabela 2 – Complexidade do Selection Sort.

No código a seguir, passamos como argumento o vetor que queremos ordenar. Ignoramos os outros dois parâmetros (`_left` e `_right`), pois eles foram colocados para apenas termos as mesmas assinaturas que as outras funções do projeto.

```
1  void selection_sort ( vector<int> & _vetor, int _left = 0, int
       _right = 0 )
2  {
3          // Tamanho do vetor
4          int size = _vetor.size();
5          // Indice do menor valor
6          int menor;
7
8          // Percorremos o vetor
9          for (int i = 0; i < size; i++)
10         {
11                 /*
12                 * Comecamos do elemento 'i', pois sempre iremos
                       garantir que os elementos menor que ci' estejam
                        ja ordenados.
13                 * Logo nao e necessario percorrer eles nesse
                       segundo 'for'
14                 */
15                 for (int j = i; j < size; j++)
16                 {
17                         // Se for o primeiro indice verificado
                               desse novo for, salva ele como o menor
                               valor ate entao
18                         if (i == j)
19                         {
20                                 menor = j;
21                                 continue;
22                         }
23                         /*
24                         * Compara se o valor verificado e menor
                               que o menor valor registrado.
25                         * Se for, salva o novo indice do menor
                               valor
26                         */
27                         if (_vetor[menor] > _vetor[j])
28                                 menor = j;
29                 }
30                 // Realiza a troca
```

```
31                  swap( _vetor, menor, i );
32          }
33  }
```

2.2.3 Bubble sort

O *Bubble sort* funciona da seguinte maneira: você tem como entrada um vetor de elementos, a cada iteração ele irá realizar trocas do elemento atual com os seus seguintes até encontrar a posição ideal do elemento. Para entender melhor, veja a imagem a seguir:

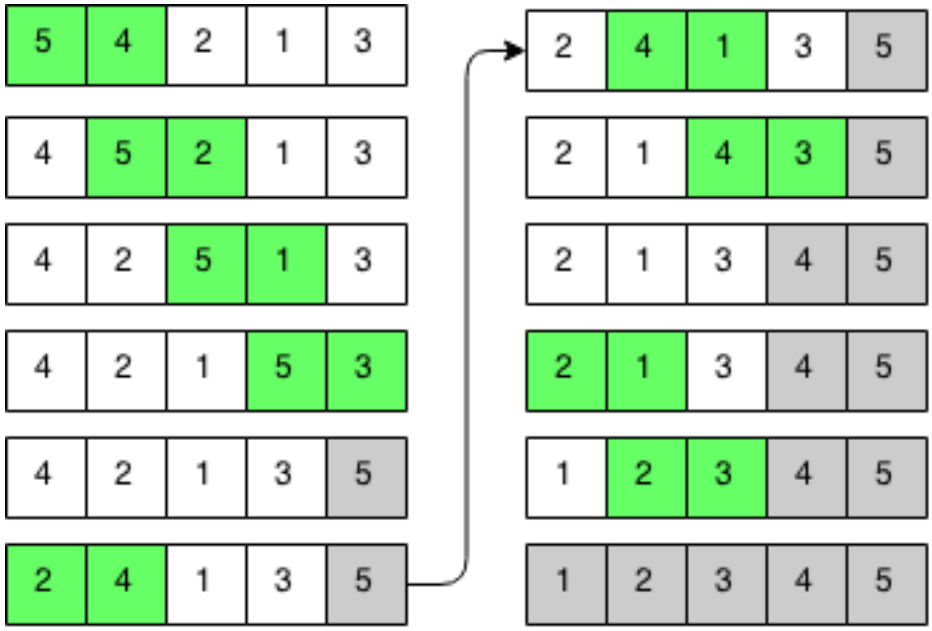

Figura 3 – Bubble Sort.

Imagem retirada de:
<http://www.coisadeprogramador.com.br/algoritmos-ordenacao-bubble-sort/>.

Suas informações referentes a complexidade são:

Complexidade pior caso	$O(n^2)$
Complexidade caso médio	$O(n^2)$
Complexidade melhor caso	$O(n)$

Tabela 3 – Complexidade do Bubble Sort.

No código a seguir, passamos como argumento o vetor que queremos ordenar. Ignoramos os outros dois parâmetros (`_left` e `_right`), pois eles foram colocados para apenas termos as mesmas assinaturas que as outras funções do projeto.

```
1  void bubble_sort ( vector<int> & _vetor, int _left = 0, int
       _right = 0 )
2  {
```

```
3           // Tamanho do vetor
4           int j = _vetor.size();
5           // Percorremos o vetor do seu ultimo indice ate o
               primeiro (direita para a esquerda)
6           for (j -= 1; j > 0; j--)
7           {
8                   /*
9                   * Percorremos o vetor da esquerda para a direita.
10                  * Vamos so ate 'j' pois sempre garantimos que os
                       elementos > 'j' ja estao ordenados
11                  */
12                  for (int i = 0; i < j; i++)
13                  {
14                          // Se o valor atual for maior que o valor
                               a sua frente
15                          if (_vetor[i] > _vetor[i+1])
16                                  // Realizamos a troca dos valores
17                                  swap( _vetor, i, i+1 );
18                  }
19          }
20  }
```

2.2.4 Shell sort

O *Shell sort* é uma extensão do algoritmo *Insertion sort*. Diferente do *Selection sort* que só permite realizar trocas de elementos adjacentes, o *Shell sort* permite trocas de elementos distantes um do outro. Ele funciona da seguinte maneira: você tem como entrada um vetor de elementos, usamos o tamanho desse vetor para decidir qual será a distância inicial entre os elementos que serão trocados. Após cada iteração, dividiremos a distância e assim sucessivamente até atingirmos a troca de elementos adjacentes. Para entender melhor, veja a imagem a seguir:

A sua análise contêm alguns problemas matemáticos muito difíceis, por causa disso, a sua complexidade ainda não é totalmente conhecida. Suas informações referentes a complexidade até então conhecidas são:

Complexidade pior caso	$O(nlog_2n)$ (melhor conhecida)
Complexidade caso médio	depende da sequência da distância
Complexidade melhor caso	$O(n)$

Tabela 4 – Complexidade do Shell Sort.

No código a seguir, passamos como argumento o vetor que queremos ordenar.

Figura 4 – Shell Sort.

Imagem retirada de: <https://pt.wikipedia.org/wiki/Shell_sort>.

Ignoramos os outros dois parâmetros (`_left` e `_right`), pois eles foram colocados para apenas termos as mesmas assinaturas que as outras funções do projeto. Perceba que dividimos a variável 'gap' em 2.2, isso ocorre porque se a divisão der, por exemplo, 2.72727, pegaremos apenas o valor inteiro, ou seja: 2. Assim não teremos problemas na execução do algoritmo.

```
1 void shell_sort ( std::vector<int> & _vetor, int _left = 0, int
     _right = 0  )
2 {
3         // Tamanho do vetor
```

```
4           size_t size = _vetor.size();
5           // Espaco entre elementos comparados
6       size_t gap = size / 2;
7       // Ate chegar a menor comparacao de elementos
8       while (gap > 0)
9       {
10          // Do primeiro indice ate o indice que permite realizar a
                comparacao
11          for (size_t i = 0; i < (size - gap); i++)
12          {
13                  // Indice do elemento que sera usado na
                        comparacao
14              size_t j = i + gap;
15              // Valor do elemento que sera usado na comparacao
16              size_t tmp = _vetor[j];
17              // Realiza as trocas
18              while ((j >= gap) && (tmp < _vetor[j - gap]))
19              {
20                  _vetor[j] = _vetor[j - gap];
21                  j -= gap;
22              }
23              _vetor[j] = tmp;
24          }
25          // Mudanca do valor da distancia dos elementos
26          if (gap == 2)
27              gap = 1;
28          else
29                  // Com 2.2, iremos arredondar o valor para baixo
30              gap /= 2.2;
31      }
32  }
```

2.2.5 Quicksort

2.2.6 Merge sort

2.2.7 Radix sort (LSD)

2.2.8 Visão geral

textbfAlgoritmo	**Complexidade**		
	Melhor caso	Caso médio	Pior caso
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n)$	Depende da sequência da distância	$O(nlog_2n)$ (melhor conhecida)
Quicksort	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$
Merge sort	$\theta(nlogn)$	$\theta(nlogn)$	$\theta(nlogn)$
Radix sort (LSD)			$\theta(nk)$

Tabela 5 – Complexidade e estabilidade do Insertion Sort

2.3 Cenários

No projeto foi-se aplicado os algoritmos anteriormente apresentados em um total de 3 cenários distintos. Informações sobre esses cenários podem ser vistos logo a seguir.

2.3.1 Arranjos com elementos aleatórios

2.3.2 Arranjos com elementos não decrescentes

2.3.3 Arranjos com elementos não crescentes

3 Resultados

4 Discussão