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

Lista de ilustrações

Lista de tabelas

Sumário

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]:

```
1 BINDIR = bin
2 SRCDIR = src
3 INCLUDEDIR = include
4 APPDIR = application
5 OBJDIR = build
6 TESTDIR = test
7
```

[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/>.

```
8  CC = g++
9  CFLAGS = -O3 -Wall -ansi -pedantic -std=c++11 -I $(INCLUDEDIR)
10 LDFLAGS =
11
12 BIN = ${BINDIR}/main
13 APP = ${APPDIR}/main.cpp
14
15 SRC = $(wildcard $(SRCDIR)/*.cpp)
16 OBJS = $(patsubst $(SRCDIR)/%.cpp,$(OBJDIR)/%.o,$(SRC))
17 APPOBJ = $(patsubst $(APPDIR)/%.cpp,$(OBJDIR)/%.o,$(APP))
18
19 _TESTS = $(wildcard $(TESTDIR)/*.cpp)
20 TESTS = $(patsubst %.cpp,%,$(_TESTS))
21
22 $(BIN): $(OBJS) $(APPOBJ)
23   $(CC) -o $(BIN) $(APPOBJ) $(OBJS) $(CFLAGS) $(LDFLAGS)
24
25 $(APPOBJ): $(APP)
26   $(CC) -c -o $@ $< $(CFLAGS)
27
28 ${OBJDIR}/%.o: $(SRCDIR)/%.cpp
29   $(CC) -c -o $@ $< $(CFLAGS)
30
31 test: $(TESTS)
32   $(info ************  Testes concluidos com sucesso!
       ************)
33
34 $(TESTDIR)/t_%: $(TESTDIR)/t_%.cpp $(OBJS)
35   $(CC) -o $@ $< $(OBJS) $(CFLAGS) $(LDFLAGS)
36   $@
37
38 clean:
39   rm -f $(BIN) $(OBJS) $(APPOBJ)
40   rm -f $(TESTS)
```

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

Para a medição de tempo, utilizou-se a biblioteca `std::chrono` do próprio `C++`, cálculando o tempo em milissegundos.

[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:

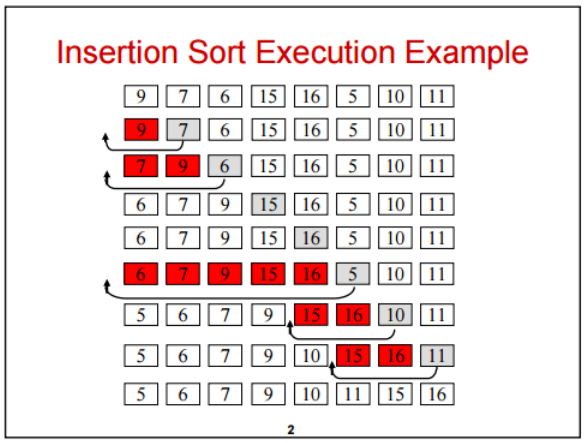

Figura 1 – Insertion Sort.

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

Ele é considerado um algoritmo estável. Suas informações referentes a complexidade e estabilidade são:

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

Tabela 1 – Complexidade 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:

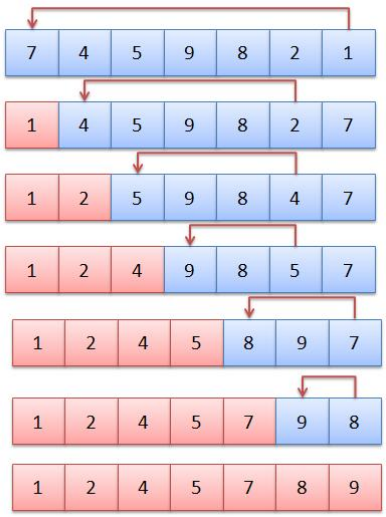

Figura 2 – Selection Sort.

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:

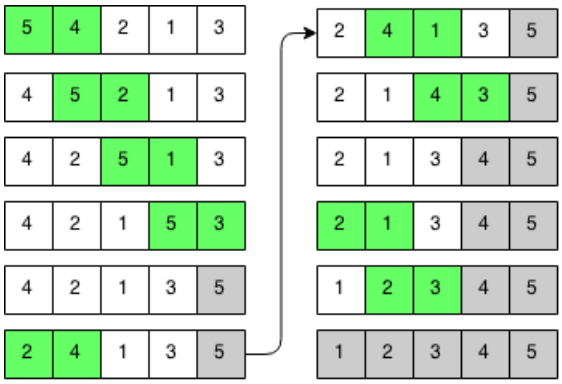

Figura 3 – Bubble Sort.

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:

Figura 4 – Shell Sort.

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

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. 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 ( 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

O *Quicksort* funciona da seguinte maneira: você tem como entrada um vetor de elementos, é então decidido um pivô e esse pivô irá dividir o vetor em duas partes (uma com números menores que ele na esquerda e outro com números maiores que ele na direita). A cada iteração, iremos aplicar essa mecânica do pivô nos subvetores, gerando outros subvetores até não conseguirmos mais gerar subvetores (vetores de 1 elemento só). Devido a essa mecânica do pivô, teremos os elementos dos subvetores organizados, logo todo o vetor original também estará organizado. Para entender melhor, veja a imagem a seguir:

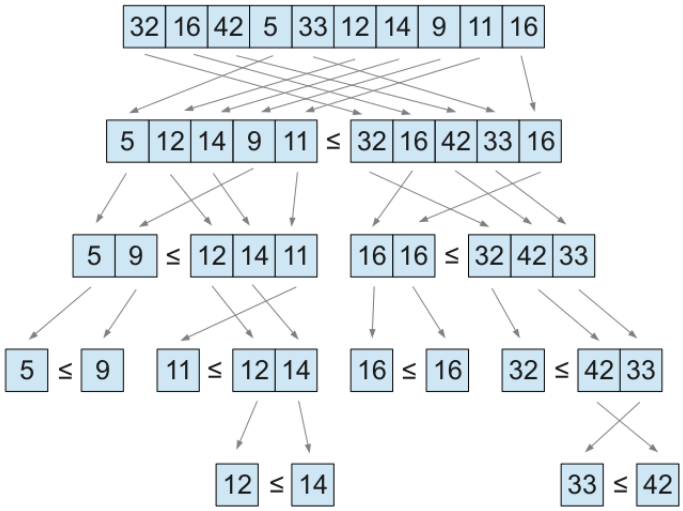

Figura 5 – Quicksort.

Imagem retirada de: <https://simpledevcode.wordpress.com/2014/06/13/quicksort-in-c/>.

Ele é considerado um algoritmo não estável. Suas informações referentes a complexidade são:

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

Tabela 5 – Complexidade do Quicksort.

No código a seguir, passamos como argumento o vetor que queremos ordenar e o índice inicial e final do vetor (respectivamente `_left` e `_right`). Como esse algoritmo é recursivo[4], usamos o valor -1 no `_right` para identificar sua primeira chamada.

```
1 void quicksort ( vector<int> & _vetor, int _left = 0, int _right
     = -1 )
2 {
3     // Usamos esse 'if' para sabermos se e a primeira vez que o
        metodo e chamado
4     if (_right == -1)
5         // Ultimo indice valido do vetor
6         _right = _vetor.size()-1;
7     // Para caso os indices cheguem a '_left >= _right', pois ai
        encerrar a funcao do quicksort naquele momento
8     if(_left < _right)
9     {
```

[4] Recursividade significa que uma sub-rotina (função ou método) que chama a si mesma até atingir uma condição para se encerrar.

```
10          // Pivo sera o primeiro indice presente na esquerda
11          int pivot = _left;
12          // Percorremos os valores do '_left'+1 (o elemento a
               direita do pivo) ate o '_right'
13          for (int i = _left + 1; i < _right; i++)
14          {
15              /*
16              * Se o numero for menor que o elemento presente no
                   indice '_left',
17              * realizamos a troca entre esse elemento e o elemento
                    a direita do pivo.
18              * Sempre iremos mover o pivo para a direita nesse
                   momento de troca
19              * (notar o '++') na esquerda do 'pivot'.
20              */
21              if (_vetor[i] < _vetor[_left])
22                  swap( _vetor, i, ++pivot );
23          }
24          // Realiza troca do primeiro indice presente na esquerda
               pelo 'pivot'
25          swap( _vetor, _left, pivot );
26          // Chama os metodos recursivos para os elementos a
               esquerda e a direita do 'pivot', respectivamente
27          quicksort( _vetor, _left, pivot );
28          quicksort( _vetor, pivot + 1, _right );
29      }
30  }
```

2.2.6 Merge sort

O *Merge sort* funciona da seguinte maneira: você tem como entrada um vetor de elementos, pegamos o indice do meio do vetor e usaremos ele para dividirmos o vetor em duas partes e usarmos o método do *Merge sort* neles, até termos vetores de apenas um elemento. Nisso, iremos aplicar a função *merge* do método e juntaremos esses subvetores formados pelas partes do vetor original. Para entender melhor, veja a imagem a seguir:

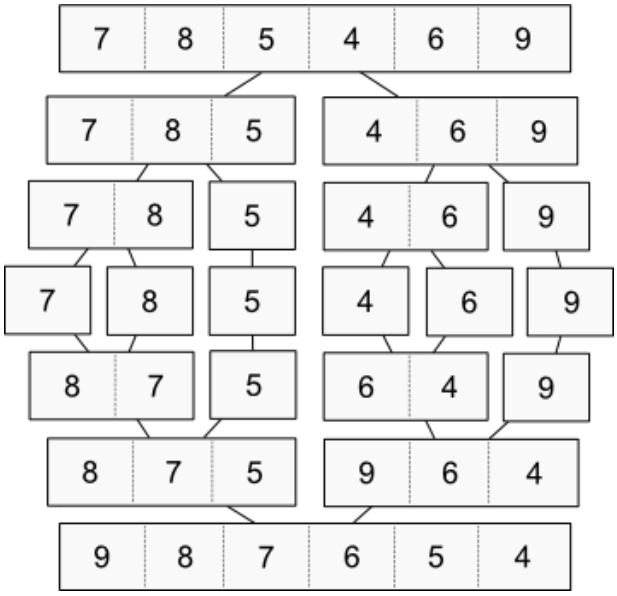

Figura 6 – Merge Sort.

Imagem retirada de: <https://www.programming-algorithms.net/article/39650/Merge-sort>.

Suas informações referentes a complexidade são:

Complexidade pior caso	$\theta(nlogn)$
Complexidade caso médio	$\theta(nlogn)$
Complexidade melhor caso	$\theta(nlogn)$

Tabela 6 – Complexidade do Merge Sort.

No código a seguir, passamos como argumento o vetor que queremos ordenar e o índice inicial e final do vetor (respectivamente `_left` e `_right`). Como esse algoritmo é recursivo, usamos o valor -1 no `_right` para identificar sua primeira chamada. A função *merge* é o que realizará a junção dos dois subvetores.

```
1  void merge ( vector<int> & _vetor, size_t _left, size_t _middle,
       size_t _right )
2  {
3      // Tamanhos do vetor pai, vetor da esquerda e vetor da
           direita, respectivamente
4      // Final - inicial + 1
5      size_t len_A = _right  - _left + 1;
6      // Meio - inicial + 1
7      size_t len_L = _middle - _left + 1;
8      // Todo - Tamanho do vetor da esquerda
9      size_t len_R = len_A - len_L;
10
```

```
11      // Vetores auxiliares, para os elementos da esquerda e da
            direita, respectivamente
12      int *L = new int[ len_L ];
13      int *R = new int[ len_R ];
14
15      // Copiar os elementos de cada metade para seus respectivos
            vetores auxiliares.
16      std::copy( _vetor.begin() + _left,         _vetor.begin() + (
            _middle + 1), L );
17      std::copy( _vetor.begin() + (_middle + 1), _vetor.begin() + (
            _right + 1),  R );
18
19      // Contadores de cada vetor
20      size_t i = 0; // Associado ao L
21      size_t j = 0; // Associado ao R
22      size_t k = _left; // Associado ao A
23
24      /*
25      * Sobrescreve os elementos do vetor pai com o menor elemento
            encontrado
26      * em um dos dois vetores auxiliares
27      */
28      while( i < len_L and j < len_R )
29          _vetor[k++] = ( L[i] < R[j] ) ? L[i++] : R[j++];
30
31      // Apos terminar um dos vetores auxiliares, falta o restante
            dos elementos do outro vetor
32      if ( i < len_L )
33          std::copy( L+i, L+len_L, _vetor.begin()+k ); // Copie o
                que sobrou de L.
34      else
35          std::copy( R+j, R+len_R, _vetor.begin()+k ); // Copie o
                que sobrou de R.
36
37      // Libera a memoria usada nos vetores auxiliares.
38      delete [] L;
39      delete [] R;
40  }
41
42  void merge_sort ( vector<int> & _vetor, int _left = 0, int _right
        = -1 )
43  {
```

```
44      // Usamos esse 'if' para sabermos se e a primeira vez que o
            metodo e chamado
45      if (_right == -1)
46          // Ultimo indice valido do vetor
47          _right = _vetor.size()-1;
48
49      // Caso base: ainda tem pelo menos 2 elementos pra ordenar.
50      if ( _left < _right )
51      {
52          // Elemento do meio, para dividir as metades.
53          int m = ( _left + _right ) / 2;
54          // Chama recursivamente o metodo para a primeira metade
                do vetor e depois para a segunda metade
55          merge_sort( _vetor, _left, m );
56          merge_sort( _vetor, m+1, _right );
57          // Realiza a operacao de juncao das duas partes
58          merge( _vetor, _left, m, _right );
59      }
60  }
```

2.2.7 Radix sort (LSD)

O *Radix sort* funciona da seguinte maneira: você tem como entrada um vetor de elementos, a cada iteração ele irá analisar o digito do elemento, podendo começar da esquerda para a direita (*MSD - Most significant digit*, digito mais significativo) ou da direita para a esquerda (*LSD - Least significant digit*, digito menos significativo), avançando de digito a cada iteração. Para entender melhor, veja a imagem a seguir:

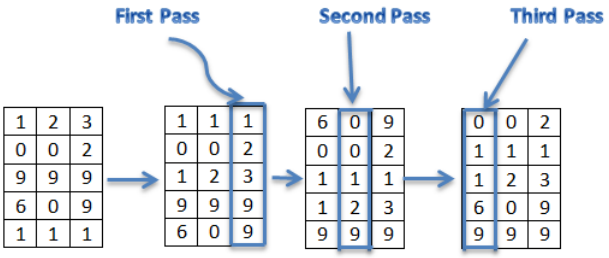

Figura 7 – Radix Sort.

Imagem retirada de: <http://scanftree.com/Data_Structure/radix-sort>.

Para todos os casos, ele possuirá o mesmo grau de complexidade. Suas informações referentes a complexidade são:

Complexidade pior caso	$\theta(nk)$
Complexidade caso médio	
Complexidade melhor caso	

Tabela 7 – Complexidade do Radix 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 radix_sort ( vector<int> & _vetor, int _left = 0, int _right
       = 0 )
2  {
3      // Vetor temporario para realizar a ordenacao
4      vector<int> temp[10];
5      // Pega o maior elemento do vetor, logicamente e o que contem
           maior quantidade de digitos
6      int max = *( max_element ( std::begin(_vetor), std::end(
           _vetor) ) );
7      // Essa variavel ira nos ajudar a percorrer cada digito dos
           numeros presentes no vetor
8      int n = 1;
9
10     while (n <= max)
11     {
12         // Percorremos o vetor
13         for(auto v : _vetor)
14         {
15             // Pegamos o digito que estamos verificando
16             int lsd = (v/n)%10;
17             // Adicionamos ele a seu respectivo lugar no vetor
                   temporario
18             temp[lsd].emplace_back(v);
19         }
20
21         int k = 0;
22         // Percorremos o vetor temporario (que agr sera uma
               especie de matriz)
23         for (auto &v: temp)
24         {
25             // Se nao existir nenhum elemento nessa parte do
                   vetor, pulamos ela
26             if (v.size() <= 0)
```

```
27                  continue;
28              // Caso exista, iremos percorrer os elementos
                   registrados aqui
29              for (auto num: v)
30                  // Passamos os elementos agora organizados para o
                        vetor original
31                  _vetor[k++] = num;
32
33              v.clear();
34          }
35          // Multiplicamos por 10 para que no calculo do 'lsd',
               estejamos verificando o proximo digito
36          n *= 10;
37      }
38  }
```

2.2.8 Visão geral

A tabela a seguire, serve para se ter uma melhor visão dos graus de complexidade de cada algoritmo implementado aqui nesse trabalho.

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	$O(nlogn)$	$O(nlogn)$	$O(n^2)$
Merge sort	$\theta(nlogn)$	$\theta(nlogn)$	$\theta(nlogn)$
Radix sort (LSD)			$\theta(nk)$

Tabela 8 – Visão geral dos algoritmos de ordenação

2.3 Cenários

Foi implementado um total de 3 cenários distintos, nos quais foram aplicados os algoritmos anteriormente apresentados. Os cenários foram:

1. Arranjos com elementos aleatórios;
2. Arranjos com elementos em ordem crescente;

3. Arranjos com elementos em ordem decrescente.

Em todos os 3 cenários pode ocorrer de haver elementos repetidos. Eles foram aplicados nessa mesma ordem apresentada anteriormente. O mesmo arranjo é usado nos 3 cenários, pois no primeiro ele finaliza com o arranjo ordenado em ordem crescente, aí então é aplicado no segundo cenário e, após, é apenas invertido sua ordem, se tornando em um arranjo decrescente.

Para garantirmos que os cenários não se repitam durante o experimento (os números aleatórios resultarem em um arranjo crescente, por exemplo), foi escrito duas funções, uma para gerar os números aleatórios no arranjo (`new_numbers`) e outra para validar o arranjo (`verify_order`). O código de ambas se encontra a seguir:

```
1  void new_numbers (vector<int> & _vetor, size_t _num )
2  {
3    // Insere novos numeros aleatorios de 0 a '_num'
4    for (size_t i = 0; i < _num; i++)
5      _vetor[i] = rand() % _num;
6    // Verifica se os elementos do vetor estao realmente aleatorios
7    bool verify = verify_order(_vetor, ALEATORIO);
8    // Se nao estiverem e o vetor tiver mais de 2 elementos, ele
       chama novamente a funcao
9    if (_num > 2 && !verify)
10     new_numbers(_vetor, _num);
11 }
12
13 bool verify_order ( vector<int> & _vetor, int _order )
14 {
15   // Verifica se os elementos do vetor estao em ordem crescente
16   if (_order == CRESCENTE)
17   {
18     for (size_t i = 0; i < _vetor.size()-1; i++)
19       if (_vetor[i] > _vetor[i+1]) return false;
20   }
21   // Verifica se os elementos do vetor estao em ordem decrescente
22   else if (_order == DECRESCENTE)
23   {
24     for (size_t i = 0; i < _vetor.size()-1; i++)
25       if (_vetor[i] < _vetor[i+1]) return false;
26   }
27   // Verifica se os elementos do vetor sao realmente aleatorios
28   else
29   {
```

```
30      if (verify_order(_vetor, CRESCENTE) || verify_order(_vetor,
            DECRESCENTE))
31        return false;
32    }
33
34    return true;
35  }
```

2.4 Geração de gráficos

Foram executados 50 vezes cada um dos 3 cenários aplicando cada uma das 7 funções para cada uma das 38 entrada, totalizando 39900 registros em um arquivo *CSV*[5]. Precisava-se, então, de gerar uma média de cada um dos cenários, em cada uma das funções e em cada uma das entradas. Para isso, utilizou-se a linguagem `Python` com a biblioteca `pandas`, os códigos foram executados no *framework* `Anaconda` em um `Jupyter notebook`. O código para gerar um novo *CSV* com as médias dos casos foi:

```
1 # Ler arquivo CSV original
2 data = pd.read_csv('/var/www/aulas/edb/sort/test/tests.csv', sep=
      ';')
3 # Agrupa os dados pela ordem de algoritmos, numero de elementos
      no vector e cada caso, nessa ordem
4 # Realiza-se uma media e depois transforma em um dataframe
5 dt = data.groupby(['algoritmo', 'n_elementos', 'caso'])['tempo'].
      mean().to_frame()
6 # Exporta os dados para um novo arquivo csv
7 dt.to_csv('tests_out.csv', sep=';')
```

Foi utilizado a biblioteca `bokeh` para gerar os gráficos. Foi atribuido as seguintes cores para cada algoritmo nos gráficos:

Algoritmo	**Cor**
Insertion Sort	Azul
Selection Sort	Verde
Bubble Sort	Laranja
Shell Sort	Vermelho
Quicksort	Roxo
Merge Sort	Marrom
Radix sort (LSD)	Preto

Tabela 9 – Algoritmos e suas respectivas cores nos gráficos.

[5] *Comma-separated values*. Arquivos de texto em formato de tabela (com linhas e cada atributo na linha é separado de outro atributo por um caractere especial, normalmente “,” ou “;”) usado para armazenamento de dados.

3 Resultados

Nessa capítulo, constará os gráficos e tabelas referentes aos resultados do experimento. Primeiro será apresentado os dados de cada um dos algoritmos e após uma visão geral.

3.1 Resultados de cada algoritmo

A seguir, poderá ser encontrado as tabelas e gráficos referentes as médias finais de cada algoritmo para sua respectiva entrada de elementos. O tempo foi medido em milissegundos. A primeira coluna consta a quantidade de elementos presente no arranjo, as outras 3 colunas constam, respectivamente, o tempo médio que o algoritmo levou para arranjos aleatórios, arranjos com elementos em ordem crescente e arranjos com elementos em ordem decrescente.

3.1.1 Insertion sort

n-elementos	aleatorio	crescente	decrescente
1	8.04e-05	7.599999999999999e-05	6.954e-05
10	0.00012396	0.00010926000000000002	0.00023034
20	0.00013946	0.00010836	0.0006446200000000003
30	0.00016641999999999994	0.00014246	0.0012635200000000002
40	0.00015675999999999996	0.00012809999999999994	0.0016115999999999995
50	0.00014634	0.00011614	0.0021676399999999985
60	0.0002198399999999999	0.00015152000000000007	0.0032595
70	0.00019559999999999998	0.00013476000000000002	0.00393508
80	0.0002027599999999999	0.0001478999999999999	0.0047943
90	0.00036754	0.00025696000000000003	0.008275399999999997
100	0.00034843999999999994	0.00024742	0.007875640000000003
200	0.0007557199999999999	0.00043473999999999987	0.031773180000000005
300	0.0010309999999999998	0.00040790000000000016	0.05496282000000001
400	0.0015114199999999994	0.0005074399999999998	0.09509326000000004
500	0.00219502	0.0006359999999999997	0.1495683
600	0.002993340000000001	0.0007875200000000003	0.2143029
700	0.003830680000000003	0.0008488999999999999	0.28881464
800	0.005107379999999999	0.0012716999999999993	0.3994657999999999
900	0.006138840000000003	0.0013167799999999998	0.5211207200000001

1000	0.007145740000000003	0.0013758599999999998	0.6369691799999998
2000	0.026273739999999997	0.00311778	2.3876307999999997
3000	0.056982519999999974	0.004042419999999999	5.718184399999998
4000	0.10421780000000001	0.005860799999999999	10.090771600000002
5000	0.15512391999999994	0.007102319999999999	16.271274000000002
6000	0.22033988	0.007708499999999997	21.739686000000006
7000	0.35038776000000005	0.00848502	28.433606000000005
8000	0.3797302600000002	0.00986584	37.877044
9000	0.4728525000000002	0.011328060000000004	48.150676
10000	0.6490004000000004	0.01405342	60.98220600000001
20000	2.75290912	0.0243977	239.72554
30000	5.509638259999997	0.036425760000000015	541.18108
40000	10.64687296	0.04892822	961.1364199999997
50000	14.350246619999993	0.06398298000000001	1446.6329999999996
60000	20.43972622	0.08489742000000003	2166.1162000000004
70000	28.421486400000006	0.08818287999999999	2985.9566000000013
80000	36.970087080000006	0.10197196	3923.987
90000	47.03672537999997	0.11550512000000003	4748.289600000001
100000	57.605872360000006	0.12649618	5841.539400000002

Tabela 10 – Média de tempo do Insertion Sort

3.1.2 Selection sort

n-elementos	aleatorio	crescente	decrescente
1	8.894000000000001e-05	7.984e-05	7.426e-05
10	0.0003354800000000001	0.00032264	0.00032304000000000013
20	0.00088672	0.00087854	0.0008772800000000001
30	0.00163884	0.00164402	0.0016425
40	0.00235116	0.0022838599999999995	0.0024117999999999987
50	0.0037228999999999999	0.0036606999999999976	0.0037776400000000014
60	0.0052438200000000002	0.0051922000000000001	0.005380499999999999
70	0.0072170199999999985	0.007260260000000002	0.007330620000000001
80	0.009653020000000005	0.00934112	0.00959626
90	0.01401454	0.013664759999999993	0.01360958
100	0.017578299999999998	0.01675234	0.016640520000000002
200	0.06357926	0.061996980000000014	0.06360304
300	0.13560870000000005	0.13545256000000006	0.13588302
400	0.24841838	0.24954946	0.2527263

500	0.3749042000000001	0.3741579799999999	0.3767620199999998
600	0.54610156	0.5467687600000001	0.5502927
700	0.7525538599999999	0.7505205599999999	0.7509273399999998
800	0.97494558	0.9773224599999996	0.97801136
900	1.2451786	1.2468340000000002	1.2516400000000003
1000	1.566225	1.5781350000000005	1.5818015999999997
2000	6.214042599999999	6.181636400000002	6.1742517999999995
3000	13.835297999999998	13.929510000000008	13.812009999999995
4000	24.69347599999999	24.693382	24.749907999999994
5000	38.498906	38.63013599999999	38.89369600000001
6000	55.63702000000002	55.665062	55.708674
7000	75.67342	75.441478	75.837246
8000	98.995134	98.68345199999996	99.21208599999996
9000	124.5302	124.4241	124.49490000000006
10000	154.11332	153.94469999999995	153.96908000000005
20000	615.3784599999998	614.3246599999999	615.98708
30000	1392.4703999999995	1393.2522	1392.0565999999994
40000	2439.398	2437.8770000000004	2437.8432000000007
50000	3816.6306	3817.2038	3818.5996000000014
60000	5482.8278	5483.8052	5483.518
70000	7454.1882	7451.157400000001	7449.478599999998
80000	9721.351200000003	9721.5888	9724.686999999998
90000	12297.200000000004	12295.282000000005	12294.612
100000	15173.762000000002	15172.421999999995	15173.320000000003

Tabela 11 – Média de tempo do Selection Sort

3.1.3 Bubble sort

n-elementos	aleatorio	crescente	decrescente
1	6.361999999999996e-05	6.432e-05	6.780000000000001e-05
10	0.0002415399999999999	0.00023302	0.00044784
20	0.00058376	0.0005693999999999999	0.00135716
30	0.00110488	0.0010634600000000002	0.0027495400000000013
40	0.0013097200000000005	0.0012513600000000002	0.002784560000000001
50	0.0019252599999999987	0.0018504000000000007	0.004268779999999997
60	0.0026985000000000025	0.002573219999999999	0.006250519999999997
70	0.0037903	0.0037311999999999996	0.00912808
80	0.004452579999999997	0.004286640000000004	0.010519800000000006

90	0.007167499999999999	0.00690576	0.018331120000000003
100	0.010191660000000002	0.009836459999999998	0.02556024
200	0.031928420000000006	0.030499060000000012	0.08034861999999998
300	0.06271342	0.058413980000000025	0.15869872
400	0.12703450000000002	0.12277264	0.32672432
500	0.15926972	0.14695972	0.39194424
600	0.24205966000000004	0.22538618	0.59441138
700	0.30831228000000005	0.28411334	0.7618253200000001
800	0.41232524	0.3887219	1.0403300000000002
900	0.5244326799999999	0.5017058999999999	1.4129602
1000	0.6632637199999999	0.6323091399999998	1.7084739999999998
2000	2.6785526000000006	2.4618416	6.852431000000003
3000	5.899901000000002	5.5032206000000015	14.892673999999996
4000	10.626763399999998	9.9530382	27.20265400000001
5000	16.889428000000002	15.702858	42.71952800000001
6000	23.651266	22.056402	60.14559200000001
7000	31.952798	29.973362000000005	80.59747800000002
8000	41.97577400000001	39.28520799999999	108.66099999999996
9000	52.945266	49.81960599999999	133.69294
10000	65.443436	61.02904800000001	164.73821999999998
20000	262.0408	243.97320000000008	664.5801800000003
30000	605.2050599999999	561.17494	1528.3691999999994
40000	1034.4937599999996	975.5939199999997	2582.915000000001
50000	1635.3270000000002	1478.5251999999996	4021.881600000001
60000	2286.8478	2155.2990000000004	5855.2588
70000	3126.2176	3182.7952	8112.5588000000025
80000	4079.674999999999	4009.7566	10354.742000000004
90000	5183.987400000001	4886.374399999999	12907.100000000002
100000	6387.3636000000015	6004.119800000001	15843.345999999996

Tabela 12 – Média de tempo do Bubble Sort

3.1.4 Shell sort

n-elementos	aleatorio	crescente	decrescente
1	8.669999999999999e-05	7.556000000000003e-05	9.596e-05
10	0.0001896	0.00018413999999999995	0.00023594
20	0.00029992	0.00028809999999999985	0.00035953999999999994
30	0.0005192199999999999	0.0004834999999999999	0.0006069600000000001

40	0.0004316400000000001	0.0003913200000000001	0.0005659599999999999
50	0.0005328000000000002	0.0005095	0.0008365400000000001
60	0.0006757799999999997	0.00065498	0.00129804
70	0.0009422800000000004	0.00088366	0.00164948
80	0.0007921600000000001	0.0007514399999999997	0.0014870999999999999
90	0.00104282	0.00099746	0.00184108
100	0.0011963599999999998	0.00110566	0.0019554200000000007
200	0.0022194400000000013	0.0020728800000000013	0.003929299999999999
300	0.00465906	0.0046492600000000005	0.007978119999999998
400	0.005206160000000003	0.004807440000000003	0.00836884
500	0.00898828	0.009464040000000002	0.01586288
600	0.00847214	0.008156859999999998	0.013151459999999998
700	0.00999584	0.00949136	0.014142320000000003
800	0.011804099999999998	0.010908580000000005	0.0196165
900	0.013808080000000006	0.01290692	0.02222180000000001
1000	0.020440859999999998	0.01997758	0.033130759999999995
2000	0.043242759999999984	0.04110132000000001	0.06234480000000001
3000	0.07187110000000001	0.06789051999999997	0.10273514
4000	0.08789107999999997	0.08007718000000001	0.11893252
5000	0.0925285	0.08704414000000005	0.13805626
6000	0.13269892	0.12264468	0.17419702000000006
7000	0.15157742	0.14018013999999998	0.20747252
8000	0.16267289999999995	0.15083636000000006	0.22027041999999994
9000	0.19120046	0.18119348	0.2729812800000001
10000	0.21462448000000006	0.1969716199999999	0.29579946
20000	0.44199366	0.40533506	0.6213983600000001
30000	0.7816365000000002	0.7256145400000001	1.0407506999999998
40000	0.9656276600000002	0.8938177200000001	1.2944742000000002
50000	1.2180288000000001	1.1319468000000001	1.6699556000000002
60000	1.4890634000000005	1.3584957999999998	2.0236593999999997
70000	1.8849568	1.7246824000000005	2.314869200000001
80000	2.1531498	1.978576	2.5989719999999994
90000	2.3803844	2.2038760000000006	3.026679000000001
100000	2.6489558000000004	2.5036772000000003	3.3127751999999986

Tabela 13 – Média de tempo do Shell Sort

3.1.5 Quicksort

n-elementos	aleatorio	crescente	decrescente
1	0.00012871999999999995	0.00013689999999999994	0.00012589999999999994
10	0.00039626	0.0003763799999999984	0.00044716
20	0.0008050400000000002	0.000775679999999997	0.00102498
30	0.0014191599999999998	0.001412140000000005	0.0016674999999999995
40	0.0018122199999999996	0.00181322	0.0020733400000000003
50	0.00259538	0.00275002	0.0030154400000000003
60	0.0036092800000000007	0.0036236200000000014	0.00424246
70	0.005393340000000001	0.0054448	0.006414800000000001
80	0.00650588	0.00610068	0.00709802
90	0.007691779999999999	0.0077901400000000036	0.008782820000000002
100	0.01147832	0.011887899999999998	0.016156040000000003
200	0.03342298000000001	0.033774820000000004	0.04122540000000001
300	0.0800768	0.08261992000000001	0.10349269999999997
400	0.13265208000000006	0.13872214000000002	0.17054222
500	0.20487036000000006	0.20763247999999992	0.2556335
600	0.29414783999999994	0.30181403999999995	0.37839278
700	0.3937792399999999	0.4026754	0.4913740000000002
800	0.5022601	0.5103693799999999	0.6075401200000002
900	0.62613752	0.63325112	0.73611716
1000	0.7853935800000003	0.7949036399999998	0.92125936
2000	3.1397559	3.1925038	3.7847432
3000	7.480459619999998	7.967535400000003	9.2145958
4000	12.50861266	12.986206000000006	15.695778
5000	19.859510499999995	20.230566	24.116454
6000	28.376703000000003	28.678648	34.644298000000006
7000	38.42625838000001	39.241603999999974	47.245038
8000	49.6324631	51.71043599999999	59.259396
9000	64.13671000000001	64.10335400000001	76.19887600000001
10000	79.55824559999999	80.86666199999998	94.18131
20000	312.9152602000001	318.56034	377.9182
30000	710.704617	725.06052	853.4606800000001
40000	1224.1749857999998	1246.1660000000002	1447.7228199999995
50000	1915.2557426000005	1956.4022000000007	2279.37444
60000	2949.207355799999	2940.0198000000005	3432.2615999999994
70000	3819.5877446000004	3936.7234	4711.543400000001
80000	4955.5327499999985	5069.6212	5887.796799999997

90000	6261.575381999997	6391.2022000000015	7419.7930000000015
100000	7705.313315999998	7809.560600000002	9107.002599999998

Tabela 14 – Média de tempo do Quicksort

3.1.6 Merge sort

n-elementos	aleatorio	crescente	decrescente
1	8.745999999999998e-05	8.271999999999998e-05	8.043999999999998e-05
10	0.002561740000000001	0.0021670000000000005	0.0022400399999999987
20	0.0039162999999999976	0.00374204	0.00374542
30	0.006140880000000002	0.005890699999999998	0.0060662999999999976
40	0.00577768	0.005675780000000003	0.005555559999999999
50	0.010145819999999998	0.00992998	0.01006036
60	0.008690020000000003	0.00870846	0.00844154
70	0.010746660000000002	0.01042974	0.01024406
80	0.012068400000000002	0.01188608	0.011587679999999998
90	0.013629999999999998	0.0134486	0.013389020000000005
100	0.022698260000000005	0.02247602	0.02262972
200	0.033423359999999985	0.03198531999999999	0.03158324
300	0.04783078	0.047321639999999984	0.04571464
400	0.07717608000000001	0.07632107999999999	0.07606866
500	0.09068790000000003	0.0902877	0.08931276000000002
600	0.09743568	0.09613956	0.09484492
700	0.12201270000000003	0.12127187999999997	0.12014502
800	0.12465428	0.12419009999999997	0.12402931999999997
900	0.15934056	0.15445996000000006	0.15348805999999993
1000	0.16415146	0.16156558	0.15910045999999994
2000	0.3310599600000001	0.33280864	0.3233439200000001
3000	0.5116175999999999	0.5079330200000001	0.5057538600000001
4000	0.7107825000000004	0.7169150199999997	0.7080439999999999
5000	0.9199447000000001	0.9157468400000004	0.9124175
6000	1.0351199	1.0147118	1.01285676
7000	1.1505416000000004	1.148931	1.1296363999999999
8000	1.3629216000000002	1.3750534	1.3455878000000001
9000	1.7060540000000002	1.6920665999999998	1.6732542
10000	1.7922128000000002	1.7980470000000002	1.7682486000000002
20000	3.4463641999999988	3.4504582000000004	3.3345266000000016
30000	5.5396887999999995	5.442347999999999	5.2768824

40000	6.949690999999999	6.9348797999999965	6.8092882
50000	8.6095736	8.676968400000002	8.6157094
60000	10.710488	10.712239999999998	10.435927999999997
70000	12.382073999999998	12.307936000000005	12.291986
80000	14.026547999999998	14.132902000000005	13.754404000000001
90000	15.851076	15.82669	15.668390000000004
100000	17.481986000000006	17.463604000000007	17.465252

Tabela 15 – Média de tempo do Merge Sort

3.1.7 Radix sort (LSD)

n-elementos	aleatorio	crescente	decrescente
1	0.00012784000000000002	0.00012076000000000003	0.00017714000000000006
10	0.00149714	0.00144342	0.0014305000000000006
20	0.00344698	0.003389179999999999	0.0034288199999999987
30	0.00507238	0.00447688	0.00444216
40	0.00402094	0.00368006	0.003653
50	0.00663616	0.00654854	0.006490940000000003
60	0.0048655	0.00480284	0.00482398
70	0.005444740000000001	0.0054082800000000006	0.005356719999999999
80	0.007354079999999997	0.00752038	0.007321079999999998
90	0.0070997399999999985	0.00711828	0.007069699999999998
100	0.008428419999999997	0.008258339999999998	0.008453219999999997
200	0.013028559999999998	0.012980599999999995	0.01306192
300	0.016468059999999996	0.017437440000000002	0.016625900000000006
400	0.029904819999999988	0.030112839999999988	0.029872779999999995
500	0.027574299999999996	0.027057280000000003	0.027487000000000004
600	0.02777642	0.027417900000000002	0.02782098
700	0.04136696	0.041383920000000005	0.04108437999999999
800	0.03417122000000001	0.033842100000000014	0.034156920000000014
900	0.0393961	0.03909072	0.039995640000000006
1000	0.04125008	0.04174422	0.04123798
2000	0.10889986	0.10946516	0.10775699999999999
3000	0.13803787999999995	0.13844994	0.14038446000000002
4000	0.20519679999999996	0.20418748	0.20233874
5000	0.24257226000000004	0.24173826	0.24388674000000005
6000	0.3376315400000001	0.33423958	0.33214738000000005
7000	0.30743894000000005	0.30584279999999997	0.30629520000000005

8000	0.3701806999999999	0.36933788	0.3755869800000003
9000	0.39881216	0.3968695999999999	0.4054190799999999
10000	0.4505964600000002	0.45749364	0.453578
20000	1.124606	1.1198513999999995	1.1084823999999998
30000	1.8236082	1.8040119999999995	1.8403962
40000	2.2719872000000003	2.2335866	2.248113
50000	2.7697327999999994	2.750054	2.7512090000000007
60000	3.4801808000000007	3.5134602	3.5083442000000002
70000	3.867741	3.879113199999998	3.874434
80000	4.4409852	4.433402600000001	4.390199200000001
90000	4.961898000000001	5.087346599999999	5.034146
100000	5.7376532000000005	5.6715044	5.770450800000001

Tabela 16 – Média de tempo do Radix Sort (LSD)

3.2 Resultados gerais

4 Discussão