Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Biológicas
Departamento de Computação

ALGORITMOS E ESTRUTURAS DE DADOS
Métodos de ordenação Interna.

Antonio Carlos de Nazaré Júnior
Professor - David Menotti Gomes

Ouro Preto
14 de novembro de 2008

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Programas e Arquivos

Capítulo 1

Introdução

Em vários momentos do dia a dia, o homem depara-se com a necessidade de consultar dados ordenados. Como exemplo, pode-se citar uma lista telefônica. Imagine como seria consultar o telefone de uma pessoa se os nomes não estivessem classificados em ordem alfabética. Por isso uma das atividades mais utilizada na computação é a ordenação.

As ordens mais utilizadas são as númericas e as lexicográficas.

Existem diversos algoritmos para ordenação interna. No presente trabalho será apresentada a implementação e os testes de sete destes métodos.

- BubbleSort
- InsertSort
- SelectSort
- ShellSort
- QuickSort
- HeapSort
- MergeSort

Os testes foram realizados com vetores de números inteiros de diferentes tamanhos (*100, 1000, 10000 e 100000*) e tipos (*ordenados em ordem crescente e decrescente, aleatórios e parcialmente ordenados com apenas 10% dos elementos fora da ordem*).

Como medidas para a comparação entre os métodos foi colhido durante cada teste:

1. Número de comparações entre chaves do vetor;
2. Número de movimentações;
3. Contagem do tempo gasto durante a execução do algoritmo;

Capítulo 2

Métodos de Ordenação

> *Ordenar corresponde ao processo de rearranjar um conjunto de objetos em ordem ascendente ou descendente. O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado. ...A atividade de colocar as coisas em ordem está presente na maioria das aplicações em que os objetos armazenados têm de ser pesquisados e recuperados...* [19]

A comparação é feita através de uma determinada chave, para este trabalho a chave escolhida foi um valor inteiro. O Programa 2.1 apresenta a estrutura do tipo Item (implementado na classe `Types`) que é armazenado pelos Vetores.

Programa 2.1: Estrutura do tipo Item.

```
typedef long TKey;

typedef struct TItem{
  TKey Key;
}TItem;
```

Um método é dito estável se a ordem relativa dos itens com a mesma chave não se altera durante o processo de ordenação. A Figura 2.1 exemplifica os métodos estáveis e instáveis de ordenação.

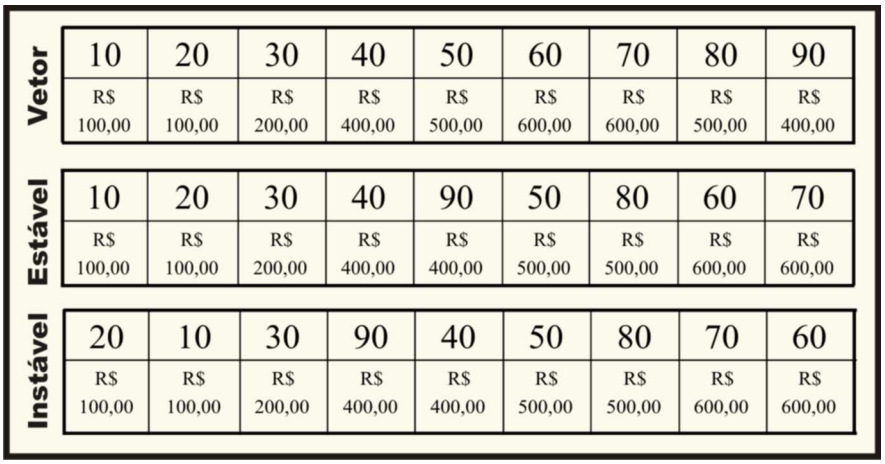

Figura 2.1: Exemplo de ordenação Estável e Instável.

Os métodos de ordenação são classificados em dois grandes grupos: ordenação interna e externa.

1. **Ordenação Interna:** São os métodos que não necessitam de uma memória secundária para o processo, a ordenação é feita na memória principal do computador;

2. **Ordenação Externa:** Quando o arquivo a ser ordenado não cabe na memória principal e, por isso, tem de ser armazenado em fita ou disco.

A principal diferença entre os dois grupos é que no método de ordenação interna qualquer registro pode ser acessado diretamente, enquanto no método externo é necessário fazer o acesso em blocos [5].

2.1 Métodos de Ordenação Interna

Durante a escolha de um algoritmo de ordenação, deve-se observar um aspecto importante, o tempo gasto durante a sua execução. Para algoritmos de ordenação interna, as medidas de complexidade relevantes contam o número de comparações entre chaves e o número de movimentações de itens do arquivo [19].

Uma outra medida que pesa na escolha é a quantidade de memória extra utilizada pelo algoritmo. Metódos que realizam a permutação dos elementos no próprio vetor são chamados ***in situ***, esses métodos são os preferidos. Os métodos que necessitam de memória extra para armazenar outra cópia do itens possuem menor importância.

Os métodos de ordenação interna são classificados em dois subgrupos [4].

- Métodos simples:
    1. BubbleSort
    2. InsertSort
    3. SelectSort
- Métodos eficientes:
    1. ShellSort
    2. QuickSort
    3. HeapSort
    4. MergeSort

2.1.1 Implementação dos métodos

Os métodos foram implementados em uma única classe ***SortMethods*** que é responsável pela ordenação dos vetores e pela medição do tempo, número de comparações e número de movimentações. A classe foi implementada utilizando programação orientada a objetos, pela necessidade do encapsulamento das variaveis que controlam as medidas de tempo, movimento e comparação. Para medição do tempo foi implementada a classe ***CTimer*** que é apresentada no Apêndice A.

O Programa 2.2 apresenta a estrutura da classe ***SortMethods***.

Programa 2.2: Estrutura da classe SortMethods.

```
#ifndef _SORTMETHODS_
#define _SORTMETHODS_

#include "CTimer.h"
#include "Types.h"

class SortMethods{

private:
  double mComparations;
  double mMoviments;
  double mTime;
  void ClearAll();
  void Partition(long Left, long Right, long *i, long *j, TItem *A);
  void ReMake(long Left, long Right, TItem *Array);
  void Build(TItem *Array, long n);
  void MethodQuick(long Left, long Right, TItem *A);
  void Merge(TItem *Array, long Left, long Means, long Right);

public:

  SortMethods();
  void BubbleSort(TItem* Array, long n);
  void SelectSort(TItem* Array, long n);
  void InsertSort(TItem* Array, long n);
  void ShellSort(TItem* Array, long n);
  void QuickSort(TItem* Array, long n);
  void HeapSort(TItem* Array, long n);
  void MergeSort(TItem* Array, long n);
  double getTime();
  double getComparations();
  double getMoviments();
};
#endif
```

As variáveis `mComparations`, `mMoviments` e `mTime`, tem como finalidade armazenar o número de comparações, movimentos e tempo gasto, respectivamente, em cada execução de um método de ordenação.

O procedimento `ClearAll()`, apresentada no Programa 2.3 é chamado a cada vez que um método é iniciado. Esta função zera os valores das variáveis de medição.

Programa 2.3: Método para zerar valores.

```
void SortMethods::ClearAll(){
  this->mComparations = 0;
  this->mMoviments = 0;
  this->mTime = 0;
}
```

Os demais métodos ***Privates*** serão apresentados juntamente com os métodos de ordenação que utilizam-os.

A consulta das váriaveis de medidas da classe ***SortMethods*** é feita através das seguintes funções que são mostrada no Programa 2.4:

- `double getTime()`: Retorna o tempo em segundos gasto pelo algoritmo de ordenação;

- `double getComparations()`: Retorna o número de comparações entre chaves;
- `double getMoviments()`: Retorna o número de movimentações (trocas) realizadas entre os itens;

Programa 2.4: Métodos de consulta às variáveis de medida.

```
double SortMethods::getTime(){
  return this->mTime;
}

double SortMethods::getComparations(){
  return this->mComparations;
}

double SortMethods::getMoviments(){
  return this->mMoviments;
}
```

Para os métodos de ordenação a entrada será a estrutura a ser ordenada (vetor) e o número de itens contido na estrutura.

O calculo da função de complexidade deste algoritmo será definido em razão a duas grandezas: O número de movimentações $[M(n)]$ e o número de comparações entre chaves $[C(n)]$.

2.2 BubbleSort

É o método mais simples em termos de implementação, porém é o menos eficiente. A idéia principal do algoritmo é percorrer o vetor $n-1$ vezes, a cada passagem fazendo flutuar para o inicio o menor elemento da sequência. Essa movimentação, ilustrada na Figura 2.2, lembra a forma como as bolhas procuram seu próprio nível, por isso o nome do algoritmo. Seu uso não é recomendado para vetores com muitos elementos [7].

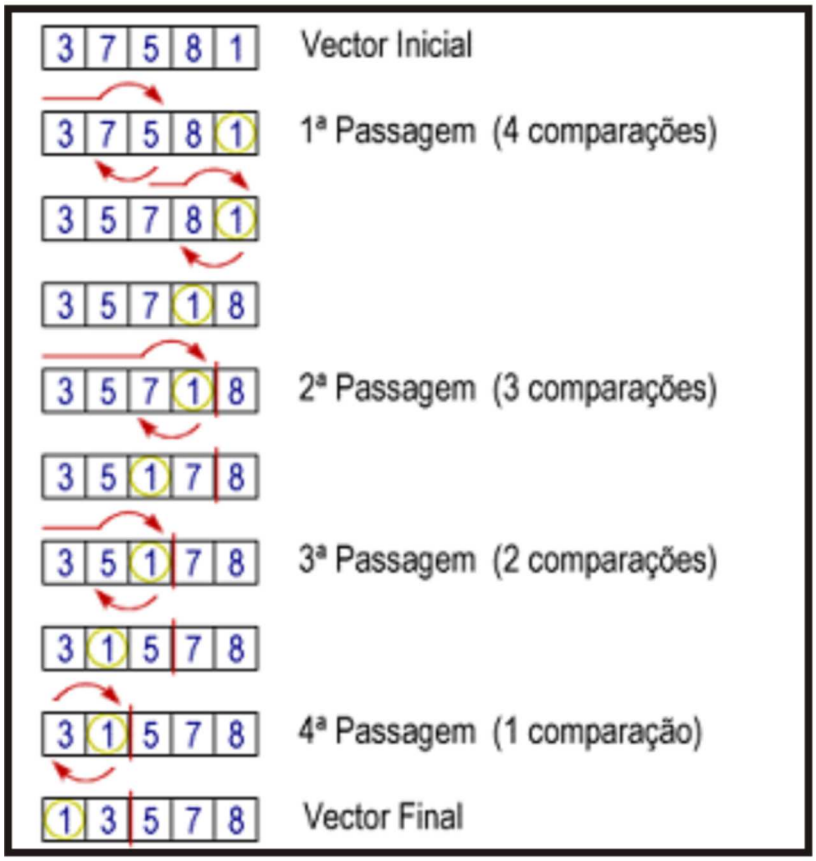

Figura 2.2: Ilustração do funcionamento do algoritmo BubbleSort.

A seguir na Figura 2.3 é mostrado o fluxograma do algoritmo.

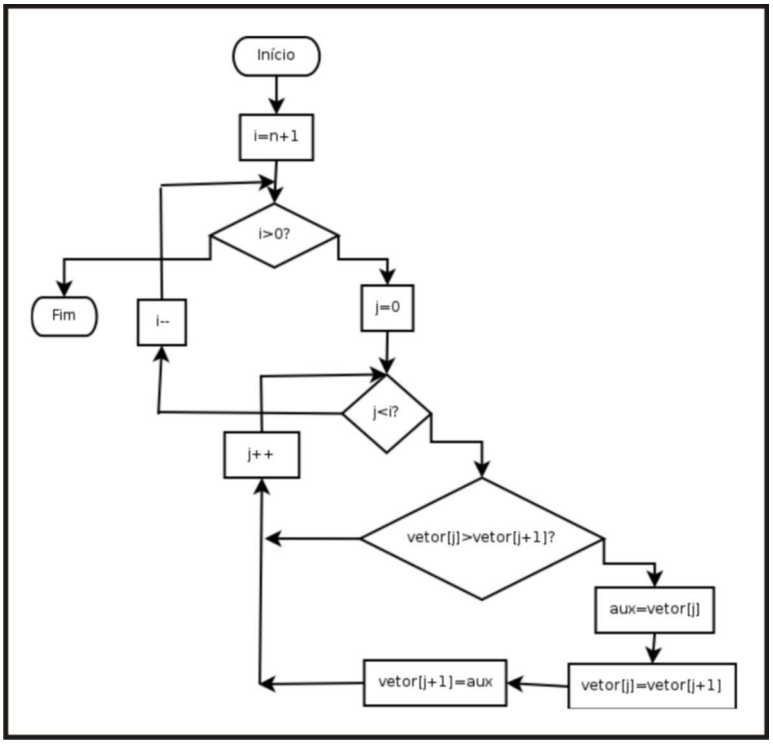

Figura 2.3: Fluxograma do algoritmo BubbleSort.

2.2.1 Implementação

Programa 2.5: Método BubbleSort.

```
void SortMethods::BubbleSort(TItem *Array, long n){
  long i, j;
  TItem aux;
  CTimer *Timer = new CTimer();
  this->ClearAll();
  Timer->start();
  for( i = 0 ; i < n-1 ; i++ ){
    for( j = 1 ; j < n-i ; j++ ){
      this->mComparations++;
      if ( Array[j].Key < Array[j-1].Key){
        aux = Array[j];
        this->mMoviments++;
        Array[j] = Array[j-1];
        this->mMoviments++;
        Array[j-1] = aux;
        this->mMoviments++;
      }
    }
  }
  Timer->stop();
  this->mTime = Timer->getElapsedTime();
}
```

O Programa 2.5 mostra a implementação do algoritmo para um conjunto de n itens, implementado como um vetor do tipo **TItem**.

O algoritmo procede da seguinte forma:

1. Zera os valores das váriáveis de medição através do método `ClearAll()`;
2. Inicia a contagem de tempo com a função `start()`;
3. Percorre o vetor, trazendo para o início o menor elemento encontrado;
4. A cada comparação incrementa a variável `mComparations` e a cada movimentação incrementa a variável `mMoviments`;
5. Pausa a contagem de tempo e calcula o tempo gasto armazenando o valor na variável `mTime`;

Uma maneira mais eficiente de implementação do ***BubbleSort*** consiste em parar o processo logo que se detecte que, ao longo de uma passagem não foram efetuadas trocas de chaves [16].

2.2.2 Estudo da Complexidade

A Equação 2.1 demonstra o calculo da função de complexidade em relação ao número de comparações. Ela é a mesma para o melhor, pior e caso médio.

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2}\left(\sum_{j=1}^{n-i-1} 1\right) \\ &= \sum_{i=0}^{n-2}(n-i-1) \\ &= \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 \\ &= n(n-1) - \frac{(0+n-2)(n-1)}{2} - (n-1) \\ &= n^2 - n - \frac{(n^2-3n+2)}{2} - (n-1) \\ &= \frac{2n^2 - 2n - n^2 + 3n - 2 - 2n + 2}{2} \\ &= \frac{n^2-n}{2} \end{aligned} \tag{2.1}$$

Ordem de Complexidade: $O(n^2)$

Para a função de complexidade em relação ao número de movimentos (Equação 2.2 basta multiplicar a função $C(n)$ por 3 que é o número de movimentações a cada iteração do algoritmo. Ela também será a mesma para os 3 casos (médio, pior e melhor).

$$\begin{aligned} M(n) &= 3 \times C(n) \\ &= 3 \times \left(\frac{n^2 - n}{2} \right) \\ &= \frac{3n^2 - 3n}{2} \end{aligned} \tag{2.2}$$

Ordem de Complexidade: $O(n^2)$

2.2.3 Análise do algoritmo

O ***BubbleSort*** é um método de simples implementação, porém a sua eficiência é a menor entre os métodos de ordenação interna [17]. Admite contudo vários melhoramentos e é também uma boa base para a construção de métodos mais elaborados [16].

A Tabela 2.1 apresenta as principais vantagens e desvantagens deste método.

Vantagens	**Desvantagens**
- Fácil Implementação;	- O fato de o arquivo já estar ordenado não ajuda em nada [7];
- Algoritmo Estável;	- Ordem de complexidade quadrática;

Tabela 2.1: Vantagens e desvantagens do Método BubbleSort.

2.3 InsertSort

InsertSort é um algoritmo elementar de ordenação [13]. É eficiente quando aplicado à um vetor com poucos elementos. Em cada passo, a partir de $i = 2$, o i-ésimo item da sequência fonte é apanhado e transferido para a sequência destino, sendo inserido no seu lugar apropriado [19]. O algoritmo assemelha-se com a maneira que os jogadores de cartas ordenam as cartas na mão em um jogo, como o pôquer, por exemplo. As Figuras 2.4 e 2.5 apresentam o seu funcionamento e o fluxograma ilustrativo, respectivamente.

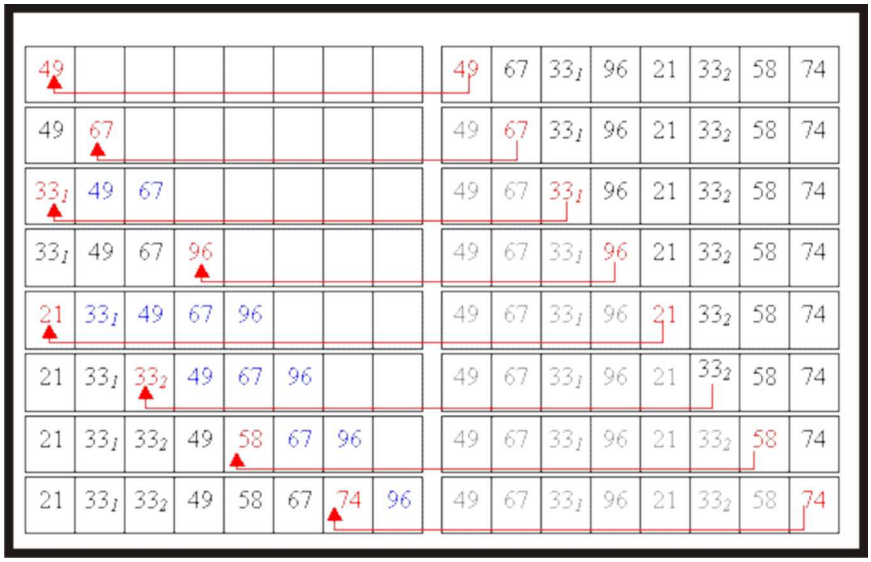

Figura 2.4: Ilustração do funcionamento do algoritmo InsertSort.

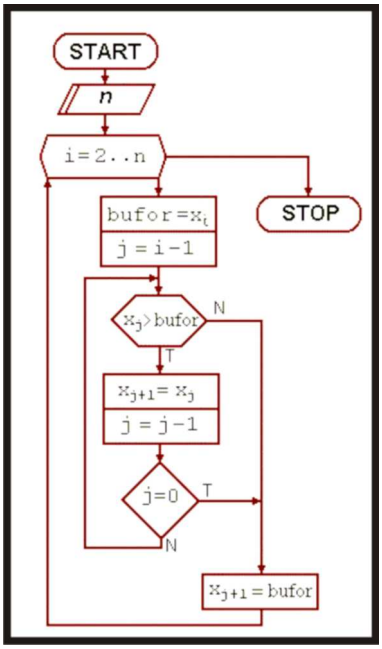

Figura 2.5: Fluxograma do algoritmo InsertSort.

2.3.1 Implementação

A colocação do item no seu lugar correto na sequência ordenanda, como demonstra o Programa 2.6, é realizada movendo os índices de maiores valores para direita e então inserindo o item na posição vazia.

Programa 2.6: Método InsertSort.

```
void SortMethods::InsertSort(TItem *Array, long n){
  long i,j;
  TItem aux;
    CTimer *Timer = new CTimer();
  this->ClearAll();
  Timer->start();
  for (i = 1; i < n; i++){
    aux = Array[i];
    this->mMoviments++;
    j = i - 1;
    this->mComparations++;
    while ( ( j >= 0 ) && ( aux.Key < Array[j].Key ) ){
      Array[j + 1] = Array[j];
      this->mMoviments++;
      j--;
    }
    Array[j + 1] = aux;
    this->mMoviments++;
  }
  Timer->stop();
  this->mTime = Timer->getElapsedTime();
}
```

O algoritmo procede da seguinte maneira:

1. Zera os valores das váriáveis de medição através do método `ClearAll()`;
2. Inicia a contagem de tempo com a função `start()`;
3. O primeiro laço de repetição tem a função de controlar o tamanho da sequencia analisada;
4. O segundo laço é responsável de colocar o novo elemento da sequencia, em relação à anterior, no lugar apropriado;
5. A cada comparação incrementa a variável `mComparations` e a cada movimentação incrementa a variável `mMoviments`;
6. Pausa a contagem de tempo e calcula o tempo gasto armazenando o valor na variável `mTime`;

Uma solução melhor, mas que não será utilizada durante os testes, é a utilização de um registro sentinela: na posição zero do vetor coloca-se o próprio registro em consideração. Assim evitando duas comparações no anel mais interno do algoritmo, porem seria necesário uma implementação do vetor a partir do índice 1, e não de 0 como proposto neste trabalho.

2.3.2 Estudo da Complexidade

A Equação 2.3 demonstra o calculo da função de complexidade em relação ao número de comparações.

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1}\left(\sum_{j=1}^{i} 1\right) \\ &= \sum_{i=1}^{n-1}(i) \\ &= \frac{(n-1)(1+n-1}{2} \\ &= \frac{(n-1)(n)}{2} \\ &= \frac{n^2-n}{2} \end{aligned} \tag{2.3}$$

Ordem de Complexidade: $O(n^2)$

A seguir na Equação 2.4 é apresentada a complexidade em função do número de movimentações;

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1}\left(2+\sum_{j=1}^{i} 1\right) \\ &= \sum_{i=1}^{n-1}(2+i) \\ &= \sum_{i=1}^{n-1} 2 + \sum_{i=1}^{n-1} i \\ &= 2(n-1) + \frac{(n-1)(1+n-1}{2} \\ &= (2n-2) + \frac{(n-1)(n)}{2} \\ &= (2n-2) + \frac{(n^2-n)}{2} \\ &= \frac{4n-4+n^2-n}{2} \\ &= \frac{n^2+3n-4}{2} \\ &= \frac{n^2+3n}{2} - 2 \end{aligned} \tag{2.4}$$

Ordem de Complexidade: $O(n^2)$

2.3.3 Análise do algoritmo

O ***InsertSort*** também é um método de simples implementação, e tem a complexidade igual ao ***BubbleSort***. Pode ser aprimorado com o uso de sentinela e outras técnicas de algoritmos. É o melhor método para se utilizar quando os arquivos já estão quase ordenados [7].

A Tabela 2.2 apresenta as principais vantagens e desvantagens deste método.

Vantagens	**Desvantagens**
- Fácil Implementação	- Número grande de movimentações
- Algoritmo Estável	- Ordem de complexidade quadrática
- O vetor já ordenado favorece a ordenação	- Ineficiente quando o vetor está ordenado inversamente;

Tabela 2.2: Vantagens e desvantagens do Método InsertSort.

2.4 SelectSort

Tem como principio de funcionamento selecionar o menor item do vetor e a seguir trocá-lo pela primeira posição do vetor. Isto ocorre para os $n-1$ elementos restantes, depois com os $n-2$ itens, até que reste apenas um elemento [19]. A principal diferença deste métodos em relação aos dois já apresentados é que ele realiza apenas uma troca por iteração. A Figura 2.6 apresenta o seu funcionamento.

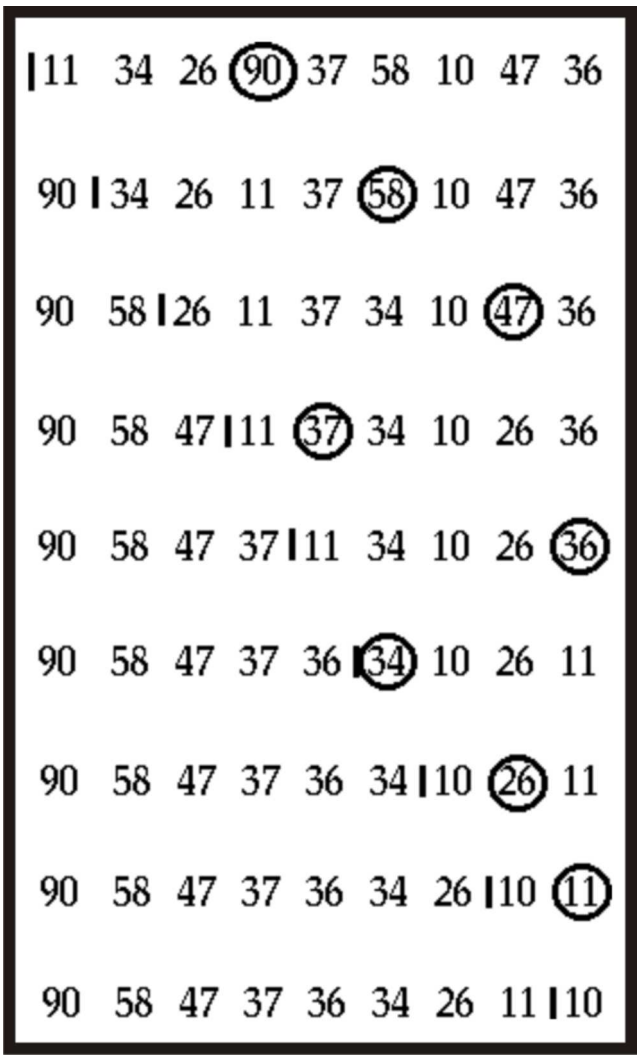

Figura 2.6: Ilustração do funcionamento do algoritmo SelectSort.

2.4.1 Implementação

A colocação do item no seu lugar correto na sequência ordenanda, como demonstra o Programa 2.7 é realizada trocando o item de menor valor pela primeira posição do vetor.

Programa 2.7: Método SelectSort.

```
void SortMethods::SelectSort(TItem *Array, long n){
  long i, j, Min;
  TItem aux;
  CTimer *Timer = new CTimer();
  this->ClearAll();
  Timer->start();
  for (i = 0; i < n - 1; i++){
    Min = i;
    for (j = i + 1 ; j < n; j++){
      this->mComparations++;
      if ( Array[j].Key < Array[Min].Key)
        Min = j;
    }
    aux = Array[Min];
    this->mMoviments++;
    Array[Min] = Array[i];
    this->mMoviments++;
    Array[i] = aux;
    this->mMoviments++;
  }
  Timer->stop();
  this->mTime = Timer->getElapsedTime();
}
```

O algoritmo procede da seguinte forma:

1. Zera os valores das váriáveis de medição através do método `ClearAll()`;
2. Inicia a contagem de tempo com a função `start()`;
3. O primeiro laço determina a dimensão de busca do menor elemento;
4. O segundo laço é responsável por realizar a busca pelo menor elemento;
5. Feito a busca é realizada a troca do menor elemento pelo primeiro elemento;
6. Após a troca o processo é realizado novamente para os $n - i$ itens restantes;
7. A cada comparação incrementa a variável `mComparations` e a cada movimentação incrementa a variável `mMoviments`;
8. Pausa a contagem de tempo e calcula o tempo gasto armazenando o valor na variável `mTime`;

2.4.2 Estudo da Complexidade

A Equação 2.5 mostra a equação da função de complexidade em relação ao número de comparações.

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2}\left(\sum_{j=1}^{n-i-1} 1\right) \\ &= \sum_{i=0}^{n-2}(n-i-1) \\ &= \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 \\ &= n(n-1) - \frac{(0+n-2)(n-1)}{2} - (n-1) \\ &= n^2 - n - \frac{(n^2-3n+2)}{2} - (n-1) \\ &= \frac{2n^2-2n-n^2+3n-2-2n+2}{2} \\ &= \frac{n^2-n}{2} \end{aligned} \tag{2.5}$$

Ordem de Complexidade: $O(n^2)$

A seguir a função de complexidade do número de movimentações é apresentado pela Equação 2.6.

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} 3 \\ &= 3(n-1) \\ &= 3n-3 \end{aligned} \tag{2.6}$$

Ordem de Complexidade: $O(n)$

2.4.3 Análise do algoritmo

O ***SelectSort*** é um método muito simples. Além disso, o algoritmo de seleção apesenta um comportamento espetacular quanto ao número de movimentos de registros, cujo tempo de execução é linear, esta particularidade é dificilmente encontrada em outros algoritmos de ordenação [19]. É o algoritmo ideal para arquivos com registros muito grandes [1].

A Tabela 2.3 apresenta as principais vantagens e desvantagens deste método.

Vantagens	**Desvantagens**
- Fácil Implementação	- O fato de o arquivo já estar ordenado não influencia em nada
- Pequeno número de movimentações	- Ordem de complexidade quadrática
- Interessante para arquivos pequenos	- Algoritmo não estável

Tabela 2.3: Vantagens e desvantagens do Método SelectSort.

2.5 ShellSort

Este algoritmo é uma extensão do método ***InsertShort*** proposto por Donald Shell em 1959. O algoritmo de inserção troca itens adjacentes quando está procurando o ponto de inserção na sequência destino. Se o menor item estiver na posição mais à direita no vetor, então o número de comparações e movimentações é igual a $n - 1$ para encontra o seu ponto de inserção [11]. O ***ShellSort*** contorna este problema, pemitindo trocas de registros distantes um do outro [19]. De maneira geral ele passa várias vezes no vetor dividindo-o em vetores menores, e nestes são aplicados o algoritmo de ordenação por inserção tradicional [2].

A Figura 2.7 ilustra o seu funcionamento.

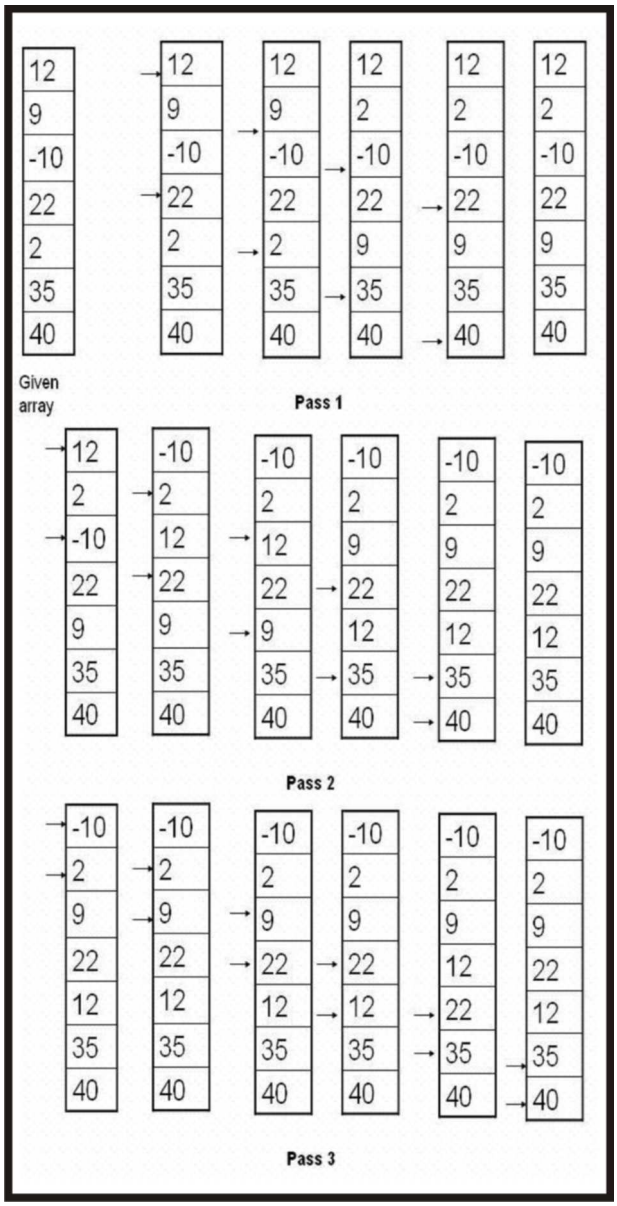

Figura 2.7: Ilustração do funcionamento do algoritmo ShellSort.

Dentre os programas de ordenação interna que tem ordem de complexidade quadrática,o ***ShellSort*** é o mais eficiente.

2.5.1 Implementação

O diferencial do ***ShellSort*** é a divisão em h intervalos que ele realiza para posteriomente aplicar o método de inserção. Várias sequências para h têm sido experimentadas. Knuth(1973) mostrou experimentalmente que a escolha do incremento para h, mostrada a seguir na Equação 2.7 é dificil de ser batida em mais de 20% dos casos em eficiência no tempo de execução [19].

$$\begin{aligned} h(s > 1) &= 3h(s-1)+1 \\ h(s = 1) &= 1 \end{aligned} \tag{2.7}$$

A implementação utilizando este calculo para h é apresentada no Programa 2.8.

Programa 2.8: Método SelectSort.

```
void SortMethods::ShellSort(TItem *Array, long n){
  int i, j;
  int h = 1;
  TItem aux;
  CTimer *Timer = new CTimer();
  this->ClearAll();
  Timer->start();
  do{
    h = h * 3 + 1;
  }while (h < n);
  do{
    h /= 3;
    for( i = h ; i < n ; i++ ){
      aux = Array[i];
      this->mMoviments++;
      j = i;
      this->mComparations++;
      while (Array[j - h].Key > aux.Key){
        this->mComparations++;
        Array[j] = Array[j - h];
        this->mMoviments++;
        j -= h;
        if (j < h)
          break;
      }
    Array[j] = aux;
    this->mMoviments++;
    }
  } while (h != 1);
  Timer->stop();
  this->mTime = Timer->getElapsedTime();
}
```

O algoritmo procede da seguinte forma:

1. Zera os valores das váriáveis de medição através do método `ClearAll()`;
2. Inicia a contagem de tempo com a função `start()`;

3. O algoritmo usa uma variável auxiliar denominada distância de comparação (h);
4. O valor de h é inicializado com um valor próximo de $n/2$;
5. A troca é feita entre elementos que estão distanciados h posições (se estiverem fora de ordem);
6. Após todas as trocas de elementos cuja distancia é h, o valor de h deve ser reduzido: conforme Equação 2.7;
7. O algoritmo é repetido até que a distância de comparação h seja igual a um ($h = 1$);
8. Para $h = 1$ (ultima passada) é executado o algoritmo de inserção;
9. A cada comparação incrementa a variável `mComparations` e a cada movimentação incrementa a variável `mMoviments`;
10. Pausa a contagem de tempo e calcula o tempo gasto armazenando o valor na variável `mTime`;

2.5.2 Estudo da Complexidade

O estudo da complexidade deste algoritmo contém alguns problemas matemáticos muito difíceis, a começar pela própria sequencia de incrementos para h.

De acordo com testes empíricos utilizando a Formula 2.7 para o calculo dos incrementos, a ordem de complexidade do ***ShellSort*** é de aproximadamente $O(n^1, 25)$ ou $O(n(\log n)^2)$ [**?**].

2.5.3 Análise do algoritmo

O ***ShellSort*** é uma ótima opção para arquivos de tamanho moderado, mesmo porque sua implementação é simples e requer uma quantidade de código pequena [19, 12].

A Tabela 2.4 apresenta as principais vantagens e desvantagens deste método.

Vantagens	**Desvantagens**
- Código Simples	- Algoritmo não estável
- Interessante para arquivos de tamanho moderado	- Tempo de execução sensível à ordem inicial do arquivo [11]

Tabela 2.4: Vantagens e desvantagens do Método ShellSort.

2.6 QuickSort

É o algoritmo mais rápido que se conhece entre os de ordenação interna para uma ampla variedade de situações. Foi escrito em 1960 e publicado em 1962 por C. A. R. Hoare após vários refinamentos [19]. Porém em raras instâncias especiais ele é lento [6]. Adotando a estratégia *Dividir para conquistar* o funcionamento resume-se a dividir o problema de ordenar um vetor de n posições em dois outros menores [2].

A Figura 2.8 ilustra o seu funcionamento.

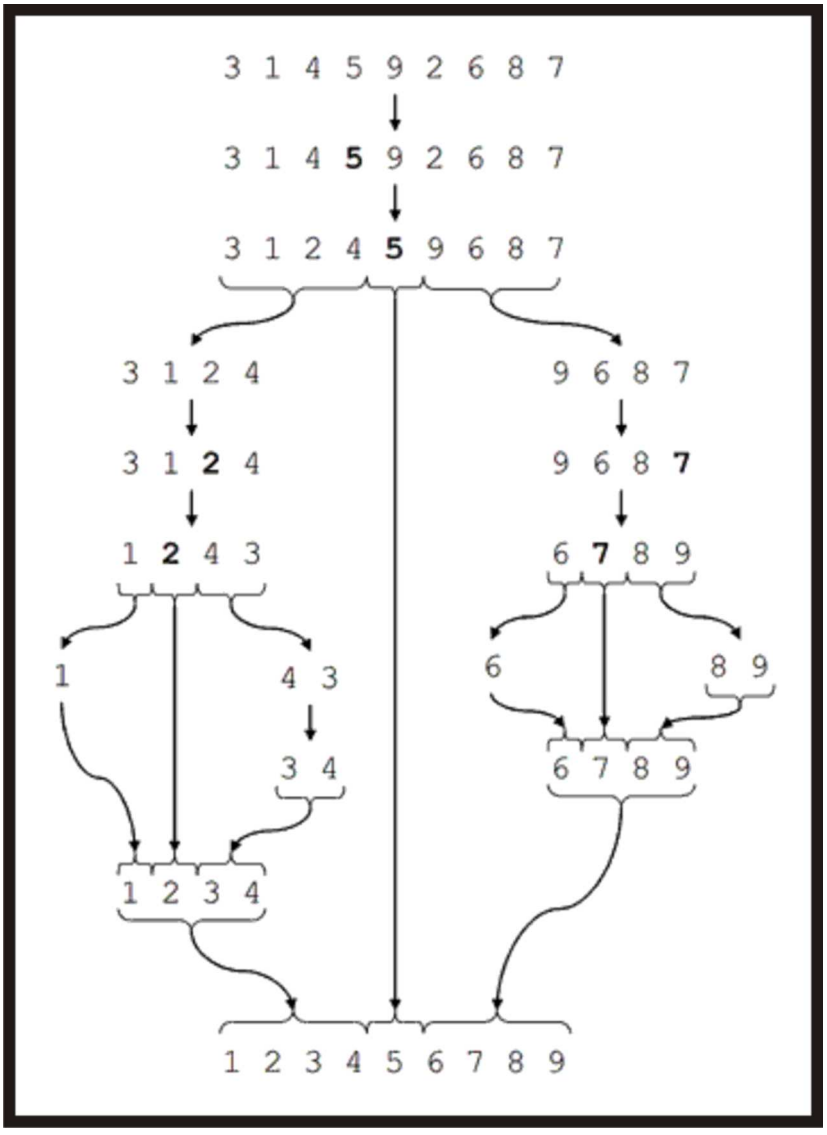

Figura 2.8: Ilustração do funcionamento do algoritmo QuickSort.

O QuickSort é provavelmente o mais utilizado, porém a sua implementação demanda um pouco mais de paciência e cautela.

2.6.1 Implementação

A parte crucial do algoritmo é o método ***Partition*** quem tem a função de rearranjar o vetor por meio da escolha de um **pivô**, de tal forma que ao final o vetor

estará particionado em uma parte esquerda com chaves menores ou iguais ao pivô e outra maiores ou iguais [19].

A implementação do método `Partition` é apresentado no Programa 2.9, para a escolha do pivô foi implementada a variação *mediana de três*, em que o pivo é escolhido usando a mediana entre a chave mais à esquerda, a chave mais à direita e a chave central [18].

Programa 2.9: Método Partition utilizado pelo QuickSort.

```
void SortMethods::Partition(long Left, long Right, long *i, long *j,
    TItem *Array){
  TItem aux1, aux2;
  *i = Left;
  *j = Right;
  aux1 = Array[(*i + *j)/2];
  do{
    this->mComparations++;
    while (aux1.Key > Array[*i].Key){
      this->mComparations++;
      (*i)++;
    }
    this->mComparations++;
    while (aux1.Key < Array[*j].Key){
      this->mComparations++;
      (*j)--;
    }
    if (*i <= *j){
      aux2 = Array[*i];
      this->mMoviments++;
      Array[*i] = Array[*j];
      this->mMoviments++;
      Array[*j] = aux2;
      this->mMoviments++;
      (*i)++;
      (*j)--;
    }
  }while (*i <= *j);
}
```

O Programa 2.11 apresenta o método `MethodQuick` que é o QuickSort propriamente dito.

Programa 2.10: Método MethodQuick.

```
void SortMethods::MethodQuick(long Left, long Right, TItem *Array){
  long i,j;
  Partition(Left, Right, &i, &j, Array);
  if (Left < j)
    MethodQuick(Left, j, Array);
  if (i < Right)
    MethodQuick(i, Right, Array);
}
```

O algoritmo procede da seguinte forma [2]:

1. Zera os valores das váriáveis de medição através do método `ClearAll()`;
2. Inicia a contagem de tempo com a função `start()`;

3. Escolha um elemento `aux1` do vetor `Array` para ser o pivô;

4. Usando `aux1` divida o vetor `Array` em duas partes, da seguinte maneira:

   - Começando com i = posição inicial, percorra o vetor em forma crescente até que um elemento `Array[i] = aux1` seja encontrado.
   - Começando com j = posição final, percorra o vetor em forma descendente até que um elemento A[j] = X seja encontrado;
   - Se `i <= j`, troque `Array[i]` com `Array[j]`;
   - Continue percorrendo o vetor procurando elementos Array[i] e Array[j] até que os contadores i e j se cruzem (j < i).

5. Repita, recursivamente, os passos 3 e 4 para as duas partes

6. A cada comparação incrementa a variável `mComparations` e a cada movimentação incrementa a variável `mMoviments`;

7. Pausa a contagem de tempo e calcula o tempo gasto armazenando o valor na variável `mTime`;

O ***QuickSort*** não foi implementado diretamente, por ser um método recursivo, as variáveis contadoras de movimentos, comparações e tempo seriam reiniciadas a toda chamada recursiva da função.

Portanto o método `QuickSort` foi implementado da maneira como o Programa 2.11 apresenta.

Programa 2.11: Método QuickSort.

```
void SortMethods::QuickSort(TItem *Array, long n){
  CTimer *Timer = new CTimer();
  this->ClearAll();
  Timer->start();
  this->MethodQuick(0,n-1,Array);
  Timer->stop();
  this->mTime = Timer->getElapsedTime();
}
```

2.6.2 Estudo da Complexidade

Assim como ***ShellSort*** a análise da complexidade deste algoritmo é dificil devido a calculos matemáticos complexos. O pior caso deste algoritmo é quando o arquivo já está ordenado e a escolha do pivô é inadequada. Nesse caso as partições serão extremamente desiguais, e o método `MethodQuick` será chamado n vezes, eliminando apenas um item em cada chamada. Essa situação é desastrosa, pois o número de comparações passa a cerca de $\frac{n^2}{2}$, e o tamanho da pilha necessária para as chamadas recursivas é cerca de n. Entretanto o pior caso pode ser evitado empregando-se modificações na escolha do pivô [10, 19, 18]. A melhor situação possível ocorre quando cada partição divide o arquivo em duas partes iguais. Logo, a Equação 2.8 demonstra o calculo da função de complexidade em relação ao número de comparações.

$$C(n) \quad = \quad n \log n - n + 1 \tag{2.8}$$

No caso médio o número de compações é apresentado na Equação 2.9, significando assim que a sua ordem de complexidade é $O(n \log n)$.

$$C(n) \quad = \quad 1,386n \log n - 0,84n \tag{2.9}$$

2.6.3 Análise do algoritmo

O ***QuickSort*** é considerado o método mais eficiente e é altamente recomendável para arquivos grandes. Quanto mais o vetor estiver desordenado, maior será sua vantagem em relação aos outros métodos. A escolha correta do pivô é essencial para a garantia de eficiência do algoritmo.

A Tabela 2.5 apresenta as principais vantagens e desvantagens deste método.

Vantagens	**Desvantagens**
- Extremamente Eficiente	- Tem um pior caso de $O(n^2)$
- Necessita apenas de um pequena pilha como memória extra	- Implementação difícil
- Complexidade $n \log n$	- Não é estável

Tabela 2.5: Vantagens e desvantagens do Método QuickSort.

2.7 HeapSort

Utilizando o mesmo princípio do ***SelectSort***, o ***HeapSort*** é um algoritmo que utiliza uma estrutura de dados conhecida como ***Heap*** binário para manter o próximo item a ser selecionado [14]. Criado em 1964 por Robert W. Floyd e J.W.J. Williams, ele é um algoritmo de Ordem de Complexidade $O(n \log n)$.

Existem dois tipos de heaps: Os heaps de máximo (max heap), em que o valor de todos os nós são menores que os de seus respectivos pais; e os heaps de mínimo (min heap), em que o valor de todos os nós são maiores que os de seus respectivos pais. Assim, em um heap de máximo, o maior valor do conjunto está na raíz da árvore, enquanto no heap de mínimo a raíz armazena o menor valor existente [3]. Os heaps podem ser representados por arranjos, nesse caso a maior(menor) chave está sempre na posição 1 do vetor. A Figura 2.9 ilustra este tipo.

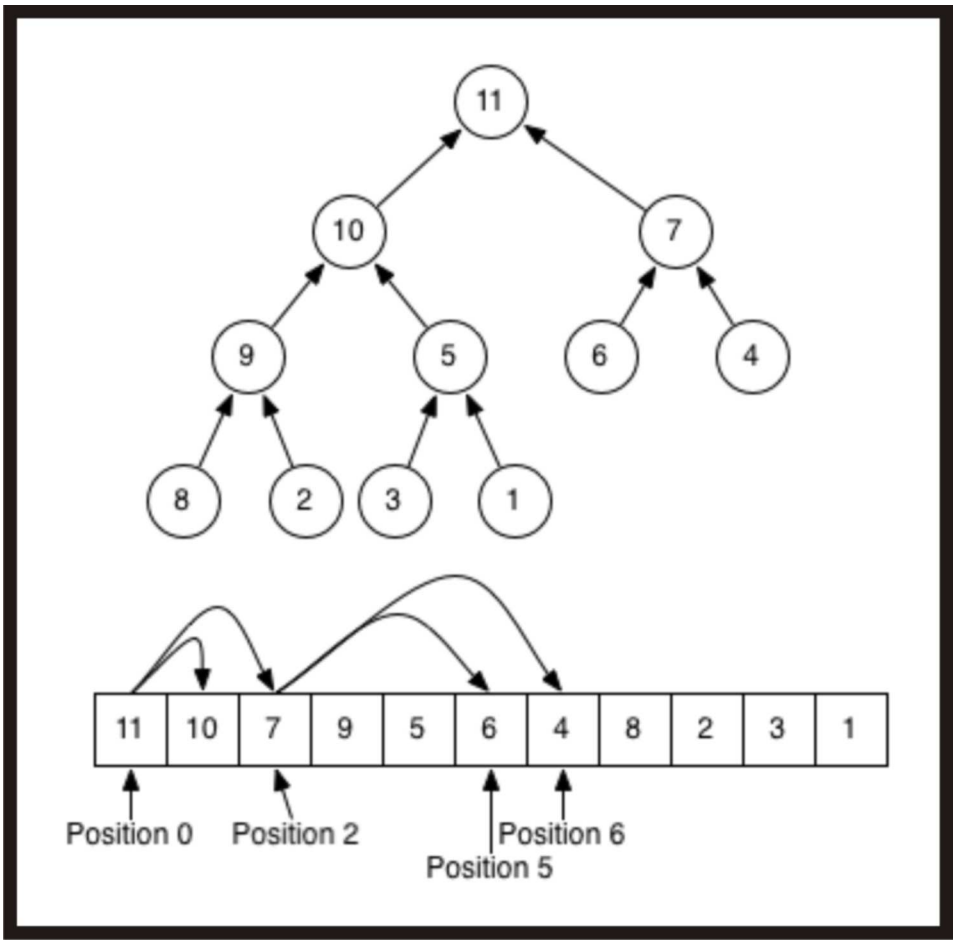

Figura 2.9: Ilustração de um Heap máximo.

Os algoritmos para implementar as operações sobre o ***heap*** operam ao longo de um dos caminhos da árvore, a partir da raiz até o nível mais profundo da árvore. Para o ***HeapSort*** são necesária apenas duas operações com o ***heap***: Remake, que refaz a estrutura e Build que a constroi. Eles são apresentados no Programa 2.12.

Programa 2.12: Operações com Heap necessárias ao HeapSort.

```
void SortMethods::ReMake(long Left, long Right, TItem *Array){
  long i = Left;
  long j;
  TItem aux;
  j = i * 2;
  aux = Array[i];
  while (j <= Right){
    if (j < Right){
      this->mComparations++;
      if (Array[j].Key < Array[j+1].Key)
        j++;
    }
    this->mComparations++;
    if (aux.Key >= Array[j].Key)
      break;
    Array[i] = Array[j];
    i = j;
    j = i * 2 ;
    this->mMoviments++;
  }
  Array[i] = aux;
  this->mMoviments++;
}
void SortMethods::Build(TItem *Array, long n){
  long Left;
  Left = n / 2 + 1;
  while (Left > 1)
  {
    Left--;
    ReMake(Left, n, Array);
  }
}
```

O método de Ordenação ***HeapSort*** é iniciado com um ***heap*** obtido através do método `Build`.

Programa 2.13: Método HeapSort.

```
void SortMethods::HeapSort(TItem *Array, long n){
  long Left, Right;
  TItem aux;
  CTimer *Timer = new CTimer();
  this->ClearAll();
  Timer->start();
  this->Build(Array, n);
  Left = 0;
  Right = n-1;
  while(Right > 0){
    aux = Array[0];
    Array[0] = Array[Right];
    Array[Right] = aux;
    this->mMoviments++;
    Right--;
    this->ReMake(Left, Right, Array);}
  Timer->stop();
  this->mTime = Timer->getElapsedTime();}
```

Depois pega-se o item na posição 0 do vetor(raiz do ***heap***) e troca-se com o item que está na posição n do vetor, como apresentado no Programa 2.13. A seguir, basta usar o método `ReMake` com o restante dos itens. A Figura 2.10 ilustra o funcionamento deste método.

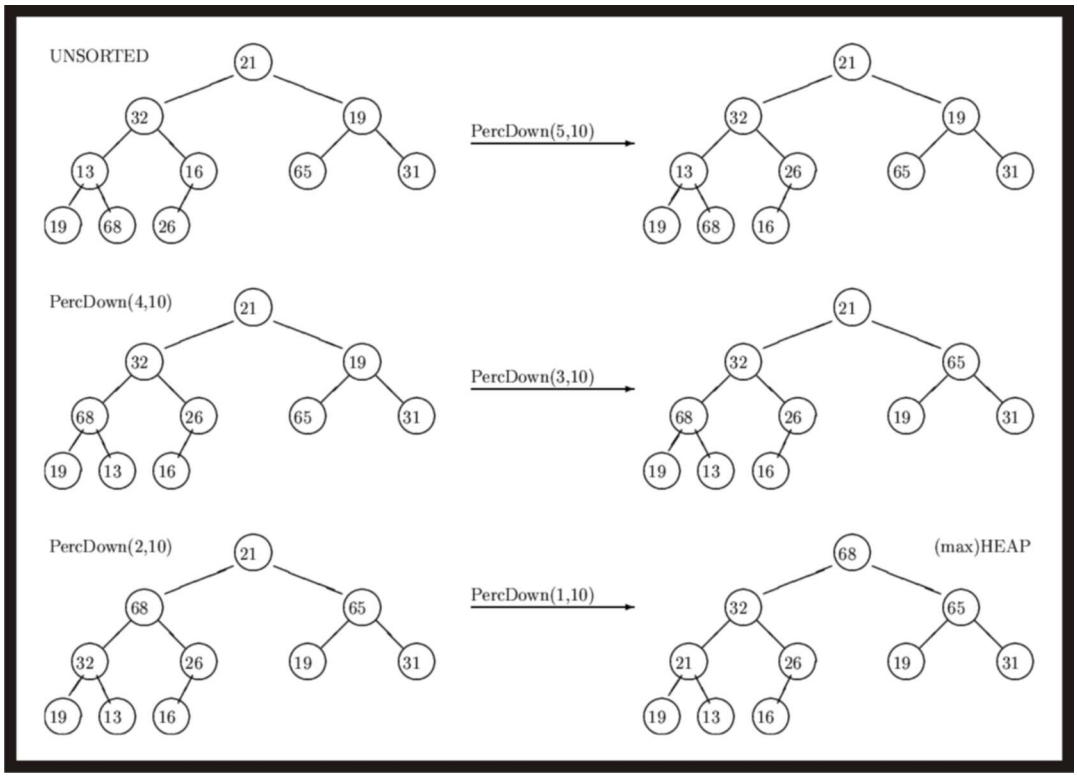

Figura 2.10: Ilustração do funcionamento do algoritmo HeapSort.

2.7.1 Implementação

Como foi apresentado o método ***HeapSort*** baseia-se em uma estrutura de dados. A implementação dos algoritmos das operações necessárias ao ***heap*** e o método propriamente dito, foram apresentadas pelos Programas 2.12 e 2.13

O algoritmo procede da seguinte forma [8]:

1. Zera os valores das váriáveis de medição através do método `ClearAll()`;
2. Inicia a contagem de tempo com a função `start()`;
3. Constroi o heap através do método `Build`;
4. Troca o item na posição 1 do vetor (raiz do heap) com o item da posição n.
5. Usa o procedimento ReMake para reconstituir o heap para os itens `Array[1]`, `Array[2]`, até `Array[`$n-1$`]`.
6. Repita os passos 4 e 5 com os $n-1$ itens restantes, depois com os $n-2$, até que reste apenas um item.

7. A cada comparação incrementa a variável `mComparations` e a cada movimentação incrementa a variável `mMoviments`;

8. Pausa a contagem de tempo e calcula o tempo gasto armazenando o valor na variável `mTime`;

2.7.2 Estudo da Complexidade

A análise de complexidade deste algoritmo também é complexa como os dois métodos apresentados anteriormente. Porém, será apresentado resultados de testes empíricos [8].

- O procedimento ReMake gasta cerca de $\log n$ operações, no pior caso;
- O método Build executa $O(n) \times \log n$(ReMake);
- O laço interno do Programa 2.13 executa $O(n) \times \log n$(ReMake);
- Logo, o ***Heapsort*** gasta um tempo de execução proporcional a $n \log n$, no pior caso.

2.7.3 Análise do algoritmo

À primeira vista, o algoritmo não parece eficiente, pois as chaves são movimentadas várias vezes. Entretanto, ele gasta um tempo de execução proporcional a $n \log n$ no pior caso [19].

Ele não é recomendado para vetores com poucos elementos por causa do tempo gasto na construção do ***heap***, sendo recomendado para aplicações que não podem tolerar eventualmente um caso desfavorável [8].

A Tabela 2.6 apresenta as principais vantagens e desvantagens deste método.

Vantagens	**Desvantagens**
- Tempo $n \log n$	- O anel interno do algoritmo é bastante complexo se comparado ao ***Quicksort***.
	- Não é estável

Tabela 2.6: Vantagens e desvantagens do Método HeapSort.

2.8 MergeSort

É outro algoritmo de ordenação do tipo *dividir para conquistar*. Sua idéia básica é criar uma sequência ordenada a partir de duas outras também ordenadas. Para isso, ele divide a sequência original em pares de dados, ordena-as; depois as agrupa em sequências de quatro elementos, e assim por diante, até ter toda a sequência dividida em apenas duas partes, como ilustrado na Figura 2.14 [15].

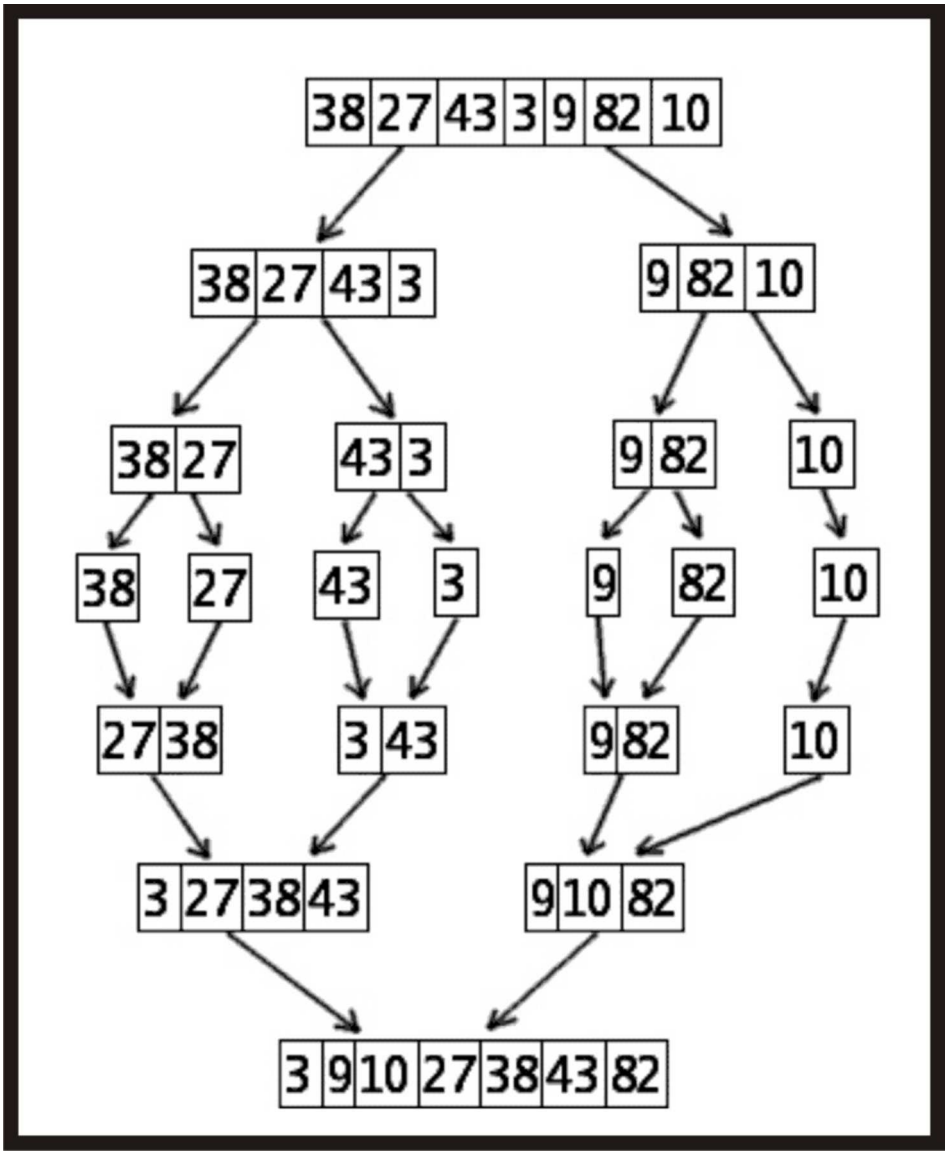

Figura 2.11: Ilustração do funcionamento do algoritmo MergeSort.

Os três passos úteis dos algoritmos dividir para conquistar, ou ***divide and conquer***, que se aplicam ao ***MergeSort*** são:

1. Dividir: Dividir os dados em subsequências pequenas;
2. Conquistar: Classificar as duas metades recursivamente aplicando o merge sort;
3. Combinar: Juntar as duas metades em um único conjunto já classificado.

2.8.1 Implementação

Primeiramente será apresentado pelo Programa 2.14 a implementação do método Merge responsável pela intercação do intervalo do vetor passado à ele.

Programa 2.14: Método Merge responsável pela intercalação.

```
void SortMethods::Merge(TItem *Array, long Left, long Means, long Right
    ) {
  TItem *ArrayAux = (TItem*)malloc ((Right-Left) * sizeof (TItem));
  long i,j,k;
  i = Left;
  j = Means;
  k = 0;
  while (i < Means && j < Right) {
    this->mMoviments++;
    this->mComparations++;
    if (Array[i].Key <= Array[j].Key)
      ArrayAux[k++] = Array[i++];
    else
      ArrayAux[k++] = Array[j++];
  }
  while (i < Means){
    ArrayAux[k++] = Array[i++];
    this->mMoviments++;
  }
  while (j < Right){
    ArrayAux[k++] = Array[j++];
    this->mMoviments++;
  }
  for (i = Left; i < Right; ++i){
    Array[i] = ArrayAux[i-Left];
    this->mMoviments++;
  }
  free (ArrayAux);
}
```

O algoritmo procede da seguinte forma:

1. É criado um vetor auxiliar do tamanho do intervalo;
2. Copia os valores do intervalo para o vetor auxiliar, de forma ordenada;
3. Copia o vetor auxiliar para o intervalo corresondente ao vetor a ser ordenado.

Para este trabalho o ***MergeSort*** foi implementado de uma forma não-recursiva. Esta implementação, Programa 2.15, favorece o desempenho do algoritmo, pois não é necessário o uso de uma pilha.

Programa 2.15: Método MergeSort.

```
void SortMethods::MergeSort(TItem* Array, long n){
  CTimer *Timer = new CTimer();
  this->ClearAll();
  Timer->start();
  long Left, Right;
  long Means=1;
  while (Means < n){
    Left = 0;
    while (Left + Means < n){
      Right = Left + 2*Means;
      if (Right > n)
        Right = n;
      this->Merge(Array, Left, Left+Means, Right);
      Left = Left + 2*Means;
    }
    Means = 2*Means;
  }
  Timer->stop();
  this->mTime = Timer->getElapsedTime();
}
```

2.8.2 Estudo da Complexidade

O calculo da análise de complexidade não será apresentada, mas de acordo com [9] a ordem de complexidade do algoritmo recursivo deste método é $O(n \log n)$.

2.8.3 Análise do algoritmo

A Tabela 2.7 apresenta as principais vantagens e desvantagens deste método.

Vantagens	**Desvantagens**
- Passível de ser transformado em estável	- Utiliza memória auxiliar
- Fácil Implementação	- Mais lento que o ***HeapSort***
- Complexidade $O(n \log n)$	

Tabela 2.7: Vantagens e desvantagens do Método MergeSort.

Capítulo 3

Testes

Neste capítulo são apresentados os resultados dos testes realizados com 4 tipos de vetores e 4 tamanhos diferentes. A análise dos resultados são feitos na Seção 3.3

3.1 Metodologia dos Testes

Os testes foram realizados em um Computador com as seguintes configurações:

- Processador Pentium D 2.8 Ghz FSB 800 Mhz;
- 1 GB de memória ram DDR 533 Mhz;
- Placa Mãe Asus P5ND2;
- Sistema Operacional Microsoft Windows XP Service Pack 3.

Os algoritmos de ordenação foram escritos na linguagem C/C++. O software utilizado para edição e compilação do código foi o Microsoft Visual Studio Express 2008.

Os gráficos foram gerados utilizando os softwares Microsoft Office Excel 2007 Service Pack 1 e R Project 2.8.0.

Para a praticidade durante os testes foram implementados métodos auxiliares que estão na classe `Operations.h`. Os detalhes da implementação desta classe é apresentada no Apêndice B.

Para realização dos testes em lotes foram utilizados arquivos textos de entrada e saída. Os arquivo de entrada, veja exemplo no Arquivo 3.1, contém as informações para os testes: Método a ser realizado, tamanho do vetor, tipo de vetor e a necessidade de exibição na tela ou não.

Arquivo 3.1: Exemplo de um arquivo de entrada para teste.

```
Merge 100 OrdC 0
Bubble 10000 OrdA 1
Quick 100 OrdP 1
Insert 532 OrdD 0
Insert 1000 OrdA 0
```

Os arquivos de saída contendo os resultados estão no Apêndice C

3.2 Resultados

Foram testados os sete métodos de ordenação, para cada método foi testado um tamanho diferente de vetor (100, 1000, 10000, 100000) e para cada tamanho um tipo diferente (Ordenado Crescente, Ordenado Decrescente, Ordenado Aleatório e parcialmente Aleatório). Para o tipo Aleatório o teste foi realizado para 10 vetores diferentes, assim o resultado aqui apresentado é a média dos valores. Os resultados dos 364 testes serão apresentados em quatro grupos, divididos pelo tipo do Vetor. Os métodos serão numerados para apresentação na tabela, conforme a lista abaixo:

1. BubbleSort
2. InsertSort
3. SelectSort
4. ShellSort
5. QuickSort
6. HeapSort
7. MergeSort

Também seram utilizados as seguintes abreviações para os tipos de vetores:

- OrdC: Vetor ordenado em ordem crescente;
- OrdD: Vetor ordenado em ordem decrescente;
- OrdP: Vetor parcialmente ordenado(10% aleatório);
- OrdA: Vetor aleatório;

3.2.1 Vetor ordenado em ordem crescente

A Tabela 3.1 apresenta a quantidade de comparações e movimentos para o vetor do tipo OrdC, enquanto a Tabela 3.2 apresenta os resultados em função do tempo.

Vetor Ordenado em Ordem Crescente								
	100		**1000**		**10000**		**100000**	
	Comp.	**Mov.**	**Comp.**	**Mov.**	**Comp.**	**Mov.**	**Comp.**	**Mov.**
1	4950	4950	499500	499500	49995000	49995000	4999950000	4999950000
2	99	198	99	1998	9999	19998	99999	199998
3	4950	297	499500	2997	49995000	29997	4999950000	299997
4	342	684	5457	10914	75243	150486	967146	1934292
5	606	189	909	1533	125439	17712	1600016	196605
6	1265	884	19562	12192	264433	156928	3312482	1900842
7	372	1376	5052	19968	71712	272640	877968	3385984

Tabela 3.1: Quantidade de comparações e movimentos dos testes no vetor OrdC.

Vetor Ordenado em Ordem Crescente				
	100	**1000**	**10000**	**100000**
1	0,000050	0,004822	0,483164	48,373736
2	0,000003	0,00002	0,000193	0,001936
3	0,000050	0,004834	0,561589	58,300695
4	0,000008	0,000108	0,001453	0,018899
5	0,000020	0,000191	0,002403	0,028667
6	0,000033	0,000412	0,008299	0,083919
7	0,000150	0,001487	0,015855	0,205009

Tabela 3.2: Tempo gasto pelos testes no vetor OrdC.

A seguir as Figuras 3.1, 3.2, 3.3 e 3.4 apresentam os gráficos em função do número de comparações e movimentos para os quatro tamanhos de vetores testados.

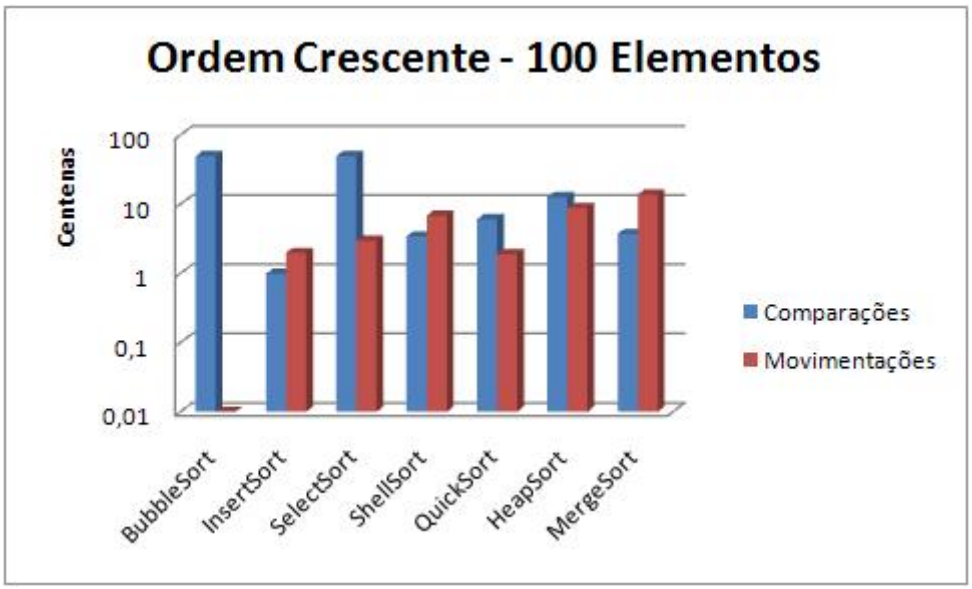

Figura 3.1: Número de comparações e movimentações do teste aplicado em um vetor de 100 posições do tipo OrdC.

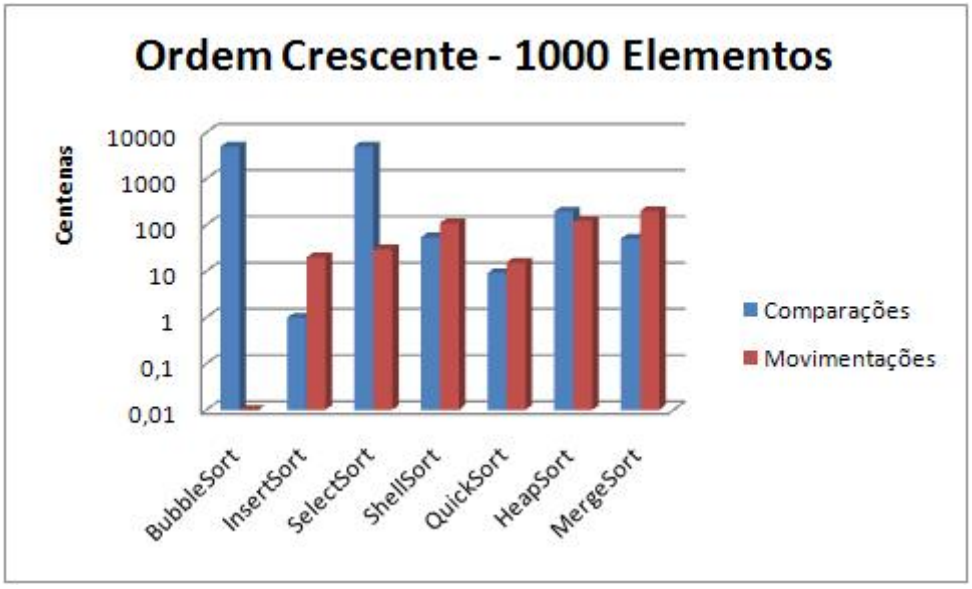

Figura 3.2: Número de comparações e movimentações do teste aplicado em um vetor de 1000 posições do tipo OrdC.

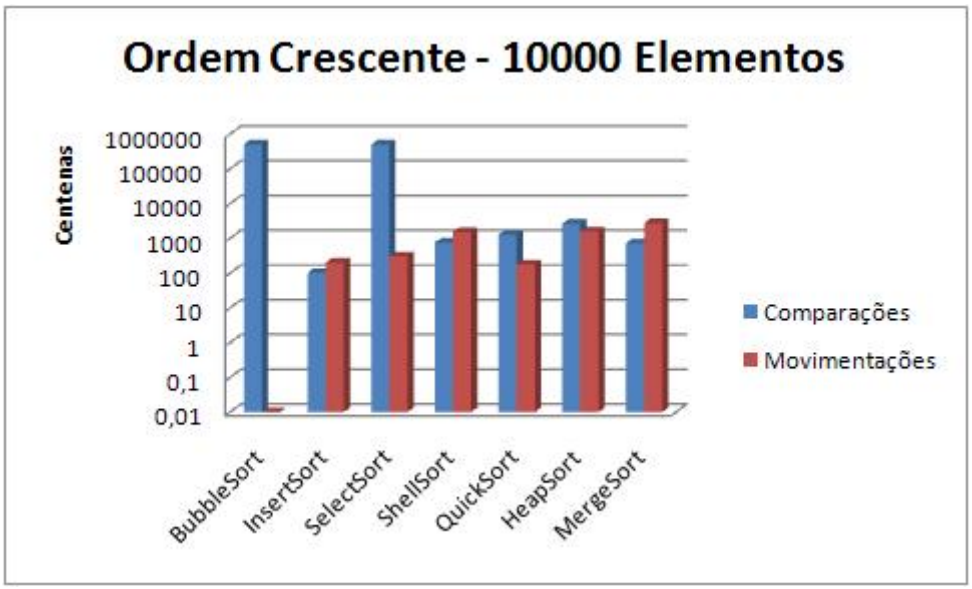

Figura 3.3: Número de comparações e movimentações do teste aplicado em um vetor de 10000 posições do tipo OrdC.

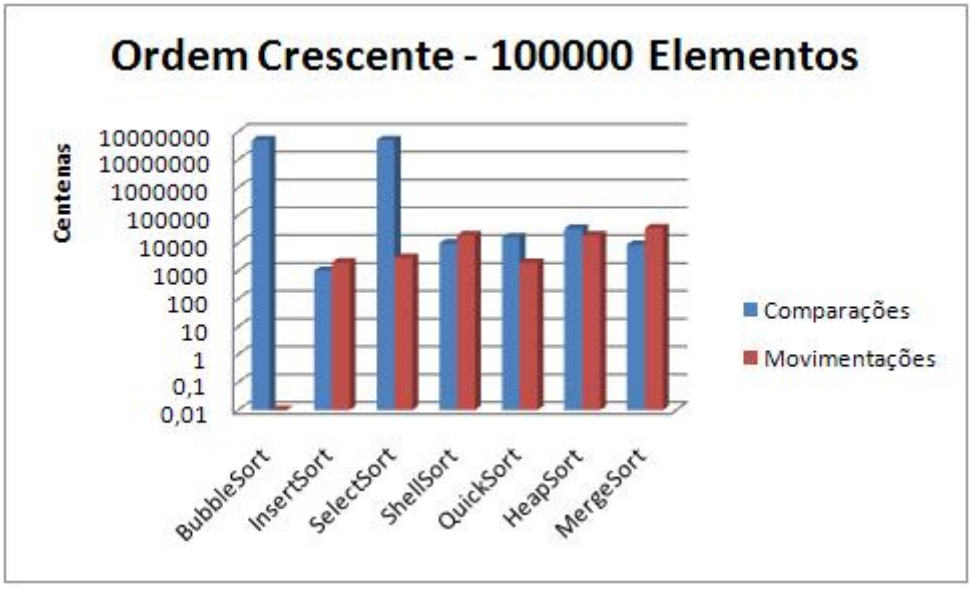

Figura 3.4: Número de comparações e movimentações do teste aplicado em um vetor de 100000 posições do tipo OrdC.

3.2.2 Vetor ordenado em ordem decrescente

A Tabela 3.3 apresenta a quantidade de comparações e movimentos para o vetor do tipo OrdD, enquanto a Tabela 3.4 apresenta os resultados em função do tempo.

Vetor Ordenado em Ordem Decrescente								
	100		**1000**		**10000**		**100000**	
	Comp.	**Mov.**	**Comp.**	**Mov.**	**Comp.**	**Mov.**	**Comp.**	**Mov.**
1	4950	4950	499500	499500	4999500	4999500	4999950000	4999950000
2	99	5148	999	501498	9999	50014998	9999	500014999
3	4950	297	499500	2997	49995000	29997	4999950000	299997
4	572	914	9377	14834	128947	204190	1586800	2553946
5	610	336	9016	303	125452	32712	1600030	346602
6	1125	747	17952	10811	246705	141975	3131748	1747201
7	316	1376	4932	19968	64608	272640	815024	3385984

Tabela 3.3: Quantidade de comparações e movimentos dos testes no vetor OrdD.

Vetor Ordenado em Ordem Decrescente				
	100	**1000**	**10000**	**100000**
1	0,000077	0,007503	0,752961	79,834753
2	0,000056	0,004897	0,667281	64,038128
3	0,000050	0,004834	0,561589	58,300695
4	0,000011	0,000164	0,002120	0,037005
5	0,000022	0,000190	0,002531	0,029430
6	0,000032	0,000381	0,007504	0,084207
7	0,000141	0,001495	0,016313	0,219894

Tabela 3.4: Tempo gasto pelos testes no vetor OrdD.

A seguir as Figuras 3.5, 3.6, 3.7 e 3.8 apresentam os gráficos em função do número de comparações e movimentos para os quatro tamanhos de vetores testados.

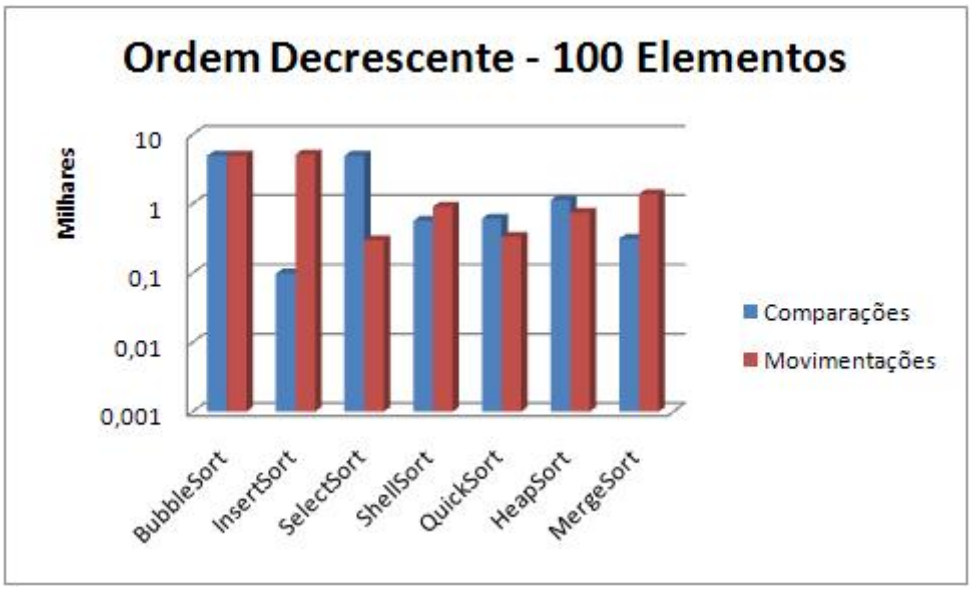

Figura 3.5: Número de comparações e movimentações do teste aplicado em um vetor de 100 posições do tipo OrdD.

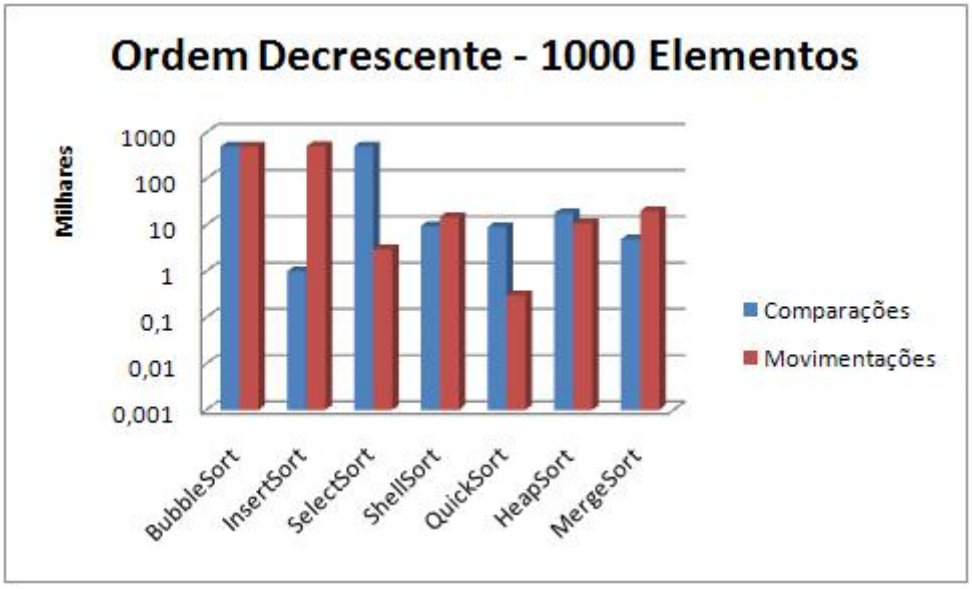

Figura 3.6: Número de comparações e movimentações do teste aplicado em um vetor de 1000 posições do tipo OrdD.

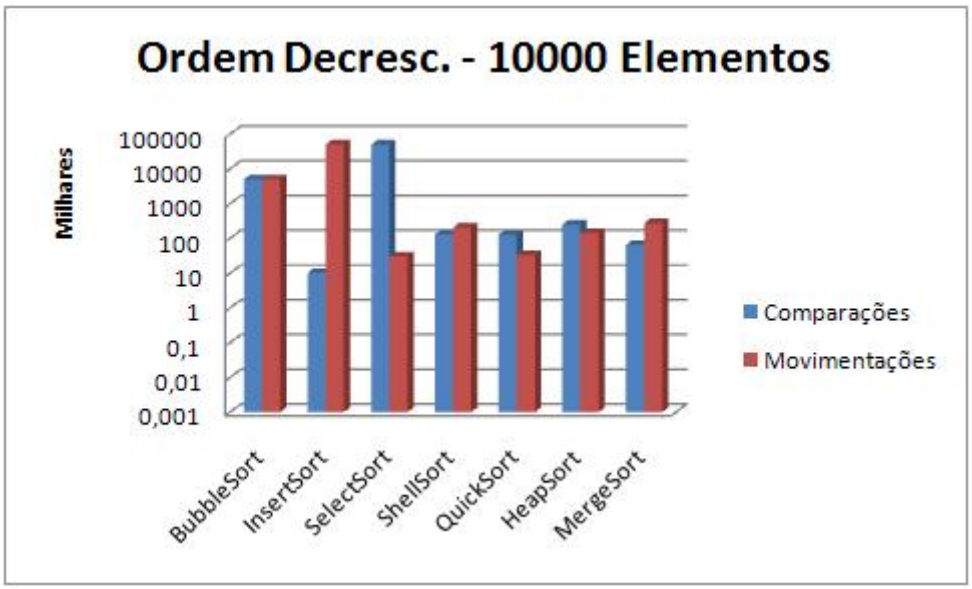

Figura 3.7: Número de comparações e movimentações do teste aplicado em um vetor de 10000 posições do tipo OrdD.

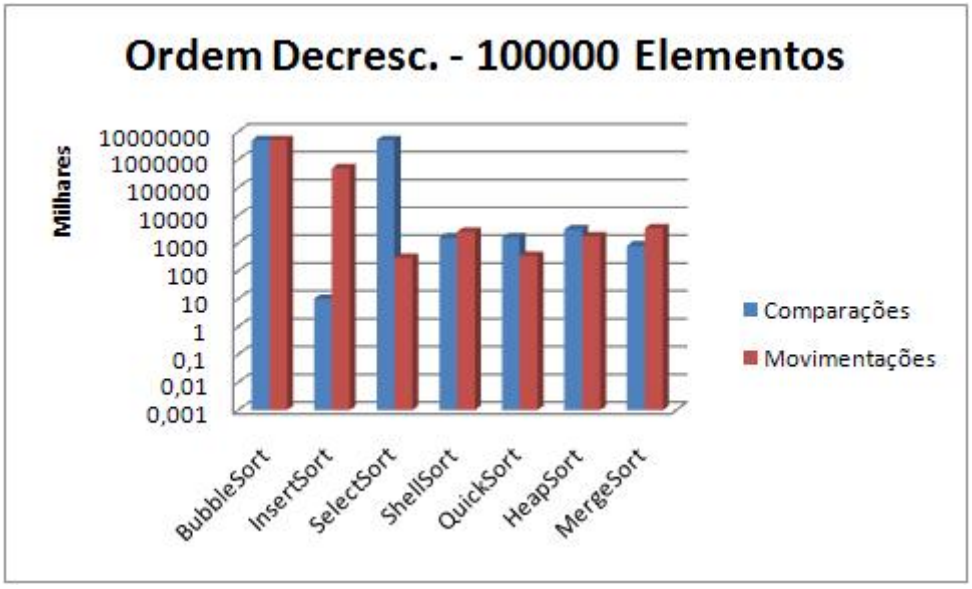

Figura 3.8: Número de comparações e movimentações do teste aplicado em um vetor de 100000 posições do tipo OrdD.

3.2.3 Vetor parcialmente ordenado

A Tabela 3.5 apresenta a quantidade de comparações e movimentos para o vetor do tipo OrdP, enquanto a Tabela 3.6 apresenta os resultados em função do tempo.

Vetor Parcialmente Ordenado								
	100		**1000**		**10000**		**100000**	
	Comp.	**Mov.**	**Comp.**	**Mov.**	**Comp.**	**Mov.**	**Comp.**	**Mov.**
1	4950	328	499500	35450	49995000	3154687	4999950000	359774720
2	99	528	99	34527	99	3410635	99	36093238
3	4950	297	499500	2997	49995000	29997	4999950000	299997
4	509	851	10857	16314	172528	247771	2214898	3182044
5	727	324	10725	3522	157689	66225	2251124	903300
6	1254	868	19525	12190	264737	156489	3294503	1887739
7	505	1376	7869	19968	114410	272640	1106062	3385984

Tabela 3.5: Quantidade de comparações e movimentos dos testes no vetor OrdP.

Vetor Parcialmente Ordenado				
	100	**1000**	**10000**	**100000**
1	0,000052	0,005348	0,526686	58,175386
2	0,000007	0,000379	0,046328	4,601265
3	0,000050	0,004834	0,561589	58,300695
4	0,000012	0,000236	0,003399	0,055055
5	0,000029	0,000309	0,003792	0,045093
6	0,000035	0,000405	0,004734	0,088935
7	0,000143	0,001518	0,023147	0,221169

Tabela 3.6: Tempo gasto pelos testes no vetor OrdP.

A seguir as figuras 3.9, 3.10, 3.11 e 3.12 apresentam os gráficos em função do número de comparações e movimentos para os quatro tamanhos de vetores testados.

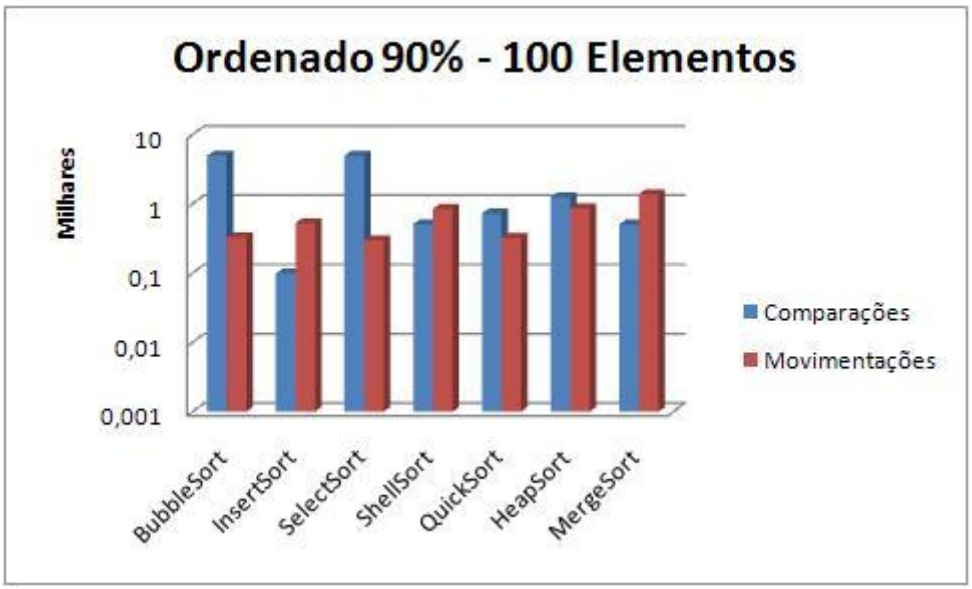

Figura 3.9: Número de comparações e movimentações do teste aplicado em um vetor de 100 posições do tipo OrdP.

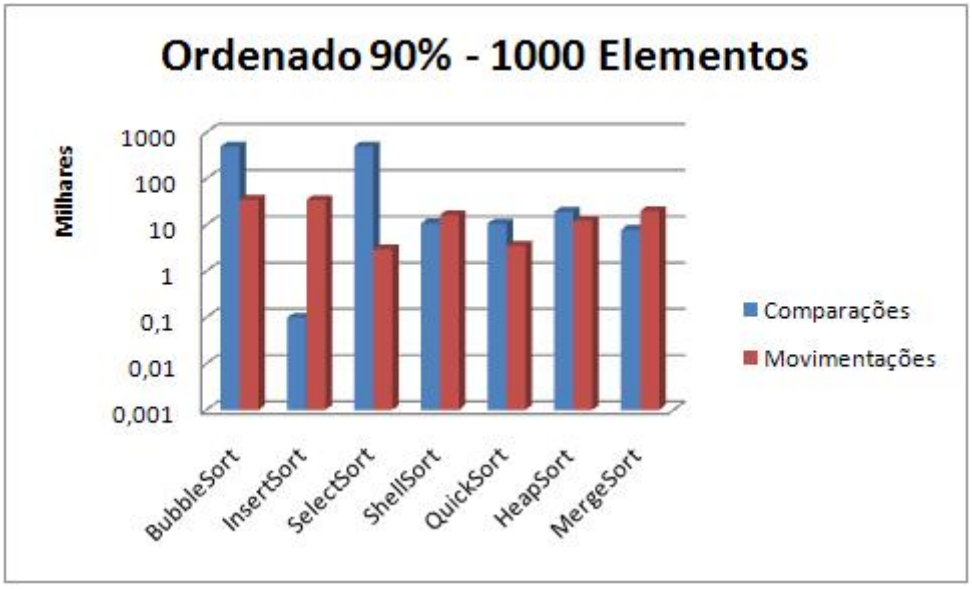

Figura 3.10: Número de comparações e movimentações do teste aplicado em um vetor de 1000 posições do tipo OrdP.

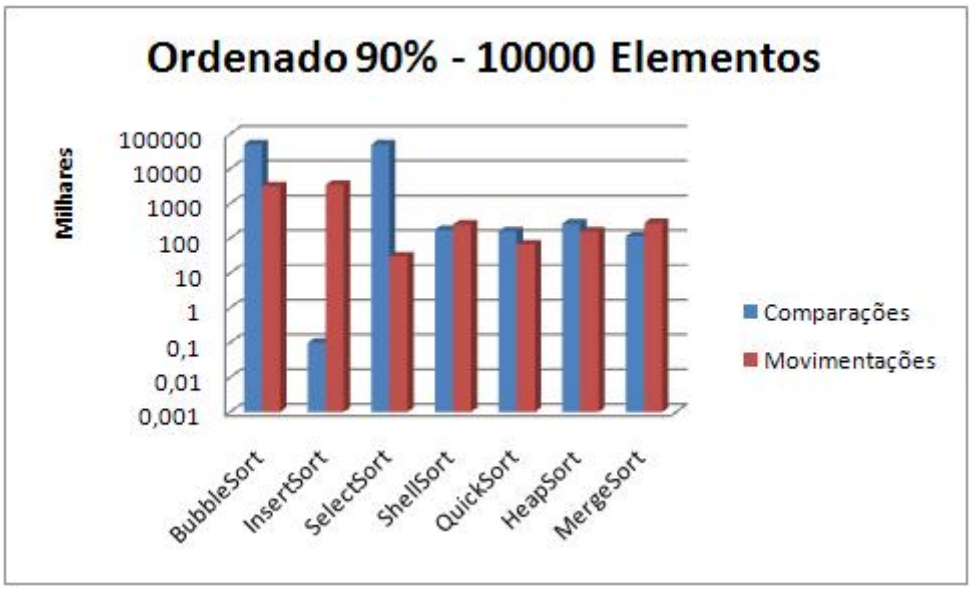

Figura 3.11: Número de comparações e movimentações do teste aplicado em um vetor de 10000 posições do tipo OrdP.

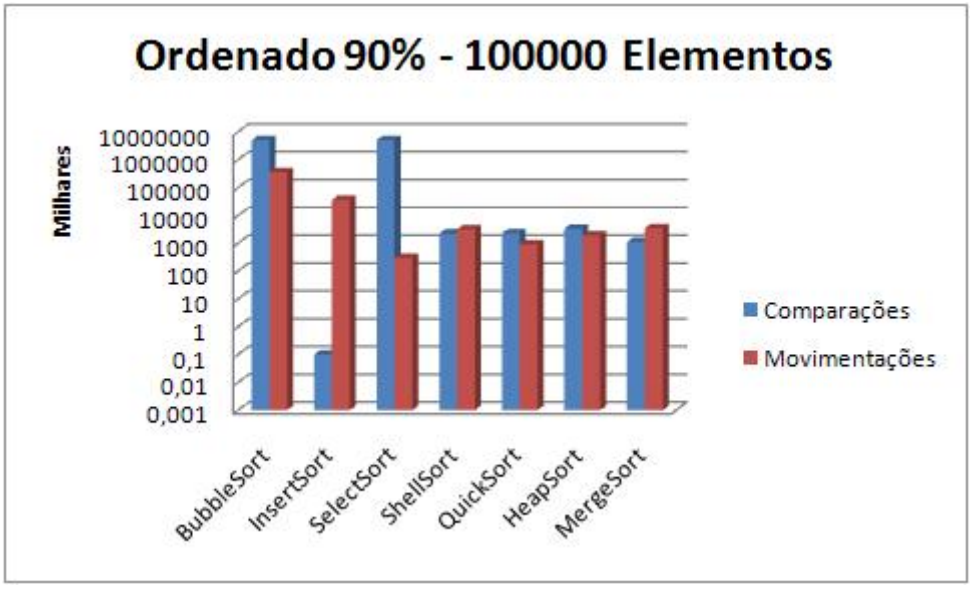

Figura 3.12: Número de comparações e movimentações do teste aplicado em um vetor de 100000 posições do tipo OrdP.

3.2.4 Vetor aleatório

A Tabela 3.7 apresenta a quantidade de comparações e movimentos para o vetor do tipo OrdA, enquanto a Tabela **??** apresenta os resultados em função do tempo.

Vetor Aleatório								
	100		**1000**		**10000**		**100000**	
	Comp.	**Mov.**	**Comp.**	**Mov.**	**Comp.**	**Mov.**	**Comp.**	**Mov.**
1	4950	2628	499500	242827	49995000	25160491	4999950000	2499136980
2	99	2387	999	253364	9999	25012754	99999	2500402956
3	4950	297	499500	2997	49995000	29997	4999950000	299997
4	818	1160	14364	19821	236735	311978	3711435	4670697
5	997	570	12852	8136	181203	103575	2114943	1310586
6	1205	817	18837	11556	255288	149150	3220006	1825075
7	558	1376	8744	19968	123685	272640	1566749	3385984

Tabela 3.7: Quantidade de comparações e movimentos dos testes no vetor OrdA.

Vetor Aleatório				
	100	**1000**	**10000**	**100000**
1	0,000076	0,008084	0,858684	114,840058
2	0,000027	0,002531	0,323982	31,500937
3	0,000050	0,004834	0,561589	58,300695
4	0,000016	0,000290	0,006158	0,104286
5	0,000031	0,000403	0,004987	0,084427
6	0,000034	0,000420	0,009227	0,087964
7	0,00015	0,001598	0,019384	0,231564

Tabela 3.8: Tempo gasto pelos testes no vetor OrdA.

A seguir as figuras 3.13, 3.14, 3.15 e 3.16 apresentam os gráficos em função do número de comparações e movimentos para os quatro tamanhos de vetores testados.

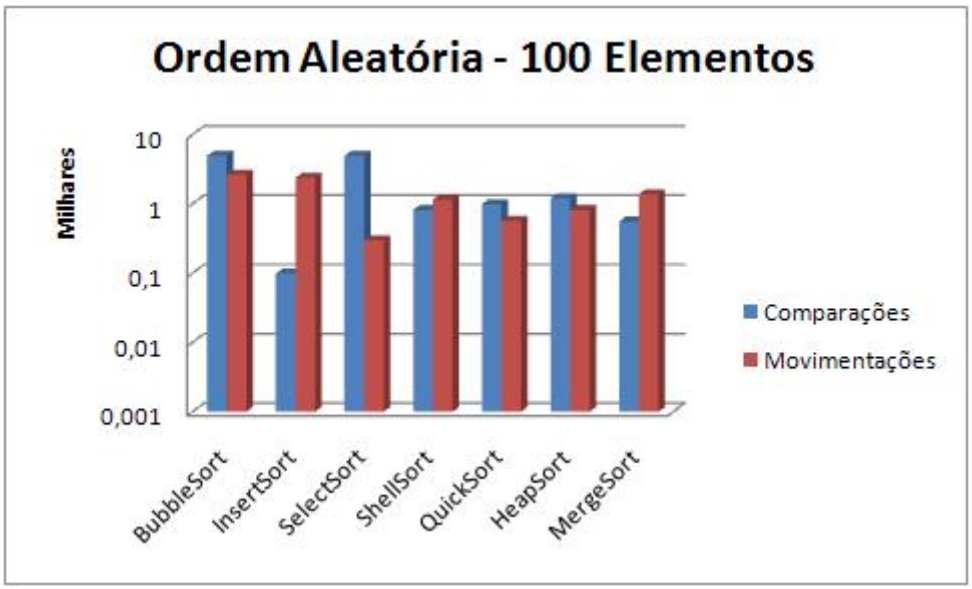

Figura 3.13: Número de comparações e movimentações do teste aplicado em um vetor de 100 posições do tipo OrdA.

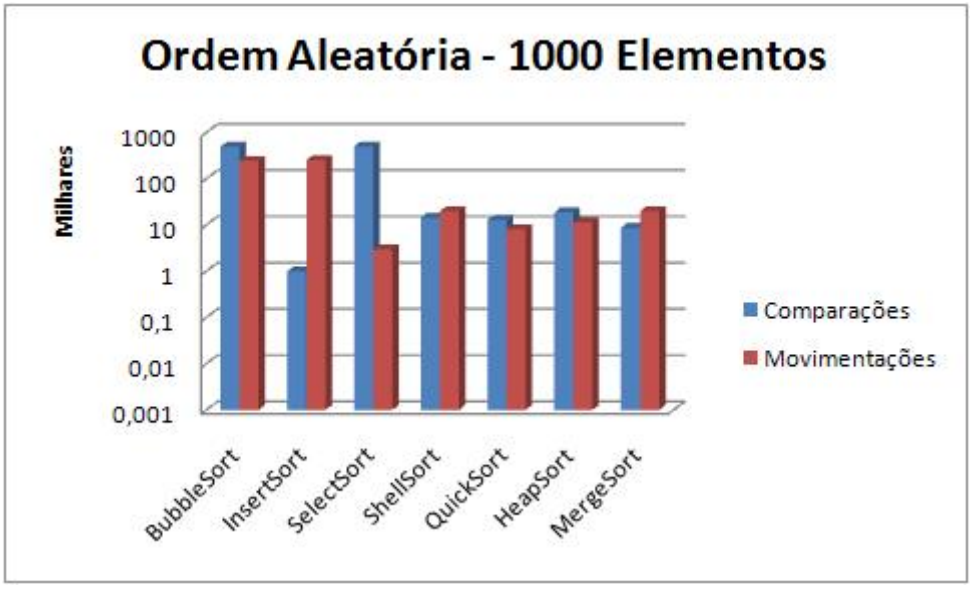

Figura 3.14: Número de comparações e movimentações do teste aplicado em um vetor de 1000 posições do tipo OrdA.

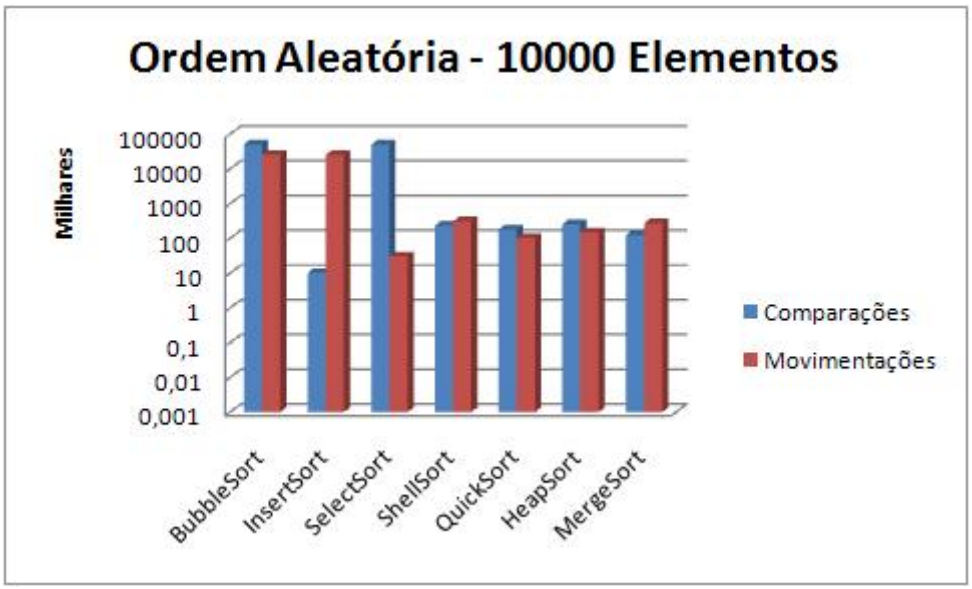

Figura 3.15: Número de comparações e movimentações do teste aplicado em um vetor de 10000 posições do tipo OrdA.

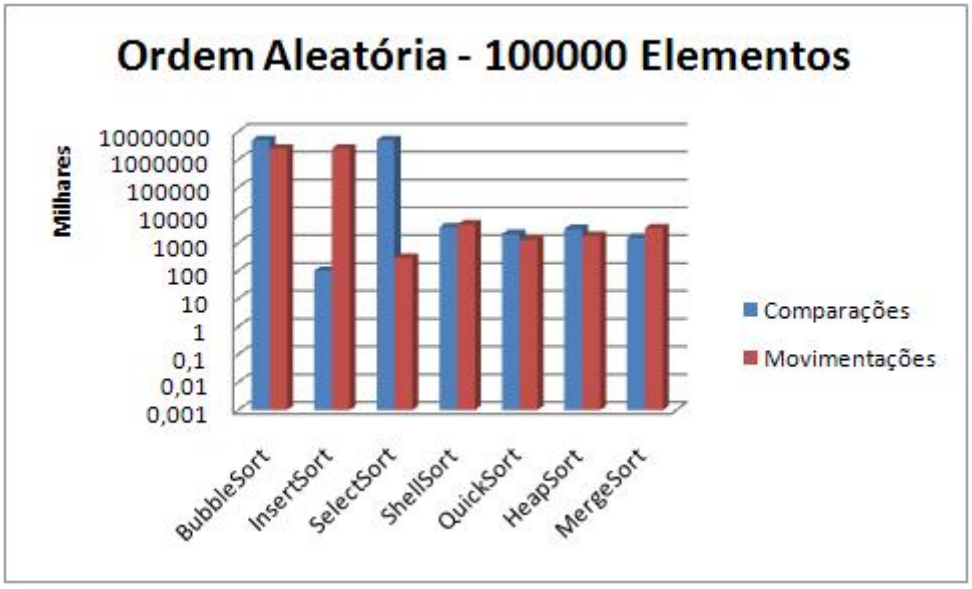

Figura 3.16: Número de comparações e movimentações do teste aplicado em um vetor de 100000 posições do tipo OrdA.

3.3 Análise dos Resultados

3.3.1 Vetor ordenado em ordem crescente

Os dois gráficos (Figuras 3.17 e 3.18) a seguir demonstram o comportamento dos métodos em relação ao tempo gasto na ordenação de um vetor do tipo OrdC nos quatro tamanhos propostos.

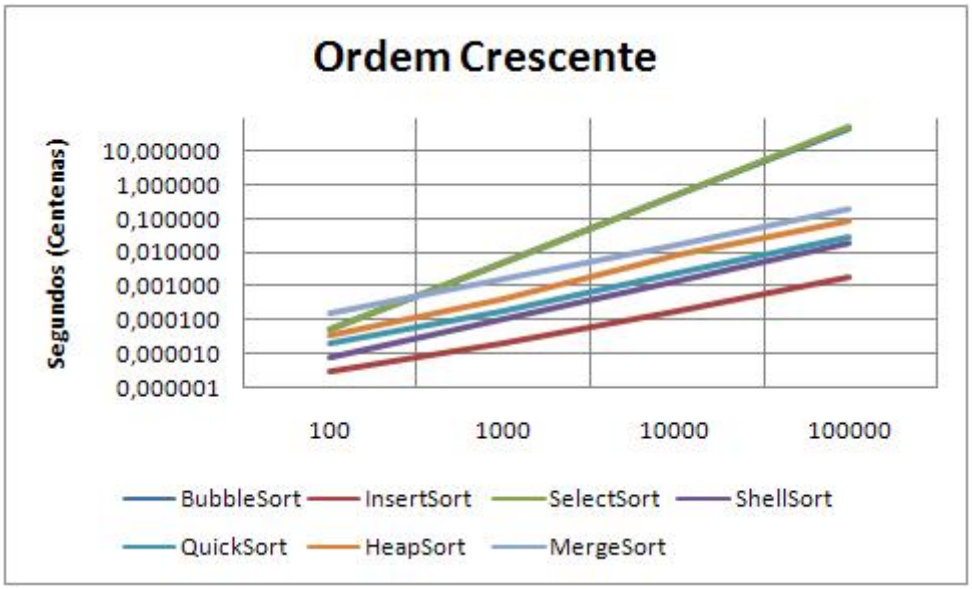

Figura 3.17: Curva de desempenho dos 7 métodos.

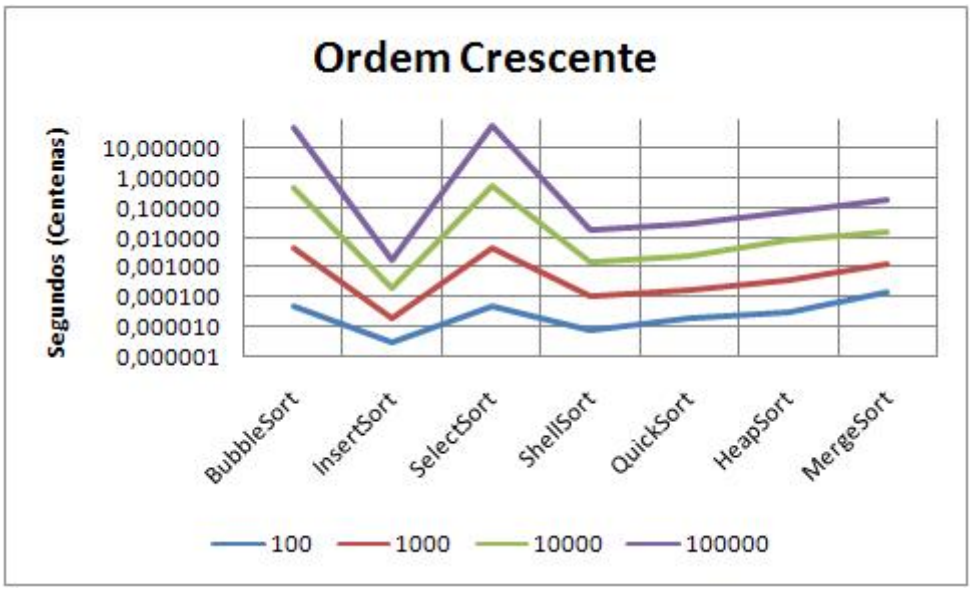

Figura 3.18: Tempo gasto pelos métodos nos quatro tamanhos de vetores propostos.

Conforme os gráficos podemos observar que:

- O ***InsertSort*** é a escolha ideal para um vetor que ja está ordenado, independente do seu tamanho;

- ***BubbleSort*** e ***SelectSort*** são os piores para vetores já ordenados.
- O algoritmo ***ShellSort*** apresenta a mesma velocidade dos algoritmos ***QuickSort***, ***HeapSort*** e ***MergeSort***, que são $O(n \log n)$.

3.3.2 Vetor ordenado em ordem decrescente

Os dois gráficos (Figuras 3.19 e 3.20) a seguir demonstram o comportamento dos métodos em relação ao tempo gasto na ordenação de um vetor do tipo OrdD nos quatro tamanhos propostos.

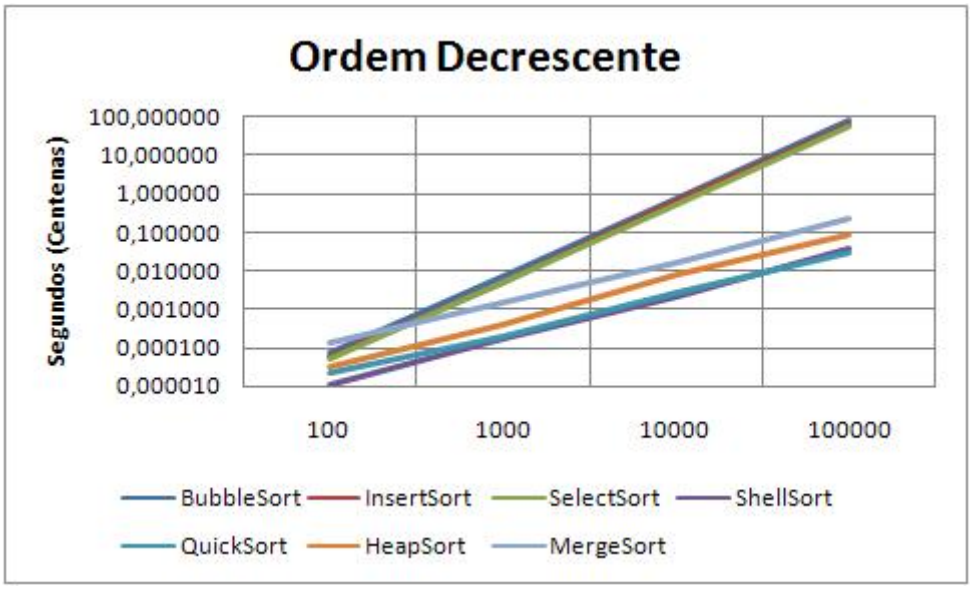

Figura 3.19: Curva de desempenho dos 7 métodos.

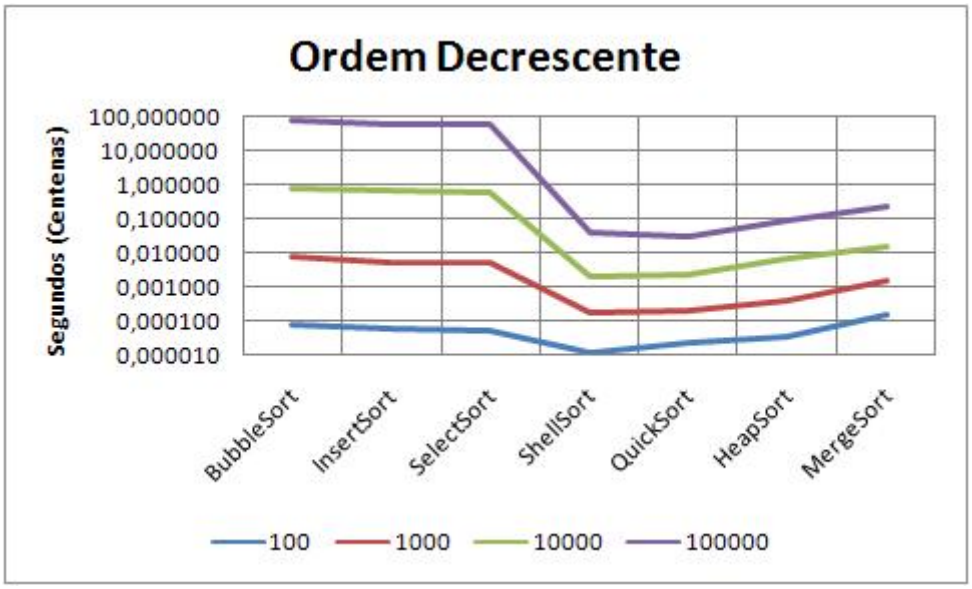

Figura 3.20: Tempo gasto pelos métodos nos quatro tamanhos de vetores propostos.

Conforme os gráficos podemos observar que:

- Para arquivos ordenados inversamente de até 10000 elementos o *ShellSort* demonstrou se a melhor escolha, acima deste número a escolha perfeita é o ***QuickSort***.

- ***BubbleSort***, ***InsertSort*** e ***SelectSort*** são os piores para vetores já ordenados inversamente.
- Dentre os algoritmos eficientes o ***MergeSort*** é o que tem o pior tempo de execução.

3.3.3 Vetor parcialmente ordenado

Os dois gráficos (figuras 3.21 e 3.22) a seguir demonstram o comportamento dos métodos em relação ao tempo gasto na ordenação de um vetor do tipo OrdP nos quatro tamanhos propostos.

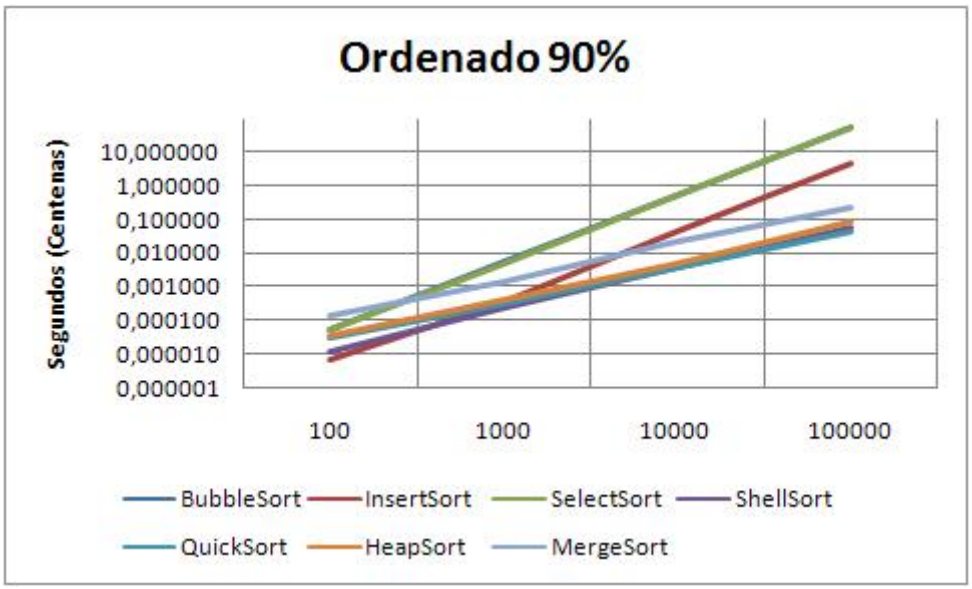

Figura 3.21: Curva de desempenho dos 7 métodos.

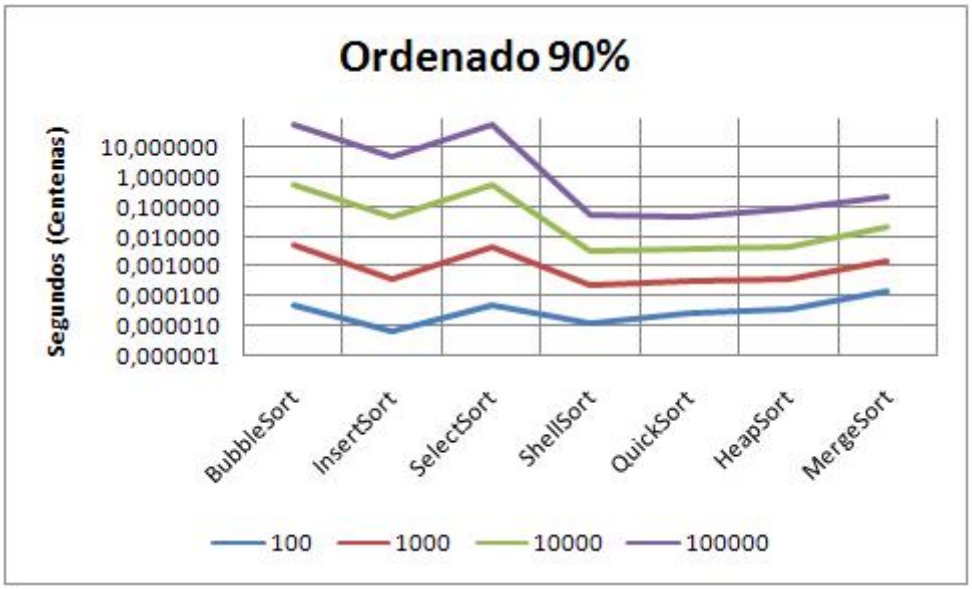

Figura 3.22: Tempo gasto pelos métodos nos quatro tamanhos de vetores propostos.

Conforme os gráficos podemos observar que:

- O ***ShellSort*** e ***QuickSort*** são os melhores para este tipo de vetor em qualquer tamanho, porém para arquivos pequenos existe uma vantagem do ***InsertSort***.
- ***BubbleSort*** e ***SelectSort*** são os piores para este tipo de vetor, independente do tamanho.

- Dentre os algoritmos eficientes o ***MergeSort*** é o que tem o pior tempo de execução.
- Até 1000 elementos o ***InsertSort*** apresentou compotamento $n \log n$, acima disto ele apresentou-se como quadrático.

3.3.4 Vetor aleatório

Os dois gráficos (figuras 3.23 e 3.24) a seguir demonstram o comportamento dos métodos em relação ao tempo gasto na ordenação de um vetor do tipo OrdA nos quatro tamanhos propostos.

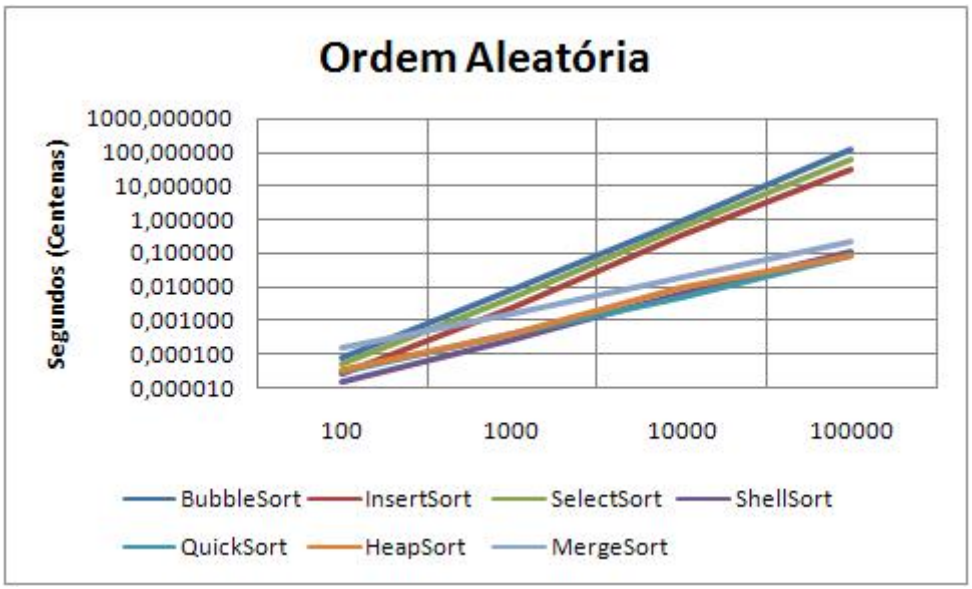

Figura 3.23: Curva de desempenho dos 7 métodos.

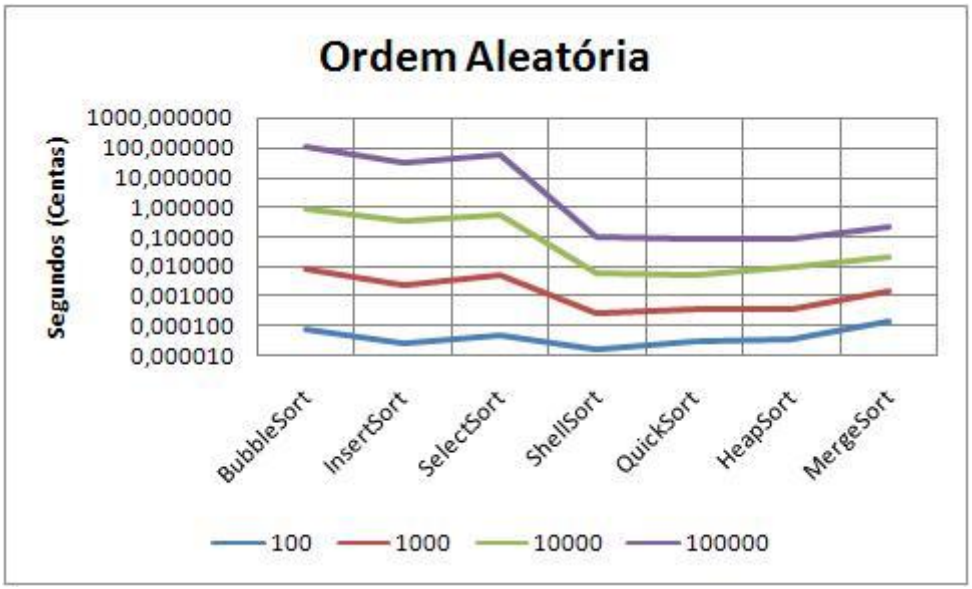

Figura 3.24: Tempo gasto pelos métodos nos quatro tamanhos de vetores propostos.

Conforme os gráficos podemos observar que:

- Para pequenos arquivos o ***ShellSort*** foi o melhor, para os maiores o ***QuickSort*** apresentou o melhor tempo de execução.
- Os algoritmos quadráticos foram os piores neste teste, sendo o ***InsertSort*** o que apresentou um vantagem entre eles.

- Dentre os algoritmos eficientes o ***MergeSort*** é o que tem o pior tempo de execução.

Capítulo 4

Conclusão

Foram apresentados sete métodos de ordenação interna mediante a comparação de chaves. A Seção 3 aprentou os dados de 364 testes realizados utilizando esses métodos. Após o estudo desses resultados e dos algoritmos, são apresentadas observações sobre cada um dos métodos.

1. ***BubbleSort***:
    - Apresentou-se como o pior método de ordenação;
2. ***InsertSort***:
    - Para arquivos pequenos é o mais interessante, sendo mais vantajoso, neste caso, do que algoritmos $n \log n$;
    - Em arranjos já ordenados apresentou um comportamento linear, sendo útil para ordenação de um arquivo com poucos elementos fora da ordem.
    - Com vetores aleatórios é o melhor entre os de complexidade quadrática.
3. ***SelectSort***:
    - Apresenta uma vantagem apenas no número de movimentações, porém seu tempo de execução é pior que o ***InsertSort***;
    - A quantidade de movimentos e comparações é a mesma, independente do tipo de vetor a ser ordenado, variando apenas em função do tamanho.
    - A sua utilização deve ser feita quando os registros dos arquivos forem grandes, em virtude da quantidade de movimentos que é realizada para a operação.
4. ***ShellSort***:
    - Apresentou-se eficiente em todos os testes;
    - Em geral é um método que se adequa a todas as situações;
    - A sua relação com o ***QuickSort*** cresce à medida que o número de elementos aumenta;
    - Em todas as situaçõe manteve-se próximo do algoritmo mais rápido do teste;

- No arquivo parcialmente ordenado ele foi o melhor.

5. ***QuickSort***:
   - Obteve o melhor resultado na maioria dos testes;
   - Teve um maior desempenho nos arquivos aleatórios;
   - Obteve uma vantagem constante em relação ao ***HeapSort***;
   - Sua eficiência foi identificada apenas nos grandes arquivos;
6. ***HeapSort***:
   - Seu desempenho não foi melhor em nenhum teste;
   - O seu comportamento foi o mesmo para todos os tipos de vetores;
   - Em arquivos aleatórios de grande tamanho o seu desempenho foi similar ao ***ShellSort***.
7. ***MergeSort***:
   - Apresentou os piores resultados entre os algoritmos de ordenação eficientes;
   - Em todos testes com 100 elementos apresentou um tempo de execução maior que os algoritmos $O(n^2)$;

Pelos testes realizados, conclui-se que a escolha de um algoritmo de ordenação depende do tipo de vetor e tamanho a ser ordenado.

Uma alternativa para obter um desempenho melhor é a combinação de métodos. Por exemplo, suponha um algoritmo de ordenação composto pelo método ***QuickSort*** e ***InsertSort***. O algoritmo verifica o tamanho do vetor a ser ordenado e em seguida aplica o ***QuickSort*** se o arquivo for grande e aleatório ou aplica o ***InsertSort*** se o arquivo for pequeno e parcialmente ordenado.

Conclui-se também que métodos eficientes, demandam uma implementação mais cautelosa do código, pois são detalhes que definem a qualidade da operação de ordenação

Apêndice A

CTimer

Implementação da classe CTimer utilizada para a medição do tempo de execução dos algoritmos de ordenação.

Programa A.1: Implementação dos métodos da classe CTimer.

```
#include "stdafx.h"
#include "CTimer.h"

CTimer::CTimer(){
  tStart.QuadPart = 0;
  tStop.QuadPart = 0;
}

double LITOSecs(LARGE_INTEGER * L){
  LARGE_INTEGER frequency;
  QueryPerformanceFrequency( &frequency ) ;
  return ((double)L->QuadPart /(double)frequency.QuadPart) ;
}


void CTimer::start(){
  QueryPerformanceCounter(&this->tStart);
}

void CTimer::stop(){
  QueryPerformanceCounter(&this->tStop);
}

double CTimer::getElapsedTime(){
  LARGE_INTEGER time;
  time.QuadPart = this->tStop.QuadPart - this->tStart.QuadPart;
  return LITOSecs(&time);
}
```

Apêndice B

Operations

Foi implementado 3 métodos auxiliares para a realização dos testes. São eles:

- `GeneratingArray`: Gera vetores no tamanho e tipo passados como argumento. Os vetores podem ser do tipo:
  - **OrdC(Type 0):** Ordenado em ordem crescente;
  - **OrdD(Type 1):** Ordenado em ordem decrescente;
  - **OrdA(Type 2):** Ordem aleatória;
  - **OrdP(Type 3):** 10% desordenado;
- `PrintArray`: Exibe o conteúdo de um vetor:
- `ProcessCommand`: Processa uma string e identifica os argumentos, retornando um TDA `Command` com os seguintes campos:
  - **Method:** Método de ordenação;
  - **mItens:** Quantidade de elentos no vetor a ser criado;
  - **TypeArray:** Tipo de vetor a ser criado;
  - **Print:** Verificação de exibição ou não do vetor a ser testado;
- `ProcessFile`: Processa as strings de comando de uma arquivo texto e guarda os resultados dos testes realizados em um arquivo de saída:

  A seguir o programa B.1 apresenta o código dos métodos auxiliares.

Programa B.1: Implementação dos métodos da classe Operations.

```
#include "stdafx.h"
#include "SortMethods.h"
#include <stdlib.h>
#include <time.h>
#include "Operations.h"
#include <iostream>
using namespace std;

TItem *GeneratingArray(long mItens, int Type){
  TItem *Array = (TItem*)malloc(sizeof(TItem) * mItens);
  srand((int)time(NULL));
```

```
  switch (Type){
    case 0:{
      for (long i=0; i<mItens; i++)
        Array[i].Key = i;
    }break;
    case 1:{
      int p=0;
      for (long i=mItens-1; i>=0; i--)
        Array[p++].Key = i;
    }break;
    case 2:{
      for (long i=0; i<mItens; i++)
        Array[i].Key = (long)rand() % mItens;
    }break;
    case 3:{
      for (long i=0; i<mItens; i++){
        if (i%10==0)
          do
            Array[i].Key = (long)rand() % mItens;
          while(Array[i].Key == i);
        else
          Array[i].Key = i;
      }
    }
  }
  return Array;
}

void PrintArray(TItem *Array, long mItens){
  cout << "================================";
  for(long i=0; i<mItens; i++){
    cout.width(6);
    cout << Array[i].Key;
    cout << " |";
  }
  cout << "================================" << endl;
}

TCommand ProcessCommand(char *StrCommand){
  TCommand CommandTemp;
  sscanf(StrCommand,"%s %d %s %d", CommandTemp.Method, &
      CommandTemp.mItens, CommandTemp.TypeArray, &CommandTemp.
      Print);
  return CommandTemp;
}

void ProcessFile(char *FileIn, char *FileOut){
  FILE *pFile;
  fopen_s(&pFile, FileIn, "r");
  FILE *pFile2;
  fopen_s(&pFile2, FileOut, "w");
  if(pFile==NULL)
    return;
  TCommand Command;
  char buffer[200];
  sprintf(buffer, "%12s %12s %12s %14s %14s %12s|n|n", "Method",
      "Quant", "Type", "mComp.", "mMovim.", "mTime");
```

```
  fputs(buffer, pFile2);
  int type;
  int i=1;
  SortMethods *Sort = new SortMethods();
  while(fgets(buffer,200,pFile)){
    Command = ProcessCommand(buffer);

    if (strcmp(Command.TypeArray, "OrdC") == 0)
      type = 0;
    else if (strcmp(Command.TypeArray, "OrdD") == 0)
      type = 1;
    else if (strcmp(Command.TypeArray, "OrdA") == 0)
      type = 2;
    else if (strcmp(Command.TypeArray, "OrdP") == 0)
      type = 3;


    TItem *Array = GeneratingArray(Command.mItens, type);


    if (Command.Print)
      PrintArray(Array, Command.mItens);

    if (strcmp(Command.Method, "Bubble") == 0)
      Sort->BubbleSort(Array,Command.mItens);
    else if (strcmp(Command.Method, "Insert") == 0)
      Sort->InsertSort(Array,Command.mItens);
    else if (strcmp(Command.Method, "Select")== 0)
      Sort->SelectSort(Array,Command.mItens);
    else if (strcmp(Command.Method, "Shell") == 0)
      Sort->ShellSort(Array,Command.mItens);
    else if (strcmp(Command.Method, "Quick") == 0)
      Sort->QuickSort(Array,Command.mItens);
    else if (strcmp(Command.Method, "Heap") == 0)
      Sort->HeapSort(Array,Command.mItens);
    else if (strcmp(Command.Method, "Merge") == 0)
      Sort->MergeSort(Array,Command.mItens);

    if (Command.Print)
      PrintArray(Array, Command.mItens);


    sprintf(buffer, "%12s %12d %12s %14.0f %14.0f %12f|n", Command
        .Method, Command.mItens, Command.TypeArray, Sort->
        getComparations(), Sort->getMoviments(), Sort->getTime());
    fputs(buffer,pFile2);

    cout << "teste " << i++ << endl;

  }
  fclose(pFile);
  fclose(pFile2);
}
```

Apêndice C

Arquivos de saída contendo os resultados

Arquivos do tipo texto contendo os resultados dos seguintes testes realizados.

- BubbleSort para 4 tipos de vetor e 4 tamanhos diferentes;
- InsertSort para 4 tipos de vetor e 4 tamanhos diferentes;
- SelectSort para 4 tipos de vetor e 4 tamanhos diferentes;
- ShellSort para 4 tipos de vetor e 4 tamanhos diferentes;
- QuickSort para 4 tipos de vetor e 4 tamanhos diferentes;
- HeapSort para 4 tipos de vetor e 4 tamanhos diferentes;
- MergeSort para 4 tipos de vetor e 4 tamanhos diferentes;

Arquivo C.1: Resultados do testes utilizando BubbleSort.

Method	Quant	Type	mComp.	mMovim.	mTime
Bubble	100	OrdC	4950	0	0.000050
Bubble	100	OrdD	4950	4950	0.000077
Bubble	100	OrdP	4950	328	0.000052
Bubble	100	OrdA	4950	2628	0.000074
Bubble	100	OrdA	4950	2628	0.000076
Bubble	100	OrdA	4950	2628	0.000076
Bubble	100	OrdA	4950	2628	0.000075
Bubble	100	OrdA	4950	2628	0.000076
Bubble	100	OrdA	4950	2628	0.000076
Bubble	100	OrdA	4950	2628	0.000076
Bubble	100	OrdA	4950	2628	0.000075
Bubble	100	OrdA	4950	2628	0.000076
Bubble	1000	OrdC	499500	0	0.004822
Bubble	1000	OrdD	499500	499500	0.007503
Bubble	1000	OrdP	499500	35450	0.005348
Bubble	1000	OrdA	499500	242827	0.008302
Bubble	1000	OrdA	499500	242827	0.008245
Bubble	1000	OrdA	499500	242827	0.008085
Bubble	1000	OrdA	499500	242827	0.008087
Bubble	1000	OrdA	499500	242827	0.008113
Bubble	1000	OrdA	499500	242827	0.008084
Bubble	1000	OrdA	499500	242827	0.008138
Bubble	1000	OrdA	499500	242827	0.008082
Bubble	1000	OrdA	499500	242827	0.008081
Bubble	1000	OrdA	499500	242827	0.008237
Bubble	10000	OrdC	49995000	0	0.483164
Bubble	10000	OrdD	49995000	49995000	0.752961
Bubble	10000	OrdP	49995000	3154687	0.526686
Bubble	10000	OrdA	49995000	24768116	0.853146
Bubble	10000	OrdA	49995000	24768116	0.851169
Bubble	10000	OrdA	49995000	25320916	0.859995
Bubble	10000	OrdA	49995000	25087802	0.856508
Bubble	10000	OrdA	49995000	25011848	0.863044
Bubble	10000	OrdA	49995000	25160491	0.858684
Bubble	10000	OrdA	49995000	24831680	0.858342
Bubble	10000	OrdA	49995000	25022032	0.854665
Bubble	10000	OrdA	49995000	25022032	0.855867
Bubble	10000	OrdA	49995000	25333035	0.857315
Bubble	100000	OrdC	4999950000	0	48.373736
Bubble	100000	OrdD	4999950000	4999950000	79.834753
Bubble	100000	OrdP	4999950000	359774720	58.175386
Bubble	100000	OrdA	4999950000	2503337665	107.334751
Bubble	100000	OrdA	4999950000	2499304061	107.940812
Bubble	100000	OrdA	4999950000	2491137250	110.872767
Bubble	100000	OrdA	4999950000	2498844837	111.346239
Bubble	100000	OrdA	4999950000	2505224085	110.317295
Bubble	100000	OrdA	4999950000	2496906642	109.635236
Bubble	100000	OrdA	4999950000	2504213408	108.712391
Bubble	100000	OrdA	4999950000	2498470030	121.313059
Bubble	100000	OrdA	4999950000	2488891637	120.359626
Bubble	100000	OrdA	4999950000	2499136980	117.840058

Arquivo C.2: Resultados do testes utilizando InsertSort.

Method	Quant	Type	mComp.	mMovim.	mTime
Insert	100	OrdC	99	198	0.000003
Insert	100	OrdD	99	5148	0.000056
Insert	100	OrdP	99	528	0.000007
Insert	100	OrdA	99	2387	0.000027
Insert	100	OrdA	99	2387	0.000027
Insert	100	OrdA	99	2387	0.000027
Insert	100	OrdA	99	2387	0.000027
Insert	100	OrdA	99	2387	0.000027
Insert	100	OrdA	99	2387	0.000027
Insert	100	OrdA	99	2387	0.000027
Insert	100	OrdA	99	2387	0.000027
Insert	100	OrdA	99	2387	0.000027
Insert	100	OrdA	99	2387	0.000027
Insert	1000	OrdC	999	1998	0.000020
Insert	1000	OrdD	999	501498	0.004897
Insert	1000	OrdP	999	34527	0.000379
Insert	1000	OrdA	999	253364	0.002623
Insert	1000	OrdA	999	253364	0.002568
Insert	1000	OrdA	999	253364	0.002535
Insert	1000	OrdA	999	253364	0.002533
Insert	1000	OrdA	999	253364	0.002540
Insert	1000	OrdA	999	253364	0.002532
Insert	1000	OrdA	999	253364	0.002531
Insert	1000	OrdA	999	253364	0.002531
Insert	1000	OrdA	999	253364	0.002579
Insert	1000	OrdA	999	253364	0.002554
Insert	10000	OrdC	9999	19998	0.000193
Insert	10000	OrdD	9999	50014998	0.667281
Insert	10000	OrdP	9999	3410635	0.046328
Insert	10000	OrdA	9999	24919368	0.362001
Insert	10000	OrdA	9999	25048430	0.325466
Insert	10000	OrdA	9999	25048430	0.313694
Insert	10000	OrdA	9999	25048430	0.332196
Insert	10000	OrdA	9999	25048430	0.336111
Insert	10000	OrdA	9999	24977078	0.323982
Insert	10000	OrdA	9999	24977078	0.341779
Insert	10000	OrdA	9999	24977078	0.320963
Insert	10000	OrdA	9999	25002784	0.345760
Insert	10000	OrdA	9999	25002784	0.325827
Insert	100000	OrdC	99999	199998	0.001936
Insert	100000	OrdD	99999	5000149998	64.038128
Insert	100000	OrdP	99999	360932382	4.601265
Insert	100000	OrdA	99999	2500686937	31.768194
Insert	100000	OrdA	99999	2502664733	32.525120
Insert	100000	OrdA	99999	2500905360	31.451376
Insert	100000	OrdA	99999	2500402956	31.500937
Insert	100000	OrdA	99999	2503185261	31.822204
Insert	100000	OrdA	99999	2496615174	31.373653
Insert	100000	OrdA	99999	2488758797	31.469461
Insert	100000	OrdA	99999	2494440950	31.235527
Insert	100000	OrdA	99999	2497932586	31.412962
Insert	100000	OrdA	99999	2502987729	31.172520

Arquivo C.3: Resultados do testes utilizando SelectSort.

Method	Quant	Type	mComp.	mMovim.	mTime
Select	100	OrdC	4950	297	0.000050
Select	100	OrdD	4950	297	0.000052
Select	100	OrdP	4950	297	0.000051
Select	100	OrdA	4950	297	0.000054
Select	100	OrdA	4950	297	0.000055
Select	100	OrdA	4950	297	0.000054
Select	100	OrdA	4950	297	0.000055
Select	100	OrdA	4950	297	0.000054
Select	100	OrdA	4950	297	0.000054
Select	100	OrdA	4950	297	0.000054
Select	100	OrdA	4950	297	0.000054
Select	100	OrdA	4950	297	0.000054
Select	1000	OrdC	499500	2997	0.004834
Select	1000	OrdD	499500	2997	0.004845
Select	1000	OrdP	499500	2997	0.006082
Select	1000	OrdA	499500	2997	0.005201
Select	1000	OrdA	499500	2997	0.005203
Select	1000	OrdA	499500	2997	0.006150
Select	1000	OrdA	499500	2997	0.005179
Select	1000	OrdA	499500	2997	0.004896
Select	1000	OrdA	499500	2997	0.004917
Select	1000	OrdA	499500	2997	0.004914
Select	1000	OrdA	499500	2997	0.004916
Select	1000	OrdA	499500	2997	0.004897
Select	1000	OrdA	499500	2997	0.004918
Select	10000	OrdC	49995000	29997	0.561589
Select	10000	OrdD	49995000	29997	0.588332
Select	10000	OrdP	49995000	29997	0.569411
Select	10000	OrdA	49995000	29997	0.554237
Select	10000	OrdA	49995000	29997	0.597913
Select	10000	OrdA	49995000	29997	0.556050
Select	10000	OrdA	49995000	29997	0.568910
Select	10000	OrdA	49995000	29997	0.609064
Select	10000	OrdA	49995000	29997	0.569972
Select	10000	OrdA	49995000	29997	0.589663
Select	10000	OrdA	49995000	29997	0.579552
Select	10000	OrdA	49995000	29997	0.576273
Select	10000	OrdA	49995000	29997	0.603351
Select	100000	OrdC	4999950000	299997	58.300695
Select	100000	OrdD	4999950000	299997	59.014263
Select	100000	OrdP	4999950000	299997	58.884310
Select	100000	OrdA	4999950000	299997	58.460898
Select	100000	OrdA	4999950000	299997	58.640138
Select	100000	OrdA	4999950000	299997	61.302155
Select	100000	OrdA	4999950000	299997	60.813733
Select	100000	OrdA	4999950000	299997	60.557073
Select	100000	OrdA	4999950000	299997	60.472528
Select	100000	OrdA	4999950000	299997	61.323460
Select	100000	OrdA	4999950000	299997	61.482899
Select	100000	OrdA	4999950000	299997	61.434110
Select	100000	OrdA	4999950000	299997	61.445988

Arquivo C.4: Resultados do testes utilizando ShellSort.

Method	Quant	Type	mComp.	mMovim.	mTime
Shell	100	OrdC	342	684	0.000008
Shell	100	OrdD	572	914	0.000011
Shell	100	OrdP	509	851	0.000012
Shell	100	OrdA	818	1160	0.000019
Shell	100	OrdA	818	1160	0.000016
Shell	100	OrdA	818	1160	0.000016
Shell	100	OrdA	818	1160	0.000017
Shell	100	OrdA	818	1160	0.000016
Shell	100	OrdA	818	1160	0.000016
Shell	100	OrdA	818	1160	0.000017
Shell	100	OrdA	818	1160	0.000015
Shell	100	OrdA	818	1160	0.000016
Shell	100	OrdA	818	1160	0.000016
Shell	1000	OrdC	5457	10914	0.000108
Shell	1000	OrdD	9377	14834	0.000164
Shell	1000	OrdP	10857	16314	0.000236
Shell	1000	OrdA	14364	19821	0.000293
Shell	1000	OrdA	14364	19821	0.000302
Shell	1000	OrdA	14364	19821	0.000300
Shell	1000	OrdA	14364	19821	0.000291
Shell	1000	OrdA	14364	19821	0.000289
Shell	1000	OrdA	14364	19821	0.000290
Shell	1000	OrdA	14364	19821	0.000297
Shell	1000	OrdA	14364	19821	0.000297
Shell	1000	OrdA	14364	19821	0.000296
Shell	1000	OrdA	14364	19821	0.000291
Shell	10000	OrdC	75243	150486	0.001453
Shell	10000	OrdD	128947	204190	0.002120
Shell	10000	OrdP	172528	247771	0.003399
Shell	10000	OrdA	236735	311978	0.004624
Shell	10000	OrdA	236735	311978	0.004745
Shell	10000	OrdA	236735	311978	0.007572
Shell	10000	OrdA	236735	311978	0.004385
Shell	10000	OrdA	236735	311978	0.004452
Shell	10000	OrdA	236735	311978	0.004395
Shell	10000	OrdA	236735	311978	0.004638
Shell	10000	OrdA	236735	311978	0.007667
Shell	10000	OrdA	236735	311978	0.004455
Shell	10000	OrdA	236735	311978	0.004624
Shell	100000	OrdC	967146	1934292	0.018899
Shell	100000	OrdD	1586800	2553946	0.037005
Shell	100000	OrdP	2214898	3182044	0.055055
Shell	100000	OrdA	3702551	4669697	0.105380
Shell	100000	OrdA	3702551	4669697	0.100341
Shell	100000	OrdA	3702551	4669697	0.111791
Shell	100000	OrdA	3702551	4669697	0.104286
Shell	100000	OrdA	3720319	4687465	0.067776
Shell	100000	OrdA	3720319	4687465	0.075897
Shell	100000	OrdA	3720319	4687465	0.080078
Shell	100000	OrdA	3720319	4687465	0.077610
Shell	100000	OrdA	3720319	4687465	0.104269
Shell	100000	OrdA	3720319	4687465	0.097616

Arquivo C.5: Resultados do testes utilizando QuickSort.

Method	Quant	Type	mComp.	mMovim.	mTime
Quick	100	OrdC	606	189	0.000020
Quick	100	OrdD	610	336	0.000022
Quick	100	OrdP	727	324	0.000029
Quick	100	OrdA	997	570	0.000035
Quick	100	OrdA	997	570	0.000033
Quick	100	OrdA	997	570	0.000032
Quick	100	OrdA	997	570	0.000032
Quick	100	OrdA	997	570	0.000032
Quick	100	OrdA	997	570	0.000032
Quick	100	OrdA	997	570	0.000031
Quick	100	OrdA	997	570	0.000031
Quick	100	OrdA	997	570	0.000032
Quick	100	OrdA	997	570	0.000032
Quick	1000	OrdC	9009	1533	0.000191
Quick	1000	OrdD	9016	3030	0.000197
Quick	1000	OrdP	10725	3522	0.000309
Quick	1000	OrdA	12852	8136	0.000409
Quick	1000	OrdA	12852	8136	0.000398
Quick	1000	OrdA	12852	8136	0.000404
Quick	1000	OrdA	12852	8136	0.000397
Quick	1000	OrdA	12852	8136	0.000403
Quick	1000	OrdA	12852	8136	0.000396
Quick	1000	OrdA	12852	8136	0.000404
Quick	1000	OrdA	12852	8136	0.000396
Quick	1000	OrdA	12852	8136	0.000404
Quick	1000	OrdA	12852	8136	0.000400
Quick	10000	OrdC	125439	17712	0.002403
Quick	10000	OrdD	125452	32712	0.002531
Quick	10000	OrdP	157689	66225	0.003792
Quick	10000	OrdA	181203	103575	0.004968
Quick	10000	OrdA	181203	103575	0.004840
Quick	10000	OrdA	181203	103575	0.004932
Quick	10000	OrdA	181203	103575	0.004946
Quick	10000	OrdA	181203	103575	0.004929
Quick	10000	OrdA	181203	103575	0.004967
Quick	10000	OrdA	181203	103575	0.005176
Quick	10000	OrdA	181203	103575	0.004931
Quick	10000	OrdA	181203	103575	0.004929
Quick	10000	OrdA	181203	103575	0.004987
Quick	100000	OrdC	1600016	196605	0.028667
Quick	100000	OrdD	1600030	346602	0.029437
Quick	100000	OrdP	2251124	903300	0.045093
Quick	100000	OrdA	2137184	1311831	0.081911
Quick	100000	OrdA	2137184	1311831	0.069456
Quick	100000	OrdA	2114943	1310586	0.054217
Quick	100000	OrdA	2114943	1310586	0.055759
Quick	100000	OrdA	2114943	1310586	0.063946
Quick	100000	OrdA	2114943	1310586	0.062869
Quick	100000	OrdA	2114943	1310586	0.084427
Quick	100000	OrdA	2114943	1310586	0.097758
Quick	100000	OrdA	2114943	1310586	0.098938
Quick	100000	OrdA	2114943	1310586	0.090165

Arquivo C.6: Resultados do testes utilizando HeapSort.

Method	Quant	Type	mComp.	mMovim.	mTime
Heap	100	OrdC	1265	884	0.000033
Heap	100	OrdD	1125	747	0.000032
Heap	100	OrdP	1254	868	0.000035
Heap	100	OrdA	1205	817	0.000034
Heap	100	OrdA	1205	817	0.000035
Heap	100	OrdA	1205	817	0.000034
Heap	100	OrdA	1205	817	0.000034
Heap	100	OrdA	1205	817	0.000034
Heap	100	OrdA	1205	817	0.000033
Heap	100	OrdA	1205	817	0.000035
Heap	100	OrdA	1205	817	0.000034
Heap	100	OrdA	1205	817	0.000034
Heap	100	OrdA	1205	817	0.000034
Heap	1000	OrdC	19562	12192	0.000412
Heap	1000	OrdD	17952	10811	0.000381
Heap	1000	OrdP	19525	12190	0.000405
Heap	1000	OrdA	18837	11556	0.000422
Heap	1000	OrdA	18837	11556	0.000420
Heap	1000	OrdA	18837	11556	0.000420
Heap	1000	OrdA	18837	11556	0.000420
Heap	1000	OrdA	18837	11556	0.000423
Heap	1000	OrdA	18837	11556	0.000426
Heap	1000	OrdA	18837	11556	0.000419
Heap	1000	OrdA	18837	11556	0.000419
Heap	1000	OrdA	18837	11556	0.000419
Heap	1000	OrdA	18837	11556	0.000419
Heap	10000	OrdC	264433	156928	0.008299
Heap	10000	OrdD	246705	141975	0.007504
Heap	10000	OrdP	264737	156489	0.004734
Heap	10000	OrdA	255288	149150	0.005324
Heap	10000	OrdA	255288	149150	0.005187
Heap	10000	OrdA	255288	149150	0.008293
Heap	10000	OrdA	255288	149150	0.008269
Heap	10000	OrdA	255288	149150	0.009227
Heap	10000	OrdA	255288	149150	0.011361
Heap	10000	OrdA	255288	149150	0.009188
Heap	10000	OrdA	255288	149150	0.010626
Heap	10000	OrdA	255288	149150	0.010120
Heap	10000	OrdA	255288	149150	0.010692
Heap	100000	OrdC	3312482	1900842	0.083919
Heap	100000	OrdD	3131748	1747201	0.084207
Heap	100000	OrdP	3294503	1887739	0.088935
Heap	100000	OrdA	3220006	1825075	0.079565
Heap	100000	OrdA	3220006	1825075	0.099474
Heap	100000	OrdA	3220006	1825075	0.102175
Heap	100000	OrdA	3220006	1825075	0.087964
Heap	100000	OrdA	3220006	1825075	0.072352
Heap	100000	OrdA	3220006	1825075	0.077337
Heap	100000	OrdA	3220006	1825075	0.073631
Heap	100000	OrdA	3220006	1825075	0.089448
Heap	100000	OrdA	3220006	1825075	0.090158
Heap	100000	OrdA	3220006	1825075	0.070367

Arquivo C.7: Resultados do testes utilizando MergeSort.

Method	Quant	Type	MComp.	MMovim.	MTime
Merge	100	OrdC	372	1376	0.000158
Merge	100	OrdD	316	1376	0.000141
Merge	100	OrdP	505	1376	0.000143
Merge	100	OrdA	558	1376	0.000148
Merge	100	OrdA	558	1376	0.000147
Merge	100	OrdA	558	1376	0.000147
Merge	100	OrdA	558	1376	0.000147
Merge	100	OrdA	558	1376	0.000151
Merge	100	OrdA	558	1376	0.000150
Merge	100	OrdA	558	1376	0.000150
Merge	100	OrdA	558	1376	0.000150
Merge	100	OrdA	558	1376	0.000150
Merge	1000	OrdC	5052	19968	0.001487
Merge	1000	OrdD	4932	19968	0.001495
Merge	1000	OrdP	7869	19968	0.001518
Merge	1000	OrdA	8744	19968	0.001598
Merge	1000	OrdA	8744	19968	0.001597
Merge	1000	OrdA	8744	19968	0.001601
Merge	1000	OrdA	8744	19968	0.001598
Merge	1000	OrdA	8744	19968	0.001690
Merge	1000	OrdA	8744	19968	0.001609
Merge	1000	OrdA	8744	19968	0.001605
Merge	1000	OrdA	8744	19968	0.001601
Merge	1000	OrdA	8744	19968	0.001603
Merge	1000	OrdA	8744	19968	0.001599
Merge	10000	OrdC	71712	272640	0.015855
Merge	10000	OrdD	64608	272640	0.016313
Merge	10000	OrdP	114410	272640	0.023147
Merge	10000	OrdA	123685	272640	0.029013
Merge	10000	OrdA	123685	272640	0.017560
Merge	10000	OrdA	123685	272640	0.017046
Merge	10000	OrdA	123685	272640	0.017231
Merge	10000	OrdA	123685	272640	0.017340
Merge	10000	OrdA	123685	272640	0.024817
Merge	10000	OrdA	123685	272640	0.016881
Merge	10000	OrdA	123685	272640	0.018917
Merge	10000	OrdA	123685	272640	0.017271
Merge	10000	OrdA	123685	272640	0.019384
Merge	100000	OrdC	877968	3385984	0.205009
Merge	100000	OrdD	815024	3385984	0.219894
Merge	100000	OrdP	1106062	3385984	0.221169
Merge	100000	OrdA	1566362	3385984	0.265624
Merge	100000	OrdA	1566362	3385984	0.243212
Merge	100000	OrdA	1566362	3385984	0.227766
Merge	100000	OrdA	1566362	3385984	0.233699
Merge	100000	OrdA	1566330	3385984	0.203184
Merge	100000	OrdA	1566330	3385984	0.253517
Merge	100000	OrdA	1566330	3385984	0.258494
Merge	100000	OrdA	1566330	3385984	0.265600
Merge	100000	OrdA	1566749	3385984	0.231564
Merge	100000	OrdA	1566749	3385984	0.250193

Referências Bibliográficas

[1] Ricardo Annes. Ordenação por seleção, 2008. http://pucrs.campus2.br/~annes/alg3_ord_sel.

[2] José Elias C. Arroyo. Inf111 - programação ii, aulas 11, 12, 13 - ordenação, 2007. www.dpi.ufv.br/disciplinas/inf111/2006_2/Ordenacao1.pdf.

[3] Julio Cesar de Andrade Vieira Lopes. Heap, 2005. www.abusar.org/ftp/pub/pitanga/Heap.pdf.

[4] Thales Castelo Branco de Castro. Métodos de ordenação, 2008. http://www.nuperc.unifacs.br/Members/thales.castro/arquivos/aulas/Ordenacao%20Interna.ppt.

[5] Alvaro Borges de Oliveira. *Métodos de Ordenação Interna*. Visual Book, São Paulo, 1st edition, 2002.

[6] Paulo Feofiloff. Projeto de algoritmos: Quicksort, 1998. http://www.ime.usp.br/~pf/algoritmos/aulas/quick.html.

[7] David Menotti Gomes. Ordenação, 2008. http://www.decom.ufop.br/prof/menotti/aedI/slides/aula14-Ordenacao.ppt.

[8] David Menotti Gomes. Ordenação - heapsort, 2008. http://www.decom.ufop.br/prof/menotti/aedI/slides/aula17-HeapSort.ppt.

[9] David Menotti Gomes. Ordenação - mergesort, 2008. http://www.decom.ufop.br/prof/menotti/aedI/slides/aula18-MergeSort.ppt.

[10] David Menotti Gomes. Ordenação - quicksort, 2008. http://www.decom.ufop.br/prof/menotti/aedI/slides/aula16-QuickSort.ppt.

[11] David Menotti Gomes. Ordenação - shellsort, 2008. http://www.decom.ufop.br/prof/menotti/aedI/slides/aula15-ShellSort.ppt.

[12] Michael Lamont. Shell sort, 2008. http://linux.wku.edu/~lamonml/algor/sort/shell.html.

[13] H. W. Lang. Insertion sort, 2008. http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/insert/insertionen.htm.

[14] Ronaldo S. Mello. Ordenação de dados(iii), 2002. www.dcc.ufam.edu.br/~alf/MATERIAL%20AED2/HEAPSORT.pdf.

[15] Marcus Ritt. Algoritmos e complexidade - notas de aula, 2007. www.inf.ufrgs.br/~buriol/cursos/complexidade07/Aula-cap5-n1.pdf.

[16] Rosália Rodrigues. Métodos de programação, 2002. http://www2.mat.ua.pt/rosalia/arquivo/MP/acetatos9.pdf.

[17] Robert Sedgewick. *Algorithms*. Addison-Wesley, Massachussets, 2st edition, 1989.

[18] Hamid Reza Shahbazkia. Quicksort. http://w3.ualg.pt/~hshah/ped/Aula%2014/Quick_final.html.

[19] Nivio Ziviani. *Projeto de Algoritmos com implementação em C++ e Java*. THOMSON Learning, São Paulo, 1st edition, 2007.