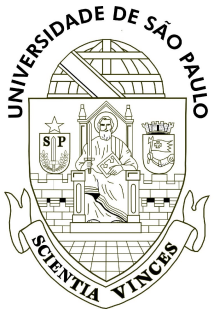

Universidade de São Paulo

ICMC - Instituto de Ciências Matemáticas e de Computação
Mestrado Profissional em Matemática, Estatística e Computação
Aplicadas à Indústria

Relatório de Pesquisa

Análise de Algoritmos de Ordenação

Robson Fernandes

Introdução à Ciência da Computação
Prof. Dr. Adenilso Simão

São Carlos/SP

2017

Análise de Algoritmos de Ordenação

Robson Fernandes
10107495

Relatório de pesquisa apresentado
como requisito parcial para
aprovação na disciplina
MAI5001-Introdução à Ciência da
Computação, Instituto de Ciências
Matemáticas e de Computação,
Universidade de São Paulo.

Prof. Dr. Adenilso Simão

São Carlos/SP

2017

Resumo

Ordenação é o ato de se colocar os elementos de uma sequência de informações, ou dados, em uma relação de ordem predefinida. Algoritmos de ordenação em ciência da computação é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem. As ordens mais usadas são a numérica e a lexicográfica. Existem várias razões para se ordenar uma sequência. Uma delas é a possibilidade se acessar seus dados de modo mais eficiente. Este trabalho visa analisar os principais algoritmos de ordenação estudados em ciência da computação, compreendendo seus detalhes, implementação, complexidade de tempo e espaço, e análise de desempenho.

Palavras-chave:
Algoritmos de Ordenação; Complexidade de Algoritmos; Linguagens de Programação;

Abstract

Ordering is done from elements of a sequence of information or data, in a predefined order relationship. Algorithms of ordering in computer science are algorithms that place the elements of a given sequence in a given order. As most commonly used orders are a number and a lexicographic. There are several reasons to order a sequence. One is a possibility to access your data more efficiently. This work aims to analyze the main ordering algorithms studied in computation, including its details, implementation, time and space complexity and performance analysis.

Key-words:
Sorting Algorithms; Complexity of Algorithms;Programming languages;

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

Os algoritmos de ordenação ou classificação consituem uma classe de algoritmos extremamente estudada e muito popular. Seu estudo é parte vital de praticamente todo curso na área da Computação e tem aplicações práticas no dia-a-dia. Em diversas aplicações a ordenação é uma das etapas, dentre muitas outras a serem efetuadas, de modo que selecionar o melhor algoritmo de ordenação é extremamente importante nestes casos. Em busca em índices previamente ordenados, por exemplo, é a maneira mais eficiente de busca por um elemento qualquer de uma coleção. Uma sequência previamente ordenada tem aplicação prática na resolução dos seguintes problemas: busca, par mais próximo (dado uma sequência de números), unicidade de elementos, distribuição de frequência, convex hull, etc.[9]

2 Algoritmos de Ordenação

2.1 Algoritmo BubbleSort

2.1.1 Introdução

O *BubbleSort* é conhecido pela sua simplicidade e pela eficácia ao realizar ordenações em um número limitado de valores. Seu princípio baseia-se na troca de valores entre posições consecutivas, fazendo com que valores altos ou baixos (dependendo da forma de ordenação desejada) "borbulhem" para o final da fila, por isso este algoritmo é chamado de *BubbleSort*.

Neste algoritmo cada elemento da posição i será comparado com o elemento da posição $i + 1$, ou seja, um elemento da posição 2 será comparado com o elemento da posição 3. Caso o elemento da posição 2 for maior que o da posição 3, eles trocam de lugar *(swap)* e assim sucessivamente. Por causa dessa forma de execução, o vetor terá que ser percorrido quantas vezes que for necessária, tornando o algoritmo ineficiente para listas muito grandes.[1]

Na Figura 2.1 temos o processo de ordenação do algoritmo *BubbleSort*.[1]

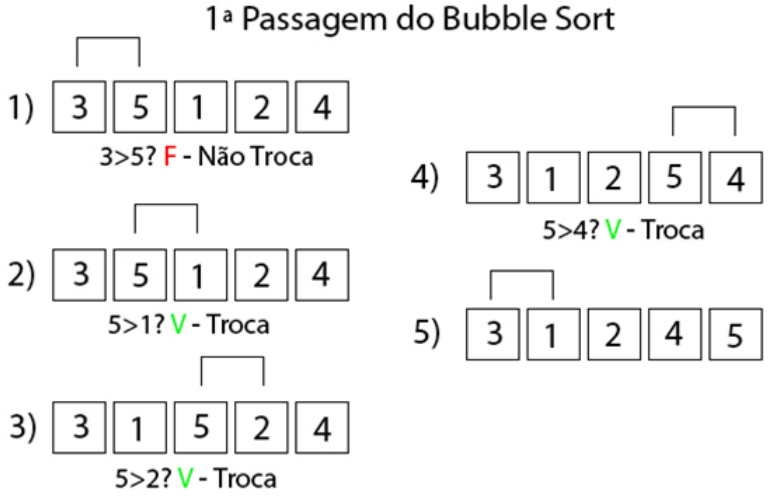

Figure 2.1: *Fluxo do Algoritmo BubbleSort*

2.1.1.1 Fluxo

1. É verificado se o 3 é maior que 5, por essa condição ser falsa, não há troca.
2. É verificado se o 5 é maior que 1, por essa condição ser verdadeira, há uma troca.
3. É verificado se o 5 é maior que 2, por essa condição ser verdadeira, há uma troca.
4. É verificado se o 5 é maior que 4, por essa condição ser verdadeira, há uma troca.

O método retorna ao início do vetor realizando os mesmos processos de comparações, isso é feito até que o vetor esteja ordenado.[1]

2.1.2 Implementação em Java

Na tabela 2.1 temos a implementação do Algoritmo *BubbleSort* em Java.[8]

Table 2.1: *Algoritmo BubbleSort - Implementação em Java*

```
1  public int[] bubbleSort(int[] vetor){
2
3          boolean troca = true;
4          int aux;
5          while (troca) {
6                  troca = false;
7                  for (int i = 0; i < vetor.length - 1; i++) {
8                          if (vetor[i] > vetor[i + 1]) {
9                                  aux = vetor[i];
10                                 vetor[i] = vetor[i + 1];
11                                 vetor[i + 1] = aux;
12                                 troca = true;
13                         }
14                 }
15         }
16         return vetor;
17 }
```

2.1.3 Complexidade de Tempo e Espaço

2.1.3.1 Tempo

1. Complexidade no pior Caso $\Theta(n^2)$;
2. Complexidade no caso médio $\Theta(n^2)$;
3. Complexidade no melhor caso $\Theta(n)$;

2.1.3.2 Espaço

1. Complexidade de espaço no pior caso é $\Theta(1)$;

2.1.3.3 Estabilidade

1. Algoritmo Estável

2.1.4 Análise de Desempenho

O algoritmo *BubbleSort* é indicado em casos que envolvam tabelas muito pequenas, quando se sabe que a tabela está quase ordenada. A desvantagem deste algoritmo é sua lentidão, de modo geral, ele considerado o mais ineficiente algoritmo de ordenação.

2.2 Algoritmo InsertionSort

2.2.1 Introdução

A ordenação por *InsertionSort* é quase tão simples quanto o algoritmo de ordenação por *Selection*, e além disso é um método estável, pois deixa os registros com chaves iguais na mesma posição relativa.

O método consiste em cada passo, a partir de *i=2*, o *i-ésimo* item da sequência fonte é apanhado e transferido para a sequência destino, sendo colocado na posição correta. A inserção do elemento no lugar de destino é efetuado movendo-se os itens com chave maiores para a direita e então inserindo-o na posição que ficou vazia.

Neste processo de alternar comparações e movimentação de registros existem duas situações que podem causar o término do processo, a primeira é quando um item com chave menor que o item em consideração é encontrado e a segunda situação é quando o final da sequência destino é atingido (à esquerda).

A melhor solução para a situação de um anel com duas condições de terminação é a utilização de uma sentinela, para isto, na posição zero do vetor colocamos o próprio registro analisado.[7]

Na Figura 2.2 temos o processo de ordenação do algoritmo *InsertionSort*.[6]

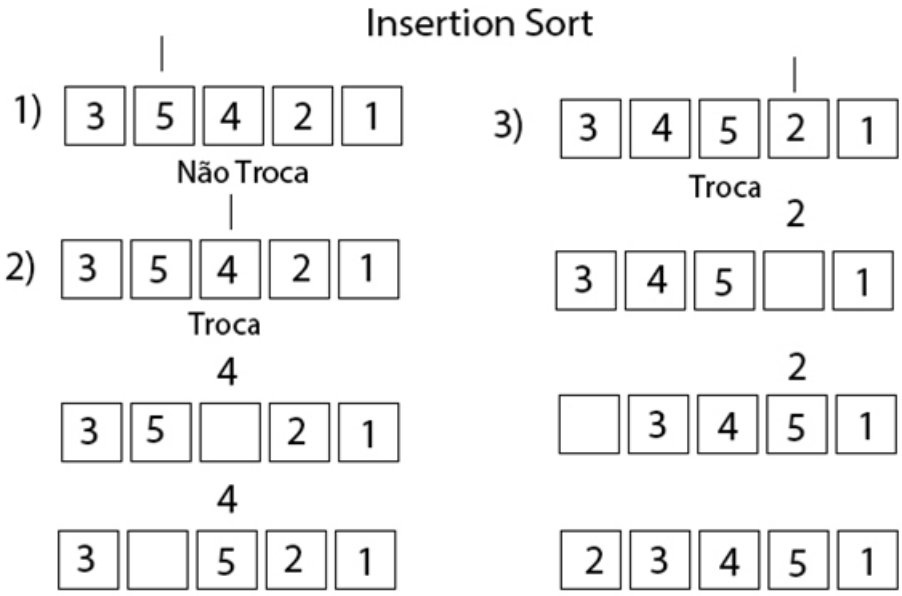

Figure 2.2: *Fluxo do Algoritmo InsertionSort*

2.2.1.1 Fluxo

1. Neste passo é verificado se o 5 é menor que o 3, como essa condição é falsa, então não há troca.

2. É verificado se o quatro é menor que o 5 e o 3, ele só é menor que o 5, então os dois trocam de posição.

3. É verificado se o 2 é menor que o 5, 4 e o 3, como ele é menor que 3, então o 5 passa a ocupar a posição do 2, o 4 ocupa a posição do 5 e o 3 ocupa a posição do 4, assim a posição do 3 fica vazia e o 2 passa para essa posição.

O mesmo processo de comparação acontece com o número 1, após esse processo o vetor fica ordenado.[1]

2.2.2 Implementação em Java

Na tabela 2.2 temos a implementação do Algoritmo *InsertionSort* em Java.[8]

Table 2.2: *Algoritmo InsertionSort - Implementação em Java*

```
1  public int[] insertionSort(int[] vetor){
2
3          for (int i = 1; i < vetor.length; i++){
4                  int key = vetor[i];
5                  int j = i;
6                  while ((j > 0) && (vetor[j-1] > key)){
7                          vetor[j] = vetor[j-1];
8                          j -= 1;
9                  }
10                 vetor[j] = key;
11         }
12         return vetor;
13 }
```

2.2.3 Complexidade de Tempo e Espaço

2.2.3.1 Tempo

1. Complexidade no pior Caso $\Theta(n^2)$;
2. Complexidade no caso médio $\Theta(n^2)$;
3. Complexidade no melhor caso $\Theta(n)$;

2.2.3.2 Espaço

1. Complexidade de espaço no pior caso é $\Theta(n)$;

2.2.3.3 Estabilidade

1. Algoritmo Estável

2.2.4 Análise de Desempenho

Este tipo de ordenação é muito eficiente em listas encadeadas, principalmente se estas já estiverem parcialmente ordenadas. É um bom método quando se deseja adicionar poucos itens a um arquivo ordenado, pois o custo é linear.

2.3 Algoritmo SelectionSort

2.3.1 Introdução

A ordenação por *SelectionSort* consiste, em cada etapa, em selecionar o maior (ou o menor) elemento e alocá-lo em sua posição correta dentro da futura lista ordenada. Durante a execução, a lista com n registros é decomposta em duas sublistas, uma contendo os itens já ordenados e a outra com os elementos ainda não ordenados. No início, a sublista

ordenada está vazia e a desordenada contém todos os elementos, no final do processo a sublista ordenada apresentará *(n-1)* elementos e a desordenada terá um elemento.[7]

Na Figura 2.3 temos o processo de ordenação do algoritmo *SelectionSort*.[1]

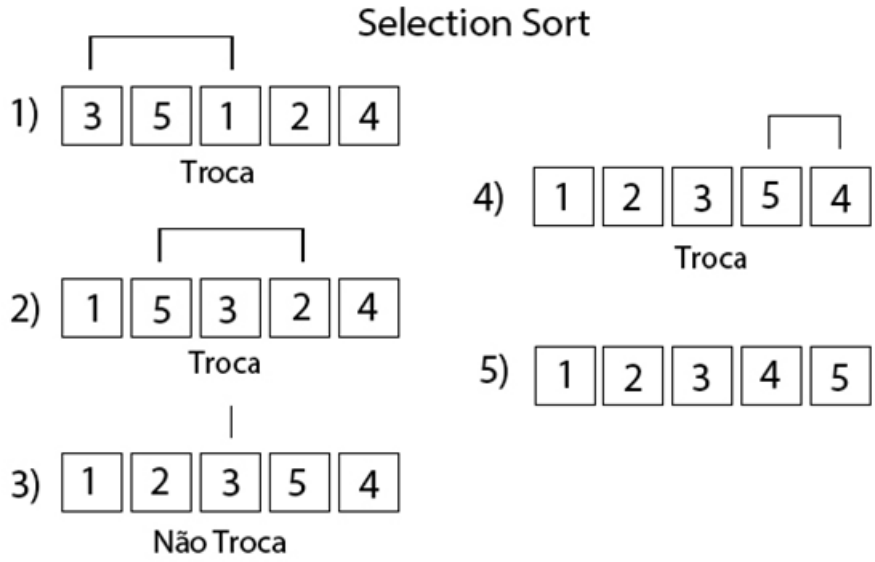

Figure 2.3: *Fluxo do Algoritmo SelectionSort*

2.3.1.1 Fluxo

1. Neste passo o primeiro número escolhido foi o 3, ele foi comparado com todos os números à sua direita e o menor número encontrado foi o 1, então os dois trocam de lugar.

2. O mesmo processo do passo 1 acontece, o número escolhido foi o 5 e o menor número encontrado foi o 2.

3. Não foi encontrado nenhum número menor que 3, então ele fica na mesma posição.

4. O número 5 foi escolhido novamente e o único número menor que ele à sua direita é o 4, então eles trocam.

5. Vetor já ordenado.

2.3.2 Implementação em Java

Na tabela 2.3 temos a implementação do Algoritmo *SelectionSort* em Java.[8]

Table 2.3: *Algoritmo SelectionSort - Implementação em Java*

```
1  public int[] selectionSort(int[] vetor){
2
3     for (int fixo = 0; fixo < vetor.length - 1; fixo++) {
4       int menor = fixo;
5
6       for (int i = menor + 1; i < vetor.length; i++) {
7          if (vetor[i] < vetor[menor]) {
8             menor = i;
9          }
10      }
11      if (menor != fixo) {
12        int t = vetor[fixo];
13        vetor[fixo] = vetor[menor];
14        vetor[menor] = t;
15      }
16    }
17    return vetor;
18 }
```

2.3.3 Complexidade de Tempo e Espaço

2.3.3.1 Tempo

1. Complexidade no pior Caso $\Theta(n^2)$;
2. Complexidade no caso médio $\Theta(n^2)$;
3. Complexidade no melhor caso $\Theta(n^2)$;

2.3.3.2 Espaço

1. Complexidade de espaço no pior caso é $\Theta(n)$;

2.3.3.3 Estabilidade

1. Algoritmo Não Estável

2.3.4 Análise de Desempenho

É o algoritmo a ser utilizado para arquivos com registros muito grandes (alto custo de movimentação), e também é interessante para listas pequenas. Custo linear no tamanho da entrada para o número de movimentos de registros. O *SelectionSort* faz poucas mudanças entre os elementos, o que o faz um algoritmo bem rápido. Sua performance é influenciada pela ordenação inicial do vetor.

2.4 Algoritmo QuickSort

2.4.1 Introdução

É um método de ordenação por troca. Entretanto, enquanto o *BubbleSort* (bolha) troca pares de elementos consecutivos, o *QuickSort* compara pares de elementos distantes, o que acelera o processo de ordenação.

É um algoritmo de troca do tipo divisão e conquista que resolve um determinado problema dividindo-o em três subproblemas, tais que:

1. o primeiro subproblema é composto dos elementos menores ou iguais ao pivô;
2. o segundo subproblema é o próprio pivô;
3. o terceiro subproblema são os elementos maiores ou iguais ao pivô;

Na Figura 2.4 temos o processo de ordenação do algoritmo *QuickSort*.[1]

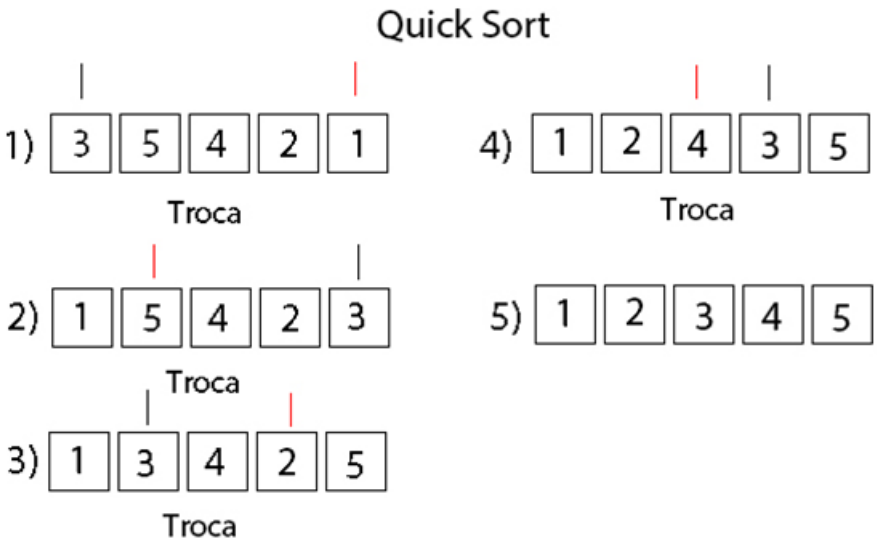

Figure 2.4: *Fluxo do Algoritmo QuickSort*

2.4.1.1 Fluxo

1. O número 3 foi escolhido como pivô, nesse passo é procurado à sua direita um número menor que ele para ser passado para a sua esquerda. O primeiro número menor encontrado foi o 1, então eles trocam de lugar.
2. Agora é procurado um número à sua esquerda que seja maior que ele, o primeiro número maior encontrado foi o 5, portanto eles trocam de lugar.
3. O mesmo processo do passo 1 acontece, o número 2 foi o menor número encontrado, eles trocam de lugar.

4. O mesmo processo do passo 2 acontece, o número 4 é o maior número encontrado, eles trocam de lugar.

O vetor desse exemplo é um vetor pequeno, portanto ele já foi ordenado, mas se fosse um vetor grande, ele seria dividido e recursivamente aconteceria o mesmo processo de escolha de um pivô e comparações.[1]

2.4.2 Implementação em Java

Na tabela 2.4 temos a implementação do Algoritmo *QuickSort* em Java.[8]

Table 2.4: *Algoritmo QuickSort - Implementação em Java*

```
1  public class QuickSort
2  {
3      public int[] sort(int[] vetor, int inicio, int fim) {
4              if (inicio < fim) {
5                      int posicaoPivo = separar(vetor, inicio,
                           fim);
6                      sort(vetor, inicio, posicaoPivo - 1);
7                      sort(vetor, posicaoPivo + 1, fim);
8              }
9              return vetor;
10     }
11
12     private int separar(int[] vetor, int inicio, int fim) {
13             int pivo = vetor[inicio];
14             int i = inicio + 1, f = fim;
15             while (i <= f) {
16                     if (vetor[i] <= pivo)
17                             i++;
18                     else if (pivo < vetor[f])
19                             f--;
20                     else {
21                             int troca = vetor[i];
22                             vetor[i] = vetor[f];
23                             vetor[f] = troca;
24                             i++;
25                             f--;
26                     }
27             }
28             vetor[inicio] = vetor[f];
29             vetor[f] = pivo;
30             return f;
31     }
32 }
```

2.4.3 Complexidade de Tempo e Espaço

2.4.3.1 Tempo

1. Complexidade no pior Caso $\Theta(n^2)$;
2. Complexidade no caso médio $\Theta(n(Log(n)))$;
3. Complexidade no melhor caso $\Theta(n(Log(n)))$;

2.4.3.2 Espaço

1. Complexidade de espaço no pior caso é $\Theta(n)$;

2.4.3.3 Estabilidade

1. Algoritmo Não Estável

2.4.4 Análise de Desempenho

O *Quicksort* é extremamente eficiente para ordenar arquivos de dados. O método necessita de apenas uma pequena pilha como memória auxiliar, e requer cerca de $n(Log(n))$ operações em média para ordenar n itens.

Como aspectos negativos tem-se:

1. A versão recursiva do algoritmo tem um pior caso que é $\Theta(n^2)$
2. A implementação do algoritmo é muito delicada e difícil;
3. O método não é estável;

Uma vez desenvolvida uma implementação robusta para o *quicksort*, este deve ser o algoritmo preferido para a maioria das aplicações. Ele é bem eficiente na manipulação de conjuntos com muitos elementos.

2.5 Algoritmo HeapSort

2.5.1 Introdução

É um método de ordenação desenvolvido em 1964 por John W. J. Williams para melhorar o desempenho do método de ordenação por seleção, que é ineficiente (O(n2) comparações) mas que tem como vantagem fazer poucas movimentações.

O *HeapSort* utiliza um *heap* binário para manter o próximo elemento a ser selecionado – *heap* binário: árvore binária mantida na forma de vetor – o *heap* é gerado e mantido no próprio vetor a ser ordenado (no segmento não-ordenado)[3]

Na Figura 2.5 temos o processo de ordenação do algoritmo *HeapSort*.

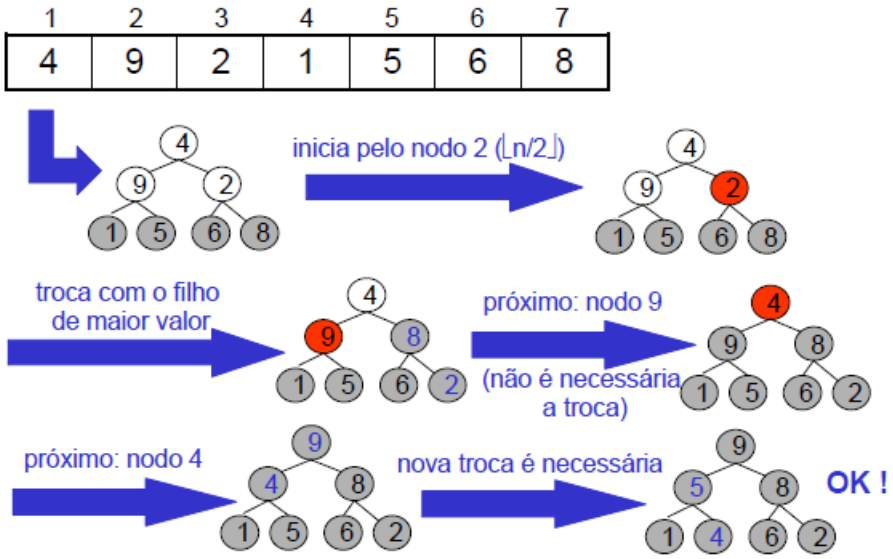

Figure 2.5: *Fluxo do Algoritmo HeapSort*

2.5.1.1 Fluxo

1. Transformação do vetor em um *heap* binário máximo (Construção do Heap)
2. Ordenação – a cada iteração seleciona-se o maior elemento (na raiz do heap) e o adiciona no início de um segmento ordenado – após cada seleção de elemento, o *heap* deve ser reorganizado para continuar sendo um *heap* binário máximo.[3]

2.5.2 Implementação em Java

Na tabela 2.5 temos a implementação do Algoritmo *HeapSort* em Java.[4]

Table 2.5: *Algoritmo HeapSort - Implementação em Java*

```java
public class HeapSort
{
    public int[] sort(int arr[])
    {
        int n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        for (int i=n-1; i>=0; i--)
        {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            heapify(arr, i, 0);
        }
        return arr;
    }

    private void heapify(int arr[], int n, int i)
    {
        int largest = i;
        int l = 2*i + 1;
        int r = 2*i + 2;

        if (l < n && arr[l] > arr[largest])
            largest = l;

        if (r < n && arr[r] > arr[largest])
            largest = r;

        if (largest != i)
        {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;
            heapify(arr, n, largest);
        }
    }
}
```

2.5.3 Complexidade de Tempo e Espaço

2.5.3.1 Tempo

1. Complexidade no pior Caso $\Theta(n(Log(n)))$

2. Complexidade no caso médio $\Theta(n(Log(n)))$

3. Complexidade no melhor caso $\Theta(n(Log(n)))$

2.5.3.2 Espaço

1. Complexidade de espaço no pior caso é $\Theta(n)$

2.5.3.3 Estabilidade

1. Algoritmo Não Estável

2.5.4 Análise de Desempenho

HeapSort não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o heap, bem como porque o anel interno do algoritmo é bastante complexo, se comparado com o anel interno do *QuickSort*. No entanto, possui um desempenho em tempo de execução muito bom em dados ordenados aleatoriamente, além do uso de memória equilibrado.

O *QuickSort* é, em média, duas vezes mais rápido que o *HeapSort*. Entretanto, o *HeapSort* é melhor que o *ShellSort* para grandes arquivos.

Deve-se observar que o comportamento do *HeapSort* é $\Theta(n(Log(n)))$ qualquer que seja a entrada. Aplicações que não podem tolerar um caso desfavorável devem usar o *HeapSort*.

2.6 Algoritmo MergeSort

2.6.1 Introdução

O *MergeSort*, ou ordenação por mistura, é um exemplo de algoritmo de ordenação por comparação do tipo dividir-para-conquistar.

Sua idéia 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.[2]

Na Figura 2.6 temos o processo de ordenação do algoritmo *MergeSort*.

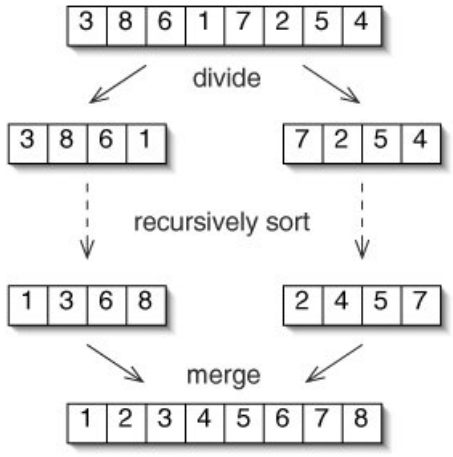

Figure 2.6: *Fluxo do Algoritmo MergeSort*

2.6.1.1 Fluxo

Com uma seqüência de entrada S com n elementos consiste de três passos:[5]

1. Divide: dividir S em duas sequencias S_1 and S_2 de aproximadamente $n/2$ elementos cada.
2. Recursão: recursivamente ordene S_1 e S_2
3. Conquista: junte S_1 e S_2 em uma única sequência ordenada.

2.6.2 Implementação em Java

Na tabela 2.6 temos a implementação do Algoritmo *MergeSort* em Java.[10]

Table 2.6: *Algoritmo MergeSort - Implementação em Java*

```
1  public class MergeSort {
2
3      private int[] numbers;
4      private int[] helper;
5      private int number;
6
7      public void sort(int[] values) {
8          this.numbers = values;
9          number = values.length;
10         this.helper = new int[number];
11         mergesort(0, number - 1);
12     }
13
14     private void mergesort(int low, int high) {
15         if (low < high) {
16                 int middle = low + (high - low) / 2;
17                 mergesort(low, middle);
18                 mergesort(middle + 1, high);
19                 merge(low, middle, high);
20         }
21     }
22
23     private void merge(int low, int middle, int high) {
24
25         for (int i = low; i <= high; i++) {
26             helper[i] = numbers[i];
27         }
28
29         int i = low;
30         int j = middle + 1;
31         int k = low;
32         while (i <= middle && j <= high) {
33             if (helper[i] <= helper[j]) {
34                     numbers[k] = helper[i];
35                     i++;
36             } else {
37                     numbers[k] = helper[j];
38                     j++;
39             }
40             k++;
41         }
42
43         while (i <= middle) {
44             numbers[k] = helper[i];
45             k++;
46             i++;
47         }
48
49     }
50 }
```

2.6.3 Complexidade de Tempo e Espaço

2.6.3.1 Tempo

1. Complexidade no pior Caso $\Theta(n(Log(n)))$
2. Complexidade no caso médio $\Theta(n(Log(n)))$
3. Complexidade no melhor caso $\Theta(n(Log(n)))$

2.6.3.2 Espaço

1. Complexidade de espaço no pior caso é $\Theta(n(Log(n)))$

2.6.3.3 Estabilidade

1. Algoritmo Estável

2.6.4 Análise de Desempenho

A vantagem do *MergeSort* é que este algoritmo é altamente paralelizável, ou seja, se existirem vários processadores disponíveis, a estratégia propiciará eficiência. Uma desvantagens é o alto consumo de memória, devido à série de chamadas recursivas.

Bibliografia

[1] B. de Almeida Honorato, *Algoritmos de ordenação: análise e comparação*, http://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261 (2016). [Online; accessed 03-May-2017].

[2] P. Drake, *Data structures and algorithms in Java*, Prentice-Hall, Inc., 2005.

[3] F. Ferreira, *Algoritmos de Ordenação*, https://codigoecafe.com/2011/01/29/algoritmos-de-ordenacao-2/ (2011). [Online; accessed 03-May-2017].

[4] GeeksforGeeks, *Heap Sort*, http://quiz.geeksforgeeks.org/heap-sort/ (2014). [Online; accessed 03-May-2017].

[5] A.R. Gonçalves, L.G.A. dos Santos, P.R. Silla, and L.B. Ruiz, *Algoritmos de ordenaçao* .

[6] T.H. Gormen, C.E. Leiserson, R.L. Rivest, C. Stein, *et al.*, *Introduction to algorithms*, MIT Press 44 (1990), pp. 97–138.

[7] W.M.P. Júnior, *Análise de algoritmos* .

[8] R. Lanhellas, *Algoritmos de Ordenação em Java*, http://www.devmedia.com.br/algoritmos-de-ordenacao-em-java/32693 (2015). [Online; accessed 03-May-2017].

[9] S.S. Skiena, *the algorithm design manual (2008)*.

[10] Vogella, *Merge Sort*, http://www.vogella.com/tutorials/JavaAlgorithmsMergesort/article.html (2009). [Online; accessed 03-May-2017].