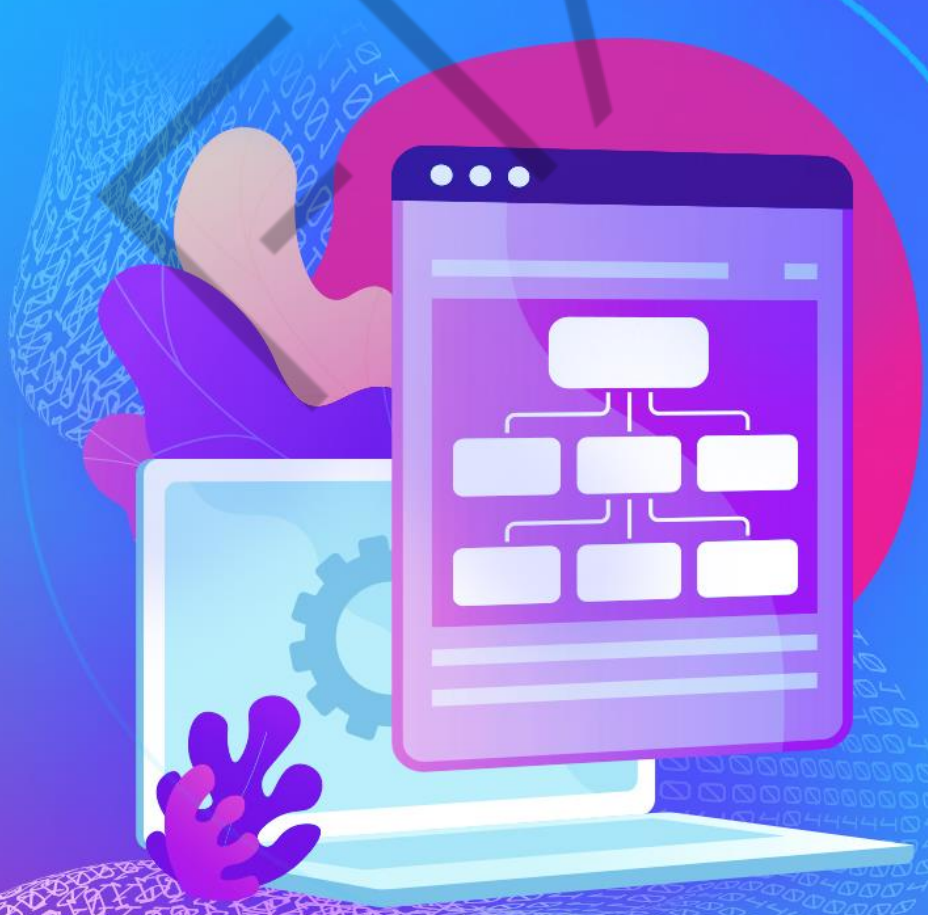
FIAP ON

CÓDIGOS DE ALTA PERFORMANCE

# ALGORITMOS DE ORDENAÇÃO

PATRICIA MAGNA



7

## LISTA DE FIGURAS

## LISTA DE CÓDIGOS-FONTE

**SUMÁRIO**

# 7 ALGORITMOS DE ORDENAÇÃO

## 7.1 Introdução

Toda vez que procuramos alguma informação nos nossos diretórios e demoramos muito até encontrar pensamos: “por que não organizei?”

Imagine a seguinte situação, uma escola faz a matrícula de cada aluno preenchendo um registro com informações como nome, RM, etc. O sistema acadêmico armazena o registro de cada aluno simplesmente por ordem de chegada no arquivo de registros. Se um determinado aluno chamado “João da Silva” chega e precisa alterar um dado do seu registro, só há uma forma de fazer a recuperação desse registro, buscando um por um. O tempo para se ter acesso ao registro seria muito grande.

Fica claro que se tivermos registros organizados (classificados) por ordem alfabética a busca poderia ser bem mais simples e eficiente.

A realização de classificação (ou ordenação) pode ser feita através de diversos métodos. Esses métodos são separados pela forma como realizam a classificação. Existem métodos de classificação por troca, por inserção, por seleção e por técnica de divisão do conjunto de dados. Cada método tem seus pontos positivos e negativos e situações onde oferecem maior ou menor eficiência. Sendo assim, vamos estudar os principais métodos de ordenação a fim de termos discernimento e clareza de quando devemos usar cada um deles.

Nesse texto vamos descrever todos os métodos de ordenação usando vetores nas explicações e programas exemplos. Porém, as mesmas considerações valem para uma abordagem mais ampla onde ao invés de vetor trabalhamos com arquivos composto por diversos registros, similar a ideia inicial desta seção.

## 7.2 Algoritmos de Classificação por Troca - Bubblesort

A ideia básica deste método da bolha ou *Bubblesort* é percorrer sequencialmente o arquivo várias vezes comparando os registros vetor[i] e vetor[i+1] e trocando esses registros quando vetor[i] > vetor[i+1].

Para compreender como esse método funciona, suponha o seguinte vetor (que representa aqui um arquivo) de dados inteiros e a utilização do método em apenas uma vez que o vetor é totalmente percorrido (primeira iteração), como mostra a Figura Exemplo de um vetor e a primeira iteração do método *Bubblesort*.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 23 | 54 | 47 | 36 | 15 | 92 | 83 | 32 | Arquivo Original |
| 23 | ~~54~~ 47 | ~~54~~ 36 | ~~54~~ 15 | 54 | ~~92~~ 83 | ~~92~~ 32 | 92 | 1ª iteração com 5 trocas |

Figura 7.1 – Exemplo de um vetor e a primeira iteração do método *Bubblesort*
Fonte: Elaborado pelo autor (2019).

Observe que o maior elemento por esse método é colocado na última posição do vetor. Esse fato dá o nome ao método, em cada vez que o vetor é percorrido (cada iteração) o maior elemento é como uma “bolha que flutua” até atingir sua posição final.

Aplicando-se esse método até que todo elemento do vetor tenha sido corretamente posicionado “bolha que flutua”, temos o vetor ordenado. Para mostrar a execução do método em cada iteração é apresentada na Figura Configuração de vetor ao final de cada iteração usando o método *Bubblesort.*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 23 | 47 | 36 | 15 | 54 | 83 | 32 | 92 | 1ª iteração |
| 23 | 36 | 15 | 47 | 54 | 32 | 83 | 92 | 2ª iteração |
| 23 | 15 | 36 | 47 | 32 | 54 | 83 | 92 | 3ª iteração |
| 15 | 23 | 36 | 32 | 47 | 54 | 83 | 92 | 4ª iteração |
| 15 | 23 | 32 | 36 | 47 | 54 | 83 | 92 | 5ª iteração |
| 15 | 23 | 32 | 36 | 47 | 54 | 83 | 92 | 6ª iteração |
| 15 | 23 | 32 | 36 | 47 | 54 | 83 | 92 | 7ª iteração |
| 15 | 23 | 32 | 36 | 47 | 54 | 83 | 92 | 8ª iteração |

Figura 7.2 – Configuração de vetor ao final de cada iteração usando o método *Bubblesort*
Fonte: Elaborado pelo autor (2019).

Por esse exemplo, pode-se observar que a cada iteração a posição n-1 do vetor já está com seu valor correto, ou seja, na 1ª iteração o valor 92 (maior valor) está na posição do índice 7 do vetor, que é a sua posição correta. Na 2ª, o valor 83 fica em sua posição (índice 6), e assim por diante. Na 8ª iteração (iteração n-1) falta apenas comparar vetor[0] e vetor[1]. Usando essa observação, podemos efetuar uma melhoria nesse método por exigir um número menor de comparações tornando-o mais eficiente.

O programa a seguir apresenta o código que lê do teclado 8 elementos de um vetor (como na Figura “Exemplo de um vetor e a primeira iteração do método Bubblesort”) e aplica as operações necessárias para aplicar o método *Bubblesort*.

```
import java.util.Scanner;

public class BubbleSortVersao1 {

        public static int N = 8;
           // define o tamanho do vetor a ser ordenado


       public static void main(String[] args)  {

            Scanner entrada = new Scanner(System.in);

            /*gera vetor fora de ordem*/
            int i;
            int vetor[] = new int[N];
            System.out.println("Digite "+N+" numeros inteiros: ");

            for(i = 0; i < N; i++)
                  vetor[i] = entrada.nextInt();


            System.out.println("Ordenando o vetor com Bubblesort");
            for (i=0;i<N-1; i++){
                  int aux;
                  for (int j=0; j<N-i-1; j++){
                        if (vetor [j]> vetor[j+1]){
                              aux= vetor[j];
                              vetor[j]= vetor[j+1];
                              vetor[j+1]=aux;
                              }
                  }
            }
            for(i = 0; i <N; i++)
                  System.out.println(i +"\t"+ vetor[i]);
            entrada.close();
            }
```

```
        }
```

Código- fonte 7.1 – Programa que usa o Método Bubblesort escrito em JAVA.
Fonte: Elaborado pelo autor (2019)

Outra alteração pode ser realizada para tornar mais eficiente o método *Bubblesort*. No Figura Configuração de vetor ao final de cada iteração usando o método *Bubblesort*, pode-se verificar que a partir da 6ª iteração nenhuma troca é realizada e quando isso acontece, significa que o arquivo já está ordenado, não sendo mais necessário percorrê-lo. Com essa otimização a ordenação do vetor com 8 elementos que necessitaria de 8 iterações pode ser feita em apenas 6.

Dessa forma, é necessário modificar o código da implementação do método *Bubblesort* para que utilize uma variável que sinalize a ocorrência de troca de posição de elementos. Essa alteração torna o método mais eficiente e é apresentada no código a seguir.

```
import java.util.Scanner;

public class BubbleSort {
      public static int N = 8;
         // define o tamanho do vetor a ser ordenado

      public static void main(String[] args)  {

           Scanner entrada = new Scanner(System.in);

           /*gera vetor fora de ordem*/
           int i;
           int vetor[] = new int[N];
           System.out.println("Digite "+N+" números inteiros: ");

           for(i = 0; i < N; i++)
                vetor[i] = entrada.nextInt();

           System.out.println("Ordenando o vetor com Bubblesort mais
eficiente ");
           boolean troca= true; /* supõe realizar troca */
           for (i=0;i<N-1 && troca==true; i++){
                int aux;
                troca= false; /* supõe não realizar troca */
                for (int j=0;j<N-i-1;j++){
                     if (vetor[j]> vetor[j+1]){
                          aux= vetor[j];
                          vetor[j]= vetor[j+1];
                          vetor[j+1]=aux;
                          troca=true;
      /*registra que houve troca na iteração*/
                          }
                     }
```

```
            }
        for(i = 0; i <N; i++)
            System.out.println(i +"\t"+ vetor[i]);
        entrada.close();
        }
}
```

Código-fonte 7.2 – Programa que usa o método Bubblesort otimizado escrito em JAVA.
Fonte: Elaborado pelo autor (2019)

### 7.2.1 Análise de Eficiência do Método *Bubblesort*

Vamos analisar a eficiência em relação aos dois Código- fonte apresentados, o primeiro, sem otimizações, e o segundo, com otimizações.

- Sem otimizações:

São (n-1) iterações e (n-1) comparações em cada iteração. Sendo assim, o número total de operações de comparação seria: (n-1)*(n-1) = $n^2$ - 2n + 1, que conforme explicado no tópico sobre eficiência de algoritmo, é O($n^2$). Claro que o número de trocas depende da sequência inicial do vetor, entretanto o número de trocas não será maior que o número de comparações. Porém, o número de trocas ocupará a maior parte do tempo do programa se comparado ao número de comparações.

- Com otimizações:

O número de comparações na iteração i é (n-i). Sendo assim, se existirem k iterações, o número total de comparações (n-1) + (n-2) +... + (n-k), que é igual a:

$$\frac{2kn + k^2 - k}{2}$$

O número médio de iterações k é O(n), portanto a expressão inteira é O($n^2$), embora o fator constante seja maior que antes. Entretanto, ocorre uma sobrecarga adicional para atribuir valor e comparar a variável troca (uma vez a cada iteração), além da atribuição de *true* quando ocorre a troca.

Podemos destacar uma vantagem do método da bolha no que se refere à exigência de espaço adicional por ser pequena, ou seja, apenas 1 registro extra para a troca (aux). Outra vantagem é que com otimizações a eficiência é O(n) para o caso do arquivo totalmente classificado.

## 7.3 Ordenação por Inserção

Os métodos de ordenação por inserção procuram colocar cada elemento em seu lugar correto diretamente. Inicialmente, supõe-se que o primeiro elemento já esteja em sua posição correta no arquivo com 1 elemento apenas. Na inserção do k-ésimo elemento, os elementos de 0 a k-1 já estarão ordenados, pois para cada inserção, os elementos maiores do que o k-ésimo serão movidos para frente. Dessa forma, após a inserção de todos os elementos o arquivo estará completamente ordenado.

Vamos conhecer dois métodos baseados na ordenação por inserção: *insertionsort* (inserção direta) e *shellsort.*

### *7.3.1* Ordenação pelo Método Inserção Direta (*INSERTION SORT)*

A explicação do método é bem simples quando realizada por meio de exemplo. Assim vamos supor o seguinte vetor a ser organizado:

| | | | |
|---|---|---|---|
| 25 | 57 | 48 | 37 |

Agora, vamos aplicar o método gerando o vetor ordenado. Primeiro elemento a ser inserido (25):

| |
|---|
| 25 |

Próximo elemento, 57, faz com que o primeiro elemento permaneça em seu lugar e ele seja inserido na posição seguinte.

| | |
|---|---|
| 25 | 57 |

Para ser inserido em sua posição correta, o elemento 48 deve fazer com que o elemento 57 passe para a posição seguinte no vetor, assim:

| | | |
|---|---|---|
| 25 | 48 | 57 |

Para ser inserido em sua posição correta, o elemento 37 deve fazer com que os elementos 48 e 57 sejam movidos para a frente no vetor, terminando a aplicação do método com o vetor já ordenado.

| | | | |
|---|---|---|---|
| 25 | 37 | 48 | 57 |

Portanto, esse procedimento deve ser aplicado até que todos os elementos do arquivo estejam inseridos em suas posições corretas.

A implementação desse método é apresentada no Código-fonte Ordenação usando método Insertion Sort escrito em JAVA. Para gerar valores dos elementos foi usado, nesse programa, o gerador de números aleatórios (random).

```
import java.util.Random;
import java.util.Scanner;

public class InsertionSort {
      public static int N = 10;
         // define o tamanho do vetor a ser ordenado

      public static void main(String[] args)  {

          Scanner entrada = new Scanner(System.in);
          Random gerador = new Random();

          int i;

      /*cria a estrutura de dados (vetor) com N elementos aleatórios*/
          int vetor[] = new int[N];
          System.out.println("Criando vetor com "+N+" elementos:");
          for(i = 0; i <N; i++)
                vetor[i] = (int)gerador.nextInt()/10000;


          System.out.println("Ordenando o vetor criado...");
          int eleito=0, j;
          for(i = 1; i <N; i++) {
                eleito = vetor[i];
                j = i-1;
                while (j >= 0 && vetor[j] > eleito) {
                      vetor[j+1] = vetor[j];
                      j = j-1;
                }
          vetor[j+1] =  eleito;
          }

          for(i = 0; i <N; i++)
                System.out.println(i +"\t"+vetor[i]);
```

```
            entrada.close();
            }
      }
```

Código-fonte 7.3 – Ordenação usando método Insertion Sort escrito em JAVA.
Fonte: Elaborado pelo autor (2019)

- **Análise de Eficiência do Método *Insertionsort***

Algumas observações sobre este método podem ser feitas em relação a sua eficiência.

Se o arquivo já estiver ordenado será feita apenas uma comparação para cada elemento, assim o algoritmo torna-se O(n).

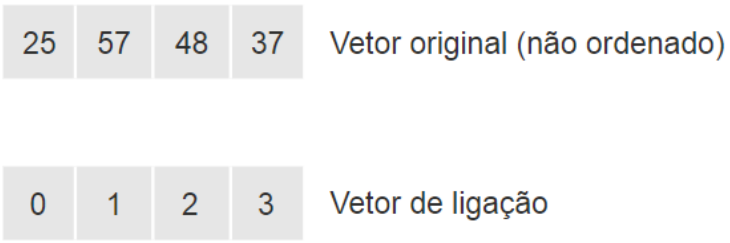
Caso o arquivo esteja na ordem inversa de classificação, para inserir o 2º elemento é necessária apenas uma comparação, para o 3º elemento duas comparações, para o 4º três e assim por diante. Dessa forma, o número de comparações em um arquivo com n elementos deve ser:

$1+2+3+ ... +(n-1) = (n-1) *n/2 \rightarrow O(n^2)$

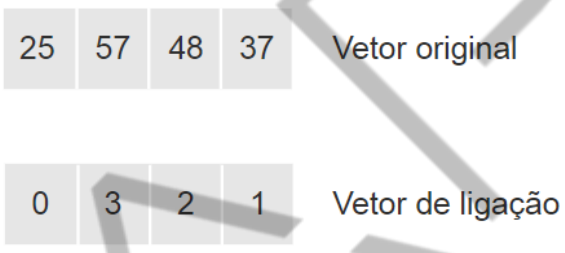
O número de trocas segue o mesmo número de comparações.

Apesar de ser $O(n^2)$ no caso de inversamente ordenado esse método tem, como ponto positivo, a exigência de apenas 1 variável temporária, ou seja, não há exigência de espaço de memória.

Quando os elementos do vetor a ser ordenado são um tipo de dado que requer um número elevado de bytes, como por exemplo, o registro de um cliente de um banco, pode-se fazer uma melhoria introduzindo uma lista de inserção, ou seja, deve-se criar um vetor de ligação que armazena a posição que cada elemento ocupa no arquivo a ser ordenado. Assim, em vez de movimentar os elementos dentro do arquivo altera-se apenas a posição indicada no vetor de ligação. O problema para esse caso é que há um aumento na ocupação de espaço na memória, passando agora para O(n).




Depois da aplicação do método com otimização não há alteração no arquivo original, mas apenas no vetor de ligação que então ficaria:




### *7.3.2* Ordenação pelo Método *Shellsort*

O algoritmo *shellsort* ou de ordenação por incremento decrescente também é um método baseado na inserção de elementos em sua posição correta. Esse método precisa de um vetor de incremento para definir o número de partições (divisões) sobre o arquivo de dados. A classificação é realizada sobre subvetores do vetor original. Esse k subvetores contém os k-ésimos elementos do vetor original. O valor de k é chamado de incremento. Supondo um vetor a ser ordenado e k=3, têm-se três subvetores:

**subvetores 1:** vetor[0], vetor[3], vetor[6],...

**subvetores 2:** vetor[1], vetor[4], vetor[7],...

**subvetores 3:** vetor[2], vetor[5], vetor[8],...

Depois que os k subvetores estiverem ordenados (geralmente por inserção) é escolhido um novo valor para k, menor que o já utilizado, por exemplo, 1.

O algoritmo do *shellsort* precisa de um vetor de incrementos para definir partições sobre o conjunto de dados. Na implementação do método *shellsort* apresentada no código a seguir, é usado um vetor de incremento pré-construído (**incre**) com o número de incrementos passado também como parâmetro (**n_incre**). De forma simplificada, foi escolhido o uso de um vetor de incremento com os elementos 3 e 1, sendo 3 o primeiro incremento utilizado. A descrição do passo a passo da execução do método é feita junto com a descrição do programa.

```
  import java.util.Scanner;
  import java.util.Random;

  public class ShellSort {
     public static int N = 10;
         // define o tamanho do vetor a ser ordenado


     public static void main(String[] args)  {

           Scanner entrada = new Scanner(System.in);
           Random gerador = new Random();

           int i;

     /*cria a estrutura de dados (vetor) com N elementos*/
           int vetor[] = new int[N];
           System.out.println("Criando vetor com " + N + " elementos:
");
           for(i = 0; i<N; i++)
                 vetor[i] = (int)gerador.nextInt()/10000;


           System.out.println("Ordenando o vetor criado...");
           int cont1, cont2, cont3, dist, elem, n_incre=2;
           int incre[] = new int[2];
           incre[0] = 3;
           incre[1] = 1;

           for(cont1 = 0; cont1<n_incre; cont1++)   {
                 //loop para percorrer vetor de incrementos
                 dist = incre[cont1];
                 for(cont2 = dist; cont2<N; cont2++)   {
                       elem = vetor[cont2];
                       for(cont3   =   cont2-dist;(cont3>=   0)   &&
(elem<vetor[cont3]); cont3 = cont3 - dist) {
                             vetor[cont3+dist] = vetor[cont3];
                             }
                       vetor[cont3+dist] = elem;
                       }
                 }

           for(i = 0; i<N; i++)
```

```
                System.out.println(i +"\t"+vetor[i]);
            entrada.close();
        }

    }
```

Código-fonte 7.4 – Ordenação usando método Shellsort escrito em JAVA.
Fonte: Elaborado pelo autor (2019)

O vetor de incremento pode ser gerado usando formas diversas de algoritmos. Para compreender melhor o método Shellsort e o significado do vetor de incremento, considere o seguinte exemplo de vetor original e vetor de incremento (incre).

| 25 | 73 | 48 | 37 | 12 | 86 |
|---|---|---|---|---|---|

| incre | 3 | 1 |
|---|---|---|

Com o primeiro valor de incremento (3) são gerados 3 subvetores que terão seus conteúdos classificados. Como o vetor tem 6 elementos cada subvetor terá apenas 2 elementos. Assim, serão comparados (e serão trocados em função dessa comparação) os elementos vetor[0] e vetor[3] (25 e 37), depois vetor[1] e vetor[4] (12 e 73, que serão trocados) e finalmente vetor[2] e vetor[5] (48 e 86).

| Subvetor1: | 0 | 3 |
|---|---|---|
| | 25 | 37 |

| Subvetor2: | 1 | 4 |
|---|---|---|
| | 12 | 73 |

| Subvetor3: | 2 | 5 |
|---|---|---|
| | 48 | 86 |

Pela ordenação usando o incremento 3 o vetor passa a ter seus subvetores já ordenados, ficando o vetor com a seguinte configuração (note que não há realmente a criação dos subvetores, sendo sua criação apenas lógica e continuamos com apenas 1 vetor):



| | | | | | |
|---|---|---|---|---|---|
| 25 | 12 | 48 | 37 | 73 | 86 |

Observe que o vetor ainda não está totalmente ordenado. Para isso é preciso ainda utilizar o segundo incremento, ou seja, 1. Com esse valor de incremento todos os elementos serão comparados 2 a 2. Isso significa que serão comparados os elementos vetor[0] e vetor[1] (25 e 12, que serão trocados), depois vetor[1] e vetor[2] (25 e 48, não faz a troca) e assim por diante.

A eficiência desse método é diretamente ligada à escolha dos valores de incrementos e à quantidade de elementos do vetor de incremento. Segundo Tenenbaum (1995, p.466) há uma forma alternativa de se obter o vetor de incremento. Os seus elementos são calculados por uma variante da função proposta por Knuth. A seguir é apresentado o programa que utiliza a proposta de Knuth.

```
import java.util.Random;
import java.util.Scanner;

public class ShellSort_Knuth {

  public static int N = 10;
    // define o tamanho do vetor a ser ordenado


  public static int Knuth(int param) {
            if(param == 1)
                  return(1);
            else
                  return(3*Knuth(param-1)+1);
            }


  public static void main(String[] args)  {

        Scanner entrada = new Scanner(System.in);
        Random gerador = new Random();
        int i;

  /*cria a estrutura de dados (vetor) com N elementos*/
        int vetor[] = new int[N];
        System.out.println("Criando vetor com"+N+" elementos: ");
```

```
            for(i = 0; i<N; i++)
                  vetor[i] = (int)gerador.nextInt()/10000;

      /*gera vetor de incrementos usando Knuth */
            int limite = 1;
            int cont;
            while (Knuth(limite) <N) {
                  limite++;
                  }
            limite = limite - 2;

            if (limite< 1)  {
                  limite = 1;
                  }
            int incre[] = new int[limite+1];
            incre[limite] = 1;
            for(cont = 0; cont<limite; cont++)  {
                  incre[cont] = Knuth(limite - cont + 1);
                  }
            limite++;


            System.out.println("Ordenando o vetor criado...");
            int cont1, cont2, cont3, dist, elem;

            for(cont1 = 0; cont1<limite; cont1++)   {
                  //loop para percorrer vetor de incrementos
                  dist = incre[cont1];
                  for(cont2 = dist; cont2<N; cont2++){
                        elem = vetor[cont2];
                        for(cont3    =    cont2-dist;(cont3>=    0)    &&
(elem<vetor[cont3]);  cont3 = cont3 - dist)
                              vetor[cont3+dist] = vetor[cont3];

                         vetor[cont3+dist] = elem;
                        }
                  }

            for(i = 0; i<N; i++)
                  System.out.println(i +"\t"+vetor[i]);
            entrada.close();
      }
   }
```

Código-fonte 7.5 – Criação do vetor de Incrementos usado pelo método Shellsort escrito em JAVA.
Fonte: Elaborado pelo autor (2019)

- **Análise de Eficiência do Método *Shellsort***

A análise de eficiência do método de *Shellsort* é muito complexa e depende diretamente da escolha de incrementos a serem aplicados sobre o arquivo. Ninguém ainda foi capaz de encontrar uma fórmula fechada para sua função de complexidade,

pois sua análise contém alguns problemas matemáticos muito difíceis, entre eles a escolha da sequência de incrementos.

Algo que se sabe é que cada incremento não deve ser múltiplo do anterior.

Foi demonstrado experimentalmente que a eficiência pode ser aproximada por $O(n (\log n)^2)$.

Esse método tem como vantagens:

- demonstrou uma ótima opção para arquivos de tamanho moderado.
- tem implementação simples e requer uma quantidade de código pequena.

Mas, devemos considerar as desvantagens que o tempo de execução do algoritmo é sensível à ordem inicial do arquivo e à escolha do vetor de incremento.

## 7.4 Ordenação por Seleção

A classificação por seleção é aquela na qual sucessivos elementos são selecionados em sequência e dispostos em suas posições corretas pela ordem.

Uma forma de implementar esse tipo de ordenação seria utilizando uma fila de prioridade crescente (*fila_prioridade),* em que a operação *fila_prioridade _remove_max* retira, selecionando, o maior elemento dessa fila. Para armazenar os elementos anteriormente em *fila_prioridade*, utiliza-se a operação *fila_prioridade _insere.* Nessa fila de prioridade a inserção estabelece alguma ordenação.

Para melhor compreender, considere o exemplo de trecho de programa:

```
/*pré-processamento para inserir em fila*/
for (i=n-1; i>=0; i--)
  fila_prioridade_insere(fila_prioridade,vetor[i]);

/*remove maior elemento da fila_prioridade */
for (i=n-1; i>=0; i--)
  vetor[i] = fila_prioridade_remove_max(fila_prioridade);
```

A existência da fila de prioridade exige então o pré-processamento do arquivo original a ser ordenado.

Um algoritmo simples denominado *selectionsort* é apresentado em Tenenbaum (1995, p. 443). Esse algoritmo utiliza uma forma em que a seleção utiliza o próprio

vetor para gerar fila de prioridade, fazendo com que sua implementação seja não eficiente, assim seu estudo será deixado como complementar.

Outros métodos usam árvores binárias para implementar a fila de prioridade, como serão discutidos a seguir.

### *7.4.1* Ordenação por Árvore de Busca Binária (ABB)

Neste método, os registros de um arquivo são inseridos em uma ABB. Depois de efetuar o pré-processamento para gerar ABB, para se obter o arquivo ordenado, deve-se percorrer a ABB em ordem.

O código a seguir apresenta a implementação desse método.

```
import java.util.*;
import java.util.Random;

public class ordenacao_ABB {

    public static int N = 10;

    private static class ARVORE{
        public int dado;
        public ARVORE dir;
        public ARVORE esq;
    }

    public static ARVORE init()
    {
        return null;
    }
    public static void main(String[] args) {
        Scanner entra = new Scanner(System.in);
        ARVORE raiz = init();

        Random gerador = new Random();

        int i;

    /*cria a estrutura de dados (vetor) com N elementos*/
        int vetor[] = new int[N];
        System.out.println("Criando vetor com" + N +"
elementos");
        for(i = 0; i<N; i++)
            vetor[i] = (int)gerador.nextInt()/10000;

        System.out.println("Pré-processamento criando ABB");
        for(i = 0; i<N; i++)
            raiz = inserir(raiz, vetor[i]);
```

```
                System.out.println("Ordenando o vetor criado...");
                i = retira_em_ordem(raiz, 0, vetor);

                for(i = 0; i<N; i++)
                      System.out.println(i +"\t"+vetor[i]);

                entra.close();
          }


          private static ARVORE inserir(ARVORE p, int info) {
                // insere elemento em uma ABB
                if (p == null) {
                      p=new ARVORE();
                      p.dado = info;
                      p.esq = null;
                      p.dir = null;
                }
                else if (info < p.dado)
                            p.esq= inserir (p.esq, info);
                      else
                            p.dir=inserir(p.dir, info);


                return p;
          }

          private static int retira_em_ordem(ARVORE root,int i, int
vetor[]) {
                // lista elementos percorrendo em ordem
                if (root != null) {
                      i = retira_em_ordem(root.esq,i,vetor);
                      vetor[i]=root.dado;
                      i++;
                      i = retira_em_ordem(root.dir,i,vetor);
                }
                return i;

          }

      }
```

Código-fonte 7.6 – Criação do vetor de Incrementos usado pelo método seleção usando árvore de busca binária (ABB) escrito em JAVA.
Fonte: Elaborado pelo autor (2019)

Quanto à eficiência fica claro que se o arquivo original já estiver ordenado a ABB teria apenas subárvores à esquerda, se o arquivo estiver em ordem decrescente. E apenas teria subárvores à direita, se o arquivo estiver em ordem crescente. Como mostra o exemplo apresentado na Figura Configuração de uma ABB inserindo elementos de um vetor a ser ordenado.

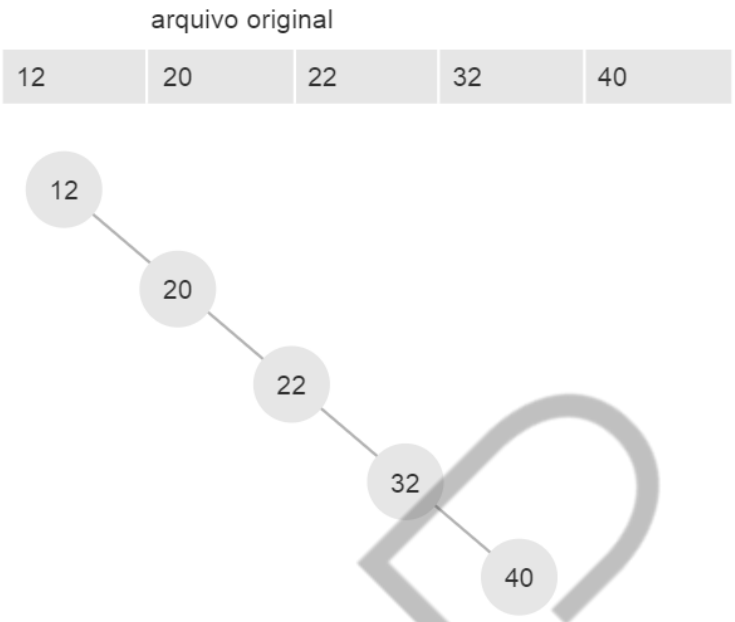



Figura 7.3 – Configuração de uma ABB inserindo elementos de um vetor a ser ordenado
Fonte: Elaborado pelo autor (2019).

Assim, para inserir o 1º registro não seria necessária nenhuma comparação. Para o 2º registro, 2 comparações são realizadas, uma para verificar se o elemento é maior que a raiz e outra para apurar que não há nó à direita, só, então o 2º elemento (valor 20, no exemplo) será inserido. Para o 3º, três comparações são necessárias, e assim por diante. Isso pode ser equacionado:

$2 + 3 +...+ (n-2) + (n-1) = n * (n+1)/2 -1 => O(n^2)$

Portanto, essa situação gera um método bastante ineficiente.

Porém, se o arquivo original estiver ordenado de forma que metade dos registros seja menor do que a raiz e metade maior, como o caso apresentado no exemplo da Figura Configuração de uma ABB inserindo elementos de um vetor a ser ordenado, o método seria mais eficiente.

Elemento 11: nenhuma comparação

Elemento 4: 2 comparações

Elemento 3: 3 comparações

Elemento 15: 2 comparações

Elemento 16: 3 comparações

Total => 2+3+2+3 = 10 comparações

De uma forma simplificada podemos fazer o seguinte raciocínio: como são 5 elementos pode-se dizer que dentro das funções usadas para descrever eficiência a mais adequada seria n log, pois seria $5*\log_2 5$, ou seja, 5 * 2.32 = 11.6. Portanto, pode-se dizer que a eficiência do algoritmo seria O(n log n) em relação ao número de comparações efetuadas.

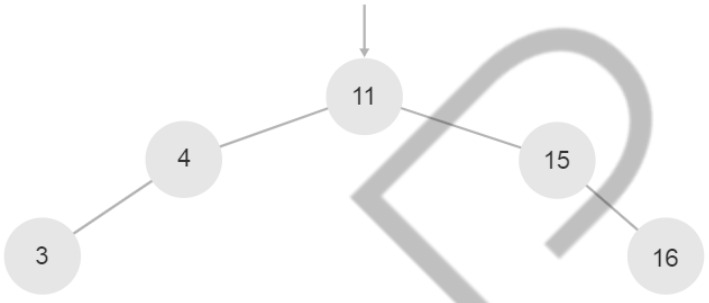



Figura 7.4 – Configuração de uma ABB inserindo elementos de um vetor a ser ordenado
Fonte: Elaborado pelo autor (2019).

Em termos de ocupação de memória esse método é bastante ineficiente, pois exige um nó para cada registro. Com a alocação dinâmica deve-se acrescer o armazenamento dos ponteiros em cada nó.

Esse fato, junto a ineficiência quando o arquivo já estiver ordenado representa uma grave deficiência do método, que será eliminado no método *Heapsort,* que veremos adiante. Antes, porém, vamos estudar uma forma de armazenar árvore binária usando vetor, ou seja, sem realizar a alocação dinâmica de cada nó da árvore.

### 7.4.2 Representação em Vetor de Árvore Binária

Uma das desvantagens em se utilizar à representação dinâmica de árvores binárias é o fato de serem necessários, no mínimo, 2 campos a mais que a informação útil para armazenar os ponteiros (esquerda e direita). A representação usando vetores pode ser mais eficiente nesse sentido.

É claro que também tem a desvantagem da necessidade de reserva de espaço de memória em tempo de compilação, mas para o uso de uma árvore binária (AB) em métodos de classificação pode ser mais eficiente se representada usando vetor.

Para explicar como usar vetor para representar uma AB será sempre usada a forma de implementação em linguagem JAVA, ou seja, o primeiro elemento está na posição de índice 0 e o maior índice será n-1, em que n é o número de elementos do vetor.

Um nó referenciado a partir de agora como nó **p** representará o nó que está na posição **p** do vetor de informações guardadas na árvore (**info**), ou seja, info[p].

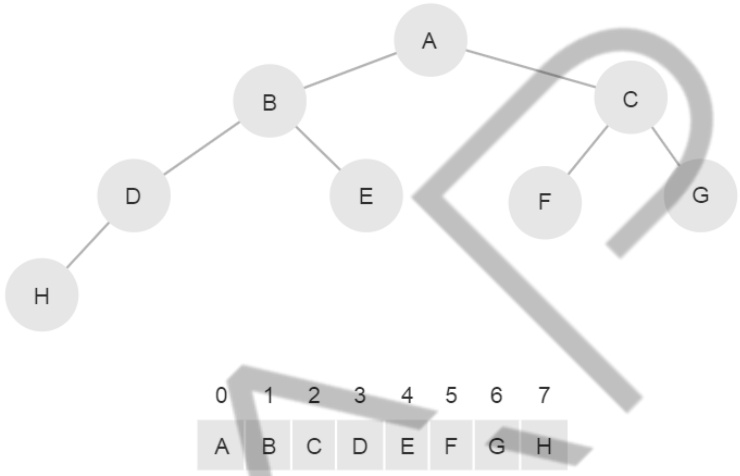



Figura 7.5 – Primeiro exemplo de uma árvore binária com seus nós armazenados como elementos de um vetor
Fonte: Elaborado pelo autor (2019).

A regra para determinar a posição dentro do vetor para cada nó da árvore binária são os filhos do nó **p** (lembre-se que é info[p]) que estarão nas posições 2p+1 (filho à esquerda) e 2p+2 (filho à direita). Por exemplo, por essa regra os filhos do nó 1 estarão nas posições 3 e 4.

Para estabelecer toda a relação dos nós deve-se observar que:

- Pai do nó p será: (p-1)/2
- Será o filho à esquerda se p%2 (resto da divisão por 2) for 0, e assim o irmão à direita estará na posição p+1.
- Será o filho à direita se p%2 for 1, e assim o irmão à esquerda estará na posição p-1.

Para uma árvore binária como a apresentada anteriormente o uso de vetor é bastante eficiente, pois os nós da árvore são colocados em posições do vetor e todas são preenchidas com as informações. Agora considere a árvore a seguir:

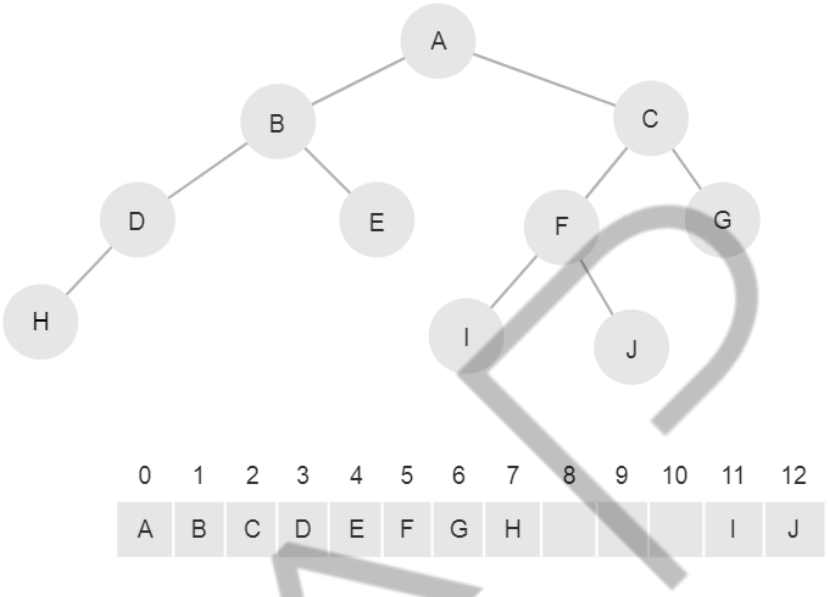



Figura 7.6 – Segundo exemplo de uma árvore binária com seus nós armazenados como elementos de um vetor
Fonte: Elaborado pelo autor (2019).

Fica claro que a implementação usando vetor traz a desvantagem de se reservar espaço contíguo na memória, mesmo que pela regra de inserção e remoção de elementos da árvore ela não seja plenamente utilizada. Essa possibilidade leva à necessidade de criar uma maneira de especificar quais posições têm elementos válidos e quais estão desocupadas. Uma forma de resolver isso é criar um campo para indicar se a posição no vetor está ocupada ou não (ocupada).

Sendo assim, pode-se declarar o vetor como sendo um vetor em que cada elemento é formado por dois atributos: info e ocupado. Assim, cada elemento inserido deve não apenas armazenar o valor do dado, mas também a sinalização de que a posição do vetor está ocupada.

O trecho de programa a seguir mostra uma forma de implementar essa solução.

```
public static class VetorAB{
      int info;
      int ocupado;
      }
```

```
      public static void main(String[] args)  {
            VetorAB vetor[] = new VetorAB[10];
            vetor[0].info = 23; //atribui valor ao elemento
            vetor[0].ocupado =1; //marca posição do vetor como ocupada

       }
```

Código-fonte 7.7 – Trecho de programa emque cada elemento é formado por dois atributos: info e ocupado.
Fonte: Elaborado pelo autor (2019)

Em Tenenbaum (1995, p.320), são encontradas funções que manipulam árvores binárias na representação sequencial (vetor) e também os aspectos negativos e positivos dessa representação em comparação com a encadeada. Para o estudo do método de *Heapsort* já são suficientes as informações aqui apresentadas.

### *7.4.3* Ordenação pelo Método *Heapsort*

Neste método é utilizado um tipo especial de árvore binária definida como *heap* decrescente, que consiste em uma árvore binária quase completa de **n** nós, tal que o conteúdo de cada nó deve ser menor ou igual ao conteúdo do nó pai.

A figura a seguir exibe um exemplo desse tipo de organização de árvore.

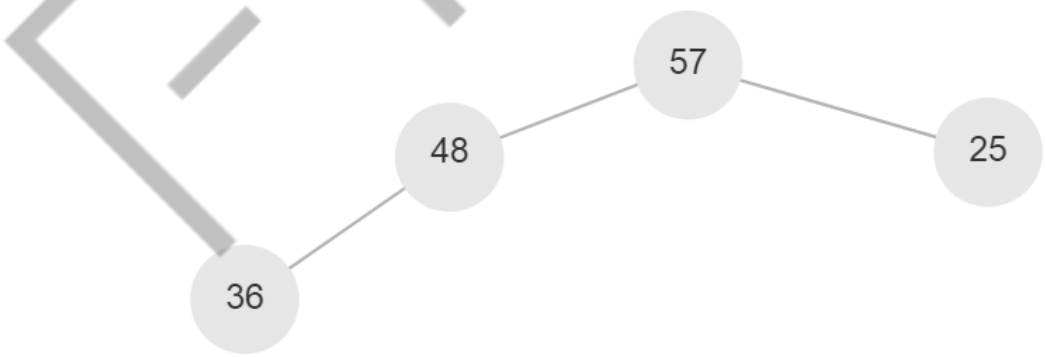



Figura 7.7 – Exemplo de uma árvore organizada como heap
Fonte: Elaborado pelo autor (2019).

Pela definição da organização da *heap* fica claro que o maior elemento de uma *heap* será sempre o nó raiz (como no exemplo é o valor 57). Dessa forma, a *heap*

será uma lista de prioridade bastante eficiente. O método *Heapsort* é apenas uma classificação por seleção usando um vetor de entrada (no código do programa a seguir será a variável chamada **vetor**) que é transformada em uma *heap* decrescente (representando a fila de prioridade). Na primeira fase é realizado o pré-processamento, no qual, a partir do vetor original a ser ordenado, cria-se o *heap* (operação fila_prioridade_insere) Na fase seguinte, fila_prioridade_remove_max seleciona retirando o maior elemento. A remoção faz com que seja necessária a redistribuição dos elementos restantes.

Tanto na inserção quanto na seleção, as iterações iniciam no 2º elemento já que vetor[0] sempre terá o maior elemento.

Um exemplo de uma *heap* criada na fase de pré-processamento é apresentada a seguir (representando a árvore da binária como vetor).

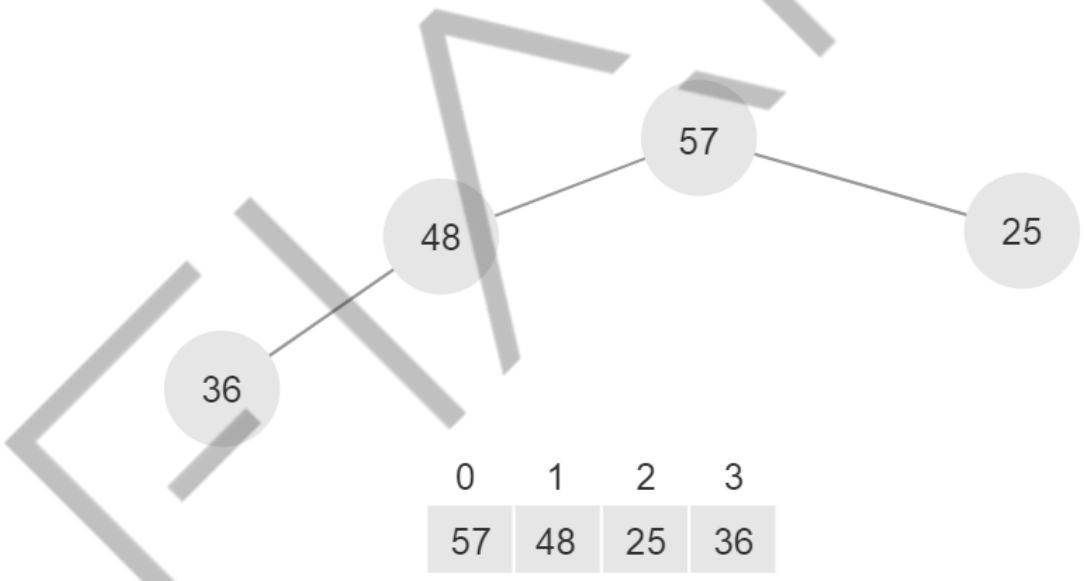



Figura 7.8 –Heap do exemplo organizada em vetor
Fonte: Elaborado pelo autor (2019).

Na fase de pré-processamento a tarefa é criar uma árvore cujo nó pai seja sempre maior que os seus filhos. Para tanto, deve-se procurar o maior elemento e a cada elemento menor encontrado deve-se deslocar para mais perto do nó folha. Isso é necessário, pois normalmente, a forma usada para representar a *heap* é como vetor. Após ter sido criada estrutura da *heap,* inicia-se a fase de real da ordenação. É

importante destacar que a ordenação será feita *in loco*, ou seja, no mesmo vetor em que foi criada a *heap.*

A ordenação inicia definindo que o elemento que deve ser selecionado e, portanto, colocado em sua posição final, é o maior elemento (que deve ocupar a posição n-1 do vetor ordenado). Pela regra de construção da *heap* o maior elemento encontra-se na raiz da árvore (posição 0). Assim, esse elemento é movido para a sua posição correta. No lugar da raiz deve ser movido o maior elemento entre seus 2 filhos. Como a raiz pode ter filho à direita e à esquerda deve-se encontrar qual dos dois é maior para ocupar o procedimento, de forma que deixe a árvore com a regra de formação da *heap*.

A Figura “Passo a passo da aplicação da ordenação usando a *heap* no mesmo vetor” mostra um exemplo em que a *heap* já está gerada no próprio vetor e a aplicação do método de seleção. Cada vez que o elemento é posicionado corretamente na figura, ele é destacado (colorido) no vetor.

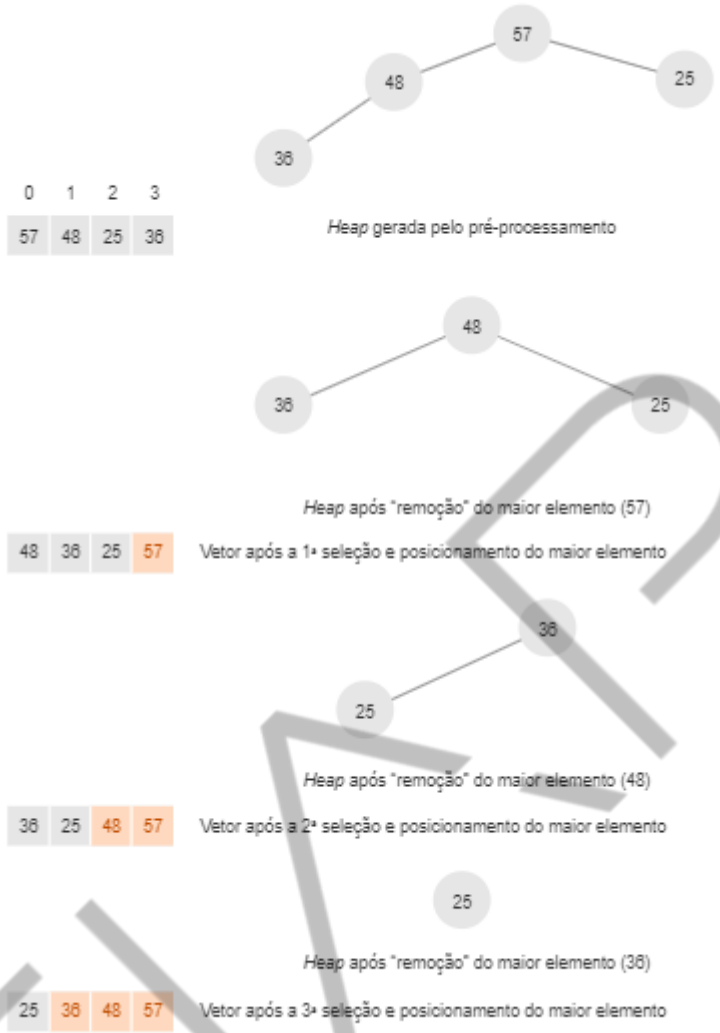



Figura 7.9 – Passo a passo da aplicação da ordenação usando a heap no mesmo vetor
Fonte: Elaborado pelo autor (2019).

O programa a seguir exibe uma implementação da seleção por meio da *heap*, ou seja, do método Heapsort.

```
import java.util.Random;
import java.util.Scanner;


public class HeapSort {

        public static int N = 10;
           // define o tamanho do vetor a ser ordenado

   public static void main(String[] args)  {
```

```
          Scanner entrada = new Scanner(System.in);
          Random gerador = new Random();
          int i;

/*cria a estrutura de dados (vetor) com N elementos*/
          int vetor[] = new int[N];
          System.out.println("Criando vetor com " + N+"  elementos");
          for(i = 0; i<N; i++)
               vetor[i] = (int)gerador.nextInt()/10000;


          System.out.println("Pré-processamento-criado heap");

/*cria heap no mesmo vetor, sempre nó pai é maior que seus filhos*/
          int elem,pai,filho,iv;
          for(i=1;i<N;i++){
               elem=vetor[i];
// elemento que se imagina ser o maior
               filho=i;
// indice do elemento (considerado filho)
               pai=(filho-1)/2;
//posição do pai do elememento
               while(filho>0 && vetor[pai]<elem){
// verifica se pai do elemento é menor
                    vetor[filho]=vetor[pai];
//sendo menor troca valor do pai e filho
                    filho=pai;
// índice do elemento muda para o filho
                    pai=(filho-1)/2;
//calcula novo indice do pai do elemento
                    }
               vetor[filho]=elem;
//menor elemento desce na árvore tornando-se nó filho
               }

          int nh=N;
     /*utiliza heap para aplicar o método de seleção*/
          for (i=N-1; i>0; i--){

               System.out.println("Configuração da Heap ...");

               for(int j= 0; j<nh; j++)
                    System.out.println(j +"\t"+vetor[j]);
               nh--;
               iv=vetor[i];
//iv guarda elemento da posição onde maior ocupará
               vetor[i]=vetor[0];
//coloca maior elemento na sua posição final inicia
// processo de busca de qual elemento que ocupará a raiz
               pai=0;
// encontra o maior filho do na raiz
               if(i==1)
                    filho=-1;
//marca com -1 que não há mais filho
               else
                    filho=1;
```

```
  //supõe que o maior filho seja o da esquerda
                if(i>2 &&vetor[2]>vetor[1])
                      filho=2;
  //verifica que o nó direito tem valor maior faz o
  //deslocamento dos elementos mantendo a estrutura da heap
                while(filho>=0 &&iv<vetor[filho]){
                      vetor[pai]=vetor[filho];
  // desloca o maior para a raiz
                      pai=filho;
  //inicia o posicionamento do filho do nó deslocado
                      filho=2*pai+1;
  //decide o nó que ficará a esquerda
                      if(filho+1<=i-1&& vetor[filho]<vetor[filho+1])
                            filho=filho+1;
  //inicia o posicionamento do filho do nó deslocado
                      if (filho>i-1)
                            filho=-1;
  //marca que o nó deslocado não tem filho
                      }
                vetor[pai]=iv;
  /*coloca o valor de elemento sobreposto pelo maior
    na posição do pai da subárvore alterada*/
                }

            System.out.println("Vetor ordenado:");
      for(i = 0; i<N; i++)
                System.out.println(i +"\t"+vetor[i]);
            entrada.close();
      }

  }
```

Código-Fonte 7.8 – Criação do vetor de Incrementos usado pelo método Heapsort escrito em JAVA.
Fonte: elaborado pelo autor (2019)

- **Considerações sobre Eficiência do *Heapsort***

Para analisar a eficiência do *Heapsort*, já que ele envolve uma árvore, deve-se considerar o estado em que essa árvore se encontra. Considerando a árvore binária completa com n nós, em que n=2m-1, isso significa que existirão $\log_2(n+1)$ níveis. Cada elemento selecionado (ou seja, o maior do vetor) seria o nó folha e considerando o deslocamento tem-se que o *Heapsort* é O(n log n). A descrição completa da análise da eficiência desse método é bem complexa, assim basta saber que para pequenos de n, esse método é muito ineficiente, já que a construção da *heap* acresce muitas operações críticas.

Para n grande a O(n log n) é mantida mesmo para os piores casos do estado inicial do vetor a ser ordenado.

### *7.5 Mergesort*

Esse tipo de método baseia-se na ideia de dividir o problema da ordenação dos elementos em partes menores até que o vetor a ser ordenado tenha apenas 1 elemento e depois combinar (operação conhecida como *merge* em inglês) as partes fazendo a ordenação final.

Esse tipo de solução utiliza uma técnica conhecida como “dividir para conquistar”.

O método segue a seguinte sequência de passos:

- O vetor com n elementos é dividido ao meio criando 2 subvetores.
- A divisão do vetor deve ser feita até que o subvetor fique com apenas 1 elemento.
- Quando os vetores de 1 elemento voltam a ser combinados os elementos são colocados usando critério de ordenação. Assim cada subvetor gerado pela combinação já está ordenado.
- A cada combinação (*merge*) os elementos são intercalados seguindo o critério de ordenação (por exemplo, em ordem crescente de valores).

Para melhor entendimento considere a figura a seguir em que o método é aplicado.

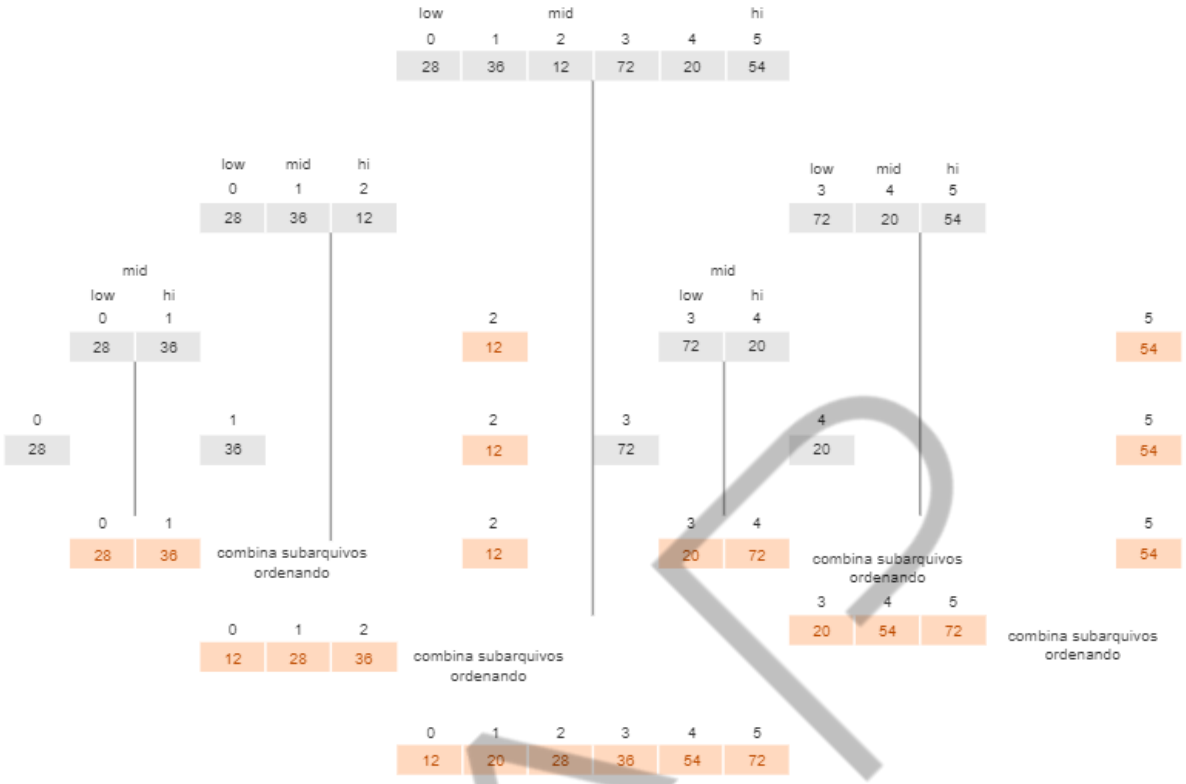



Figura 7.10 – Passo a passo da aplicação da ordenação usando a mergesort
Fonte: Elaborado pelo autor (2019).

No exemplo da Figura Passo a passo da aplicação da ordenação usando a Mergesort, inicialmente o vetor possui 6 elementos. Os índices mais baixos (low=0) e mais altos (hi=5) são usados para calcular o índice médio:

mid = (hi+low)/2 = (5+0)/2 = 2, apenas a parte inteira do resultado da divisão.

Assim são criados 2 subvetores de 3 elementos, o primeiro com elementos dos índices de 0 a 2 e o segundo com os índices de 3 a 5.

O método é reaplicado em cada subvetor. Analisando o primeiro o subvetor low=0, hi=2 e mid=(0+2)/2=1, criando mais 2 subvetores, um com 2 elementos e outro com apenas 1 elemento (portanto, já ordenado, por isso aparece em verde na figura). O método é reaplicado para o vetor com 2 elementos, low=0, hi=1 e mid=(0+1)/2=0, criando mais 2 subvetores com apenas 1 elemento (já ordenados).

Nesse ponto do método, é feita a combinação dos subvetores já ordenados. Como 28 é menor do que 36, as posições dos elementos no sub-vetor combinado são mantidas. Quando ocorre a combinação do vetor com elementos 28 e 36 com o

subvetor com elemento 12, deve-se fazer a intercalação dos elementos para que o sub-vetor final fique ordenado como representado na figura.

Processo idêntico é efetuado no subvetor de índices 3 a 5, e quando ocorre a combinação dos 2 subvetores de índices de 0 a 2 e de índices de 3 a 5, a intercalação ocorre a fim de que o vetor final fique completamente ordenado.

Pela descrição do método fica claro que a forma de reaplicar o método a cada subvetor é feita recursivamente.

O programa apresentado a seguir traz o código em que é usada a função recursiva merge() na qual a condição de parada da recursividade é quando o vetor possui apenas 1 elemento. A combinação com a ordenação dos elementos dos subvetores é efetuada pela função chamada de intercala().

Note que a divisão é apenas lógica e que o vetor é apenas analisado de forma separada e não são criados novos vetores para armazenar os subvetores.

```
import java.util.Random;
import java.util.Scanner;

public class MergeSort{

 public static int N = 10;
   // define o tamanho do vetor a ser ordenado

 public static void main(String[] args)  {

       Scanner entrada = new Scanner(System.in);
       Random gerador = new Random();

       int i;

 /*cria a estrutura de dados (vetor) com N elementos*/
       int vetor[] = new int[N];
       System.out.println("Criando vetor com"+N+" elementos");
       for(i = 0; i<N; i++)
            vetor[i] = (int)gerador.nextInt()/10000;

       System.out.println("Ordenando o vetor criado...");
       merge(vetor,0,N-1);

       System.out.println("Vetor ordenado:");
       for(i = 0; i<N; i++)
            System.out.println(i +"\t"+vetor[i]);
      entrada.close();
 }
```

```
public static void merge(int vet[], int low, int hi) {
      int mid;
      if (low < hi) {
            mid = (low + hi)/2;
            merge (vet,low,mid);
            merge (vet,mid+1,hi);
            intercala(vet,low,hi,mid);
            }
      }


public static void intercala(int vet[], int low, int hi, int mid)
      {
      int poslivre, low_v1, low_v2, i;
      int aux[] = new int[N];
      low_v1 = low;
      low_v2 = mid+1;
      poslivre = low;
      while (low_v1 <= mid && low_v2 <= hi) {
            if(vet[low_v1] <= vet[low_v2]) {
                  aux[poslivre] = vet[low_v1];
                  low_v1++;
                  }
            else {
                  aux[poslivre] = vet[low_v2];
                  low_v2++;
                  }
            poslivre++;
            }
      for (i = low_v1; i<=mid; i++) {
            aux[poslivre] = vet[i];
            poslivre++;
            }
      for (i = low_v2; i<=hi; i++) {
            aux[poslivre] = vet[i];
            poslivre++;
            }
      for (i=low; i<=hi; i++)
            vet[i] = aux[i];
      }
}
```

Código-fonte 7.9 – Criação do vetor de Incrementos usado pelo método Mergesort escrito em JAVA.
Fonte: elaborado pelo autor (2019)

- **Considerações sobre Eficiência do Merge*sort***

Para analisar a eficiência desse método é preciso considerar a complexidade da função intercalada. Cada subvetor deve ser percorrido com comparações entre

seus elementos fazendo m comparações, em que m é o número de elementos de cada subvetor.

Quanto à divisão recursiva do vetor original e de seus subvetores, a análise é mais simples se considerarmos a Figura Profundidade (altura) da divisão recursiva do vetor no método Mergesort, em que é mostrado o esquema gráfico da recursividade. É claro que cada vez que a divisão ocorre o vetor será dividido ao meio, supondo vetor com 8 elementos, assim ocorrem 3 ($\log_2 8$) divisões:

- Vetor original com 8 elementos é dividido em 2 vetores com 4 elementos
- Vetores com 4 elementos são divididos em 2 vetores com 2
- Vetores com 2 elementos são divididos em 2 vetores com 1

Assim o número de vezes que a operação de divisão é efetuada é da ordem de log n.

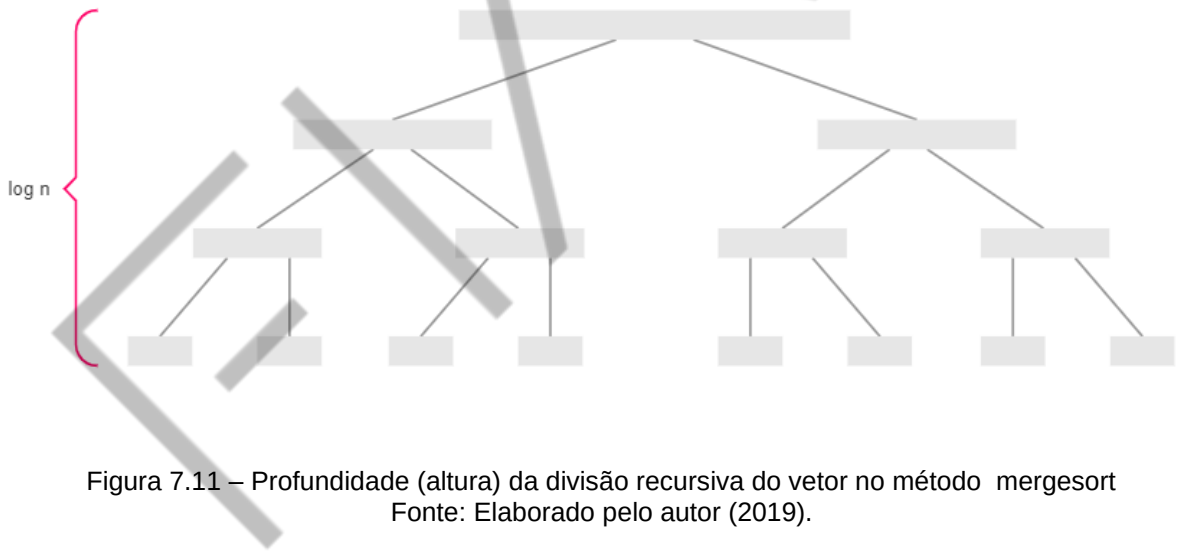



Figura 7.11 – Profundidade (altura) da divisão recursiva do vetor no método mergesort
Fonte: Elaborado pelo autor (2019).

Portanto, a eficiência desse método é da ordem de m log n, como m varia de 1 até n, concluímos que o método é O(n log n).

### *7.6 Quicksort*

Da mesma forma que o método *Mergesort* o método *Quicksort* baseia-se na técnica de dividir para conquistar. Nesse método deve-se escolher um elemento, chamado de pivô, em uma posição específica dentro do vetor, por exemplo, o primeiro elemento.

Suponha que os elementos do vetor estejam separados (particionados) de modo que a seja colocado na posição j e as seguintes condições sejam observadas:

- os elementos das posições 0 à j são menores do que pivô.
- os elementos das posições j+1 a n-1 sejam maiores do que o pivô.

Com essas duas condições satisfeitas o elemento a estará em sua posição final depois que o vetor estiver ordenado.

Se o processo é repetido para os subvetores vetor[0]... vetor[j-1] e vetor[j+1]... vetor[n-1] e com quaisquer subvetores criados pelo método em recursivas iterações, o resultado será o vetor ordenado.

Ao final do processo recursivo tem-se o arquivo ordenado. Para realizar esse processo descrito do método Quicksort, suponha a função:

```
public static void quicksort(int x[], int li, int ls) {
/*li=limite inferior do vetor
  ls=limite superior do vetor*/

 int j;
 j = particiona (x, li, ls);
 quicksort(x, li, j-1);
 quicksot(x, j+1, ls);
 }
```

Código Fonte 7.10 – Exemplo de função recursiva para o método quicksort.
Fonte: Elaborado pelo autor (2019)

Nessa função recursiva *Quicksort*(), é utilizada a função particiona() para determinar a posição correta do pivô dentro do vetor, retornada com o valor j. Assim, depois deve-se aplicar novamente a função *quicksort* para o subvetor formado pelos elementos vetor[0] a vetor[j-1], que serão todos menores que o pivô, e para o sub-vetor formado pelos elementos vetor[j+1] a vetor[n-1], formado por todos os elementos maiores do que o pivô.

Podemos resumir da seguinte forma:

```
quicksort (vetor, limite inferior, limite superior)
      Particionar vetor:
            descobrir posição do pivô no vetor (j)
            separar maiores e menores elementos em relação ao pivô
      quicksort (vetor, limite inferior, j-1)
      quicksort (vetor, j+1, limite superior)
```

Código Fonte 7.11 – Resumo da função recursiva *quicksort*()
Fonte: elaborado pelo autor (2019)

Para implementar o Quicksort com recursão, falta ainda definir condição de parada que, então, seria quando o limite inferior se tornasse maior que o limite superior.

A Figura Esquema da aplicação método Quicksort apresenta um exemplo de um vetor em que o método Quicksort é aplicado. Escolhendo como pivô o primeiro elemento, ou seja, o valor 25, o vetor é dividido em 2 subvetores, o primeiro subvetor com elementos menores do que 25, que é formado por apenas 1 elemento, portanto já se encontra ordenado. Dessa forma, apenas deve-se repetir o processo para o subvetor com os elementos maiores do que 25.

Reaplicando o método para o subvetor composto pelos elementos maiores do que 25 (pela aplicação da função particiona() descrita posteriormente esse vetor fica como apresentado na figura). Inicia com a escolha do pivô=48, e mais uma vez o vetor é dividido.

O subvetor com elementos menores do que 48 é composto pelos elementos 33 e 37. Com o pivô =33 a divisão tem apenas o subvetor com 1 elemento e essa parte do vetor conclui já ordenada. Assim, fica ainda faltando ordenar o subvetor maior do que 48, o que é feito repetindo o processo, escolhendo o pivô=92.

Dessa vez é gerado apenas o vetor com elementos menores do que 92, formado por 57 e 86. Agora com o pivô =57 a divisão tem apenas o subvetor com 1 elemento maior concluindo a ordenação.

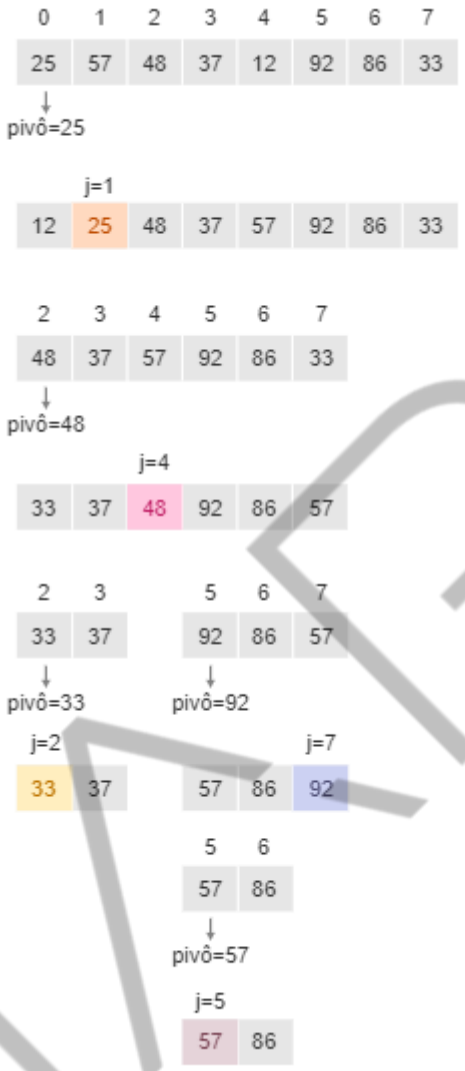



Figura 7.12 – Esquema da aplicação método quicksort
Fonte: Elaborado pelo autor (2019).

O programa a seguir mostra a implementação completa da aplicação do método Quicksort.

```
import java.util.Random;
import java.util.Scanner;

public class QuickSort {

     public static int N = 8;
          //define quantidade de elementos do vetor

     public static void main(String[] args)  {

          Scanner entrada = new Scanner(System.in);
          Random gerador = new Random();
```

```
        int i;

    /*cria a estrutura de dados (vetor) com N elememento*/
        int vetor[] = new int[N];
        System.out.println("Criando vetor com " + N + " elememento");
        for(i = 0; i<N; i++)
            vetor[i] = (int)gerador.nextInt()/10000;



        System.out.println("Ordenando o vetor criado...");
        quicksort(vetor, 0, N-1);

        for(i = 0; i<N; i++)
            System.out.println(i +"\t"+vetor[i]);
        entrada.close();
    }



    public static int particiona (int x[], int li, int ls)
        {
        int pivo,down,temp,up;
        pivo=x[li];
        System.out.println("pivo: " + pivo);
        up=ls;
        down=li;
        while(down<up)
            {
            while(x[down]<=pivo && down<ls)
                down++;
            while (x[up]>pivo)
                up--;
            if (down<up){
                temp=x[down];
                x[down]=x[up];
                x[up]=temp;
                }
            }
        x[li]=x[up];
        x[up]=pivo;
        return up;
    }


    public static void quicksort(int x[],int li,int ls)
        {
        int j;
        if (li<ls){
            j = particiona(x, li, ls);
            quicksort(x, li, j-1);
            quicksort(x, j+1,ls);
            }
```

```
            }

    }
```

Código Fonte 7.12 – Criação do vetor de Incrementos usado pelo método Quicksort escrito em JAVA.
Fonte: elaborado pelo autor (2019)

A função particiona() tem como objetivo encontrar a posição exata onde deve ser colocado o pivô quando o vetor estiver já ordenado.

A ideia é definir um pivô a tal que, pivô = vetor[li], sendo que li é o limite inferior do vetor em que o pivô seria o 1º elemento.

Para tanto, serão utilizados 2 ponteiros up e down, inicializados como:

- up = ls (limite superior)
- down = li (limite inferior)

Esses ponteiros são movimentados um em direção ao outro seguindo os seguintes passos:

1) down é incrementado até vetor[down] > pivô

2) up é decrementado até vetor[up] ≤ pivô

3) se up < down então trocar vetor[down] por vetor[up]

Esses três passos devem ser repetidos enquanto down < up. Quando atingida a condição de parada, o ponteiro up indica a posição final de a que deve, então, ser trocado, ou seja, trocar vetor[up] e vetor[li].

- **Eficiência do Quicksort**

Em um arquivo com n registros, supondo que o pivô fique sempre no meio, tem-se em relação ao número de comparações:

n + 2*(n/2) + 4*(n/4) + 8*(n/8)+...+ n*(n/n)

n + n + n + n + n... (m vezes) => m * n

Se $n= 2^m$ => $m = \log_2 n$

Assim, m*n = log n * n => O(n log n)

Essa é uma função de eficiência considerada relativamente eficiente.

Outra condição é quando o vetor já está ordenado e o pivô é vetor[li]. Nessas condições, verifica-se que o pivô está na posição correta e o subvetor a ser ordenado é 1 a n-1 elementos. Assim, deverão ser ordenados (n-1) subvetores, sendo que o 1º com n, o 2º com n-1 e assim por diante.

n + (n-1) + (n-2) + ...+ 2 => n*n + constante => $O(n^2)$

Portanto, com essa análise de eficiência fica claro que o *Quicksort* é um método que é mais eficiente quando o arquivo se encontra totalmente desordenado.

Uma forma de se obter uma eficiência razoável quando o arquivo está ordenado ou quase ordenado é fazer com que a escolha do pivô seja diferente. Uma solução adotada é escolher o pivô como a mediana. Existe nesse caso ainda possibilidade de ter-se a eficiência $O(n^2)$, mas sendo muito mais rara.

## 7.7 Exercícios Propostos

1) Uma forma de analisar como muda a eficiência do método Shellsort em função do vetor de incrementos é aplicando o método sobre o mesmo vetor com valores dos 1.000 elementos gerados aleatoriamente contando o número de comparações (neste caso específico entre elem e v[cont3]):

- Com vetor de incremento incre = {3,1}
- Gerando os incrementos pelo algoritmo proposto por Knuth.

Para poder comparar deve-se usar o mesmo vetor assim é necessário alterar a criação do vetor:

```
int N =1000;
int vetor[] = new int[N];
int vetork[] = new int[N];
System.out.println("Criando vetor com 1000 elementos: ");
for(i = 0; i<N; i++) {
      vetor[i] = (int)gerador.nextInt()/10000;
      vetork[i] = vetor[i];
       }
```

Inserir contador de comparações no loop com incrementos 3 e 1:

```
int comp=0;
for(cont1 = 0; cont1<n_incre; cont1++)   {
      dist = incre[cont1];
      for(cont2 = dist; cont2<N; cont2++)   {
```

```
            elem = vetor[cont2];
            for(cont3 = cont2-dist;(cont3>= 0) && (elem<vetor[cont3]); cont3 =
cont3 - dist) {
                  comp++;
                  vetor[cont3+dist] = vetor[cont3];
                  }
            vetor[cont3+dist] = elem;
            }
          }
```

Inserir contador de comparações no loop com incrementos Knuth:

```
int compk=0;
for(cont1 = 0; cont1<limite; cont1++)   {
      dist = incre[cont1];
      for(cont2 = dist; cont2<N; cont2++)   {
            elem = vetork[cont2];
            for(cont3 = cont2-dist;(cont3>= 0) && (elem<vetork[cont3]);cont3 =
cont3 - dist) {
                  compk++;
                  vetork[cont3+dist] = vetork[cont3];
                  }
            vetor[cont3+dist] = elem;
            }
      }
```

Fazendo a execução e criando vetor com 1000 elementos:

Número de comparações com incrementos 3 e 1: 92736

Vetor de incrementos gerado por Knuth

0 364

1 121

2 40

3 13

4 4

5 1

Número de comparações usando Knuth: 52053

2) Qual a diferença de abordagem entre os métodos de classificação por troca e seleção?

O método por troca, como o Bubblesort, baseia-se na comparação direta (e possível troca) de cada elemento com os demais elementos do vetor.

Já o método de classificação efetua um pré-processamento gerando uma “fila de prioridade” para que dessa “fila” seja selecionado e posicionado o elemento no vetor ordenado.

3) É adequado aplicar o método de balanceamento de ABB para implementar este método? Explique.

Principalmente, pelo fato de ser apenas viável o uso de vetor para armazenar a ABB é preciso que a árvore seja gerada de forma balanceada, pois assim o vetor não teria posições "vazias". Diminuindo, desta forma o uso de memória.

4) Identifique na função gera_heap as operações críticas. Execute a função para o seguinte vetor e verifique quantas vezes as operações críticas são executadas.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 87 | 23 | 17 | 32 | 45 | 21 | 55 | 67 |

```
int elem,pai,filho,iv;
for(i=1;i<N;i++){
	elem=vetor[i];
	filho=i;
	pai=(filho-1)/2;
	while(filho>0 && vetor[pai]<elem){ //OPERAÇÃO SIGNIFICATIVA COMPARAÇÃO
		vetor[filho]=vetor[pai];  // OPERAÇÃO SIGNIFICATIVA DE TROCA
		filho=pai;
		pai=(filho-1)/2;
		}
	vetor[filho]=elem; //OPERAÇÃO SIGNIFICATIVA DE TROCA
	}
```

Configuração da montagem da heap a cada iteração de i:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 87 | 23 | 17 | 32 | 45 | 21 | 55 | 67 |
| 87 | ~~23~~ | 17 | ~~32~~ | 45 | 21 | 55 | 67 |
| 87 | ~~32~~ | 17 | 23 | ~~45~~ | 21 | 55 | 67 |
| 87 | 45 | ~~17~~ | 23 | 32 | ~~21~~ | 55 | 67 |
| 87 | 45 | ~~21~~ | 23 | 32 | 17 | ~~55~~ | 67 |
| 87 | ~~45~~ | 55 | ~~23~~ | 32 | 17 | 21 | ~~67~~ |
| 87 | 67 | 55 | 45 | 32 | 17 | 21 | 23 |

Comparações = 13

Trocas = 6

5) Elabore um programa que conte o número de comparações e trocas efetuadas no programa *Quicksort* apresentado neste texto e aplique o método sobre o vetor do exercício anterior.

As comparações e trocas ocorrem apenas no particionamento do vetor assim, apenas na função particiona é a contagem deve ser feita. Apenas somar cont e troca de cada chamada de particiona.

```
public static int particiona (int x[], int li, int ls)
	{
	int pivo,down,temp,up;
	int cont = 0;
	int troca = 1;
	pivo=x[li];
	System.out.println("pivo: " + pivo);
	up=ls;
	down=li;
	while(down<up)
		{
		while(x[down]<=pivo && down<ls) {
			cont++;
			down++;
			}
		 cont++; // conta também quando comparou mas não entrou no loop
		while (x[up]>pivo) {

			up--;
			cont++;
			}
		if (down<up){
			troca++;
			temp=x[down];
			x[down]=x[up];
			x[up]=temp;
			}
		}
		x[li]=x[up];
		x[up]=pivo;
		System.out.println("troca: " + troca + "comparações: " + cont);
		return up;
		}
```

# REFERÊNCIAS

ASCÊNCIO, A. F. G.; ARAUJO, G. S. **Estruturas de dados:** algoritmos, análise de complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice-Hall, 2010.

DEITEL, P. J.; DEITEL, H. M. **Java:** como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

PEREIRA, S. L. **Estruturas de dados fundamentais:** conceitos e aplicações. São Paulo: Érica, 1996.

TENEMBAUM, A. M. et al. **Estruturas de dados usando C.** São Paulo: Makron Books, 1995.