CAPÍTULO II – ORDENAÇÃO

II.1 Introdução

A ordenação é uma operação fundamental em Ciência da Computação, pois muitos programas a utilizam como uma etapa intermediária. Por isso, foram desenvolvidos muitos algoritmos de ordenação.

☺ O que é ordenar?

Rearranjar um conjunto de objetos em uma ordem crescente ou decrescente.

☺ Objetivo

Facilitar a recuperação posterior, dos dados armazenados.
Pesquisar e recuperar dados.

☺ Como os algoritmos trabalham?

Sobre os registros de um arquivo, através de uma **"chave"** de pesquisa que é um campo especial do registro que controla a ordenação. O registro pode conter outros campos que independem da **"chave**". Em JAVA, esses registros são representados por objetos de uma classe.

```
class Item {
  private tipoChave chave;
  // outros atributos
  // construtor(es) e métodos para manipular os atributos,
  // dentre eles:
  public tipoChave getChave ( ){
    return chave;
  }
}
```

O tipo da **CHAVE** pode ser qualquer um que se possa ordenar de forma bem definida. A ordem mais comum é numérica ou alfabética.

Existem vários algoritmos de ordenação. A escolha do melhor algoritmo para uma aplicação depende do número de itens a ser ordenado, de quantos itens já estão ordenados de algum modo, de possíveis restrições aos valores dos itens, do dispositivo de armazenamento utilizado, etc.

☺ Ambiente de Classificação -

O meio de armazenamento afeta o processo de classificação

- ***Ordenação interna*** – Ocorre quando o arquivo a ser ordenado cabe todo na memória principal. O número de registros é pequeno o suficiente para caber em um array, definido por uma linguagem de programação. Neste tipo de ordenação, qualquer registro pode ser imediatamente acessado.

- ***Ordenação externa*** – Ocorre quando o arquivo não cabe na memória principal e deve ser armazenado em fita ou disco (memória externa – tempo maior). Neste tipo de ordenação, os registros são acessados seqüencialmente ou em grandes blocos.

A maioria dos métodos de ordenação é baseada em comparação de chaves.

☺ Medidas de complexidade relevantes

Para se escolher um algoritmo de ordenação, é necessário saber o tempo gasto para ordenar um arquivo. Para um algoritmo de ordenação interna, é necessário saber:

- ***Comparações de chaves - C (n)***
- ***Movimentações de itens - M (n)***

Onde ***n*** é o número de itens do arquivo.

Também é necessário saber a quantidade de memória auxiliar utilizada pelo algoritmo, pois é necessário fazer uso de maneira econômica da memória.

☺ Apresentação do resultado

☺ Contiguidade física

25	**37**	**15**	**12**	**20**
1	**2**	**3**	**4**	**5**

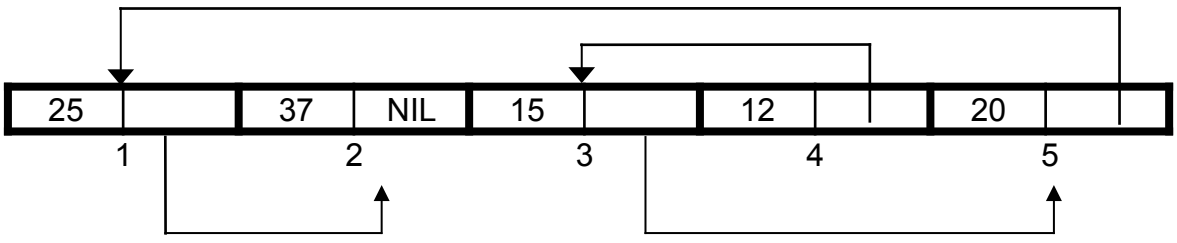
12	**15**	**20**	**25**	**37**
1	**2**	**3**	**4**	**5**

☺ Vetor indireto de ordenação

25	**37**	**15**	**12**	**20**
1	**2**	**3**	**4**	**5**

4	**3**	**5**	**1**	**2**
1	**2**	**3**	**4**	**5**

☺ Encadeamento



Primeiro = 4

Os métodos preferidos são aqueles que utilizam **vetor** como estrutura de dados e fazem **permutação de itens no próprio veto**r. Os métodos que utilizam listas encadeadas são utilizados em situações muito especiais porque utilizam ***n*** palavras extras para armazenar os apontadores. Os métodos que precisam de uma quantidade extra de memória para armazenar uma outra cópia dos itens que serão ordenados são menos importantes.

☺ Algoritmos de Ordenação Interna

- **Métodos simples –** O (n^2) comparações – arquivos pequenos. Produzem programas pequenos e fáceis de entender por isso, em muitos casos, é melhor usar métodos mais simples do que mais sofisticados.

- **Métodos eficientes –** O (n logn) comparações - arquivos grandes. As comparações usadas são mais complexas nos detalhes.

Ao implementar um algoritmo de ordenação interna em Java será utilizada a seguinte classe:

```
class Dados{
     private Item[] vetor; //referência a um vetor de itens
     private int nElem;  //número de itens de dados
     //construtor(es)  e métodos desta classe
}
```

☺ Método de Ordenação Estável

Um método de ordenação é dito ser estável quando a ordens dos itens com **chaves** iguais mantém-se inalterada pelo processo de ordenação.

Por exemplo, uma lista, em ordem crescente de matrícula dos funcionários de uma empresa, é ordenada pelo campo nome. Se o método de ordenação utilizado for estável, ele produzirá uma lista onde funcionários com o mesmo nome aparecerão ordenados por matrícula.

18	**7**	**10**	**7**	**3**
Andréa Costa	**João da Silva**	**João da Silva**	**Luiz Dantas**	**Maria Freitas**
1	*2*	*3*	*4*	*5*

Alguns métodos mais eficientes não são estáveis. Mas, para um método não-estável, a estabilidade pode ser forçada, se ela for importante.

II.2 SELEÇÃO DIRETA (SELECTION SORT)

É um dos algoritmos mais simples de ordenação.

FUNCIONAMENTO:

- Selecione o menor item do vetor

- Troque-o com o item da primeira posição
- Repita a operação com os n-1 itens restantes
- Depois com os n-2 itens, até que reste apenas 1 item,

Exemplo:

Vamos mostrar, passo a passo, um exemplo de uso deste algoritmo:

520	450	254	310	285	179	652	351	423	861
i					min				
179	450	254	310	285	520	652	351	423	861
	i	min							
179	**254**	450	310	285	520	652	351	423	861
		i		min					
179	**254**	**285**	310	450	520	652	351	423	861
			i						
			min						
179	**254**	**285**	**310**	450	520	652	351	423	861
				i			min		
179	**254**	**285**	**310**	**351**	520	652	450	423	861
					i			min	
179	**254**	**285**	**310**	**351**	**423**	652	450	520	861
						i	min		
179	**254**	**285**	**310**	**351**	**423**	**450**	652	520	861
							i	min	
179	**254**	**285**	**310**	**351**	**423**	**450**	**520**	652	861
								i	
								min	
179	**254**	**285**	**310**	**351**	**423**	**450**	**520**	**652**	**861**

→ Ordenado!!!

Baseado no exemplo anterior e no princípio de funcionamento do método, nós vamos desenvolver o algoritmo de ordenação por seleção em Java, supondo que este método está na classe Dados:

```
public void seleçãoDireta (){
  int i, j, minimo;
  Item temp;
  for (i=0; i< this.nElem-1;i++){
    minimo = i;
    for (j=i+1; j< this.nElem; j++)
      if (this.vetor[j].getChave ()< this.vetor[minimo].getChave())
        minimo = j;
    temp = this.vetor[minimo];
    this.vetor[minimo] = this.vetor[i];
    this.vetor[i] = temp;
  }
}
```

Após o desenvolvimento do algoritmo, vamos calcular seu custo:

COMPARAÇÕES

$$C(n) = \frac{n^2}{2} - \frac{n}{2}$$

MOVIMENTAÇÕES

$$M(n) = 3(n - 1)$$

Observem que o número de movimentações é linear no tamanho da entrada

CONSIDERAÇÕES SOBRE O MÉTODO:

- Número de movimentações é linear
- Bom para arquivos com registros grandes
- Bom para arquivos com até 1000 registros se a chave tem tamanho igual a 1 palavra
- Se o arquivo já está ordenado ou quase, isso não ajuda em nada. C(n) continua quadrático
- Algoritmo não é estável.

Exemplo da não estabilidade do algoritmo:

Seja o vetor abaixo, onde o primeiro campo é um código e o segundo um nome. Supondo que o vetor esteja ordenado por código e deseja-se reordená-lo por nome. Se o algoritmo fosse estável, ao aparecer nomes iguais, viria primeiro aquele com menor código.

1	2	3	4	5	6
KÁTIA	CARLOS	LUCAS	JOSÉ	LUCAS	JÚLIA

Ao simular o algoritmo, notamos que o vetor final, ordenado por nome ficaria:

2	4	6	1	5	3
CARLOS	JOSÉ	JÚLIA	KÁTIA	LUCAS	LUCAS

Portanto, este algoritmo não é estável.

II.3 BOLHA (BUBBLESORT)

É um método simples e de fácil entendimento e implementação. É um dos mais conhecidos e difundidos métodos de ordenação de arranjos.

Mas, não é um algoritmo eficiente. Ele é estudado apenas visando o desenvolvimento de raciocínio.

O princípio do Bubblesort é a troca de valores entre posições consecutivas, fazendo com que os valores mais altos (ou mais baixos) "borbulhem" para o final do arranjo (daí o nome Bubblesort).

FUNCIONAMENTO:

- Chaves na posição 1 e 2 são comparadas. Se tiverem fora de ordem são trocadas.
- Logo após repete o processo com as chaves 2 e 3, 3 e 4, ..., n–1 e n.
- Começa o processo de novo de 1 até a comparação entre n–2.
- Repita o processo até que sobrem apenas as 2 primeiras chaves

Vamos mostrar, passo a passo, um exemplo de uso deste algoritmo:

520	450	254	310	285	179	652	351	423	861
j									Lsup
450	**520**	254	310	285	179	652	351	423	861
j									Lsup
450	254	**520**	310	285	179	652	351	423	861
	j								Lsup
450	254	310	**520**	285	179	652	351	423	861
		j							Lsup
450	254	310	285	**520**	179	652	351	423	861
			j						Lsup
450	254	310	285	179	**520**	652	351	423	861
				j					Lsup
450	254	310	285	179	520	351	**652**	423	861
						j			Lsup
450	254	310	285	179	520	351	423	**652**	861
							j		Lsup
450	254	310	285	179	520	351	423	**652**	**861**
j							Lsup		
254	**450**	310	285	179	520	351	423	**652**	**861**
j							Lsup		
254	310	**450**	285	179	520	351	423	**652**	**861**
	j						Lsup		
254	310	285	**450**	179	520	351	423	**652**	**861**
		j					Lsup		
254	310	285	179	**450**	520	351	423	**652**	**861**
			j				Lsup		
254	310	285	179	450	351	**520**	423	**652**	**861**
					j		Lsup		
254	310	285	179	450	351	423	**520**	**652**	**861**
						j	Lsup		
254	**310**	285	179	450	351	423	**520**	**652**	**861**
j						Lsup			
254	285	**310**	179	450	351	423	**520**	**652**	**861**
	j					Lsup			
254	285	179	**310**	450	351	423	**520**	**652**	**861**
		j				Lsup			
254	285	179	310	351	**450**	423	**520**	**652**	**861**
				j		Lsup			
254	285	179	310	351	423	**450**	**520**	**652**	**861**
					j	Lsup			
254	**285**	179	310	351	423	**450**	**520**	**652**	**861**
j					Lsup				
254	179	**285**	310	351	423	**450**	**520**	**652**	**861**
	j				Lsup				
254	179	**285**	**310**	**351**	**423**	**450**	**520**	**652**	**861**
j	Lsup								

→ Ordenado!!!

179	254	285	310	351	423	450	520	652	861

Lsup

j

Baseado no exemplo anterior e no princípio de funcionamento do método, nós vamos desenvolver o algoritmo Bubblesort:

```
public void bubblesort (){
  int LSup, i, j;
  Item temp;

  LSup = this.nElem-1;
  do{
    j = 0;
    for (i = 0; i < LSup; i++)
      if (this.vetor[i].getChave() > this.vetor[i+1].getChave()){
        temp = this.vetor[i];
        this.vetor[i] = this.vetor[i+1];
        this.vetor[i+1] = temp;
        j = i;
      }
    LSup = j;
  }while (LSup >= 1);
}
```

Após o desenvolvimento do algoritmo, vamos calcular seu custo:

COMPARAÇÕES

MELHOR CASO: Vetor já está ordenado

$$C(n) = n - 1$$

PIOR CASO: Vetor ordenado em ordem contrária

$$C(n) = \frac{n^2}{2} - \frac{n}{2}$$

MOVIMENTAÇÕES

MELHOR CASO: Vetor já está ordenado.

$$M(n) = 0$$

PIOR CASO: Vetor ordenado em ordem contrária

$$M(n) = \frac{3n^2}{2} - \frac{3n}{2}$$

CONSIDERAÇÕES SOBRE O MÉTODO:

- Parece o algoritmo de Seleção
- Faz muitas trocas, o que o torna o menos eficiente dos métodos Simples ou Diretos
- É um método estável
- Se o arquivo estiver quase ordenado ele costuma ser eficiente, mas deve-se tomar cuidado com o caso em que o arquivo está ordenado, com exceção do menor elemento que está na última posição. Neste caso, o algoritmo fará o mesmo número de comparações do pior caso.

- É um método lento, pois só compara posições adjacentes.
- Cada passo aproveita muito pouco do passo anterior
- Comparações redundantes, pois o algoritmo é linear e obedece a uma sequência fixa de comparações.

II.4 COQUETELEIRA (SHAKERSORT)

Este método é semelhante ao Bubblesort. Seu objetivo é melhorar o desempenho do Bubllesort alternando o sentido do movimento a cada passo e mantendo os indicadores das posições da última troca.

FUNCIONAMENTO:

- Sejam ***esq*** (indicador da esquerda), ***dir*** (indicador da direita) e ***K*** (indicador da última troca).
- *Esq = 2* e *dir = n*
- A primeira passada é feita do final para o começo do arquivo, a segunda do começo para o final e assim sucessivamente, colocando os menores no início e os maiores no final.
- Assim, a cada passada do final para o começo do vetor, está garantido que os elementos a_1, a_2, ..., a_{k-1} estão ordenados, então *esq:*= *k*+1. Da mesma forma, a cada passada do início para o final, garante que os elementos a_k, a_{k+1}, ..., a_n estão ordenados, então *dir:*= *k*–1.

Vamos mostrar, passo a passo, um exemplo de uso deste algoritmo:

520	450	254	310	285	179	652	**351**	423	**861**
	esq								**dir/j**
520	450	254	310	285	179	**351**	652	423	**861**
	esq						**j**		**dir**
520	450	254	310	**179**	285	351	652	423	**861**
	esq				**j**				**dir**
520	450	254	**179**	310	285	351	652	423	**861**
	esq			**j**					**dir**
520	450	**179**	254	310	285	351	652	423	**861**
	esq		**j**						**dir**
520	**179**	450	254	310	285	351	652	423	**861**
	esq	**j**							**dir**
179	520	450	254	310	285	351	652	423	**861**
	esq/j								**dir**
179	**520**	450	254	310	285	351	652	423	**861**
	j	**esq**							**dir**
179	450	**520**	254	310	285	351	652	423	**861**
		esq/j							**dir**
179	450	254	**520**	310	285	351	652	423	**861**
		esq	**j**						**dir**
179	450	254	310	**520**	285	351	652	423	**861**
		esq		**j**					**dir**

179	450	254	310	285	**520**	351	652	423	**861**
		esq			j				dir
179	450	254	310	285	351	**520**	652	423	**861**
		esq				j			dir
179	450	254	310	285	351	520	423	**652**	**861**
		esq						j	dir
179	450	254	310	**285**	351	520	**423**	**652**	**861**
		esq					dir	j	
179	450	254	310	**285**	351	**423**	520	**652**	**861**
		esq					dir/j		
179	450	254	**285**	310	351	423	**520**	**652**	**861**
		esq		j			dir		
179	**254**	450	285	310	351	423	**520**	**652**	**861**
		esq/j					dir		
179	**254**	285	**450**	310	351	423	**520**	**652**	**861**
		j	esq				dir		
179	**254**	285	310	**450**	351	423	**520**	**652**	**861**
			esq	j			dir		
179	**254**	285	310	351	**450**	423	**520**	**652**	**861**
			esq		j		dir		
179	**254**	285	310	351	423	**450**	**520**	**652**	**861**
			esq		dir	j			
179	**254**	**285**	**310**	**351**	**423**	**450**	**520**	**652**	**861**
					dir	j	esq		

→ Ordenado!!!

Baseado no exemplo anterior e no princípio de funcionamento do método, nós vamos desenvolver o algoritmo Shakersort :

```
public void shakersort (){
  int esq, dir, i, j;
  Item temp;
  esq = 1;
  dir = this.nElem - 1;
  j = dir;
  do{ //leva as menores chaves para o início
    for (i = dir ; i >= esq; i-- )
      if (this.vetor[i-1].getChave() > this.vetor[i].getChave()){
        temp = this.vetor[i];
        this.vetor[i] = this.vetor[i-1];
        this.vetor[i-1] = temp;
        j = i;
      }
    esq = j+1;
    //leva as maiores chaves para o final
    for (i = esq ; i <= dir; i++)
      if (this.vetor[i-1].getChave() > this.vetor[i].getChave()){
        temp = this.vetor[i];
        this.vetor[i] = this.vetor[i-1];
        this.vetor[i-1] = temp;
        j = i;
    }
```

```
    dir = j-1;
  }while (esq <= dir);
}
```

Após o desenvolvimento do algoritmo, vamos calcular seu custo:

COMPARAÇÕES

MELHOR CASO: Vetor já está ordenado

$$C(n) = n - 1$$

PIOR CASO: Vetor ordenado em ordem contrária

$$C(n) = \frac{n^2}{2} - \frac{n}{2}$$

MOVIMENTAÇÕES

MELHOR CASO: Vetor já está ordenado.

$$M(n) = 0$$

PIOR CASO: Vetor ordenado em ordem contrária

$$M(n) = \frac{3n^2}{2} - \frac{3n}{2}$$

CONSIDERAÇÕES SOBRE O MÉTODO:

- Parece o algoritmo BUBBLESORT
- Faz muitas trocas, o que o torna um dos menos eficientes dentre os métodos Simples ou Diretos
- É um método estável

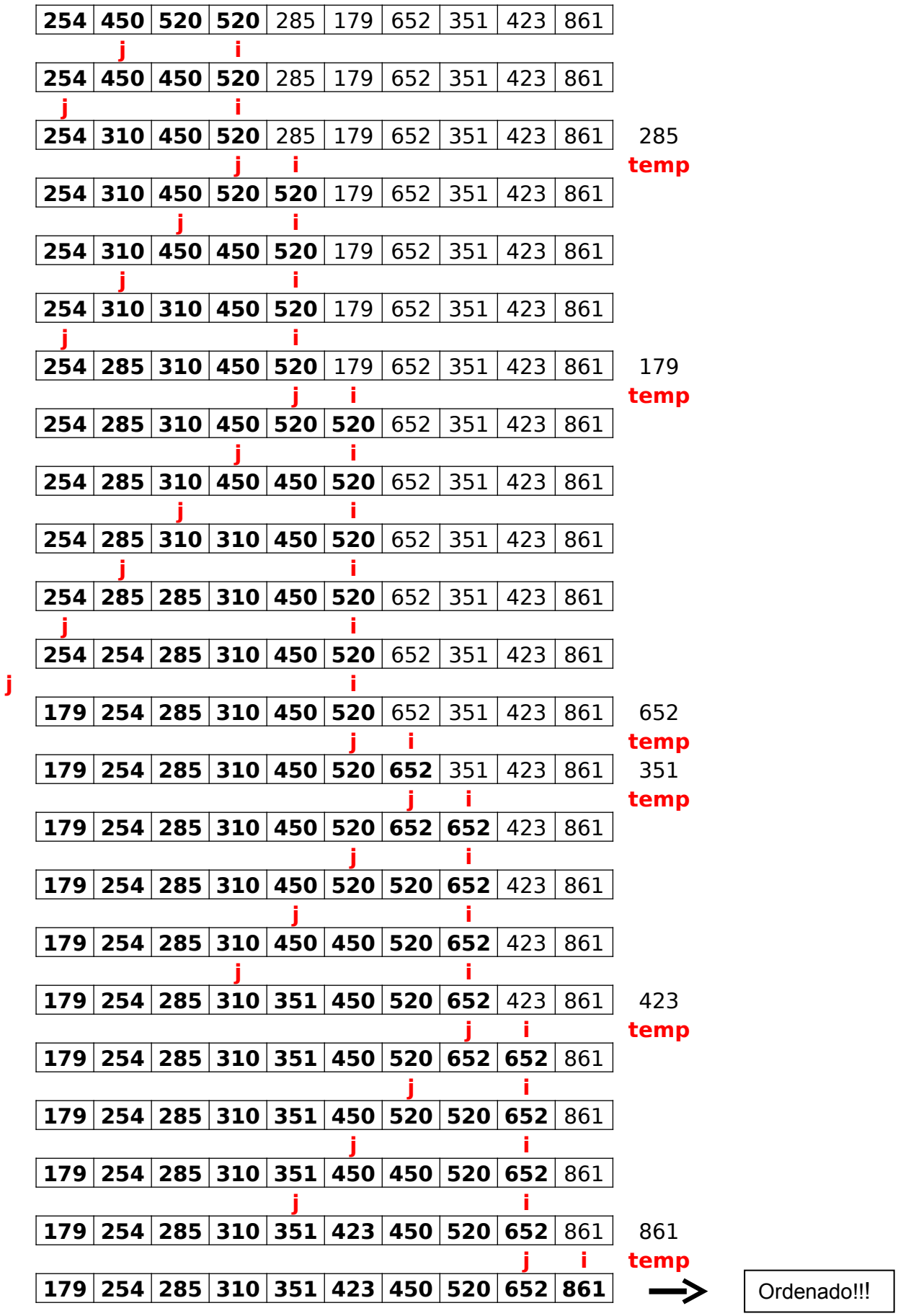
II.5 INSERÇÃO DIRETA (INSERTION SORT)

FUNCIONAMENTO:

- Vetor é dividido em 2 subvetores
- Inicialmente o primeiro segmento contém um único elemento e conseqüentemente está ordenado. O segundo segmento contém os (n-1) elementos restantes.
- A cada passo, a partir de i=2, o i-ésimo elemento é transferido do segundo segmento para o primeiro segmento, sendo inserido na sua posição apropriada.

Exemplo passo a passo:

	520	450	254	310	285	179	652	351	423	861	450
	j	i									**temp**
	450	**520**	254	310	285	179	652	351	423	861	
j		i									
	450	**520**	254	310	285	179	652	351	423	861	254
		j	i								**temp**
	450	**520**	**520**	310	285	179	652	351	423	861	
	j		i								
	450	**450**	**520**	310	285	179	652	351	423	861	
j			i								
	254	**450**	**520**	310	285	179	652	351	423	861	310
			j	i							**temp**



Baseado no exemplo anterior e no princípio de funcionamento do método, nós vamos desenvolver o algoritmo de Inserção Direta:

```
public void inserçãoDireta(){
    int i, j;
    Item temp;

    for (i=1; i < this.nElem; i++){
        temp = this.vetor[i];
        j = i-1;
        while ((j >= 0) &&
                (this.vetor[j].getChave() > temp.getChave())){
            this.vetor [j+1] = this.vetor[j];
            j--;
        }
        this.vetor [j+1] = temp;
    }
}
```

Após o desenvolvimento do algoritmo, vamos calcular seu custo:

COMPARAÇÕES

MELHOR CASO: Vetor já está ordenado

$$C(n) = n - 1$$

PIOR CASO: Vetor ordenado em ordem contrária

$$C(n) = \frac{n^2 - n}{2}$$

MOVIMENTAÇÕES

MELHOR CASO: Vetor já está ordenado.

$$M(n) = 2(n - 1)$$

PIOR CASO: Vetor ordenado em ordem contrária

$$M(n) = \frac{n^2 + 3n - 4}{2}$$

CONSIDERAÇÕES SOBRE O MÉTODO:

- Bom para vetores quase ordenados
- Bom quando se deseja adicionar poucos itens, de forma ordenada, a um arquivo já ordenado, pois a ordem, neste caso, é linear.
- Método estável.

II.6 SHELLSORT

Este método é uma extensão do algoritmo de ordenação por inserção.

- **Método de inserção** ⇒ Se o menor item está na posição mais à direita, então o número de comparações e movimentações para encontrar seu ponto de inserção é (n – 1).
- **Método Shell** ⇒ Permite troca de registros de posições distantes (h posições). Eles são comparados e trocados. A sequência ordenada para um certo h, dizemos que esta sequência está h-ordenada. Quando h=1, o algoritmo funciona como o algoritmo de inserção.

Exemplo passo a passo:

0	1	2	3	4	5	6	7	8	9		
520	450	254	310	**285**	179	652	351	423	861	285	h = 4
j-4				**i/j**						**temp**	
285	**450**	254	310	520	**179**	652	351	423	861	179	
	j-4				**i/j**					**temp**	
285	179	**254**	310	520	450	**652**	351	423	861	652	
		j-4				**i/j**				**temp**	
285	179	254	**310**	520	450	652	**351**	423	861	351	
			j-4				**i/j**			**temp**	
285	179	254	310	**520**	450	652	351	**423**	861	423	
				j-4				**i/j**		**temp**	
285	179	254	310	**520**	450	652	351	520	861	423	
j-4				**j**				**i**		**temp**	
285	179	254	310	423	**450**	652	351	520	**861**	861	
					j-4				**i/j**	**temp**	
285	179	**254**	310	423	450	652	351	520	861	254	h = 2
j-2		**i/j**								**temp**	
254	**179**	285	**310**	423	450	652	351	520	861	310	
	j-2		**i/j**							**temp**	
254	179	**285**	310	**423**	450	652	351	520	861	423	
		j-2		**i/j**						**temp**	
254	179	285	**310**	423	**450**	652	351	520	861	450	
			j-2		**i/j**					**temp**	
254	179	285	310	**423**	450	**652**	351	520	861	652	
				j-2		**i/j**				**temp**	
254	179	285	310	423	**450**	652	**351**	520	861	351	
					j-2		**i/j**			**temp**	
254	179	285	**310**	423	**450**	652	450	520	861	351	
			j-2		**j**		**i**			**temp**	
254	179	285	310	423	351	**652**	450	**520**	861	520	
						j-2		**i/j**		**temp**	
254	179	285	310	**423**	351	**652**	450	652	861	520	
				j-2		**j**		**i**		**temp**	
254	179	285	310	423	351	520	**450**	652	**861**	861	
							j-2		**i/j**	**temp**	
254	179	285	310	423	351	520	450	652	861	179	h = 1
j-1	**i/j**									**temp**	
179	**254**	285	310	423	351	520	450	652	861	254	
	j-1	**i/j**								**temp**	
179	**254**	**285**	310	423	351	520	450	652	861	...	
		j-1	**i/j**							**temp**	
179	**254**	**285**	**310**	**423**	351	520	450	652	861	351	
				j-1	**i/j**					**temp**	
179	**254**	**285**	**310**	**351**	**423**	520	450	652	861	520	
					j-1	**i/j**				**temp**	

179	254	285	310	351	423	520	450	652	861	450
						j-1	i/j			temp
179	254	285	310	351	423	450	520	652	861	...
							j-1	i/j		temp
179	254	285	310	351	423	450	520	652	861	

→ Ordenado!!!

Já houve várias experiências de sequência s de h. Knuth mostrou, experimentalmente, que existe uma sequência que melhora a eficiência do tempo de execução em cerca de 20%. Esta sequência é:

1, 4, 13, 40, 121, ...

ou seja,

h(s) = 3h(s-1) + 1, para s > 1
h(s) = 1, para s = 1

Baseado no exemplo anterior e no princípio de funcionamento do método, nós vamos desenvolver o algoritmo Shellsort:

```
public void shellsort (){
    int i, j, h;
    Item temp;

    h = 1;
    do{
        h = 3*h+1;
    }while (h < this.nElem);

    do{
        h = h/3;
        for (i=h; i < this.nElem; i++){
            temp = this.vetor[i];
            j = i;
            while (this.vetor[j-h].getChave() > temp.getChave()){
                this.vetor[j] = this.vetor[j-h];
                j -= h;
                if (j < h)
                    break;
            }
           this.vetor [j] = temp;
        }
    }while (h != 1);
}
```

ANÁLISE:

- Ninguém ainda foi capaz de analisar este algoritmo. Portanto, ninguém sabe por que ele é eficiente.
- Uma pergunta comum é: Como escolher os incrementos?
- Não sabemos, mas cada incremento não deve ser múltiplo do anterior.
- Quanto à complexidade, conjeturas apontam para:

1) $C(n) = O(n^{1,25})$
2) $C(n) = O(n (\ln n)^2)$

CONSIDERAÇÕES

- Ótima opção para arquivos com ±5.000 registros
- Implementação é simples
- Quantidade de código é pequena
- O tempo de execução é sensível à ordem inicial do arquivo
- Método não é estável

II.7 QUICKSORT

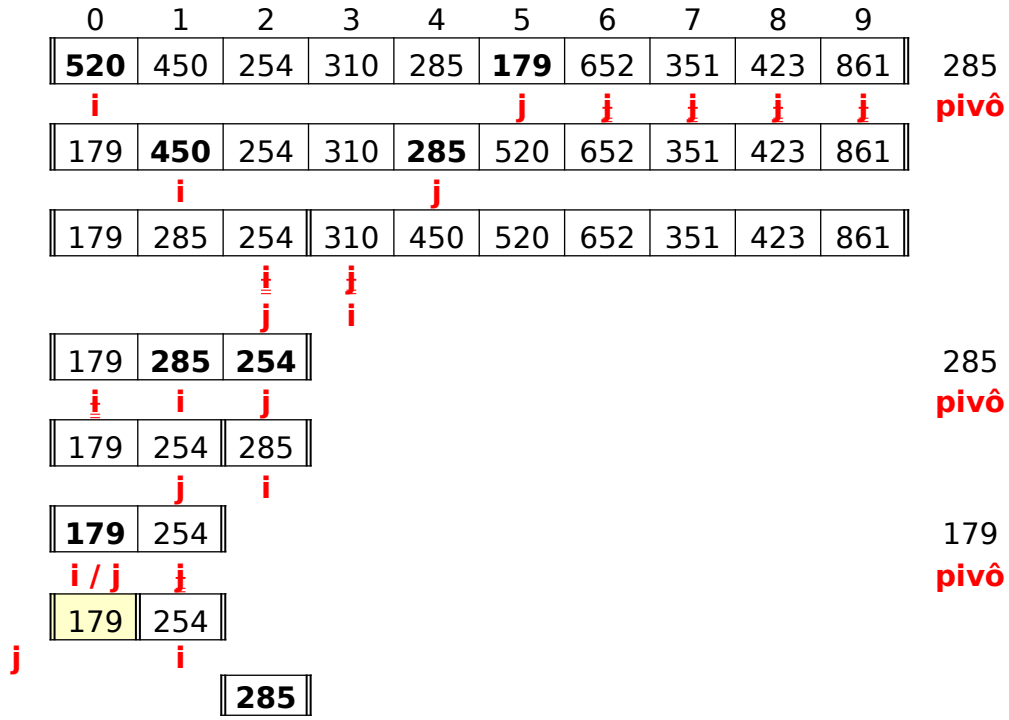
Princípio de funcionamento:

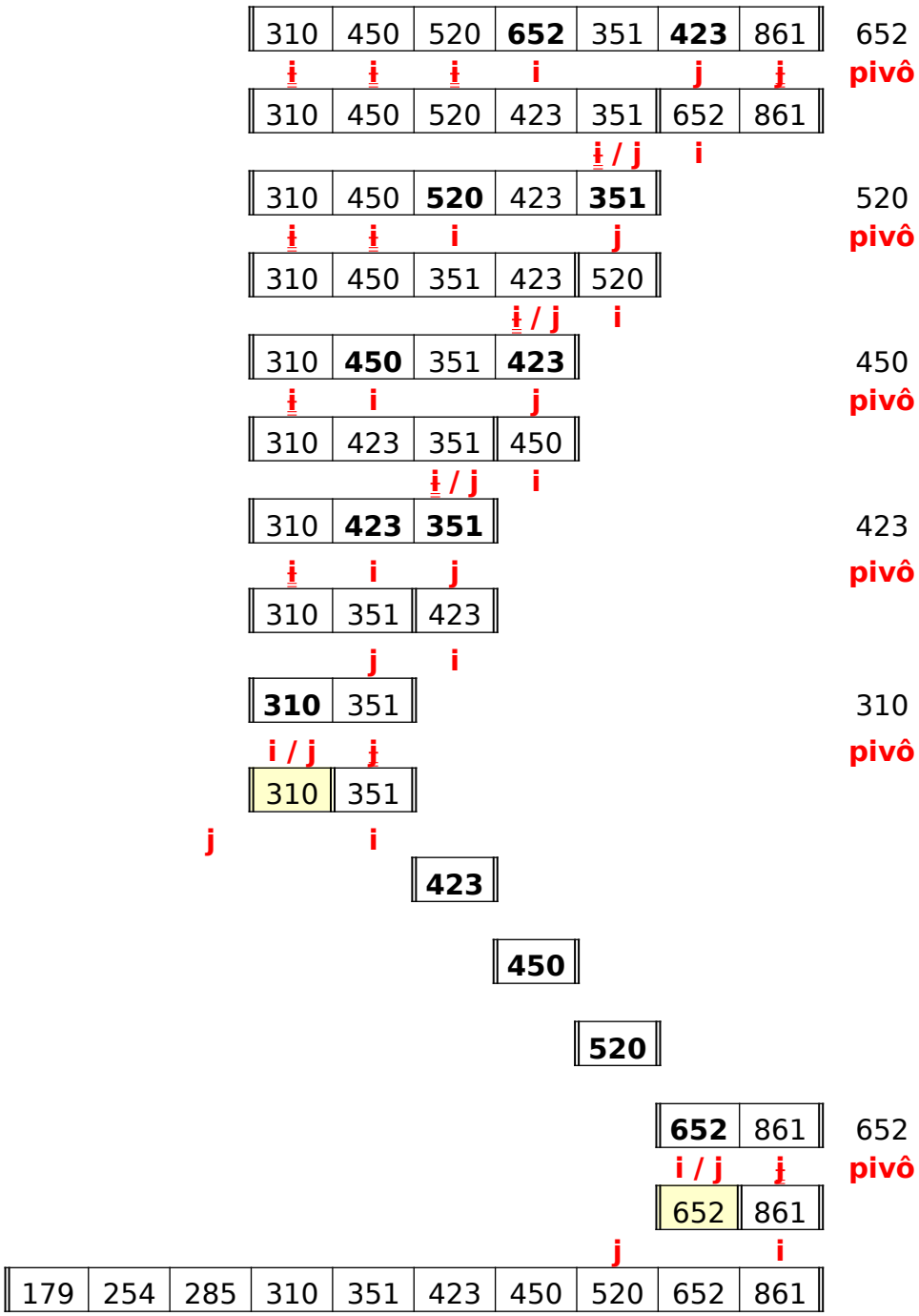
a) escolher arbitrariamente um item do vetor como pivô
b) percorrer o vetor a partir de seu início, até encontrar um item com chave maior ou igual à chave do pivô índice i
c) percorrer o vetor a partir do final, até encontrar um item com chave menor ou igual à chave do pivô índice j
d) trocar os itens v[i] e v[j]
e) continuar o percurso-e-troca até que os dois índices se cruzem

Obs: quando i e j se cruzam, temos dois grupos:

- `v[l],..., v[j] <= pivô`
- `v[j+l],..., v[n] >= pivô`

Exemplo passo a passo:






```
public void quicksort (){
    ordena (0, this.nElem-1);}

private void ordena (int esq, int dir){
    int pivo, i = esq, j = dir;
    Item temp;

    pivo = this.vetor[(i+j)/2].getChave();
    do {
        while (this.vetor[i].getChave() < pivo)
            i++;
        while (this.vetor[j].getChave() > pivo)
            j--;
        if (i <= j) {
```

```
                temp = this.vetor[i];
                this.vetor[i] = this.vetor[j];
                this.vetor[j] = temp;
                i++;
                j--;
            }
        } while (i <= j);
        if (esq < j)
            ordena (esq, j);
        if (dir > i)
            ordena (i, dir);
}
```

PIOR CASO: Escolha como pivô de um dos extremos do arquivo já ordenado. Há n chamadas recursivas, onde será eliminado um elemento por vez. Logo, necessita de uma pilha auxiliar para as chamadas recursivas de tamanho n e, o número de comparações é:

$$C(n) = n^2 / 2$$

MELHOR CASO: Dividir o arquivo ao meio.

$$C(n) = 2C(n/2) + n$$

Onde C(n/2) é o custo de ordenar cada metade e n é o custo de examinar cada item.
Logo, $C(n) \leq 1{,}4\ n \log n$
Sendo, em média, o tempo de execução O(n logn).

CONSIDERAÇÕES:

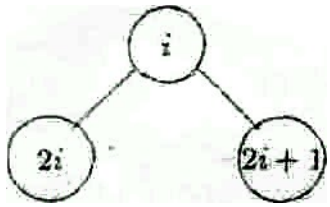
- Não é bom para arquivos já ordenados quando a escolha do pivô não é boa.
- É um algoritmo muito eficiente
- Precisa, em média n logn operações para ordenar n itens
- Necessita de uma pequena pilha como memória auxiliar
- Método não é estável
- A versão recursiva tem como pior caso $O(n^2)$ operações
- Implementação é delicada e difícil. Um pequeno engano pode levar a efeitos inesperados

Como evitar o pior caso?
Escolha 3 itens quaisquer do arquivo e use a mediana dos 3 como item divisor na partição.

II.8 HEAPSORT – Seleção em Árvore

O princípio de funcionamento deste método baseia-se na idéia de uma heap. Uma heap é uma árvore binária com as seguintes características:

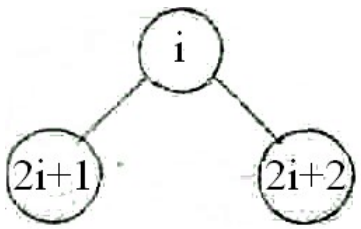
- A maior chave está sempre na raiz da árvore.
- O sucessor à esquerda do elemento de índice **i** é o elemento de índice **2i** e o sucessor à direita é o elemento de índice **2i + 1**, sendo que a chave de cada nó é maior ou igual às chaves de seus filhos.



V[i].chave >= V[2i].chave
V[i].chave >= V[2i +2].chave

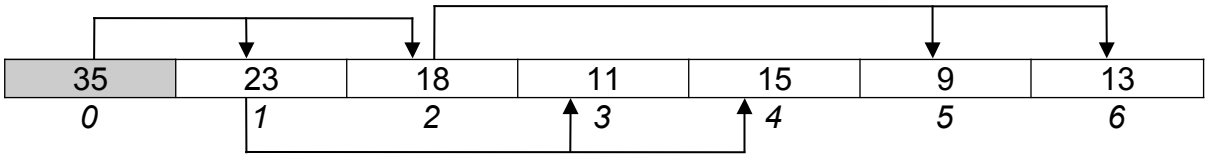
OBS.: Esta relação é baseada em um vetor cujo primeiro elemento encontra-se na posição 1, o que acontece em linguagens como Pascal e Delphi.
Mas, em linguagens como Java, C e C++, onde a primeira posição do vetor é a posição 0, a relação é:

- O sucessor à esquerda do elemento de índice **i** é o elemento de índice **2i+1** e o sucessor à direita é o elemento de índice **2i + 2**, sendo que a chave de cada nó é maior ou igual às chaves de seus filhos.



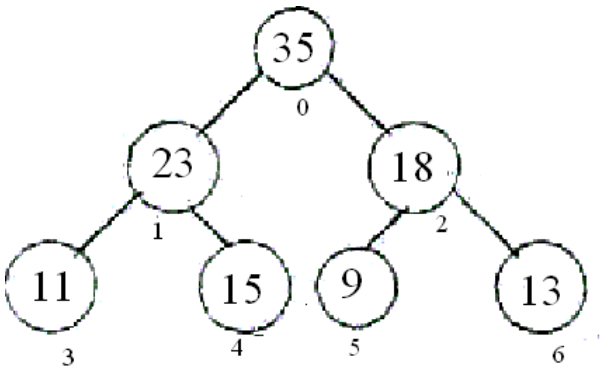
V[i].chave >= V[2i+1].chave
V[i].chave >= V[2i +2].chave

Exemplo: O vetor abaixo é uma HEAP!



Assim, podemos montar a seguinte tabela:

i	0	35
2i+1	1	23
2i+2	2	18
i	1	23
2i+1	3	11
2i+2	4	15
i	2	18
2i+1	5	9
2i+2	2	13



HEAPSORT – ETAPAS

a) Montar a heap. A transformação é feita do último nível da árvore para a raiz, colocando em cada nó o elemento de maior chave entre ele e seus filhos, até que se obtenha a heap.

b) Trocar o elemento da raiz (que possui a maior chave) com o elemento do nó que está na última posição do vetor. A seguir isolar esse elemento e repetir o processo com os elementos restantes.

Exemplo passo a passo:

0	1	2	3	4	5	7	8	9	
520	450	254	310	**285**	179	351	423	**861**	285
				i				**MF**	raiz
520	450	254	310	**861**	179	351	423	**861**	
								i	
520	450	254	310	861	179	351	423	**285**	
520	450	254	**310**	861	179	351	**423**	285	310
			i			~~**MF**~~	**MF**		raiz
520	450	254	**423**	861	179	351	**423**	285	
							i		
520	450	254	423	861	179	351	**310**	285	
							i		
520	450	**254**	423	861	179	351	310	285	254
		i			~~**MF**~~				raiz
520	450	**652**	423	861	179	351	310	285	
520	450	**652**	423	861	179	351	310	285	
520	**450**	652	423	**861**	179	351	310	285	450
	i		~~**MF**~~	**MF**					raiz
520	**861**	652	423	**861**	179	351	310	**285**	
				i				**MF**	
520	861	652	423	**450**	179	351	310	285	
520	**861**	652	423	450	179	351	310	285	520
i	**MF**								raiz
861	**861**	652	423	**450**	179	351	310	285	
	I		~~**MF**~~	**MF**					
861	**520**	652	423	450	179	351	310	285	

ÁRVORE HEAP

TROCA V[0] COM V[9]

861	520	652	423	450	179	351	310	**285**	
								dir	
285	520	652	423	450	179	351	310	861	
							dir		
285	520	**652**	423	450	179	351	310	285	
i	~~**MF**~~	**MF**						**raiz**	

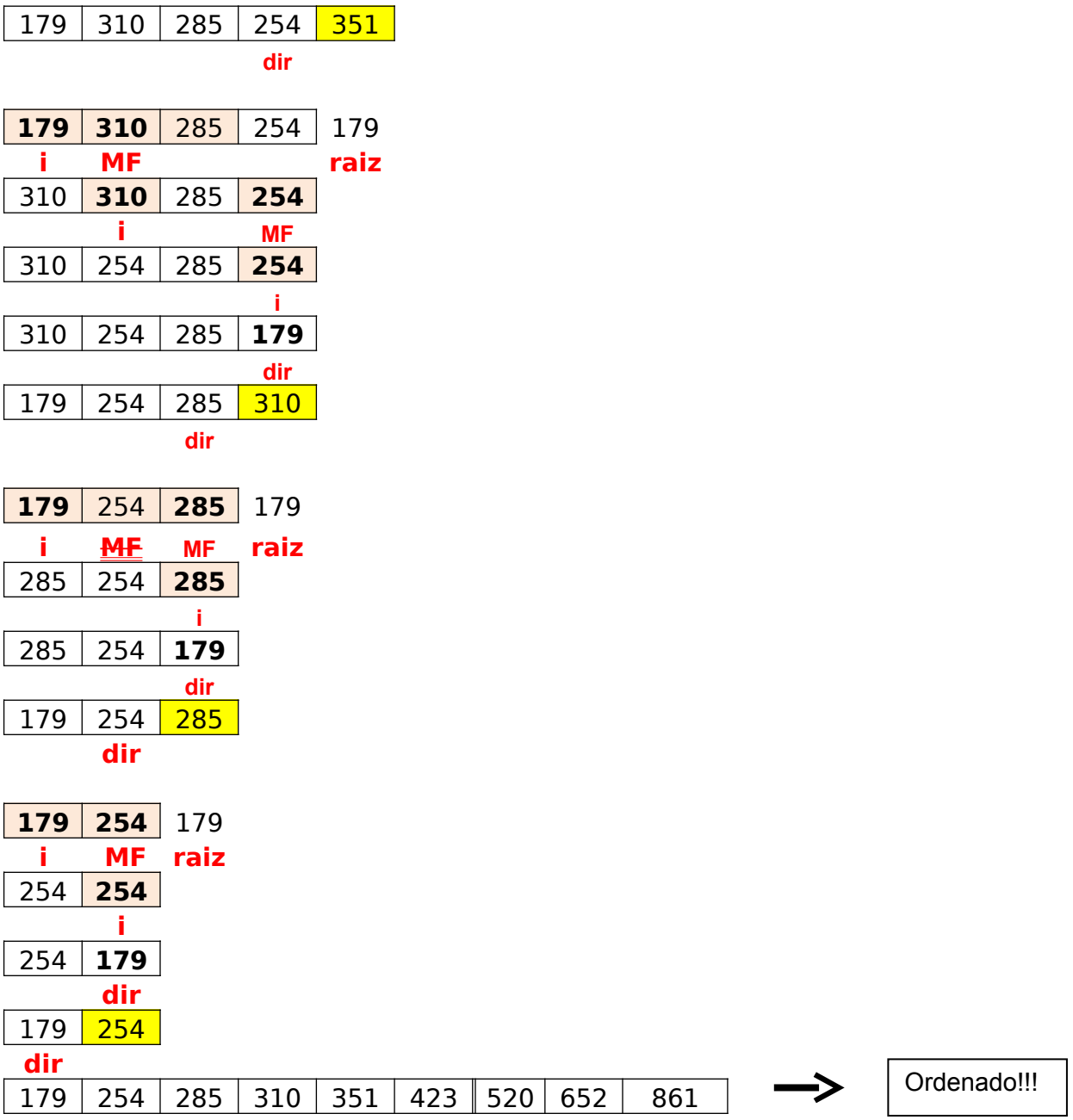
652	520	**652**	423	450	179	351	310
		i			**MF**		
652	520	**285**	423	450	179	351	310
							dir
310	520	285	423	450	179	351	652
						dir	

310	**520**	285	423	450	179	351	310
i	**MF**						**raiz**
520	**520**	285	423	**450**	179	351	
	i		**MF**	**MF**			
520	450	285	423	**450**	179	351	
				i			
520	450	285	423	**310**	179	351	
						dir	
351	450	285	423	310	179	520	

351	**450**	285	423	310	179	351
i	**MF**					**raiz**
450	**450**	285	**423**	310	179	
	i		**MF**			
450	423	285	**423**	310	179	
			i			
450	423	285	**351**	310	179	

254	423	285	351	310	179
					dir
254	**423**	285	351	310	179
i	**MF**				
423	**423**	285	**351**	310	179
	i		**MF**		
423	351	285	**351**	310	179
			i		
423	351	285	**254**	310	179
					dir
179	351	285	254	310	423
				dir	

179	**351**	285	254	310	179
i	**MF**				**raiz**
351	**351**	285	254	**310**	
	i		**MF**	**MF**	
351	310	285	254	**310**	
				i	
351	310	285	254	**179**	
				dir	



O índice inicial do vetor é 0 (zero), o sucessor à esquerda do elemento de índice i será *2i+1* e o da direita, *2i+2*.

```
public void heapSort (){
    int dir = nElem-1;
    int esq = (dir-1)/2;
    Item temp;

    while (esq >= 0){
         refazHeap (esq, this.nElem-1);
         esq--;
     }
    while (dir > 0){
         temp = this.vetor[0];
         this.vetor [0] = this.vetor [dir];
         this.vetor [dir] = temp;
         dir--;
```

```
            refazHeap(0, dir);
        }
}

private void refazHeap (int esq, int dir){
   int i = esq;
   int mF = 2*i+1; // maior filho
   Item raiz = this.vetor[i];
   boolean heap = false;

   while ((mF <= dir) && (!heap)){
      if ( mF < dir)
         if (this.vetor[mF].getChave() < this.vetor[mF+1].getChave())
                  mF ++;
      if (raiz.getChave() < this.vetor[mF].getChave()) {
         this.vetor[i] = this.vetor[mF];
         i = mF;
         mF = 2*i+1;
      }
      else
         heap = true;
   }
   this.vetor[i] = raiz;
}
```

Encontrando o maior filha

Se a raiz for menor que o maior filho, ocorre a troca. Volta e verifica se sub-árvore continuará heap.

CONSIDERAÇÕES:

- Este algoritmo não é estável
- Não é recomendado seu uso para arquivos com poucos registros devido ao tempo necessário para se construir a heap

ANÁLISE DO ALGORITMO:

- Melhor caso seria o vetor em ordem decrescente
- Pior caso seria o vetor em ordem crescente
- Como a complexidade não é tão simples de ser calculada, temos que, no caso médio:

C (n) = O(n logn) , bem como M(n) = O(n logn)

Exercício:

1) Simule o processo de ordenação heapSort para o vetor abaixo:

23	1	44	32	67	9	23	5	70	10

II.9 COMPARAÇÃO ENTRE OS MÉTODOS DE ORDENAÇÃO INTERNA

A tabela abaixo reproduz as tabelas apresentadas por Wirth[1] e ilustra o comportamento dos algoritmos estudados. Os métodos foram implementados em Modula-2 e executados em um mesmo PC. Assim, são mostrados os tempos (em segundos):

[1] Wirth, Algoritmos e Estrutura de Dados

Método	Quadro Comparativo = Tempo de Execução					
	N = 256			N = 2048		
	Ordenado	Aleatório	Invertido	Ordenado	Aleatório	Invertido
Inserção Direta	0.02	0.82	0.64	0.22	50.74	103.80
Seleção Direta	0.94	0.96	1.18	58.18	58.34	73.46
Bubblesort	1.26	2.04	2.80	80.18	128.84	178.66
Shakersort	0.02	1.66	2.92	0.16	104.44	187.36
Shellsort	0.10	0.24	0.28	0.80	7.08	12.34
Heapsort	0.20	0.20	0.20	2.32	2.22	2.12
Quicksort	0.08	0.12	0.08	0.72	1.22	0.76

Podemos concluir que:

- Em quase todos os casos o Bubblesort é o pior deles. Isso só não acontece quando o arranjo está em ordem invertida, pois assim, o pior deles é o Shakersort.
- Quando o arranjo está ordenado, os melhores são Inserção Direta e Shakersort, sendo que ao aumentar o tamanho do arranjo, o Shakersort fica melhor do que o Inserção.
- Para arranjos aleatórios e invertidos, o Quicksort é o melhor.
- O heapsort não altera muito seu tempo de execução ao alterar o tipo de entrada e, o seu tempo de execução ao aumentar o tamanho do arranjo aumenta menos se compararmos com os outros métodos.

Podemos também observar as tabelas apresentadas pelo Prof. Nívio Ziviane[2] que estuda o comportamento de cinco desses algoritmos (Seleção e Inserção – $O(n^2)$ comparações – e Shell, Quick e Heap – O(n logn) comparações) ao ordenar arranjos de 500, 5.000, 10.000 e 30.000 registros.

Em cada tabela, não temos tempo real, mas um comparativo onde o método que levou menos tempo para executar a ordenação recebeu valor 1 e os outros receberam valores relativos a ele. Assim, quem recebeu valor 2, levou o dobro de tempo do que quem recebeu valor 1.

	Ordenado				Aleatório				Invertido			
	500	**5000**	**10000**	**30000**	**500**	**5000**	**10000**	**30000**	**500**	**5000**	**10000**	**30000**
Inserção	1	1	1	1	11.3	87	161	–	40.3	305	575	–
Seleção	128	1524	3066	–	16.2	124	228	–	29.3	221	417	–
Shellsort	3.9	6.8	7.3	8.1	1.2	1.6	1.7	2	1.5	1.5	1.6	1.6
Quicksort	4.1	6.3	6.8	7.1	1	1	1	1	1	1	1	1
Heapsort	12.2	20.8	22.4	24.6	1.5	1.6	1.6	1.6	2.5	2.7	2.7	2.9

Podemos observar, pelas tabelas acima que:

- Shellsort, Quicksort e Heapsort têm a mesma ordem de grandeza;
- Se a entrada de dados for aleatória, o Quicksort é o mais rápido para todos os tamanhos experimentados;

[2] Projeto de Algoritmos com Implementação em Pascal e C

- Para entrada aleatória em arquivos pequenos (500 registros) o Shellsort é mais rápido do que o Heapsort, mas quando o tamanho da entrada cresce, esta relação se inverte.
- O método de Inserção é o mais rápido, para qualquer tamanho, se os elementos já estão ordenados (melhor caso) e o mais lento, para qualquer tamanho, se os elementos estão em ordem invertida (pior caso).
- Entre os algoritmos $O(n^2)$, o Inserção é o melhor para todos os tamanhos aleatórios experimentados

Foi feito também um experimento para saber a influência da ordem inicial do arquivo sobre cada um dos três métodos eficientes. Para cada método, tem-se também o tempo proporcional, ou seja, 1 é o melhor tempo e os outros são relativos ao 1.

	SHELLSORT			***QUICKSORT***			***HEAPSORT***		
	5000	**10000**	**30000**	**5000**	**10000**	**30000**	**5000**	**10000**	**30000**
Ord	1	1	1	1	1	1	1.1	1.1	1.1
Inv	1.5	1.6	1.5	1.1	1.1	1.1	1	1	1
Ale	2.9	3.1	3.7	1.9	2.0	2.0	1.1	1.1	1.1

1. **SHELLSORT** ⇒É sensível à ordenação ascendente ou descendente, ou seja, para arquivos de mesmo tamanho, executa mais rápido com arquivo ordenado ou invertido do que aleatório.
2. **QUICKSORT** ⇒Assim como o shellsort, é sensível à ordenação ascendente ou descendente, ou seja, para arquivos de mesmo tamanho, executa mais rápido com arquivo ordenado ou invertido do que aleatório. Mas é o mais rápido se os elementos estiverem em ordem ascendente.
3. **HEAPSORT** ⇒Praticamente não é sensível à entrada.

II.10 CONCLUSÃO

☺ **INSERÇÃO:** é o mais interessante para arquivos com menos de 20 elementos. O método é estável e seu comportamento é melhor que o BUBBLESORT que também é estável. Sua implementação é simples. Quando se deseja adicionar alguns elementos a um arquivo já ordenado e depois obter um outro arquivo ordenado, seu custo é linear.

☺ **SELEÇÃO:** Somente é vantajoso quando a arquivo possui registros muito grandes, desde que o tamanho do arquivo não ultrapasse 1000 registros

☺ **SHELLSORT:** é o método escolhido para a maioria das aplicações por ser muito eficiente para arquivos com até 10000 registros. Para arquivos grandes, apesar de ser mais lento do que o Quicksort, ele demora apenas o dobro do tempo deste, mas, em compensação, sua implementação é mais simples, fácil de funcionar corretamente e resulta em um programa pequeno. E, além disso tudo, não possui um pior caso e, quando encontra um arquivo parcialmente ordenado trabalha menos.

☺ **QUICKSORT:** é o algoritmo mais eficiente para várias situações, mas é um método frágil, pois pode haver muita dificuldade para detectar erros de implementação. O algoritmo é recursivo, necessitando de uma pequena quantidade de memória

adicional. No pior caso sua ordem é $O(n^2)$ operações. Se obtiver uma implementação robusta, ele deve ser o método utilizado. Ë necessário ter cuidado ao escolher o **pivô. SOLUÇÃO**: escolher uma pequena amostra do arranjo e usar a mediana da amostra como pivô. Geralmente se usa uma amostra de 3 elementos. **Melhoria:** Se o sub-arquivo da chamada recursiva tiver menos que 25 elementos, pode-se usar Inserção Direta nesse sub-arquivo. A melhoria chega a 20%.

☺ **HEAPSORT:** ë um método elegante e eficiente apesar de ser mais lento do que o QUICKSORT mas, não precisa de memória adicional. É sempre O(n logn) portanto, é bom para aplicações que não podem tolerar variações no tempo esperado, de acordo com a entrada.

II.11 EXERCÍCIOS

1) Preencha a tabela abaixo adequadamente:

Algoritmo	Custo para o melhor caso		Custo para o pior caso		Observações
	M(n)	C(n)	M(n)	C(n)	
Seleção Direta					
BubbleSort					
ShakerSort					
Inserção Direta					
Shellsort					
MergeSort					
QuickSort					
HeapSort					

2) Coloque em ordem crescente, o comportamento assintótico das funções mais utilizadas em análise de algoritmos

O(1) < O( ) < O( ) < O( ) < O( ) < O( ) < O( )

3) Em cada caso abaixo, diga qual algoritmo de ordenação, dentre os vistos, seria mais indicado para cada caso ? Justifique.

a) Um vetor com 1000 registros inversamente ordenados.
b) Um vetor com 10000 registros inversamente ordenados.
c) Um vetor com 5000 registros, ordenados, e que pode sofrer poucas inclusões ou remoções.
d) Arquivo com 70 registros, sendo que estes registros são muito grandes, o que torna o custo de movimentação de itens crítico em relação a qualquer outra operação.
e) Um arquivo com 500 registros, de 20 campos cada, ordenados em ordem inversa.
f) Um arquivo com 750 registros quase ordenados.
g) Um arquivo com 3.500 registros aleatórios.
h) Um arquivo com 650 registros, ordenados, exceto o último elemento que é o menor de todos.
i) Um arquivo com 1.500 registros, onde se deseja estabilidade.
j) Um arquivo com 2.000 registros aleatórios.

4) Sugira um algoritmo para ordenação de uma lista simplesmente encadeada. Justifique.

5) Para um vetor pouco volátil, pequeno e com baixo potencial de crescimento, qual o algoritmo de ordenação, dentre os vistos, é o mais indicado? Explique.

6) Se você deseja ordenar vários arquivos, de vários tamanhos, podendo variar o tamanho dos registros, qual ou quais algoritmos de ordenação, dentre os vistos, usaria se houvesse restrições quanto à estabilidade? Justifique.

7) No problema (6), qual usaria se houvesse intolerância para o pior caso, ou seja, não permite que a execução demore muito mesmo que eventualmente? Justifique.

8) Supondo que você estava trabalhando em um computador a ordenar um conjunto muito grande de números inteiros, cada número formado de muitos dígitos, usando um método O(n logn). O seu computador foi atacado por um vírus que alterou, aleatoriamente os 4 bits menos significativos de cada número. Você quer ordenar novamente os novos números. Escolha um algoritmo capaz de ordenar os novos números em O (n). Justifique.

9) Suponha que você tenha que ordenar vários arquivos de 100, 500 e 2.000 números inteiros. Para os três tamanhos de arquivos é necessário realizar a ordenação no menor tempo possível, utilizando um método visto.

a) Que algoritmo de ordenação você usaria para cada tamanho de arquivo? Justifique.
b) Se for necessário manter a estabilidade, que algoritmo de ordenação você usaria para cada tamanho de arquivo? Justifique.

10) Suponha um vetor que irá armazenar informações de telefones dos habitantes de uma determinada cidade. A finalidade do programa que utiliza este vetor é realizar a leitura de todos os telefones e nomes das respectivas pessoas e então realizar a ordenação para imprimir a lista telefônica. Qual o melhor algoritmo de ordenação para este caso. Justifique.

11) Suponha um site de busca que encontra todas as páginas relacionadas a uma determinada palavra. Este site deve então ordenar todas as páginas encontradas pelo número de ocorrências da palavra na página, ou seja, a página onde a palavra apareceu mais vezes deverá ser a primeira da lista, e assim sucessivamente. Dos algoritmos estudados qual seria o melhor para utilização neste site? Explique.

12) O que cada trecho de código assinalado (A, B, C, D e E) faz no algoritmo abaixo.

```
public void heapSort (){
    int dir = nElem-1;
    int esq = (dir-1)/2;
    Item temp;

  A while (esq >= 0)
        refazHeap (esq--, this.nElem-1);

  B while (dir > 0){
        temp = this.vetor[0];
        this.vetor [0] = this.vetor [dir];
        this.vetor [dir--] = temp;
       refazHeap (0, dir);
    }
}
```

```
private void refazHeap (int esq, int dir){
  int i = esq;
  int MaiorFolha = 2*i+1;
  Item raiz = this.vetor[i];
  boolean heap = false;

  while ((MaiorFolha <= dir) && (!heap)){
     if (MaiorFolha < dir)
  C    if (this.vetor[MaiorFolha].getChave()<this.vetor[MaiorFolha+1].getChave())
          MaiorFolha++;
     if (raiz.getChave() < this.vetor[MaiorFolha].getChave()) {
  D       this.vetor[i] = this.vetor[MaiorFolha];
          i = MaiorFolha;
  E       MaiorFolha = 2*i+1;
       }
       else
           heap = true;

  }
  this.vetor[i] = raiz;
}
```

13) Simule os métodos vistos para organizar os vetores abaixo.

a) Ao simular o Shellsort, utilize H = 5, H = 3 e H = 1:

05	10	09	07	14	08	20	02	06	18	01

b) Ao simular o Shellsort, utilize H = 4, H = 2 e H = 1:

30	20	15	12	22	18	07	09	11

14) Calcule o custo dos algoritmos abaixo:

```
a)
boolean procura (int[][] mat, int num){
  int i=0, j;
  boolean achou = false;
  while (i<mat.length && !achou){
    j = 0;
    while (j<mat[i].length && !achou){
      if (mat[i][j] == num)
        achou = true;
      j++;
    }
    i++;
  }
  return achou;
}

b)
void calculo (int[][] mat, int[] vet){
  int i, j;
  for (i=0; i<mat.length; i++)
    for (j=0; j< mat[i].length; j++)
      vet[i] = vet[i] + mat[i][j];
}
```

```
c)
int maior (int[][] mat){
  int temp, i, j;
  temp = mat[0][0];
  for (i=0; i<mat.length; i++)
    for (j=0; j< mat[i].length; j++)
      if (mat[i][j] > temp)
        temp = mat[i][j];
  return temp;
}

d)
boolean simetrica (int[][] mat){
  int i, j;
  for (i=1; i<mat.length; i++)
    for (j=0; j< i; j++)
      if (mat[i][j] != mat[j][i])
        return false;
  return true;
}
```

15) Coloque em ordem crescente as seguintes funções pela dominação assintótica.
A(n) = 50n B(n) = 3 n^2 /log n D(n) = $\log^2 n$ E(n) = $2^{\log n}$ F(n) = log(n + 5/n)