MÉTODOS DE ORDENAÇÃO

Ordenação

- Ordenar é rearranjar um conjunto de objetos em uma ordem crescente ou decrescente.
- Objetivo:
  - Facilitar a recuperação posterior, dos dados armazenados.
  - Pesquisar e recuperar dados.

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.
- Os algoritmos trabalham através de uma **"chave"** de pesquisa.

Classe Item

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

Ambiente de Classificação

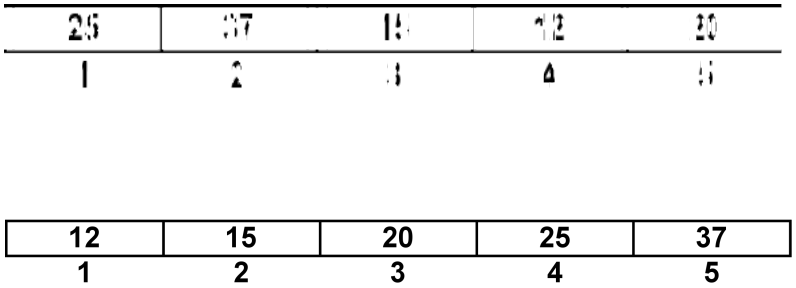
- ***Ordenação interna*** – Ocorre quando o arquivo a ser ordenado cabe todo na memória principal. Neste tipo de ordenação, qualquer registro pode ser imediatamente acessado.
- ***Ordenação externa*** – Ocorre quando o arquivo não cabe na memória principal e deve ser armazenado em fita ou disco. Neste tipo de ordenação, os registros são acessados seqüencialmente ou em grandes blocos.

Medidas de complexidade relevantes

- Para saber o tempo gasto para ordenar um arquivo, é necessário calcular :
  - ***Comparações de chaves - C (n)***
  - ***Movimentações de itens - M (n)***
- Também é necessário saber a quantidade de memória auxiliar utilizada pelo algoritmo.

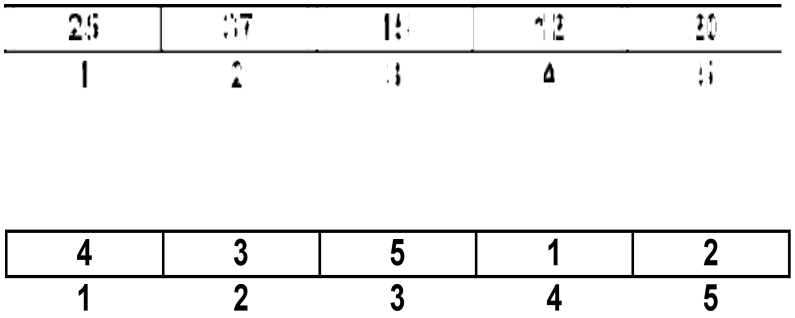
Estrutura de dados

- ***Contiguidade física***



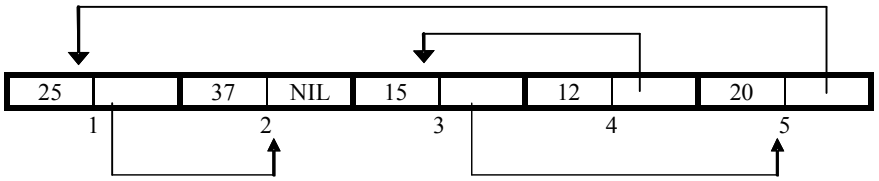
Estrutura de dados

- ***Vetor indireto de ordenação***



Estrutura de dados

- ***Encadeamento***



Primeiro = 4

Estrutura de Dados

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

Ordenação Interna

- **Métodos simples**
  - O(n²) comparações
  - Produzem programas pequenos e fáceis de entender.
- **Métodos eficientes**
  - O (n logn) comparações.
  - As comparações usadas são mais complexas nos detalhes.

Tipo de dados

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

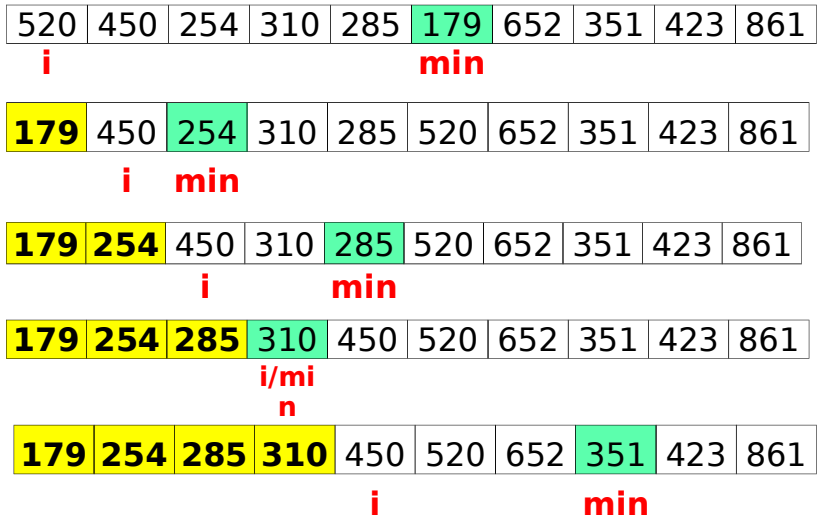
Método de Ordenação Estável

- Quando a ordens dos itens com **chaves** iguais mantém-se inalterada pelo processo de ordenação.

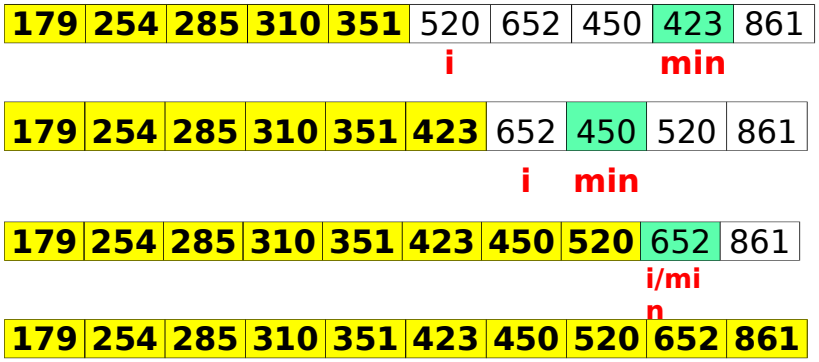
18	7	10	7	3
Andréa Costa	João da Silva	João da Silva	Luiz Dantas	Maria Freitas
1	*2*	*3*	*4*	*5*

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

Seleção Direta – Selection Sort



Seleção Direta – Selection Sort



Algoritmo

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

Custo

- Comparações

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

- Movimentações

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

Considerações

- 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.

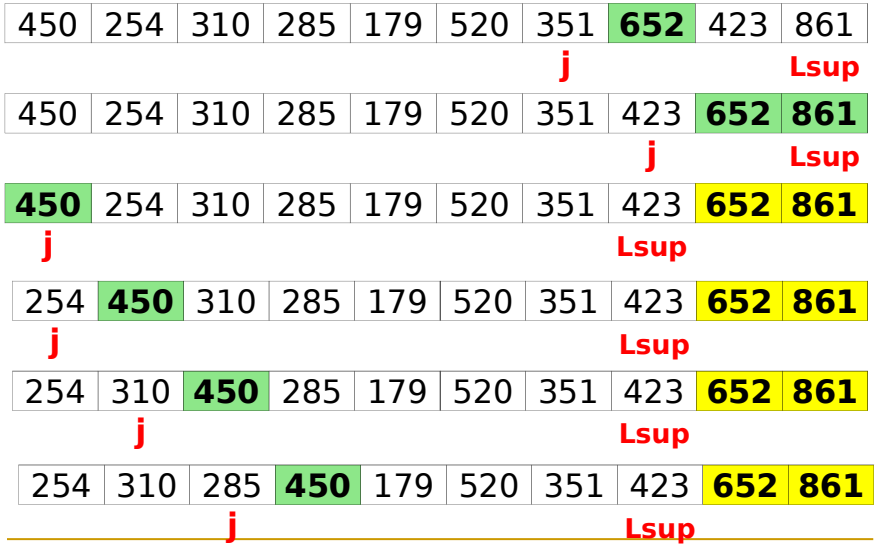
Não é estável

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

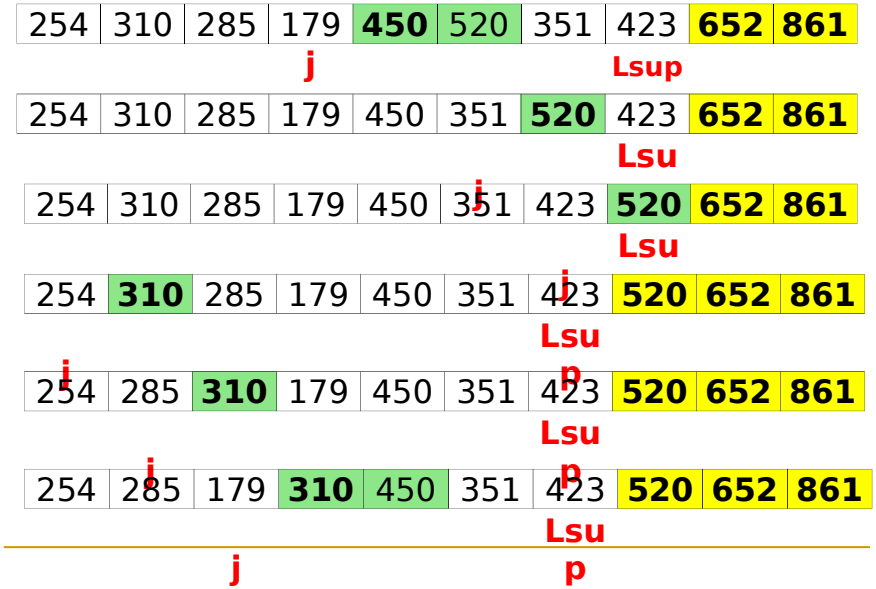
Bolha - BubbleSort

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

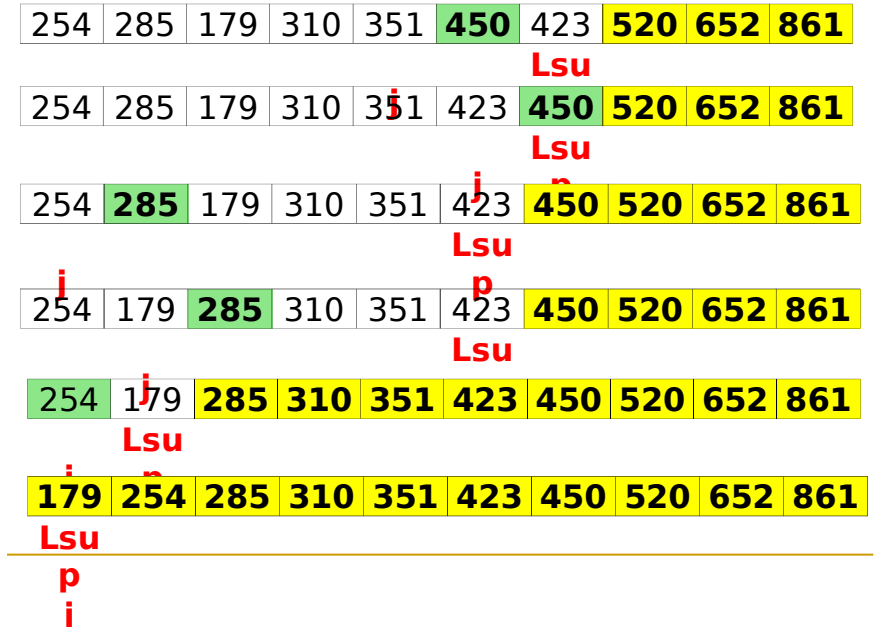
Bolha – BubbleSort



Bolha – BubbleSort



Bolha – BubbleSort



BubbleSort - Algoritmo

```
public void bubbleSort () {
  int LSup, iBolha ;
  Item temp;

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

     LSup = iBolha;
  while (LSup >= 1) ;
}
```

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

- Parece o algoritmo de Seleção
- É um método **estável**
- Faz muitas trocas, o que o torna o menos eficiente dos métodos Simples ou Diretos
- 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 seqüência fixa de comparações.

Considerações

- 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.

ShakerSort

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

ShakerSort

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

ShakerSort

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		

ShakerSort

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		

ShakerSort - Algoritmo

```
public void shakersort (){
  int esq =1, dir = this.nElem-1, i, j=dir;
  Item temp;
  do{
     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;
    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);
}
```

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.

$$M(n) = 0$$

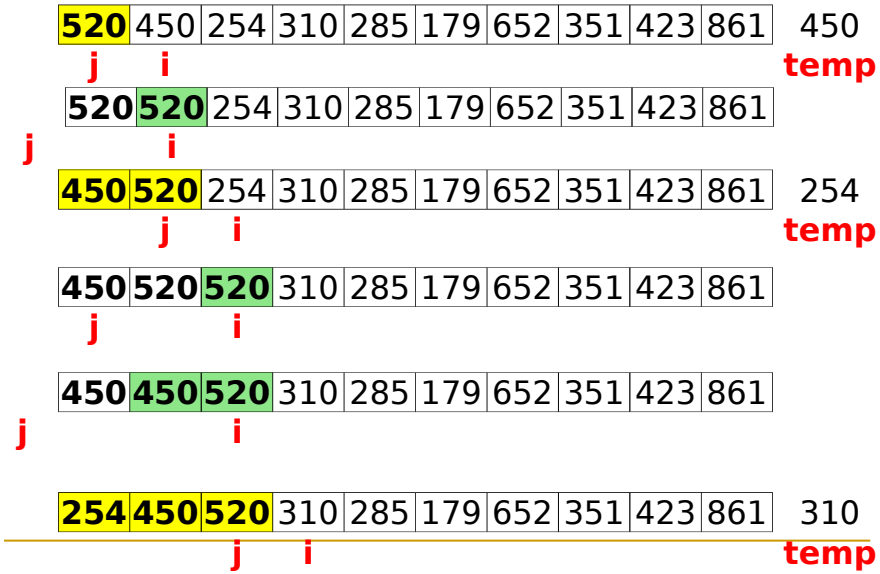
PIOR CASO: Vetor ordenado em ordem contrária

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

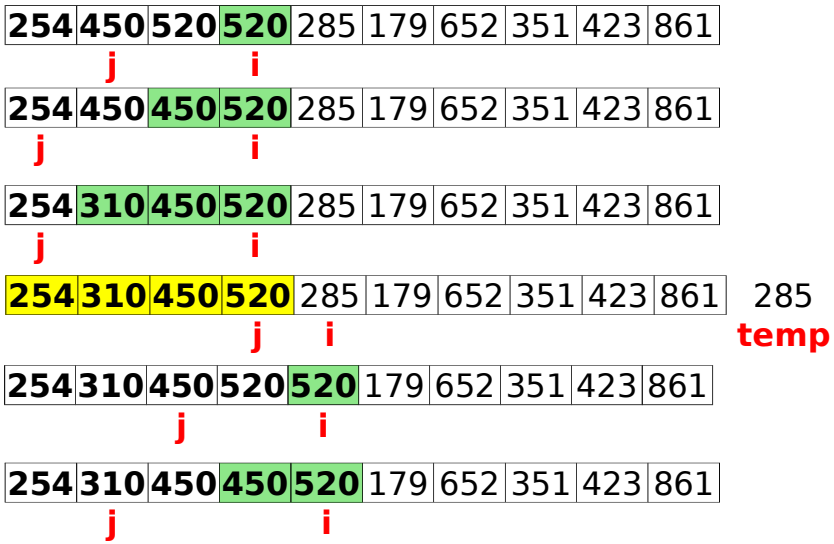
Considerações

- 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

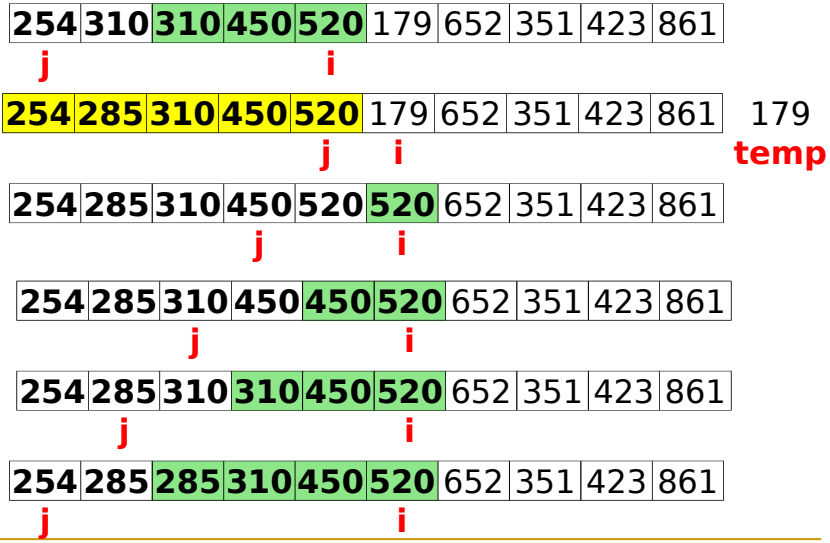
Inserção Direta - InsertionSort



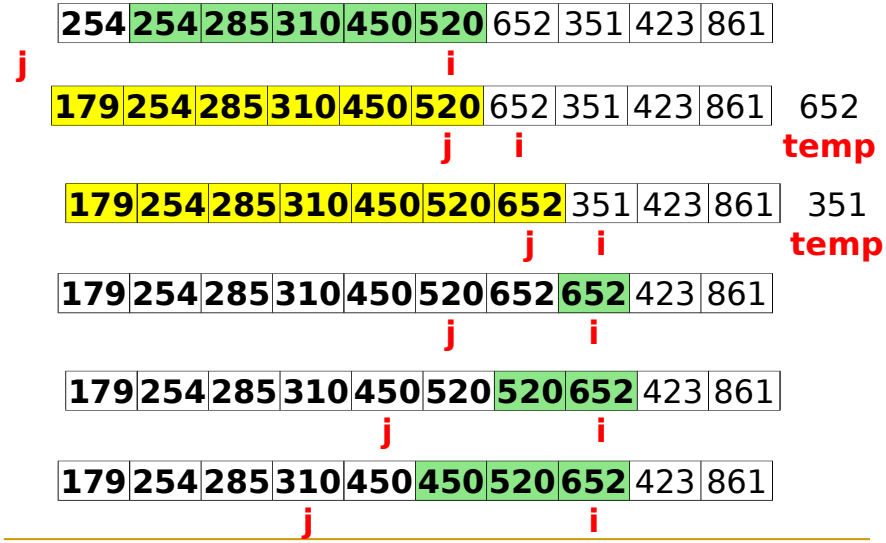
Inserção Direta - InsertionSort

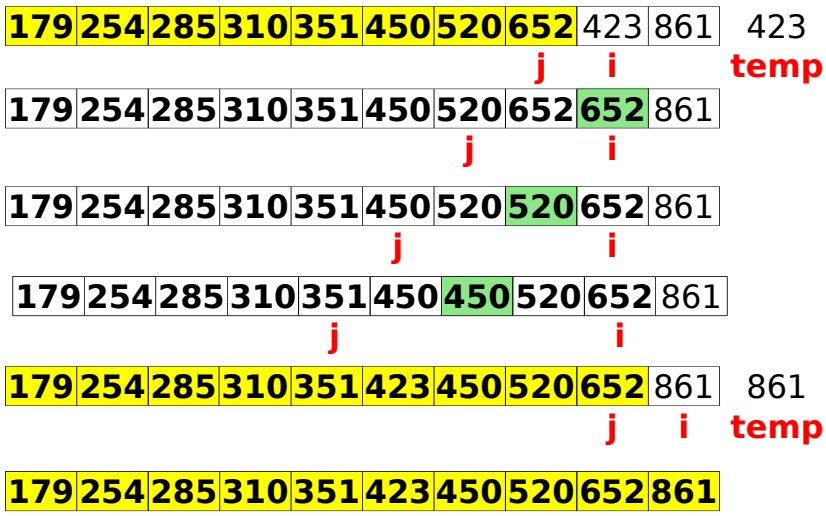


Inserção Direta - InsertionSort



Inserção Direta - InsertionSort





InsertionSort - Algoritmo

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

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

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.

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

PIOR CASO: Vetor ordenado em ordem contrária

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

Considerações

- 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

ShellSort

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**		

ShellSort

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**		

ShellSort

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**

ShellSort

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**	

ShellSort - Algoritmo

```
public void shellSort() {
    int h, j, n;
    Item temp;

    n = ;
    do{
      n = (n/3)+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);
```

Custo

- Ninguém ainda foi capaz de analisar este algoritmo. Portanto, ninguém sabe porque ele é eficiente.
- Como escolher os incrementos?
- Quanto à complexidade, conjecturas apontam para:

$$O(n^{1.2})$$

$$O(n \log n)$$

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

MergeSort

0	1	2	3	4	5	6	7	8	9
520	450	254	310	285	179	652	351	423	861
esq				meio					dir
520	450	254	310	285	179	652	351	423	861
esq		meio		dir					
520	450	254	310	285	179	652	351	423	861
esq	meio	dir							
520	450	254	310	285	179	652	351	423	861
esq	dir								
meio									
450	**520**								

MergeSort

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

MergeSort

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

MergeSort

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

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

MergeSort - Algoritmo

```
public void mergeSort() {
  mergeSort(0, this.tamanho-1);
}

private void mergeSort(int esq, int dir) {
  int meio;

  if (esq == dir)
    return;
  else{
    meio = (esq+dir)/2;
    mergeSort(esq, meio);
    mergeSort(meio+1, dir);
    merge(esq, meio+1, dir);
  }
}
```

MergeSort - Algoritmo

```
private void merge(int esq, int dir, int limSup){
  int limInf = esq;
  int meio = dir-1;
  int n = limSup - limInf + 1;
  VetorInteiro temp = new VetorInteiro(n);

  while (esq <= meio && dir <= limSup)
    if (this.vetor[esq] < this.vetor[dir])
      temp.insereFim(this.vetor[esq++]);
    else
      temp.insereFim(this.vetor[dir++]);
  while (esq <= meio)
    temp.insereFim(this.vetor[esq++]);
  while (dir <= limSup)
    temp.insereFim(this.vetor[dir++]);

  for (int j=0; j<n; j++)
    this.vetor[limInf+j] = temp.getElem(j);
}
```

Custo

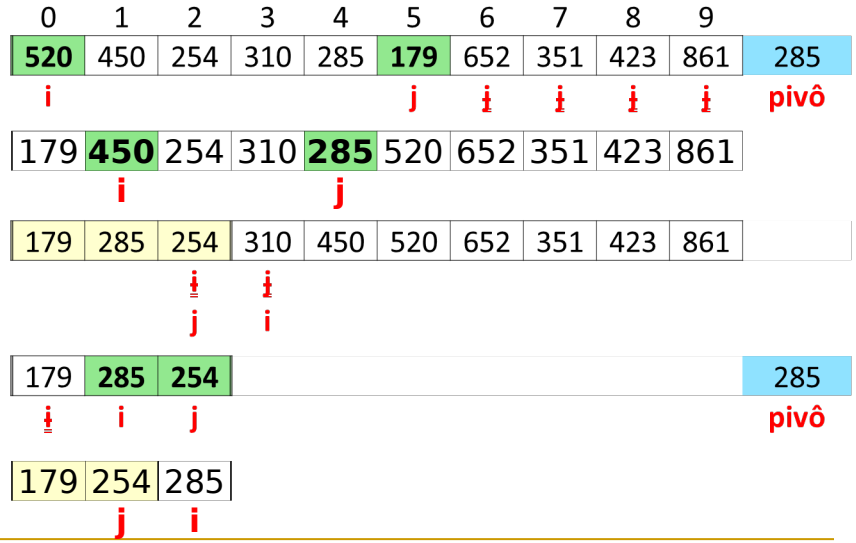
Não existe pior caso nem melhor caso.

Seu custo é da ordem: $C(n) = n \log n$

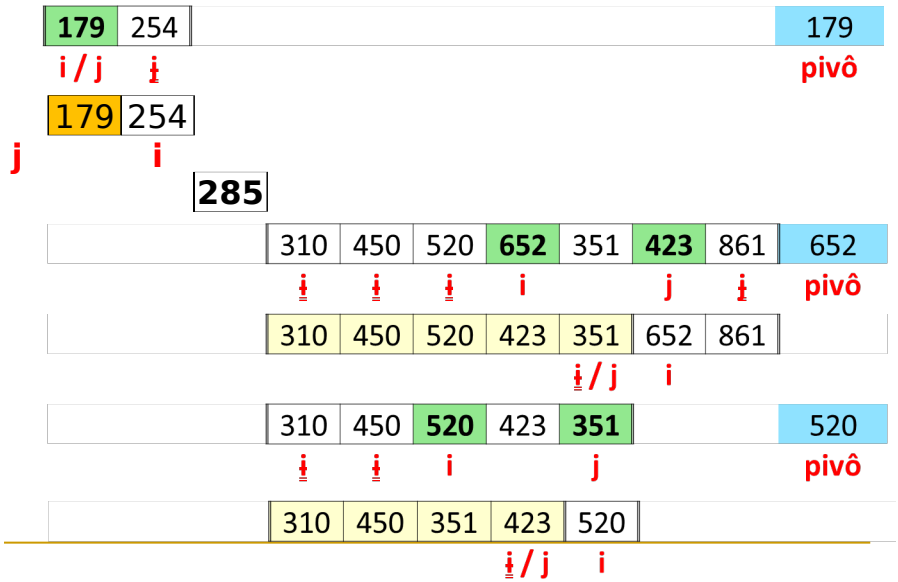
Considerações

- É um algoritmo fácil de implementar.
- Requer o dobro de memória, ou seja, precisa de um vetor com as mesmas dimensões do que se quer ordenar.

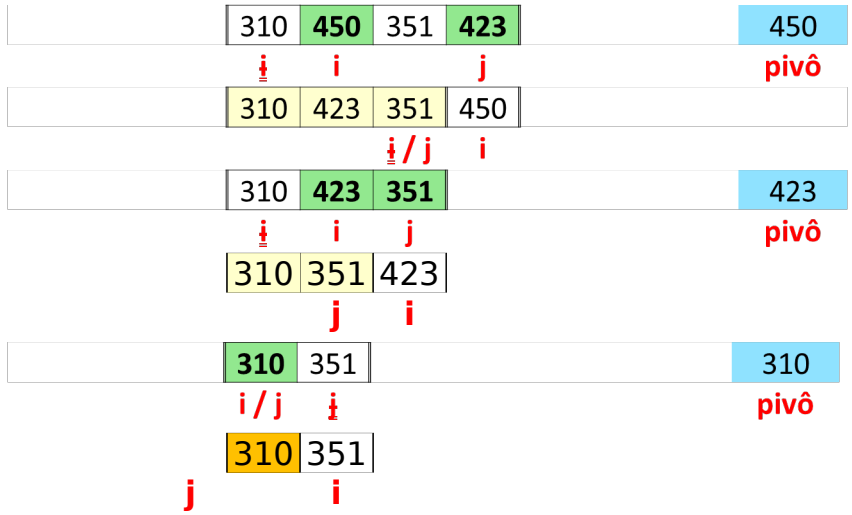
QuickSort



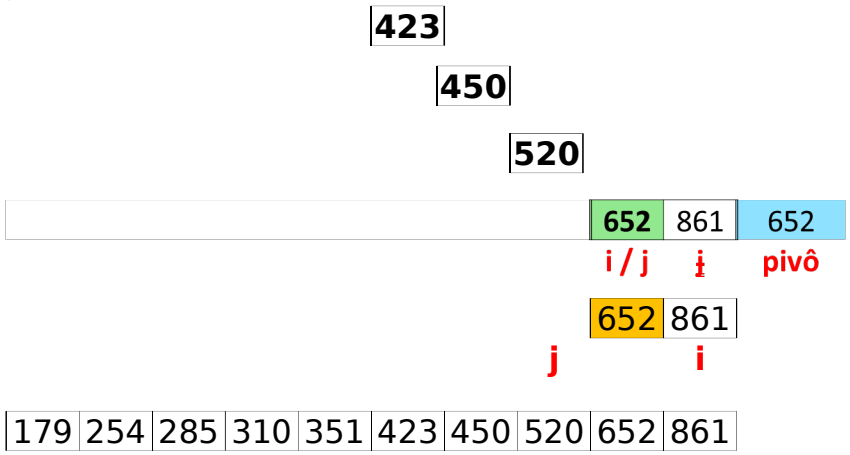
QuickSort



QuickSort



QuickSort



QuickSort - Algoritmo

```
public void quickSort ()
    ordena (0, this.nElem-1);

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

    pivo = this.vetor[(esq + dir)/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);
}
```

Custo

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

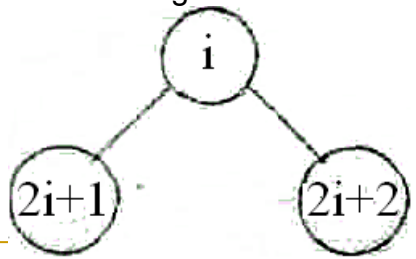
- É um algoritmo muito eficiente
- Precisa, em média n logn operações para ordenar n itens
- Necessita de uma pequena pilha como memória auxiliar
- Método não é estável
- Implementação é delicada e difícil.
- Um pequeno engano pode levar a efeitos inesperados

Considerações

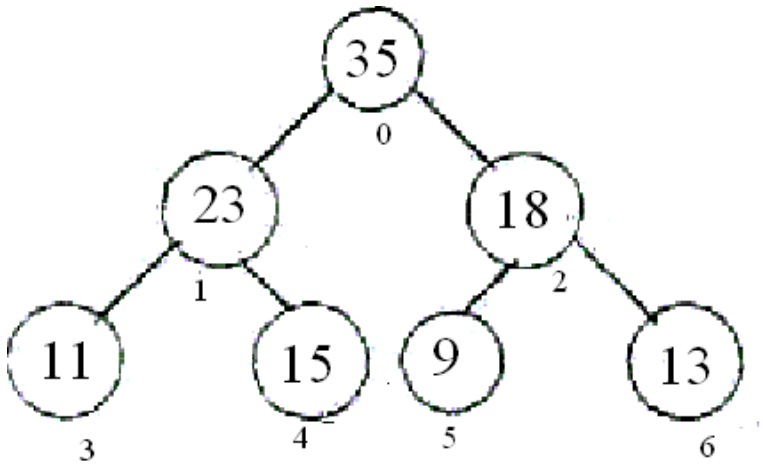
- Não é bom para arquivos já ordenados quando a escolha do pivô não é boa.
- A versão recursiva tem como pior caso O( ) operações
- Como evitar o pior caso?

HeapSort

- 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+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.

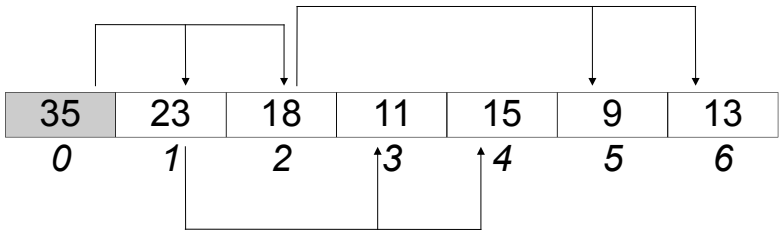


HeapSort



HeapSort

Vetor que representa a heap anterior



HeapSort - Etapas

- 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.
- 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.

HeapSort

0	1	2	3	4	5	6	7	8	9	
520	450	254	310	**285**	179	652	351	423	**861**	285
				i					MF	raiz
520	450	254	310	**861**	179	652	351	423	**861**	
									i	
520	450	254	310	**861**	179	652	351	423	**285**	

520	450	254	**310**	861	179	652	351	**423**	285	310
			i				~~MF~~	MF		raiz
520	450	254	**423**	861	179	652	351	**423**	285	
								i		
520	450	254	**423**	861	179	652	351	**310**	285	
								i		

HeapSort

520	450	**254**	423	861	179	**652**	351	310	285	254
		i			~~MF~~	MF				raiz
520	450	**652**	423	861	179	**652**	351	310	285	
						i				
520	450	**652**	423	861	179	**254**	351	310	285	
						i				

520	**450**	652	423	**861**	179	254	351	310	285	450
	i		~~MF~~	MF						raiz
520	**861**	652	423	**861**	179	254	351	310	**285**	
				i					MF	
520	**861**	652	423	**450**	179	254	351	310	285	

HeapSort

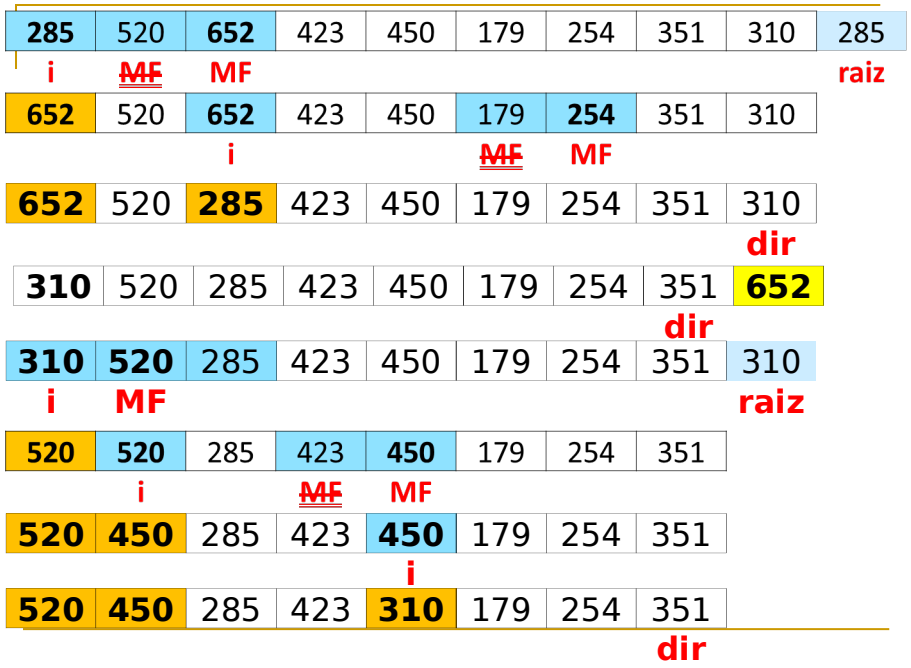
520	**861**	652	423	450	179	254	351	310	285	520
i	MF									raiz
861	**861**	652	423	**450**	179	254	351	310	285	
	i		~~MF~~	MF						
861	**520**	652	423	450	179	254	351	310	285	

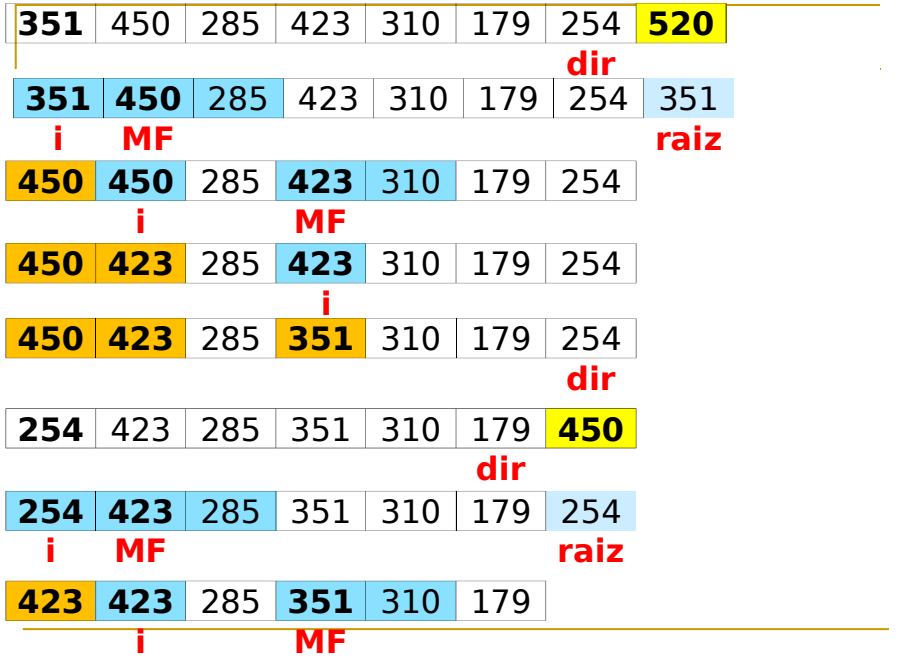
ÁRVORE HEAP

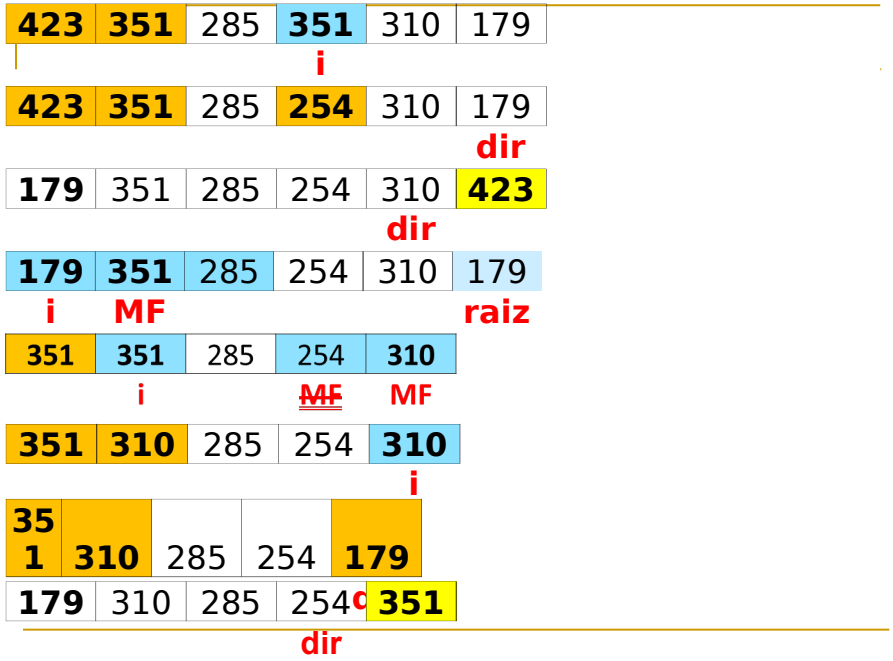
TROCA V[0] COM V[9]

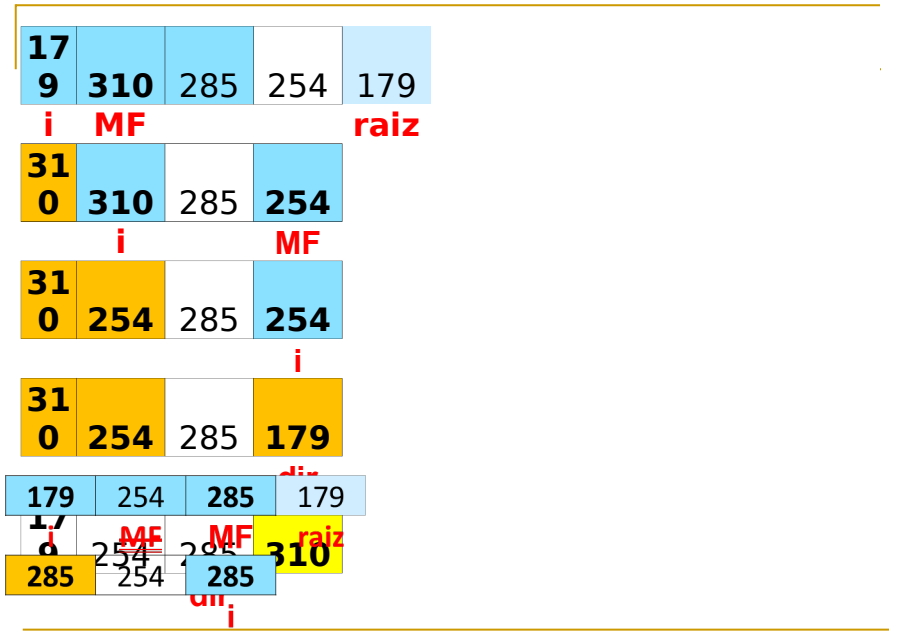
861	520	652	423	450	179	254	351	310	**285**
									dir

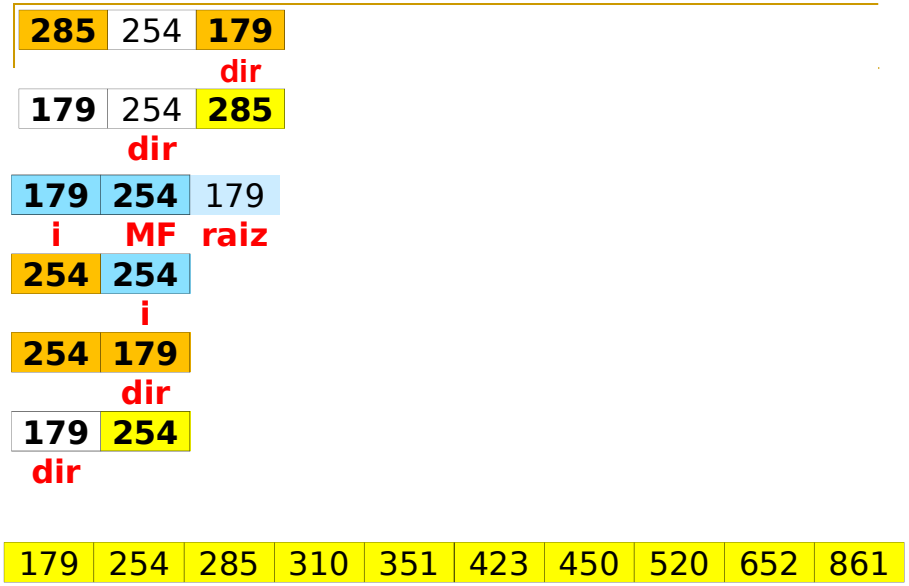
285	520	652	423	450	179	254	351	310	861
									dir


285 520 652 423 450 179 254 351 310 285
i MF MF raiz
652 520 652 423 450 179 254 351 310
i MF MF
652 520 285 423 450 179 254 351 310
dir
310 520 285 423 450 179 254 351 652
dir
310 520 285 423 450 179 254 351 310
i MF raiz
520 520 285 423 450 179 254 351
i MF MF
520 450 285 423 450 179 254 351
i
520 450 285 423 310 179 254 351
dir


351 450 285 423 310 179 254 520
dir
351 450 285 423 310 179 254 351
i MF raiz
450 450 285 423 310 179 254
i MF
450 423 285 423 310 179 254
i
450 423 285 351 310 179 254
dir
254 423 285 351 310 179 450
dir
254 423 285 351 310 179 254
i MF raiz
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
179 310 285 254 351
dir


179 310 285 254 179
i MF raiz
310 310 285 254
i MF
310 254 285 254
i
310 254 285 179
179 254 285 179
MF MF raiz
285 254 285
i



HeapSort - Algoritmo

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

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

HeapSort Algoritmo

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

Custo

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

$$C(n) = O(n \log n)$$

$$M(n) = O(n \log n)$$

Quadro Comparativo – Tempo de Execução

Método	*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

Conclusão

- 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.

Quadro Comparativo – Tempo de Execução

	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

Conclusão

- **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;
- 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 de ordem quadrática, o **Inserção** é o melhor para todos os tamanhos aleatórios experimentados.

Quadro Comparativo – Ordem inicial do arquivo

	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

Conclusão

- **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.
- **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.
- **HEAPSORT** ⇒ Praticamente não é sensível à entrada.

Conclusão

- **INSERÇÃO:**
  - é o mais interessante para arquivos com menos de 20 elementos.
  - é estável e seu comportamento é melhor que o BUBBLESORT que também é estável.
  - 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

Conclusão

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

Conclusão

- **QUICKSORT:**
  - é o algoritmo mais eficiente para várias situações
  - é um método frágil
  - algoritmo é recursivo.
  - No pior caso sua ordem é quadrática.
  - Se obtiver uma implementação robusta, ele deve ser o método utilizado.
  - É necessário ter cuidado ao escolher o **pivô.**
  - **Melhoria:** Se o sub-arquivo da chamada recursiva tiver menos que 25 elementos, pode-se usar Inserção Direta nesse sub-arquivo.

Conclusão

- **HEAPSORT:**
  - é um método elegante e eficiente apesar de ser mais lento do que o QUICKSORT.
  - 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.