Estruturas de Dados

Engenharia Informática

1º Ano - 2º Semestre

Francisco Morgado
Escola Superior de Tecnologia e Gestão de Viseu

1. Algoritmos de ordenação e algoritmos de pesquisa em arrays

- Algoritmos de Ordenação
  - Seleção Direta
  - Troca Direta ou BubbleSort
    - ShakerSort
  - Inserção Direta
    - ShellSort
  - QuickSort
    - QuickSort recursivo
    - QuickSort não recursivo
- Algoritmos de Pesquisa
  - Pesquisa Linear ou Sequencial
    - Pesquisa Linear com Sentinela
  - Pesquisa Binária
    - Algoritmo não recursivo
    - Algoritmo recursivo

1. Métodos de ordenação de arrays

A ordenação é genericamente entendida como sendo o processo de rearranjo de um determinado conjunto de objetos por uma ordem específica.
O objetivo principal da ordenação é facilitar o processo de pesquisa de um dos membros desse conjunto (Niklaus Wirth).

EXEMPLOS DE APLICAÇÃO

- ☑ **Lista telefónica**
- ☑ **Gestão de clientes**
- ☑ **Bibliotecas**
- ☑ **Dicionários**
- ☑ **etc.**

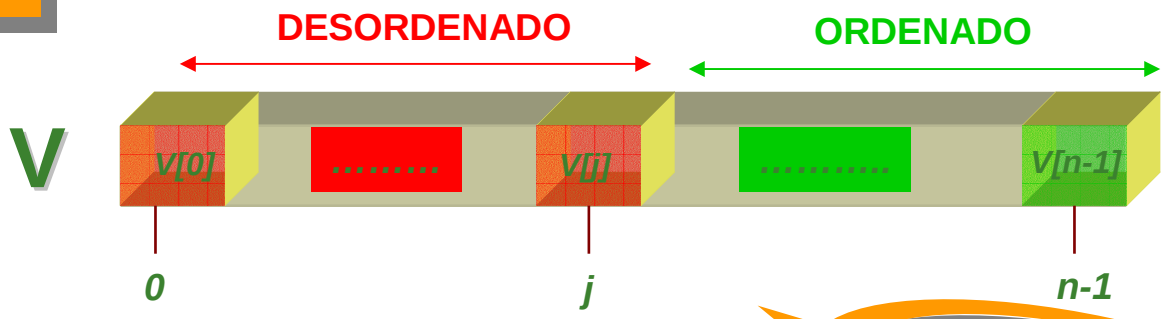
```
void sort(TipoDados vetor[], int tamanho);
```

1.1 Ordenação por Seleção Direta

Neste algoritmo, a estratégia consiste em ir selecionando em cada iteração o maior valor do *array* num determinado intervalo e colocá-lo no extremo desse intervalo.

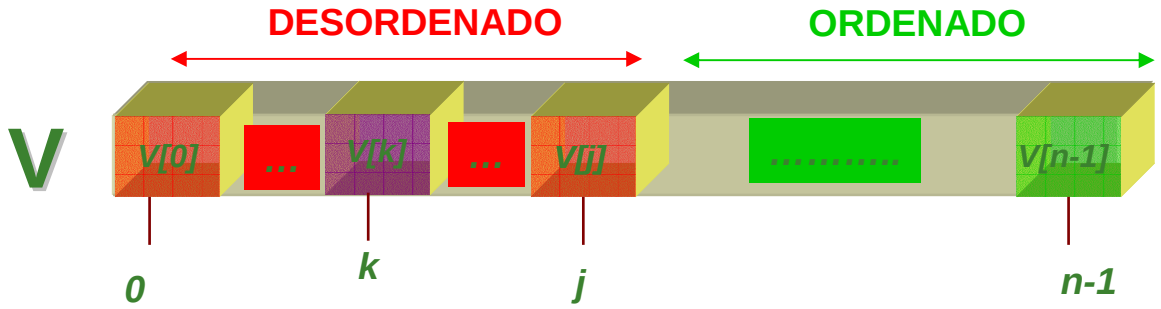
- **Primeiramente procuramos o maior valor a partir da primeira posição e colocamo-lo em último lugar**
- **Em seguida procuramos o segundo maior valor e colocamo-lo na penúltima posição**
- **O processo é sucessivamente repetido até chegar ao fim do *array***

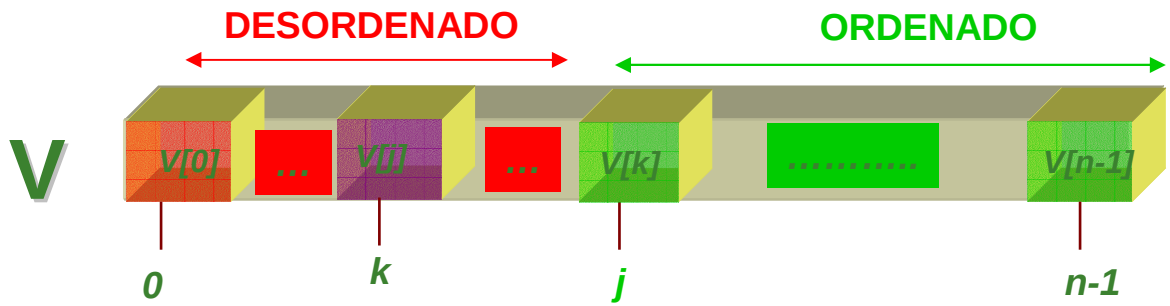
ILUSTRAÇÃO



Após j passos, falta ordenar o conjunto {v[0], ... v[n-1-j] }.
Sendo v[k] o maior elemento desse conjunto, trocar v[k] com v[j].

ESTRATÉGIA





Ordenação por Seleção Direta - Algoritmo

```
void selecao(TipoDados vetor[], int tamanho)
{
    int i, j, posMax;
    for (j = tamanho-1; j>0; j--)
    {
        //procura o máximo
        posMax = 0;
        for (i = 1; i<=j; i++)
            if (vetor[ i ] > vetor[posMax])
                posMax = i;

        troca(&vetor[j], &vetor[posMax]);
    }
}
```

FUNÇÃO troca()

Efetua a troca de valores entre duas variáveis a e b:

```
void troca(TipoDados* a, TipoDados* b)
{
        TipoDados tmp = *a;
        *a = *b;
        *b = tmp;
}
```

1.2 Ordenação por Troca Direta (*BubbleSort*)

A estratégia neste algoritmo consiste em percorrer a tabela a ordenar e comparar cada par de elementos adjacentes, trocando-os caso estejam fora de ordem.

- **Em geral, não basta uma única passagem pela tabela para que fique ordenada.**

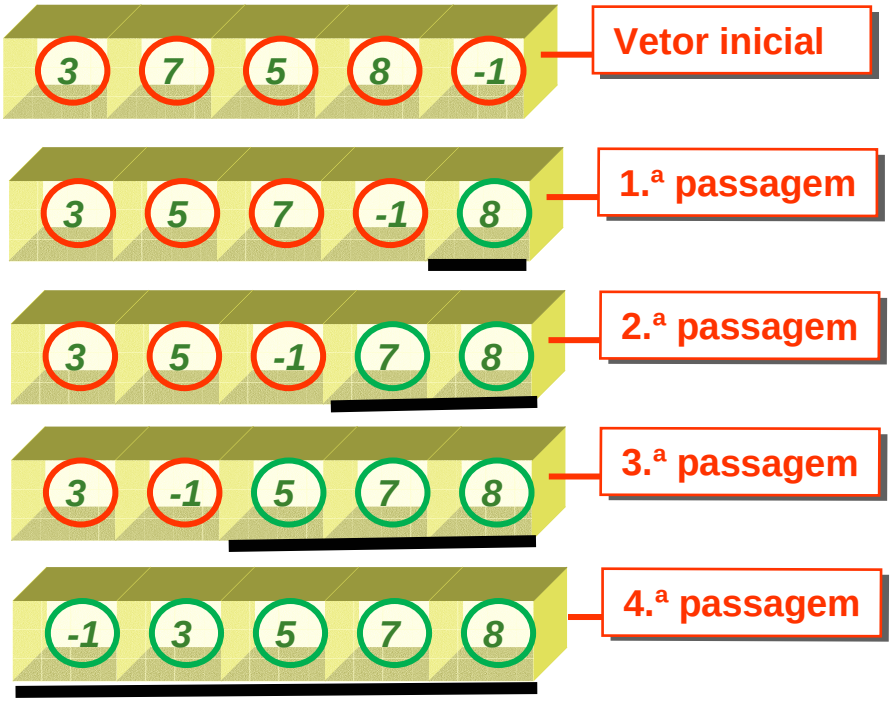
  →*Na primeira passagem o maior valor é colocado no fim da tabela (posição n).*

  →*Na segunda passagem, fica o segundo maior valor na penúltima posição.*

  →*Assim, após a iteração i+1 colocamos na posição n-i o elemento correto. Após n-1 passagens, temos a certeza de que todos os elementos da tabela estão ordenados.*

- **Este método é conhecido por *BubbleSort* por analogia com bolhas de ar dentro de água: as bolhas mais leves (valores menores) vão subindo lentamente, deslocando-se para o início do *array*. Contrariamente, os elementos maiores deslocam-se rapidamente para o fim do *array*.**

EXEMPLO de ordenação dum vetor pelo método *BubbleSort*



1.2.1 Ordenação por Troca Direta ou *BubbleSort* - Algoritmo

```
void bubbleSort(TipoDados vetor[], int tamanho)
{
      int i, j;

    for (i=0; i < tamanho-1; i++)          //passagens
       for (j=0; j<tamanho-1-i; j++)       //percorre apenas os que faltam
          if (vetor[j] > vetor[j+1])       //compara elementos adjacentes, se estão
             troca(&vetor[j], &vetor[j+1]); // fora de ordem troca-os
}
```

QUANDO HÁ UMA PASSAGEM SEM QUALQUER TROCA, O *ARRAY* ENCONTRA-SE ORDENADO.

ESTA CONSTATAÇÃO SUGERE A ESCRITA DE UM NOVO ALGORITMO.

1.2.2 BubbleSort mais eficiente (após uma passagem sem trocas, parar) - Algoritmo

```
void novoBubbleSort(TipoDados vetor[], int tamanho)
{
	int j;
	bool houveTroca;
	do
	{
		houveTroca = false;
		for (j= 0; j< tamanho-1; j++)
		   if (vetor[j] > vetor[j+1])    //compara elementos adjacentes
		   {
		      houveTroca = true;
		      troca(&vetor[j], &vetor[j+1]);
		   }
	} while (houveTroca);
}
```

Algoritmo ShakerSort

Novo aperfeiçoamento do algoritmo BubbleSort

- Dado que o algoritmo BubbleSort apenas compara elementos adjacentes, se um elemento pequeno estiver muito afastado da sua posição final, necessita de muitas passagens até que seja colocado na posição correta. De notar que os elementos pequenos apenas se deslocam uma posição de cada vez, enquanto que os maiores vão rapidamente para a sua posição. Consequentemente, o algoritmo BubbleSort não é uma boa opção para ordenar uma tabela.

- POSSÍVEIS MELHORAMENTOS:
  - Fazer as passagens em sentidos alternados de modo que, além dos elementos grandes, também os pequenos se aproximem mais rapidamente da sua posição correta.
  - Em vez de apenas se verificar quando não há trocas, para terminar o algoritmo, pode-se também registar o índice em que foi efetuada a última troca. Assim, considerando uma passagem no sentido dos índices crescentes, se a última troca efetuada foi a dos elementos j e j+1 então os elementos j+1 a N já estão ordenados.

- Estes dois melhoramentos conduzem a um novo método de ordenação – o algoritmo ShakerSort.

1.2.3 Algoritmo ShakerSort

```
void shakerSort(TipoDados vetor[], int tamanho)
{
        int i, inf = 0, sup = tamanho-1, ultimaTroca = 1;
        while (inf < sup)
        {
            for (i = inf; i < sup; i++)
                if (vetor[i] > vetor[i+1]) //compara elementos adjacentes
                {
                        troca(&vetor[i], &vetor[i+1]);
                        ultimaTroca = i;
                }
         sup = ultimaTroca;
            for (i = sup; i > inf; i--)
                if (vetor[i] < vetor[i-1]) // compara elementos adjacentes
                {
                        troca(&vetor[i], &vetor[i-1]);
                        ultimaTroca = i;
                }
        inf = ultimaTroca;
        }
}
```

Desloca os valores maiores para o fim do *array*

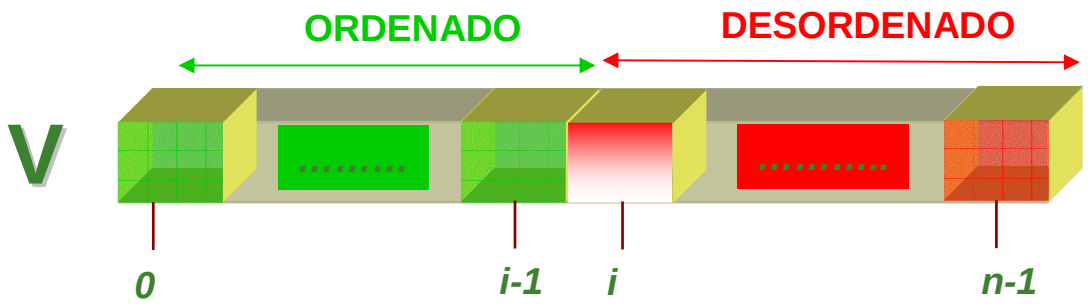
Desloca os valores menores para o início do *array*

1.3 Ordenação por Inserção Direta

Este método consiste, basicamente, em determinar a posição correta na sub-tabela já ordenada de um elemento da sub-tabela ainda não ordenada, deslocando os elementos da sub-tabela ordenada, de modo a deixar vaga uma posição para inserir o novo elemento.

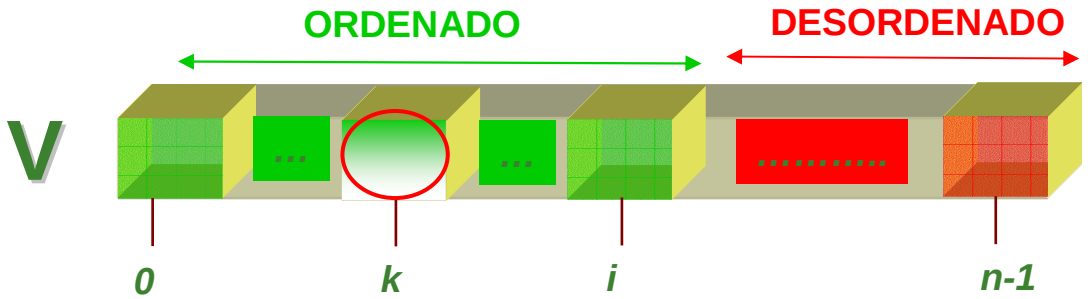
- **Começa-se por ordenar os dois primeiros elementos da tabela**
- **Seguidamente, coloca-se o terceiro elemento na sua posição correta relativamente aos dois elementos já ordenados**
- **O quarto elemento é inserido na posição correta relativamente à sub-tabela de três elementos já ordenada**
- **O processo é repetido até que toda a tabela fique ordenada**

ILUSTRAÇÃO



ESTRATÉGIA

Uma vez ordenado o conjunto {v[0], ... ,v[i-1] }, insere-se o elemento v[i] nesse conjunto, "empurrando" de uma posição os elementos que lhe ficam à direita:



1.3.1 Ordenação por Inserção Direta - Algoritmo

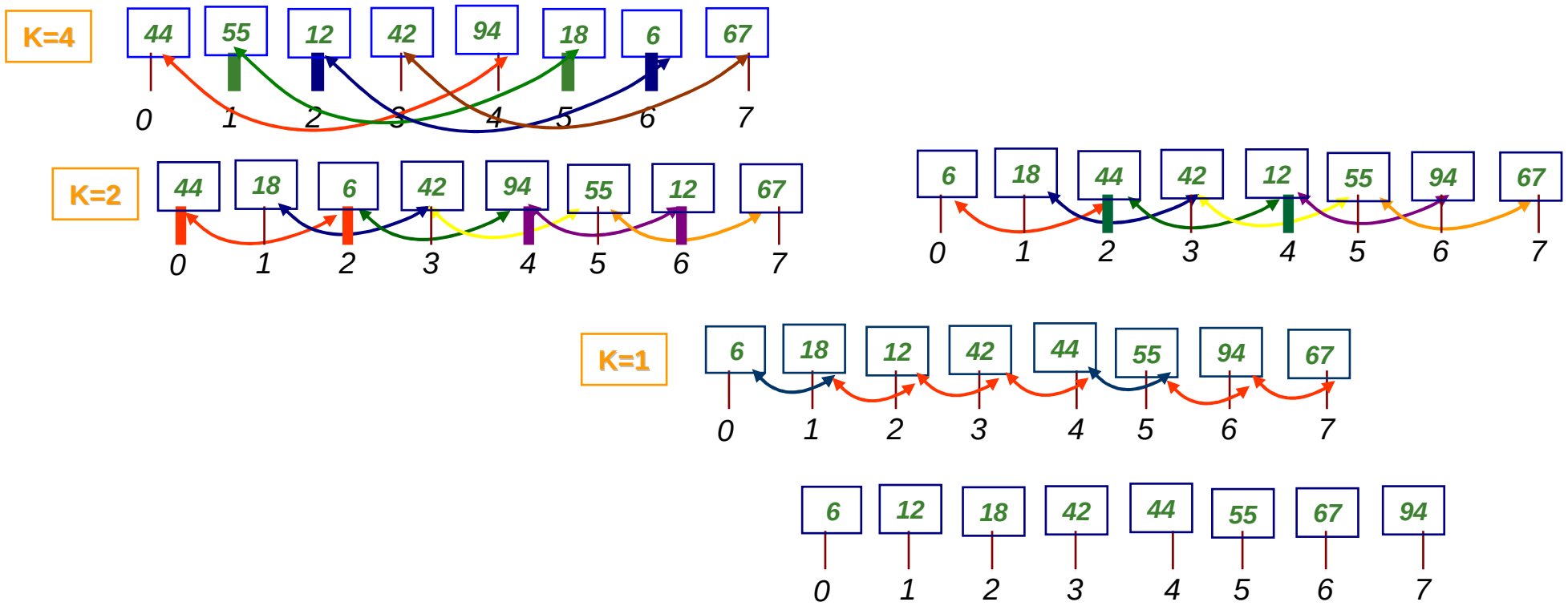
```
void insercaoOrdenada(TipoDados elemento, TipoDados vetor[], int tamanho)
{     int j = tamanho;
      //desloca os valores maiores para a direita
      while (j > 0 && vetor[j-1] > elemento)
      {
            vetor[j] = vetor[j-1];
            j = j-1;
      }
      vetor[j] = elemento;
}
```

```
void insercaoDireta(TipoDados vetor[], int tamanho)
{
      int i;
      for (i = 1; i < tamanho; i++)        // começa no 2º elemento
            insercaoOrdenada(vetor[i], vetor, i);
}
```

Algoritmo ShellSort - aperfeiçoamento do método de inserção direta

- O algoritmo *ShellSort* é uma melhoria relativamente ao algoritmo de inserção direta e consiste em dividir a tabela original em sequências de elementos, espaçados de *k* elementos, que já estão ordenadas.
- Após ordenar estas sequências-*k*, é escolhido um valor menor para *k* e a tabela é dividida num novo conjunto de sequências. Este método é repetido até que *k* seja igual a 1 e a sequência constituída por todos os elementos da tabela esteja ordenada.

Exemplo



1.3.2 Algoritmo ShellSort

```
void shellSort(TipoDados vetor[], int tamanho)
{
    int i, intervalo = tamanho/2;
    bool houveTroca;

    while (intervalo > 0)
   {
      do
      {
            houveTroca = false;
            for (i = 0; i < tamanho - intervalo; i++)
                  if (vetor[i] > vetor[i+intervalo])
                  {
                        troca(&vetor[i], &vetor[i+intervalo]);
                        houveTroca = true;
                  }
      } while (houveTroca);
      intervalo = intervalo/2;
   }
}
```

- O uso de sequências de elementos separados visa mover os elementos mais rapidamente para a sua posição correta. Existem muitas versões deste algoritmo, dependendo do método de ordenação das sequências.
- A versão que apresentamos usa o algoritmo BubbleSort para ordenar as sequências e faz o intervalo *k* igual a metade da dimensão da tabela. A amplitude dos intervalos seguintes é sempre escolhida como sendo metade da anterior (até atingir 1). Quando a amplitude do intervalo chega a 1, o algoritmo é, efetivamente, o BubbleSort. Nessa altura, como a tabela está quase ordenada, o algoritmo é bastante eficiente.

1.4 O algoritmo QuickSort

O algoritmo de ordenação *QuickSort* consiste em particionar a tabela a ordenar e, recursivamente, ordenar cada uma das partições. O princípio subjacente ao algoritmo pode traduzir-se nos seguintes passos:

(1) Escolher um dos elementos da tabela (chamado *pivot*)
(2) Colocar à sua esquerda todos os elementos que lhe são inferiores
(3) Colocar à sua direita todos os elementos superiores
(4) Ordenar a sub-tabela à esquerda
(5) Ordenar a sub-tabela à direita

Se o elemento escolhido inicialmente for ***x*** e após os passos 2 e 3 ***x*** estiver na posição **j**, e a tabela a ordenar tiver *N* elementos,

- cada elemento nas posições 0 a j-1 é menor ou igual a x
- cada elemento nas posições j+1 a N-1 é maior ou igual a x.

Após todos os elementos terem sido ordenados, o elemento *x* permanecerá na posição *j.*

Repetindo este processo para as sub-tabelas t[0] .. t[*j*-1] e t[*j+1*] .. t[*N-1*] e para todos as sub-tabelas criadas pelo processo, nas sucessivas iterações, obtém-se no final uma tabela ordenada.

1.4.1 Algoritmo QuickSort recursivo

O método **QuickSort** pode ser implementado usando uma função recursiva, expressa do seguinte modo:

```
Função QuickSort(lista, inf, sup)
  - Se (inf >= sup)
      Então return;
  - escolhe pivot (por exemplo o elemento na
                   primeira posição);
  - coloca os menores à esquerda e os maiores
     à direita, ficando o pivot na posição j
  - QuickSort(lista,inf,j-1); // ordena sub-tabela esquerda
  - QuickSort(lista,j+1,sup); // ordena sub-tabela direita
```

A sua concretização passa por desenvolver um algoritmo para colocar o elemento *pivot* na sua posição, todos os elementos menores em posições à esquerda e todos os elementos maiores em posições à direita do pivot, tarefa que é usualmente chamada partição da tabela.

Uma das formas de efetuar a partição consiste em usar dois "marcadores", um *acima* e outro *abaixo*, sendo o marcador abaixo inicializado com os índices *inferior* da sub-tabela a ordenar.

Em qualquer ponto da execução do algoritmo, todos os valores à esquerda do "marcador" *abaixo* são menores ou iguais ao pivot. Estes dois "marcadores" são movidos do seguinte modo:

- percorre-se o *array*
  - Se o elemento atual for menor que o pivot
    - aumenta-se a sub-tabela dos menores
    - move-se o elemento para esta sub-tabela

Este processo é repetido até que *se percorra todo o array*.

Nessa altura coloca-se o *pivot* na sua posição correta.

PARTIÇÃO DA TABELA

```
int particao(TipoDados vetor[], int inf, int sup)
{
        int abaixo, acima;
        TipoDados pivot = vetor[inf];
        abaixo = inf;

        for (acima = inf + 1; acima <= sup; acima++)
        {
                if (vetor[acima] < pivot)
                {
                        abaixo++;
                        troca(&vetor[abaixo], &vetor[acima]);
                }
        }
        troca(&vetor[inf], &vetor[abaixo]);

        return abaixo;
}
```

Algoritmo QuickSort – recursivo

```
void quickSortAux(TipoDados vetor[], int inf, int sup)
{
    int posPivot;

    if (inf < sup)
    {
        posPivot = particao(vetor, inf, sup);
        quickSortAux(vetor, inf          , posPivot - 1); // ordena sub-tabela esquerda
        quickSortAux(vetor, posPivot + 1, sup          ); // ordena sub-tabela direita
    }
}
```

```
void quickSort(TipoDados vetor[], int
tamanho)
{
        quickSortAux(vetor, 0, tamanho-1);
}
```

1.4.2 QuickSort não recursivo

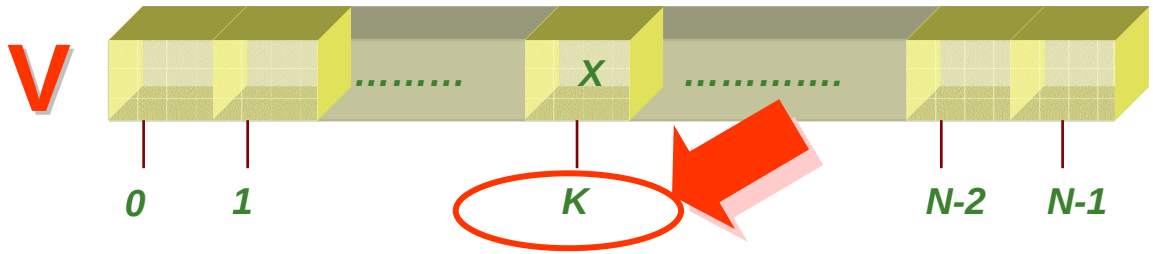
```
void quickSortNaoRecursivo(TipoDados vetor[], int tamanho)
{
        const int max = 12;
        int i, j, esq, dir, s = 0;
        TipoDados  elemento;
        struct {int l, r;} stack[max];
        stack[0].l = 0; stack[0].r = tamanho-1;
        do // atende o pedido do topo da lista
         {
                esq = stack[s].l; dir = stack[s].r; s = s-1;
                do  // parte t[esq] ... t[dir]
                 {
                        i = esq; j = dir; elemento = vetor[(esq+dir)/2];
                        do
                        {
                                while (vetor[i] < elemento) i = i+1;
                                while (vetor[j] > elemento) j = j -1;
                                if (i < j)
                                        troca(&vetor[i], &vetor[j]);
                                i = i+1; j = j-1;
                        } while (i <= j);
                        if (i < dir) // pedido colocado na pilha para ordenar a partição direita
                        {
                                s = s+1; stack[s].l = i; stack[s].r = dir;
                        }
                        dir = j;
                } while (esq < dir);
        } while (s > 0);
}
```

ESTRATÉGIA

Os limites das sub-tabelas são guardados numa pilha (*stack*) para posterior ordenação

2. Métodos de pesquisa em arrays

Os algoritmos de pesquisa têm como objectivo determinar a posição (*k*) em que se encontra um determinado valor *X* numa tabela unidimensional de *N* elementos.



- Os algoritmos que seguidamente apresentamos referem-se ao caso da pesquisa de um valor num *array*. Devolvem a posição onde o valor foi encontrado (índice), ou -1 se o valor não existir no *array*.
- A forma de pesquisa mais simples consiste em percorrer o *array* sequencialmente até o valor ser encontrado ou até ser atingido o fim do *array*.

int pesquisa(TipoDados chave, TipoDados vetor[], int tamanho)

2.1 Pesquisa Linear ou Sequencial

- A função a seguir apresentada permite percorrer um *array* **vetor** de dimensão **tamanho**, procurando um determinado valor (**chave**). Devolve a posição onde se encontra o valor procurado ou -1 se ele não existir no *array*.

```
int pesquisaLinear(TipoDados chave, TipoDados vetor[], int tamanho)
{
        int i = 0;
        while (vetor[i] != chave && i < tamanho)
                i++;
        if (i < tamanho)
                return i;
        else
                return -1;
}
```

Chamada da função

```
 ...
printf("\nQual o valor a procurar? ");
scanf("%d", &chave);
indice = pesquisaLinear(chave, vetor, tamanho);
if (indice > -1)
       printf("\nPosicao do valor dado: %d", indice);
else printf(“\nNao existe esse valor no array”);
...
```

2.2 Pesquisa Sequencial com Sentinela

No ciclo correspondente à pesquisa sequencial `while (vetor[i] != chave && i < tamanho)` **são feitos dois testes: verifica-se se o elemento pretendido foi encontrado e se já foi atingido o fim da tabela.**

Se existisse a garantia de que o elemento se encontrava na tabela, poderíamos prescindir do segundo teste, tornando a pesquisa mais eficiente.

Uma forma de eliminar esse teste, consiste em inserir o que normalmente se designa *sentinela*.

SOLUÇÃO

```
int PesqSequencialSentinela(TipoDados chave, TipoDados vetor[],int tamanho)
{
  vetor[tamanho] = chave;
  int i = 0;
  while (vetor[i] != chave)
       i++;
  if( i < tamanho)
    return i;
  else
    return -1;
}
```

Barreira

Atenção: o array terá de ser declarado com tamanho+1 elementos...

2.3 Pesquisa Binária

A pesquisa sequencial é muito simples, mas pode exigir a inspeção de todos os elementos da tabela.

O algoritmo de **Pesquisa Binária** é uma alternativa, regra geral mais eficiente - demora menos tempo a encontrar o valor - mas **requer que o vetor em que se vai efetuar a pesquisa esteja ordenado**.

No algoritmo seguinte suporemos que o vetor a pesquisar se encontra ordenado por ordem **crescente**.

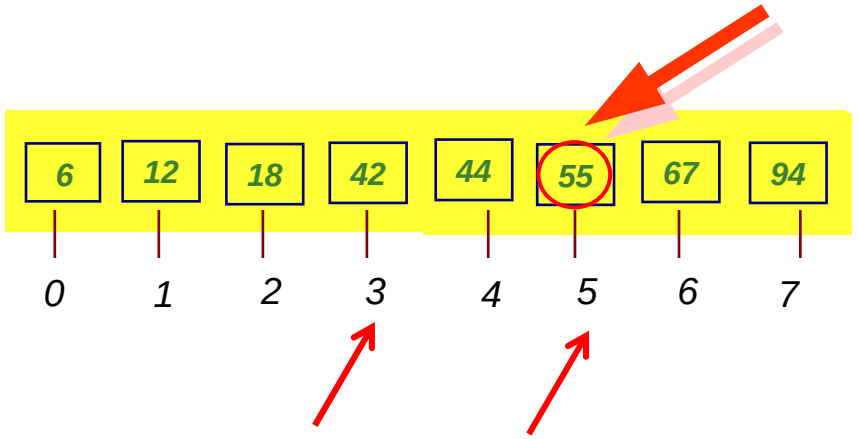
ESTRATÉGIA

Seleciona-se um valor no meio da lista e compara-se com o valor que se está a procurar.
Se o valor a procurar for maior que o valor selecionado, repete-se o processo para a metade da lista posterior ao valor selecionado. Se for menor, repete-se o processo para a metade da lista anterior ao valor selecionado.
O processo é repetido até o valor ser encontrado, ou até que a metade em que a procura irá ser feita seja vazia (neste caso o valor não está na lista).

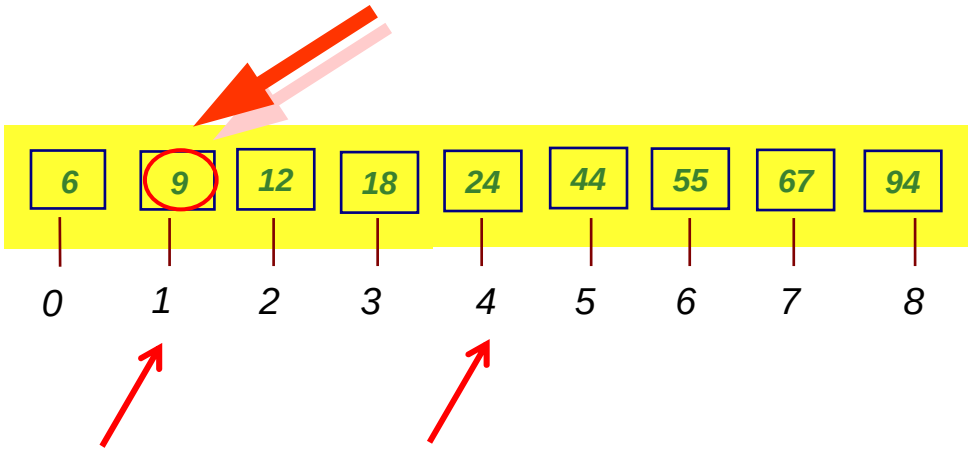
Em cada passo, a pesquisa binária reduz a metade o número de elementos a considerar. Daí o nome do algoritmo.

Exemplos

N par



N ímpar



Pesquisa binária – algoritmo

```
int pesquisaBinaria (TipoDados chave, TipoDados vetor[], int tamanho)
{
   int posicao, inf, sup, indMeio;
   posicao = -1; inf = 0; sup = tamanho-1;
   while (sup >= inf  &&  posicao==-1)
   {
            indMeio = (sup+inf)/2;
            if  (vetor[indMeio] == chave)
                        posicao = indMeio;
            else
                if (vetor[indMeio] < chave)
                        inf = indMeio+1;
                else //vetor[indMeio] > chave
                        sup = indMeio-1;
    }
    return posicao;
}
```

Sendo **vetor** um *array* ordenado com **N** elementos e **chave** o valor que se procura, a função poderá ser chamada através da instrução de atribuição:

```
posicao = pesquisaBinaria(chave, vetor, N);
```

Pesquisa binária – algoritmo recursivo

```
int pesquisaBinariaRecursiva(TipoDados chave, TipoDados vetor[], int inf, int sup)
{
    int pos, indMeio;
    if (inf > sup)
      pos = -1;                  // ou return -1;
    else
    {
      indMeio = (sup+inf)/2;
      if (vetor[indMeio] == chave)
          pos = indMeio;    // ou return indMeio;
      else
          if (vetor[indMeio] < chave)
              pos = pesquisaBinariaRecursiva(chave, vetor, indMeio +1, sup);
          else
              pos = pesquisaBinariaRecursiva(chave, vetor, inf, indMeio -1);
    }
    return pos;
}
```

Sendo **vetor** um *array* ordenado de **N** elementos e **chave** o valor procurado, a função poderá ser chamada através da instrução de atribuição:

posicao = pesquisaBinariaRecursiva(chave, vetor, 0, N-1);