departamento de informática

engenharia informática

Algoritmos e Programação

1º Ano - 1º Semestre

5. Métodos de Ordenação e Pesquisa

Escola Superior de Tecnologia e Gestão de Viseu

Agradecimentos a Francisco Morgado, Carlos Simões e Jorge Loureiro

5. Métodos de Ordenação e Métodos de Pesquisa

Os programas apresentados ao longo desta secção utilizam um vetor do tipo inteiro, de dimensão N (N definido como constante).

5.1 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

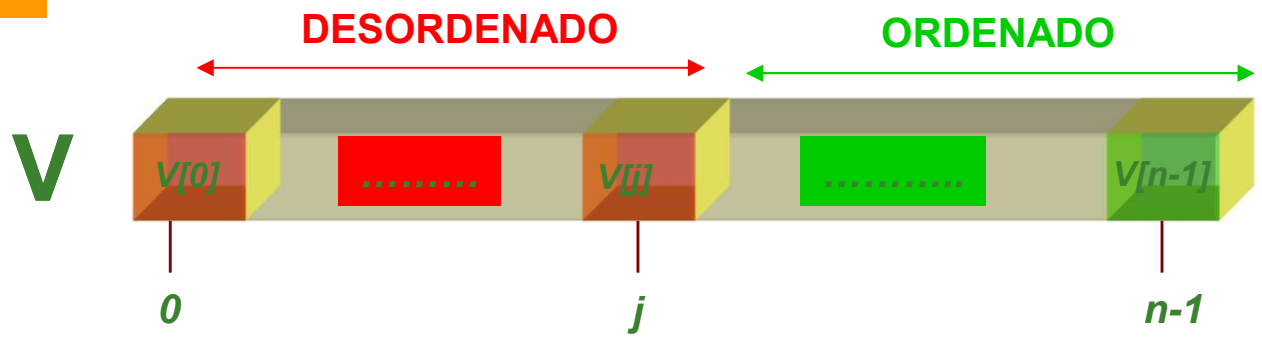
- ☑ **Lista telefónica**
- ☑ **Gestão de clientes**
- ☑ **Bibliotecas**
- ☑ **Dicionários**
- ☑ **etc.**

5.1.1 Ordenação por Seleção Direta

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

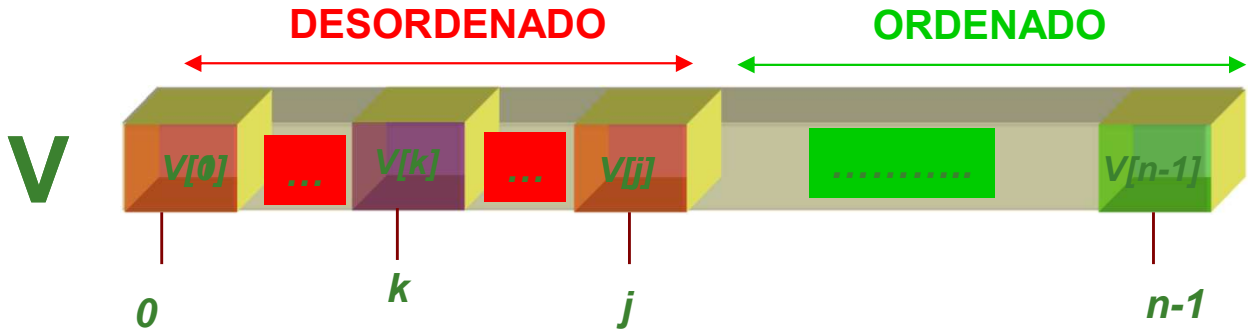
- Primeiramente procuramos o maior valor a partir da primeira posição e colocamo-lo em último lugar
- Em seguida procuramos o segundo maior valor e colocamo-lo na penúltima posição
- O processo é sucessivamente repetido até chegar ao fim do *array*

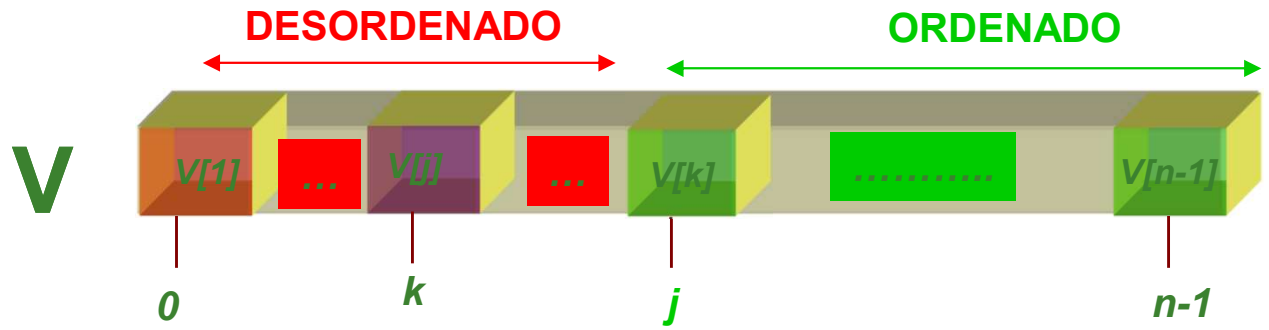
ILUSTRAÇÃO



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





EXEMPLO de ordenação dum vetor pelo método *Seleção Direta: procurar máximo*

Procura-se o máximo e coloca-se no local respetivo

Para a ordenação deste vetor de 8 elementos foram necessárias 7 passagens. Generalizando, **se o vetor tivesse N elementos seriam necessárias N-1 passagens**.

Este algoritmo é relativamente dispendioso em termos de operação. São necessárias N-1 comparações, o mínimo tem de ser determinado N vezes e o número total de comparações é quadrático em N, ou seja é de ordem N^2.

Ordenação por Seleção Direta – Algoritmo: pesquisa do máximo

```
void SelectionSort(int * vet, int N)
{
    int i,j,posMax;
    int temp;
    for (i= N-1; i>0; i--)
    {
        posMax=i;
        for (j=0; j<=i-1;j++)
        {
            if (vet[j]>vet[posMax])
                posMax=j;
         }
         // troca de valores entre vet[i] e vet[posMax]
         temp=vet[i];
         vet[i]=vet[posMax];
         vet[posMax]=temp;
     }
}
```

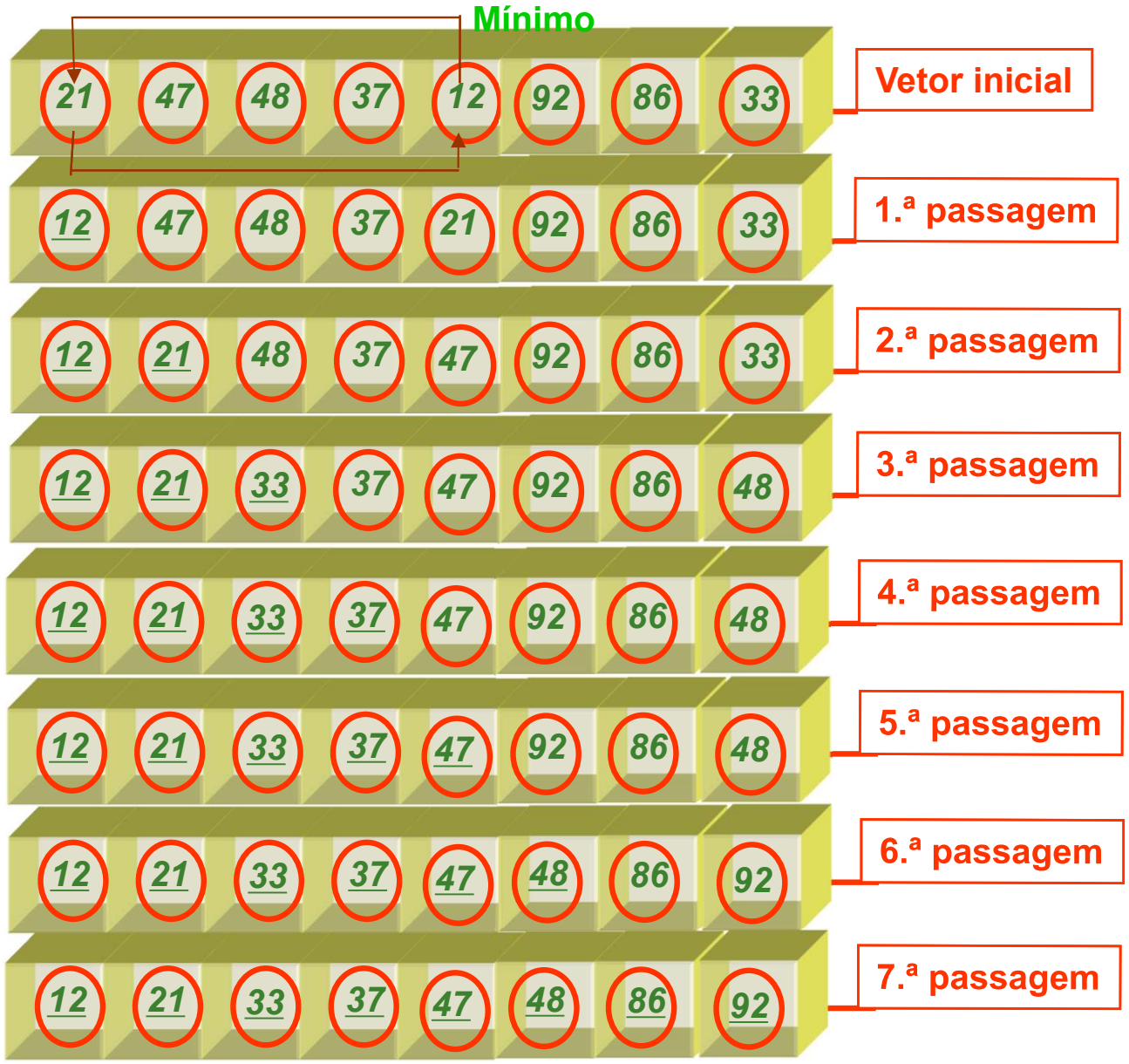
Estabelece o intervalo de procura: começa com todo o vetor e vai sucessivamente diminuindo;

Procura o maior valor dentro do intervalo de procura corrente

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

```
temp := a;
a := b;
b := temp;
```

EXEMPLO de ordenação dum vetor pelo método *Seleção Direta: procurar mínimo*



Ordenação por Seleção Direta – Algoritmo: pesquisa do mínimo

```
void SelectionSort(int * vet, int N)
{
    int i,j,posMin;
    int temp;
    for (i=0; i<N-1; i++)
    {
        posMin=i;
        for (j=i+1; j<N; j++)
        {
            if (vet[j]<vet[posMin])
                posMin=j;
         }
         // troca de valores entre vet[i] e vet[posMin]
         temp=vet[i];
         vet[i]=vet[posMin];
         vet[posMin]=temp;
     }
}
```

Estabelece o intervalo de procura: começa no início e percorre o vetor, não necessitando ir até ao último

Procura o menor valor entre o índice i e o fim

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

```
temp := a;
a := b;
b := temp;
```

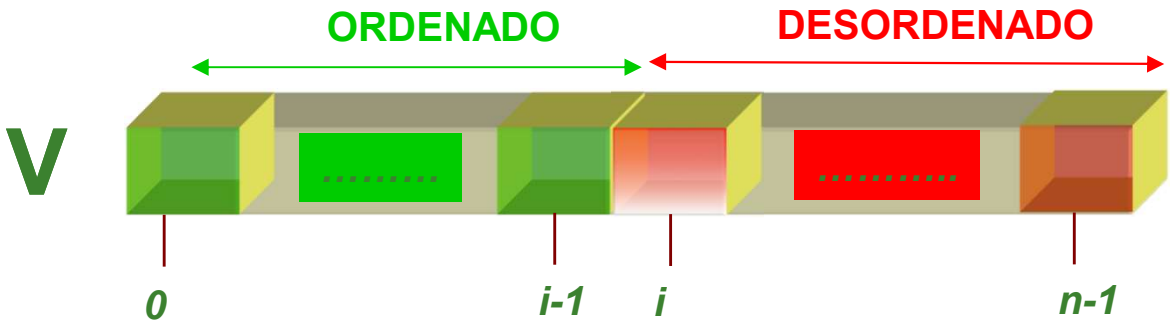
Coloca o elemento na posição inicial

5.1.2 Ordenação por Inserção Direta

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

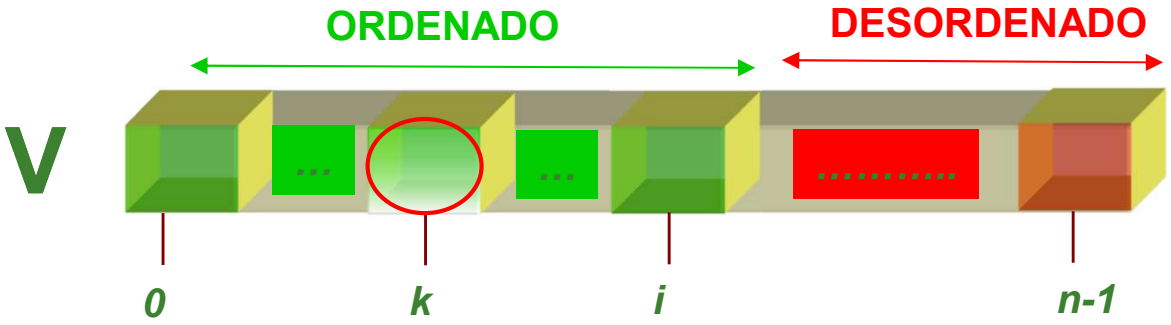
- Começa-se por ordenar os dois primeiros elementos do vetor;
- Seguidamente, coloca-se o terceiro elemento na sua posição correta relativamente aos dois elementos já ordenados;
- O quarto elemento é inserido na posição correta relativamente à sub-vetor de três elementos já ordenada;
- O processo é repetido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

ILUSTRAÇÃO



ESTRATÉGIA

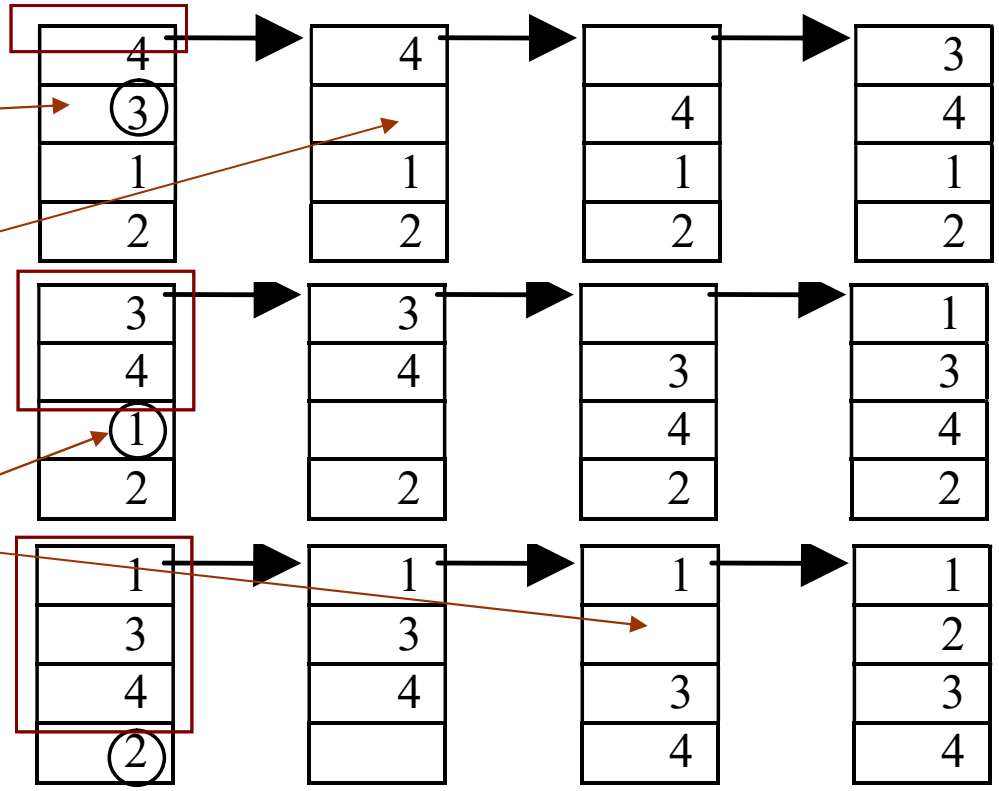
Uma vez ordenado o conjunto {v[0], ... ,v[i-1] }, insere-se o elemento v[i] nesse conjunto, "empurrando" uma posição os elementos que lhe ficam à direita:



EXEMPLO de ordenação dum vetor pelo método *Inserção Direta*

Passos:

- Considera-se um elemento do vetor, começando no segundo (dado que o primeiro elemento não pode ser comparado com nenhum anterior).
- Compara-se o elemento com os elementos anteriores a ele no vetor (no sentido decrescente dos índices) e deslocam-se estes elementos uma posição para a frente até que se encontre um elemento menor que o elemento considerado. Este elemento menor não é deslocado, ficando assim uma posição vaga a seguir a ele.
- Quando é encontrado um elemento inferior ao elemento a inserir, este é colocado na posição imediatamente posterior ao elemento menor.
- Finalmente, é considerado o próximo elemento do vetor e o processo é repetido até que todos os elementos estejam ordenados.



Para a ordenação deste vetor de 8 elementos foram necessárias 7 passagens. Generalizando, **se o vetor tivesse N elementos seriam necessárias N-1 passagens**.

Este algoritmo é relativamente dispendioso em termos de operação. São necessárias N-1 comparações, o mínimo tem de ser determinados N vezes e o número total de comparações é quadrático em N, ou seja é de ordem N^2.

Ordenação por Inserção Direta - Algoritmo

```
void InsertionSort(int * vet, int N)
{
   int i, j;
   int temp;
   for (i = 1; i < N; i++)
   {
      temp=vet[i];
      j=i;
      while(j>0 && vet[j-1] > temp) // procura posição do elemento i
       {
          vet[j]=vet[j-1]; // desloca o elemento j-1para a direita
          j--;
       }
       vet[j]=temp; // coloca o elemento i na posição correcta
    }
}
```

Inicia no 2.º elemento

temp fica com o elemento a comparar seleccionado no loop externo

Compara o elemento selecionado no loop externo com o próximo elemento no vetor: Se o elemento selecionado no loop externo for < do que o elemento selecionado pelo loop interno, há o shift para criar espaço para o elemento

Análogia: ordenação de ficha de alunos

1.ª ficha (Simão): coloca-se no canto sup. esq. da secretária

2.ª ficha (Carlos): move-se Simão para a direita e coloca-se carlos na posição de Simão

3.ª ficha (Tiago): pode ser colocado à direita sem mover quaisquer fichas

4.ª ficha (António): tem de colocar no início da lista. Movem-se todas as fichas para a direita para deixar a 1.ª posição livre.

5.1.3 Ordenação por Troca Direta ou *BubbleSort*

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

- **Em geral, não basta uma única passagem pelo vetor para que fique ordenada.**

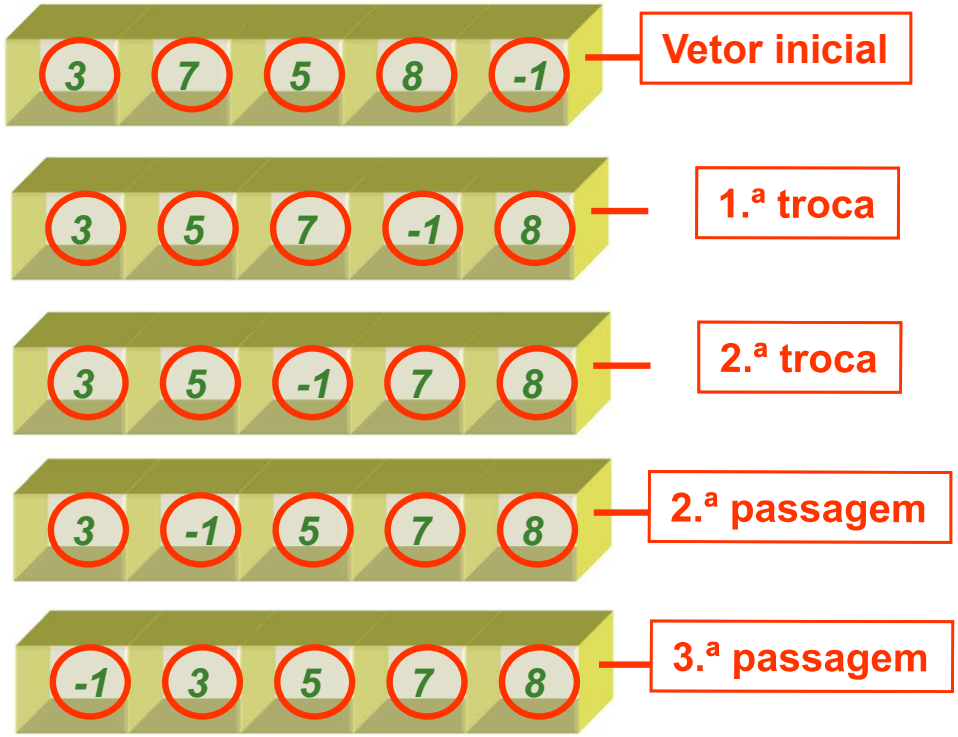
  → *Na primeira passagem o maior valor é colocado no fim do vetor (posição n).*

  → *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 do vetor 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*



Descrição do funcionamento:

1.ª Passagem

- Primeiro compara-se o 3 com o 7 e como estão em ordem, não se trocam;
- Seguidamente compara-se o 7 com o 5 que se trocam;
- Como o 7 está agora na posição do 5, vai comparar-se o 7 com o -1 e trocam-se;
- Finalmente compara-se o 7 e o 8 que já estão na posição correcta.

Para a ordenação deste vetor de 5 elementos foram necessárias 3 passagens. Generalizando, **se o vetor tivesse N elementos seriam necessárias N-1 passagens**.

Ordenação por Troca Direta ou *BubbleSort* - Algoritmo

```
public void BubbleSort(int * vet, int N)
{
    int i,j,
    int temp;
    for (i=0; i<N-1; i++)
    {
        for (j=0;j<N-i-1;j++)
        {
            // compara elementos adjacentes para ver se estão fora de ordem
            if (vet[j] > vet[j+1]) // se estiverem
            {
                temp=vet[j];
                vet[j]=vet[j+1];
                vet[j+1]=temp;
            }
        }
    }
}
```

Ordenação por ordem crescente

NOTAR QUE …

QUANDO HÁ UMA PASSAGEM SEM QUALQUER TROCA, O *ARRAY* JÁ SE ENCONTRA ORDENADO, PODENDO TERMINAR-SE A EXECUÇÃO.

ESTA CONSTATAÇÃO SUGERE A ESCRITA DE UM NOVO ALGORITMO.

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

```
void FastBubbleSort(int * vet, int N)
{
    int i,j, houveTroca=1;
    int temp;
    for(i=0; i<N-1 && houveTroca; i++)
    {
        houveTroca=0;
        for (j=0;j<N-i-1;j++)
        {
            // compara elementos adjacentes para ver se estão fora de ordem
            if (vet[j] > vet[j+1]) // se estiverem
            {
               houveTroca=1;
                temp=vet[j];  // troca vet[j] com vet[j+1]
                vet[j]=vet[j+1];
                vet[j+1]=temp;
            }
        }
    }
}
```

5.1.4 Complexidade Relativa dos Três Algoritmos

Designando por

- **n o número de itens a ordenar**
- **C a operação de comparação entre dois itens**
- **M o movimento de um item**

apresenta-se no vetor seguinte a complexidade relativa dos três algoritmos de ordenação estudados, em termos do número de operações exigido por cada um deles.

Método	Operações	#Mínimo	#Médio	#Máximo
Inserção direta	C	n-1	$(n^2+n-2)/4$	$(n^2-n)/2-1$
	M	2(n-1)	$(n^2-9n-10)/4$	$(n^2+3n-4)/2$
Seleção direta	C	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	M	3(n-1)	n(ln n+ 0.75)	$n^2/4+3(n-1)$
Troca direta "Bubblesort"	C	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	M	0	$(n^2-n)*0.75$	$(n^2-n)*1.5$

5.1.5 Algoritmo de Ordenação Avançados

ShellSort

O nome é devido ao seu criador, Donald Shell.

Este algoritmo é, fundamentalmente, uma melhoria da ordenação por um dos métodos vistos atrás, mas, muitas vezes, a inserção direta.

O conceito chave neste algoritmo é que ele compara itens que estão distantes e não adjacentes. Divide-se o vetor em sequências de elementos espaçados de k elementos que são ordenadas. Se k (normalmente denominado incremento ou intervalo) for 5, os elementos vet[0], vet[5], vet[10], ..., fazem parte da 1.ª sequência, vet[1], vet[6], vet[11], ..., constituem a 2.ª sequência, etc. Após ordenar estas k sequências (normalmente utilizando inserção direta), é escolhido um valor menor para k e o vetor dividida num novo conjunto de sequências, depois ordenadas. O processo é repetido até que k seja igual a 1 e a sequência constituída por todos os elementos seja ordenada.

Existem muitas versões deste algoritmo, variando especialmente na geração de k e no algoritmo utilizado para a ordenação de cada sequência.

EXEMPLO de ordenação dum vetor pelo método *ShellSort*

A divisão do vetor em sequências, e sua ordenação, compreende-se melhor se imaginarmos que:

- **o vetor inicial é rearranjado sob a forma bidimensional**
- **ordenar as colunas do array resultante**

→ *O efeito é que a sequência de dados é parcialmente ordenada;*

→ *O processo é repetido, mas de cada vez com um menor número de colunas;*

→ *O nível de ordenação da sequência é sucessivamente maior, até que, no último passo, os dados estão completamente ordenados;*

→ *Contudo, o número de operações de ordenação necessárias em cada passo é limitado, devido à pre-ordenação da sequência obtida nos passos anteriores.*

- Seja 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2 a sequência de dados a ser ordenada.
- Rearranjando-a sob a forma dum array de 7 colunas (k=7) (à esquerda) e ordenando as colunas (à direita)

3 7 9 0 5 1 6		3 3 2 0 5 1 5
8 4 2 0 6 1 5	→	7 4 4 0 6 1 6
7 3 4 9 8 2		8 7 9 9 8 2

Os elementos 8 e 9 foram para o fim da sequência, mas um elemento menor (2) ainda está lá.

No próximo passo,vamos reordenar a sequência em 3 colunas.

3 3 2		0 0 1
0 5 1		1 2 2
5 7 4		3 3 4
4 0 6	→	4 5 6
1 6 8		5 6 8
7 9 9		7 7 9
8 2		8 9

Agora a sequência está quase ordenada.

No último passo, rearranja-se o array numa coluna e só o 6, 8 e 9 têm de mover-se um pouco para a sua posição correta.

Versão do Algoritmo ShellSort que utiliza o BubbleSort para ordenar as sequências

```
void ShellSortV1(int * vet, int N)
{
    int i, intervalo, houveTroca;
    int temp;
    intervalo=N/2;
    do
    {
        do
        {
            houveTroca = 0;
            for (i = 0; i < N - intervalo; i++)
            {
                if (vet[i] > vet[i + intervalo])
                {
                    temp = vet[i];  // troca elementos
                    vet[i]=vet[i+intervalo];
                    vet[i+intervalo]=temp;
                    houveTroca = 1;
                }
            }
        }
        while (houveTroca);
        intervalo /= 2;
    }
    while (intervalo > 0);
}
```

Outra Versão do Algoritmo ShellSort

```
void ShellSortV2(int * vet, int N)
{
     int inner, h=1;
     int temp;
     // procurar o maior valor h possível
     while (h <= N / 3)
          h = h * 3 + 1;
     while (h > 0)
     {
          for (int outer = h; outer <= N - 1; outer++)
          {
               temp = vet[outer];
               inner = outer;
               while ((inner > h - 1) && vet[inner - h] >= temp)
               {
                    vet[inner] = vet[inner - h];
                    inner - = h;
                }
                vet[inner] = temp;
          }
          h = (h - 1) / 3;
     }
}
```

Outros Algoritmos de Ordenação Avançados

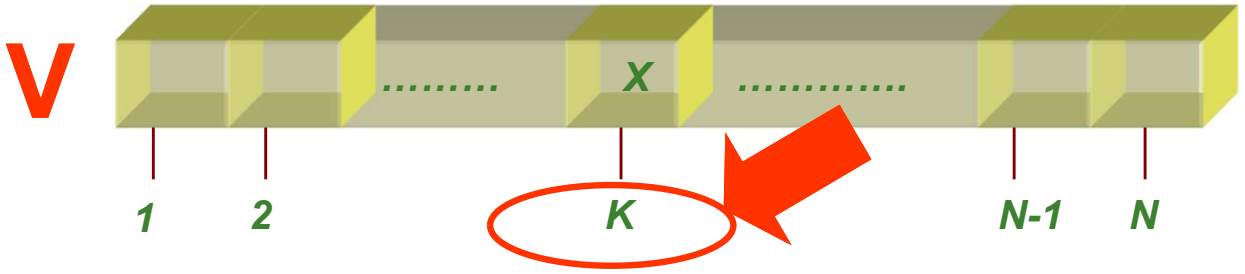
Há muitos outros algoritmos de ordenação que procuram uma maior eficiência do processo de ordenação:

- → ShakerSort;
- → HeapSort;
- → MergeSort;
- → QuickSort (reputado como o mais rápido), especialmente verdade para grandes arrays, maioritariamente não ordenados;
- → Outros.

O estudante pode procurar os algoritmos e respetiva descrição na Internet ou em livros como “Data Structures and Algorithms Using C#”, Michael McMillan, Cambridge University Press, ISBN 978-0-521-67015-2, cota [004.43 C# MCM DAT]. Facilmente convertem o código para C.

5.2 Métodos de pesquisa num vetor

- Os algoritmos de pesquisa têm como objetivo determinar a posição (k) em que se encontra um determinado valor X num vetor unidimensional de N elementos.



Os algoritmos que seguidamente apresentamos referem-se ao caso da pesquisa de um valor num vetor. Devolvem a posição onde o valor foi encontrado (índice), ou -1 se o valor não existir no vetor.

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.

5.2.1 Pesquisa Linear ou Sequencial

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

Descrição Algorítmica → ***Codificação em C***

```
...
índice ← 0
Enquanto (t[indice] <> chave) E (índice < N)
Faz
          índice ← índice +1
Fim Enquanto
Se  indice <N Então
   posicao ← i
Senão posicao ← -1
Fim Se
...
```

```c
int PesquisaLinear(int * vet, int N, int chave)
{
  int indice=0;
  while (indice < N && vet[indice] != chave)
  {
    indice++;
  }
  if (indice < N)
     return indice; // encontrou a chave
  else
     return -1; // não encontrou a chave
}
```

Chamada do método

```
 ...
posicao = PesquisaLinear(vet, N, chave); // Posicao é variável inteira
if (posicao>0)
  printf(”O valor procurado encontra-se na posição %d”, posicao)
else printf(“Não existe esse valor no vetor”);

 ...
```

5.2.2 Pesquisa Sequencial com Sentinela

No ciclo correspondente à pesquisa sequencial, no programa do diapositivo 25

while (arr[indice] != chave && indice < N)

são feitos dois testes:

1. verifica-se se o elemento pretendido foi encontrado e
2. se já foi atingido o fim do vetor.

Se existisse a garantia de que o elemento se encontrava no vetor, 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***, o elemento a procurar, evitando assim estar sempre a testar o fim do array no ciclo while.

Codificação em C

```
int PesqLinearSentinel(int *vet, int N, int chave)
{
  int indice=0;
  vet [N] = chave;
  while (arr[indice] != chave)
  {
    indice++;
  }
  IF (indice < N)
     return indice; // encontrou a chave
  ELSE
     return -1; // não encontrou a chave
END;
```

O vetor vet tem de ter uma última posição vazia, para incluir a sentinela, sendo agora definido como int vet[N+1]

Sentinela

A utilização de sentinela exige a existência de uma posição adicional no array e a primeira operação a efetuar (antes da pesquisa) é a inserção da sentinela no array.

5.2.3 Pesquisa Binária

- A pesquisa linear é muito simples, mas pode exigir a inspeção de todos os elementos do vetor.

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

No algoritmo seguinte suporemos que o vetor a pesquisar se encontra ordenada 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.

Exemplo

A ESTRATÉGIA de pesquisa é semelhante ao adivinhar um número escolhido por um amigo: vamos supor no intervalo entre 1 e 100.

Cada tentativa tem uma resposta: ou o número é correto ou o número indicado foi alto ou baixo.

A melhor estratégia consiste em indicar 50 na primeira tentativa. Se a resposta for "muito alto", escolhe-se 25; escolhe-se 75, de contrário. De cada vez que é feita nova tentativa, seleciona-se o ponto intermédio do intervalo menor ou maior, dependendo da resposta obtida. O número escolhido pelo colega será encontrado.

Jogo da Adivinha – O número a adivinhar é 82

1 50 82 100

primeira tentativa 50; resposta: muito baixo

51 75 82 100

segunda tentativa 75; resposta: muito baixo

76 82 88 100

terceira tentativa 88; resposta: muito alto

76 81 82 87

quarta tentativa 81; resposta: muito baixo

82 84 87

quinta tentativa 84; resposta: muito alto

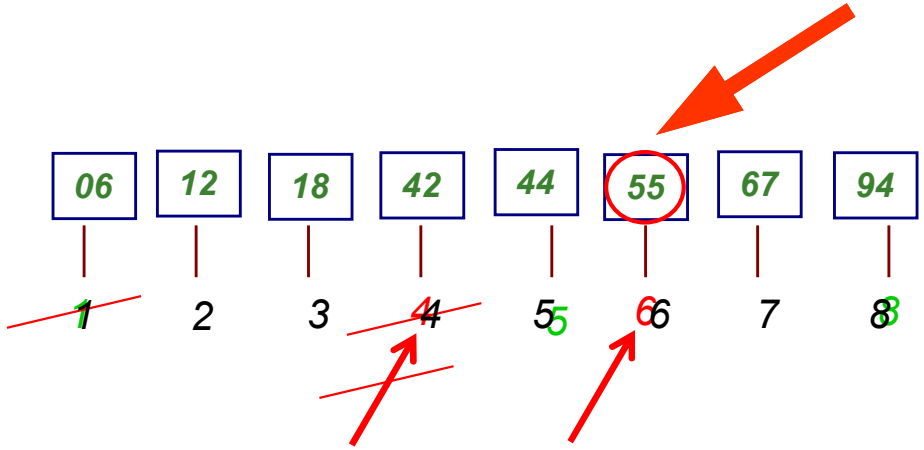
82 83

sexta tentativa 82;
resposta: **correcto**

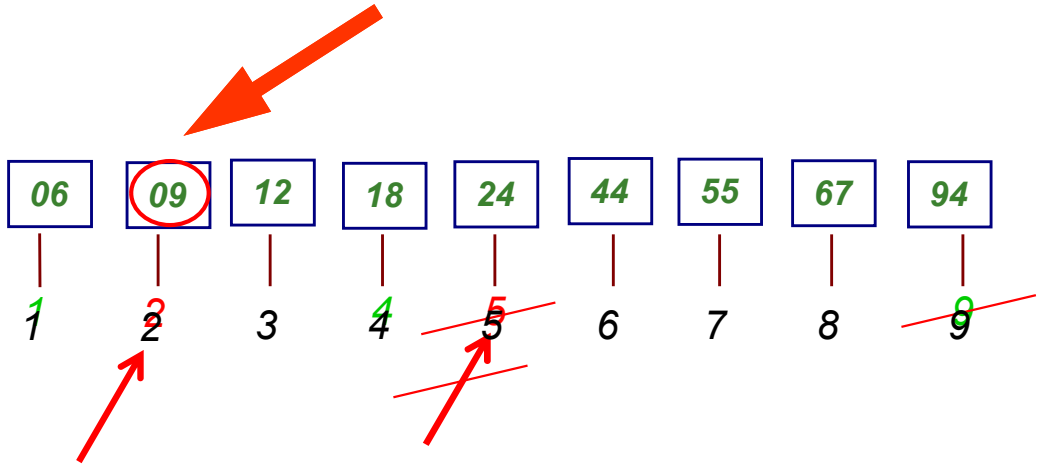
Ponto intermédio é 82.5: truncado para 82

Exemplos

N par



N ímpar



Pesquisa binária – algoritmo iterativo

```
int PesquisaBinaria(int * vet, int N, int chave)
{
   int indInf, indSup, indMeio;
   indInf = 0; indSup = N - 1;
   while (indSup>=indInf)
   {
     indMeio = (int)(indInf + indSup) / 2;
     if (vet[indMeio] == chave)
        return indMeio;
     else
     {
        if (vet[indMeio] < chave)
          indInf = indMeio + 1;
        else
          indSup = indMeio -1;
     }
   }
   return -1;
}
```

Sendo vet um vetor ordenado de tamanho N e x o valor que se procura, a função poderá ser chamada através da instrução de atribuição:

```
Posicao = PesquisaBinaria(vet, N, x);
```

Pesquisa binária – algoritmo recursivo

```
int PesqBinRec(int * vet, int indInf, int indSup, int chave)
{
   int ind_meio;
   if (indInf > indSup)
      return -1;
   else
   {
     indMeio =(int)(indInf + indSup) / 2;
     if (chave < vet[indMeio])
        return PesqBinRec(vet, indInf, indMeio-1, chave);
     else if (vet[indMeio] == chave)
        return indMeio;
     else
        return PesqBinRec(vet, indMeio+1, indSup, chave);
   }
}
```

Sendo vet um *array* ordenado com N elementos e x o valor que se procura (determinar o índice da sua posição no *array*), a função poderá ser chamada através da instrução de atribuição:

```
Posicao = PesquisaBinaria(vet, 0, N-1, x);
```

Posicao é a variável inteira a que se pretende atribuir o índice do valor x no array.