# Pesquisa e Ordenação

***SC121 - Introdução à Programação***

ICMC - USP - São Carlos
Maio de 2009

# Algoritmo de Procura

- Algoritmo de Procura
  - O problema de procurar, pesquisar alguma informação numa tabela ou num catálogo é muito comum
- Exemplo:
  - procurar o telefone de uma pessoa no catálogo
  - procurar o nº da conta de um certo cliente
  - consultar o seu saldo em um terminal automático

# Algoritmo de Procura

- A tarefa de “pesquisa”, “procura” ou “busca” é, como se pode imaginar, *uma das funções mais utilizadas*
- rotinas que a executem de uma forma *eficiente*
- *Eficiente*: uma rotina que faça a busca no menor tempo possível
- O TEMPO GASTO pesquisando dados em tabelas depende do TAMANHO da tabela.

# Algoritmo de Procura

O tempo gasto para se descobrir um telefone na lista de São Paulo

>

O tempo gasto para se descobrir um telefone na lista de uma cidade do interior com 3000 habitantes

# Algoritmo de Procura

- TEMPO GASTO pode variar muito dependendo do algoritmo utilizado
- Para os algoritmos de busca que se seguem vamos denotar por:
  - **Tab** - um vetor de n posições
  - **Dado** - elementos que devemos procurar em **Tab**
  - **Achou** - indica o sucesso ou falha na pesquisa
  - **Ind** - aponta para a posição do elemento encontrado

# Parte que se repetirá...

(*)

```
Tab[100] of integer;    {tabela de pesquisa}
Int Ind;{retorna a posição do elemento}
boolean Achou; {sucesso ou falha na busca}
Int Dado;  {valor a ser procurado}
Int I;     {auxiliar}
```

# Algoritmo 1 - Busca em Tabela

```
#includes
(*)
Int main(){
       printf("Entre com os valores da  tabela");
       for (i=0;i<N;i++)
      scanf("%d", &Tab[I]);
       printf("Entre com o valor a ser procurado");
       scanf("%d",&Dado);
       Achou=false;
       for (i=0;i<N;i++)
            if Tab[I] == Dado{
                  Achou=true;
                  Ind = I;
               }
       if Achou
        printf("%d %d se encontra na posição", Dado,
Ind)
       else printf("%d não encontra na tabela",
Dado);
   getch();
}
```

**N comparações**

**( ~ percorrer 1 dicionário todo)**

# Algoritmo 2 - Busca em tabela

- Pára-se o processo de busca quando o dado for encontrado.
- 1º modo

```
I ← 1
Enquanto (Tab [I] <> dado) and (I<=n) faça
        I= 1 + 1
```

- Este algoritmo não funciona pois

`Ind ← N +1`

e

`Tab [N+1]` - referência inválida

# 2º Modo (Com o uso da variável BOOLEAN)

```
Program Alg2;
(*)
Int Main(){
       Achou:= false;  Procura:= true;
       Ind:= 0;
       printf("Entre com os valores da  tabela");
       for (i=0;i<N;i++)
     scanf("%d", &Tab[I]);
       printf("Entre com o
       scanf("%d",&Dado);
        while Procura{
       Ind++;
       if Ind > N Procura= false;
       else Procura = Tab[Ind] <> Dado
        }
       if Ind <= N
        Achou = true;
       if Achou  printf("%d se encontra na posição %d",Dado,
Ind);
   else printf(" %d não encontra na tabela", Dado);
}
```

**Dado pode estar na 1ª posição ou Dado pode estar na última**

**Na média: N/2 comparações**

**Obs: 2 testes**
- **Procura = true**
- **Ind > N**

# Algoritmo 3: Busca com Sentinela

- Se soubermos que o dado se encontra na tabela na precisaríamos fazer o teste **Ind > N**
- **INSERIR** o Dado no final da tabela

```
Program Alg3; {Bem + simples}
(*)
begin
    printf(“Entre com os valores da  tabela”);
       for (i=0;i<N;i++)
     scanf(“%d”, &Tab[I]);
       printf(“Entre com o valor a ser procurado”);
       scanf(“%d”,&Dado);
    Achou= false;
    Ind= 0;
    Tab[N]= Dado;
 while (Tab[Ind] != Dado){
            Ind++;
      }
    Achou= Ind != N;
    if Achou
    printf(“%d se encontra na posição %d”,Dado, Ind);
   else printf(“ %d não se encontra na tabela”, Dado);
end.
```

# Algoritmo 4 - Busca binária (+ eficiente)

**Dicionário - *Tarol***

- Abre-se o dicionário ao meio → letra J
- Abandonamos a 1ª metade
- Tomamos a metade a partir de J → letra P
- Abandonamos a 1ª metade
- Tomamos a metade a partir de P → letra S (pág. 1318)
- Dividimos novamente, chegamos a palavra *Tomo* (pág. 1386)

∴ palavra está entre 1318 e 1386

# Algoritmo 4 - Busca binária

- A cada passo dividimos a área de pesquisa à metade
- Caso o dicionário tenha 1500 palavras

1500/2 → 750 24/2 → 12

750/2 → 3 6

376/2 → 1

188/2 → 94 3/2 → 1,5

94/2 → 47 2/2 → 1

47/2 → 23,5

**11 pesquisas = $\log_2$ 1500**

**N ⇒ $\log_2$ N**

**32.000 ⇒ 15 comparações**

```
Program Alg4;
(*)
Inicio, Fim, Meio: Integer;
begin
       printf(“Entre com os valores da  tabela”);
       for (i=0;i<N;i++)
      scanf(“%d”, &Tab[I]);
       printf(“Entre com o valor a ser procurado”);
       scanf(“%d”,&Dado);
       Achou= false;
       Inicio=1; Fim= N; Meio = (1+N)/2;
        while (Dado <> Tab[Meio]) and (Inicio <> Fim){
          if Dado > Tab[Meio]
           Inicio = Meio + 1;
          else Fim = Meio;
          Meio= (Inicio + Fim) / 2;
    }
       Achou = Dado = Tab[Meio];
       if Achou
       printf(“%d se encontra na posição %d”,Dado, Meio);
    else printf(“ %d não encontra na tabela”, Dado);
end.
```

# Algoritmos de Ordenação

# Algoritmos de Ordenação

- Da mesma forma que a BUSCA, a ORDENAÇÃO é uma das tarefas básicas em processamento de dados
- A *ordenação* de um vetor significa fazer com que os seus elementos estejam colocados de acordo com algum critério de ordenação

# Algoritmos de Ordenação (continuação)

- Supor que os elementos de um vetor sejam inteiros
- Critério de ordenação: ordem crescente ou decrescente
- Assim, se o vetor tem N elementos:

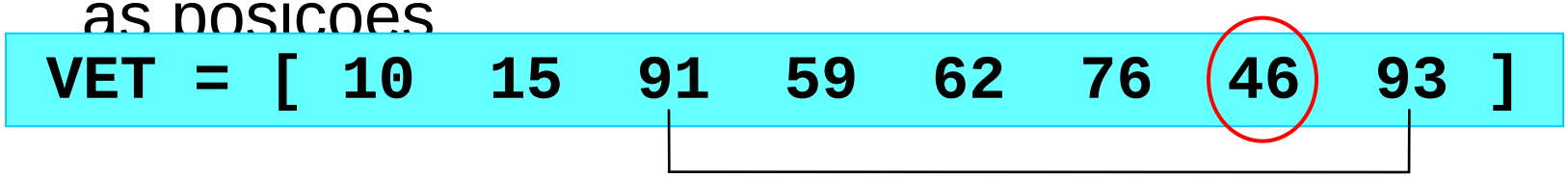
`VET[I] >= VET[J]` **se** `I > J` → **crescente**

`VET[I] <= VET[J]` **se** `I > J` → **decrescente**

# Método da Seleção

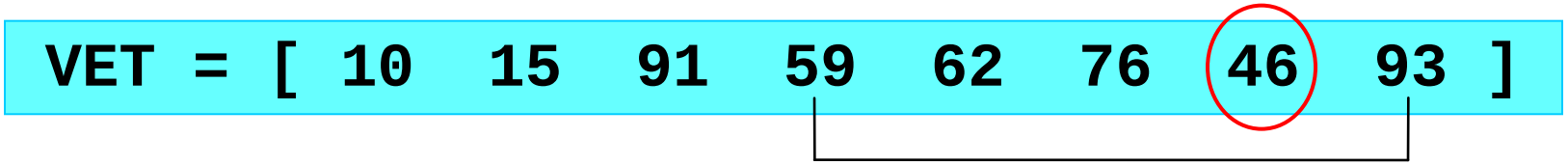
```
VET = [ 46   15   91   59   62   76   10   93 ]
```

- Ordem crescente
  - **1º passo:-** para saber quem fica na posição **1**, determina-se o menor elemento do vetor da posição **2** até a **8**
    - Compara-se com o elemento VET[**1**]
    - se for menor troca-se as posições

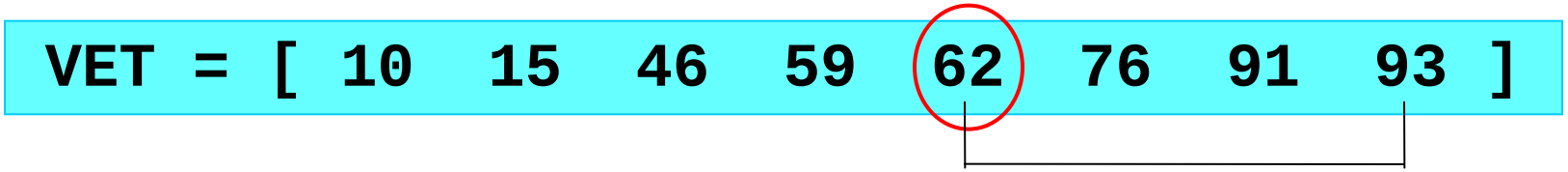


# Método da Seleção (continuação)

– **2º passo:-** procura-se o menor elemento da posição **3** até a **8** e repetimos o processo

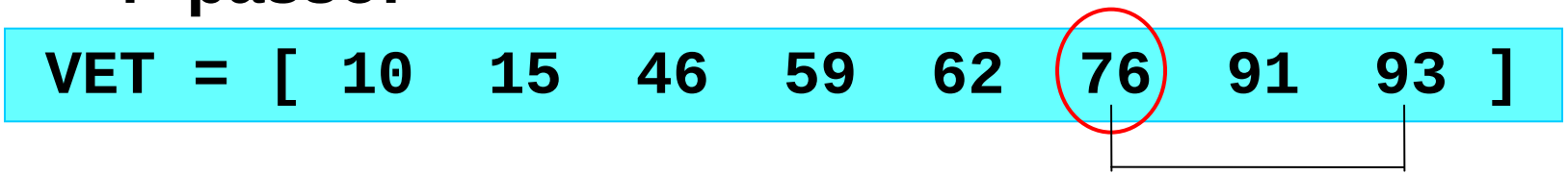



– **3º passo:**

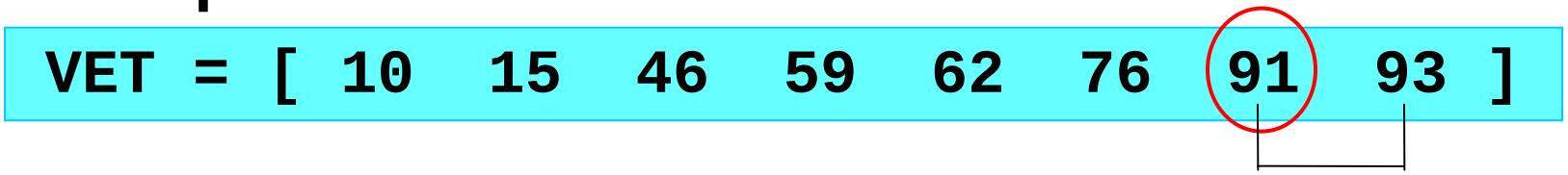


# Método da Seleção (continuação)

– 4º passo:




– 5º passo:




– 6º passo:


– 7º passo:


Ok!

# Algoritmo1 - Ordenação por Seleção

```
program ordena1;
#includes
.....
  int Tab[10] {tabela de pesquisa}
(**) int Ind1, Ind2; {marcadores}
  int Aux; {posição para a troca}
  int IndMin; {posição do menor
  elemento}
```

# Algoritmo1 - Ordenação por Seleção (continuação)

```
Int Main(){
(**)
        printf("Entre com os valores da  tabela");
   for (i=0;i<N;i++)
      scanf("%d", &Tab[I]);
   for(Ind1=0;Ind1< N – 1;Ind1++){
                IndMin= Ind1;
                for(Ind2= IndMin + 1;I<N;I++){
                      if Tab[Ind2] < Tab[IndMin]
                          IndMin= Ind2;
                if Tab[IndMin] < Tab[Ind1]{
                        Aux = Tab[IndMin];
                        Tab[IndMin]= Tab[Ind1];
                  Tab[Ind1]= Aux;
                }
        }
   printf("O vetor ordenado e\n");
   for (Ind1=0;Ind1<N;Ind1++){
                printf("%d",Tab[Ind1]);
}
```

# Esforço Computacional

1º passo: 7 comparações
2º passo: 6 comparações
3º passo: 5 comparações
4º passo: 4 comparações
5º passo: 3 comparações
6º passo: 2 comparações
7º passo: 1 comparação

**N elementos : soma do N-1 primeiros inteiros**

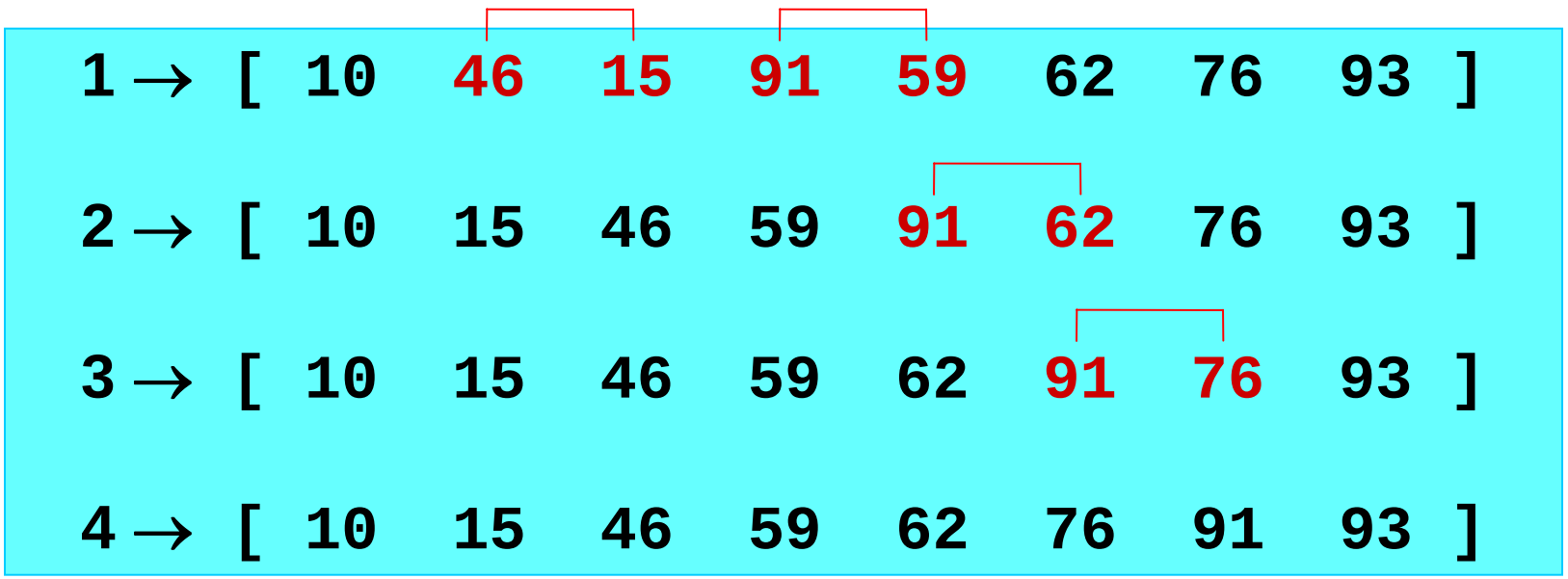
$$\therefore \quad N_C = (N-1) \cdot N / 2$$

$N_T = (N-1)$ {*no máximo*}

# Método da Bolha

- É mais popular que o anterior, embora seu desempenho seja inferior

```
Tab = [ 10  46  15  91  59  62  76  93 ]
```



# Algoritmo2 - Método da Bolha

```
Int Main(){
(**)
   boolean Troquei;  {indicador de parada}
   printf("Entre com os valores da  tabela");
   for (i=0;i<N;i++)
     scanf("%d", &Tab[I]);
   Ind2= N;
   do{
              Troquei= false;
              for (Ind1=0;Ind1<Ind2 – 1;Ind1++){
                  if Tab[Ind1] > Tab[Ind1+1]{
                          Aux= Tab[Ind1];
                          Tab[Ind1]= Tab[Ind1+1];
                          Tab[Ind1+1]= Aux;
                          Troquei:= true;
     }}}
   while(Troquei);
   writeln('O vetor ordenado e');
   printf("O vetor ordenado e\n");
   for (Ind1=0;Ind1<N;Ind1++){
              printf("%d",Tab[Ind1]);
   printf("O vetor ordenado e\n");
   for (Ind1=0;Ind1<N;Ind1++){
              printf("%d",Tab[Ind1]);
}
```

# Esforço Computacional

- Método da Bolha

$$N_C = (N-1)\,N / 2$$

$$N_T = 3\,(N-1)\,N / 4$$

$N_C$ = número de comparações

$N_T$ = número de trocas

# Método Quicksort

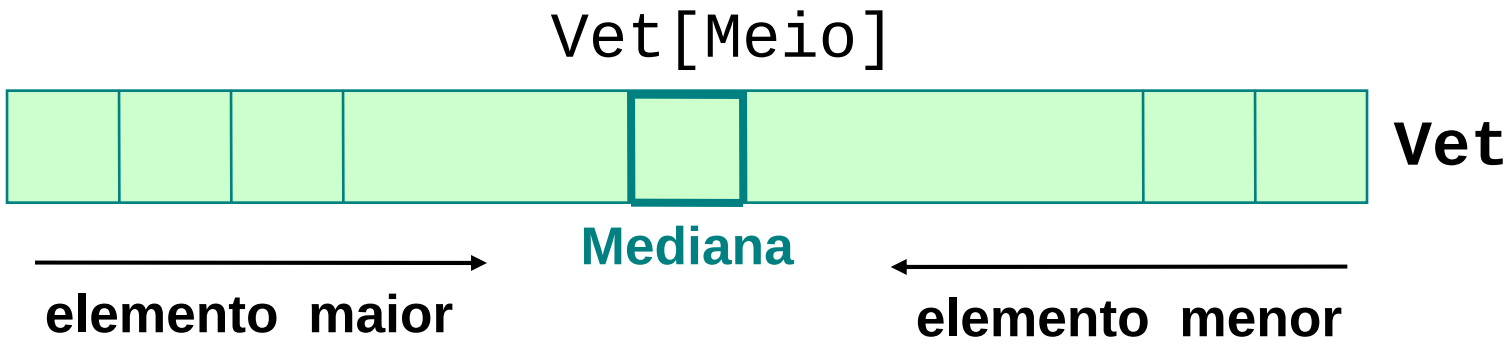
- Vamos, agora, examinar um processo de ordenação chamado ORDENAÇÃO POR PARTIÇÃO, que é vantajoso em relação aos processos anteriores
- O esforço computacional cresce com

  N log N

  e não mais quadraticamente

# Idéia Básica (Quicksort)

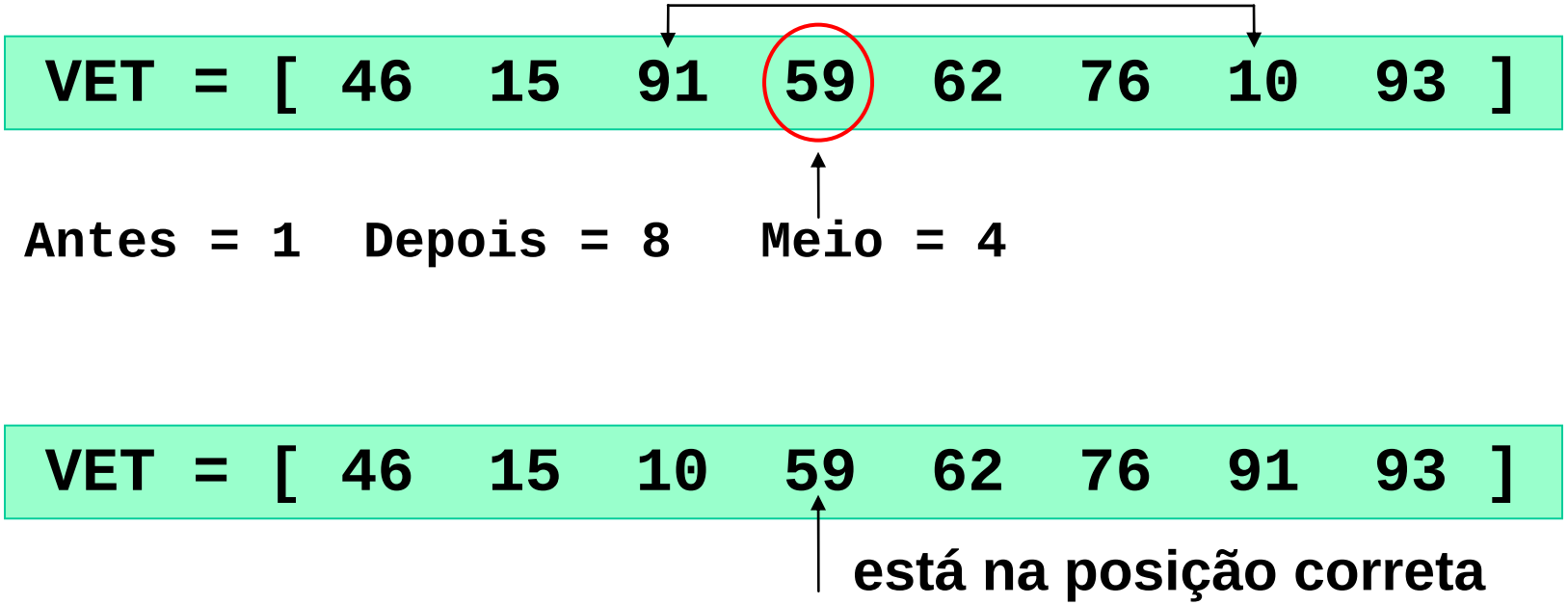
- Trocas em distâncias maiores do que **1**
- Três marcadores:
  - *Antes*, *Depois* e *Meio*



# Idéia Básica (Quicksort)

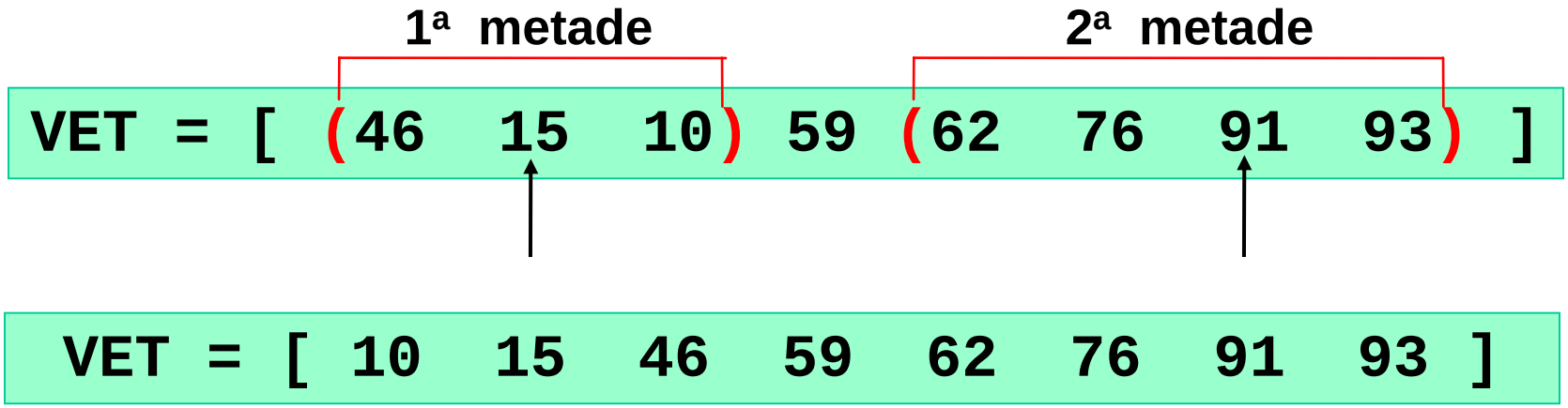
- Procura-se de um lado um elemento menor que a Mediana e por outro lado um elemento maior que a Mediana e troca-se suas posições
- Se existir apenas um elemento nas cond. anteriores, entao, troca-se esse elemento com a Mediana.
- Repete-se o processo até que *Antes* seja maior que *Depois*

# Exemplo


VET = [ 46 15 91 59 62 76 10 93 ]
Antes = 1 Depois = 8 Meio = 4
VET = [ 46 15 10 59 62 76 91 93 ]
está na posição correta

# Exemplo (continuação)

- Repete-se o processo, isto é, a partição, para cada uma das METADES




Ordenado!

# Subprogramas Recursivos (1 de 5)

- O Escopo de um subprograma é delimitado desde de sua definição até o fim do bloco que está definido.
- Sendo assim, um subprograma pode ser chamado por um outro subprograma ou até mesmo por si próprio.
- Quando um subprograma contém uma chamada a si próprio, ele é dito um **subprograma recursivo**.

# Subprogramas Recursivos (2 de 5)

- Exemplos de definições recursivas:

(i)
$$fat(n) = n * fat(n-1) \quad , n >= 1$$
$$fat(n) = 1 \quad , n = 0$$

(ii)
$$fib(n) = fib(n-1)+fib(n-2) \quad , n >= 2$$
$$fib(n) = 1 \quad , n = 1$$
$$fib(n) = 0 \quad , n = 0$$

# Subprogramas Recursivos (3 de 5)

- Todas as definições recursivas têm em comum:
  - um índice para cada definição;
  - pelo menos, uma definição não-recursiva (condição de saída) que, ao ser alcançada garante a interrupção da recursão;

- Cada chamada faz a função ser executada novamente, com um argumento diferente para cada nova execução.

# Subprogramas Recursivos (4 de 5)

fib(n) = fib(n-1)+fib(n-2) , n >= 2
fib(n) = 1 , n = 1
fib(n) = 0 , n = 0

```
Int function fib(int n){
   if n = 0
      fib := 0;
   else if n = 1  fib := 1;
        else fib = fib(n-1)+fib(n-2)
}
```

# Subprogramas Recursivos (5 de 5)

- Programas recursivos consomem **tempo de execução** e **espaço em memória** (pilha de ativação para cada procedimento recursivo) e **podem ser ineficientes** para alguns casos.
- Por exemplo, o cálculo do número de Fibonacci iterativo gasta menos recursos computacionais do que o cálculo recursivo.
- **Faça o programa fib(n) iterativo** para comprovar a eficiência perante o recursivo.

# Algoritmo Quicksort

- O algoritmo para executar esta tarefa é recursivo

Início do **Programa Quicksort**

```
Program Quicksort;

variaveis
        int Tab[20];   {tabela a ser ordenada}
        int Ind;  {marcador}
```

## Continuação do Programa Quicksort

```
void Quick (int Inicio, int Fim);
Int Antes, Depois;
Int    Mediana, Aux;

       Antes= Inicio;
       Depois= Fim;
       Mediana= Tab[(Inicio + Fim) / 2];
       do{
              while Tab[Antes] < Mediana  {  Antes= Antes +
1};
              while Tab[Depois] > Mediana  { Depois= Depois –
1};
              if Antes <= Depois{
                     Aux := Tab[Antes];
                     Tab[Antes] := Tab[Depois];
                     Tab[Depois] := Aux;
                     Antes := Antes + 1;
                     Depois := Depois  - 1;
                   }
       while( Antes > Depois);
       If Inicio < Depois    Quick(Inicio, Depois);
       if Antes < Fim    Quick(Antes, Fim);
} /*Quick
```

## Final do **Programa Quicksort**

```
Int main(){
        printf("Entre com os valores da  tabela");
        for (i=0;i<N;i++)
     scanf("%d", &Tab[I]);
        Quick(1,N);
  printf("O vetor ordenado e\n");
        for (Ind1=0;Ind1<N;Ind1++){
                        printf("%d",Tab[Ind1]);

}  {program quicksort}
```

# Esforço Computacional

- Método do Quicksort

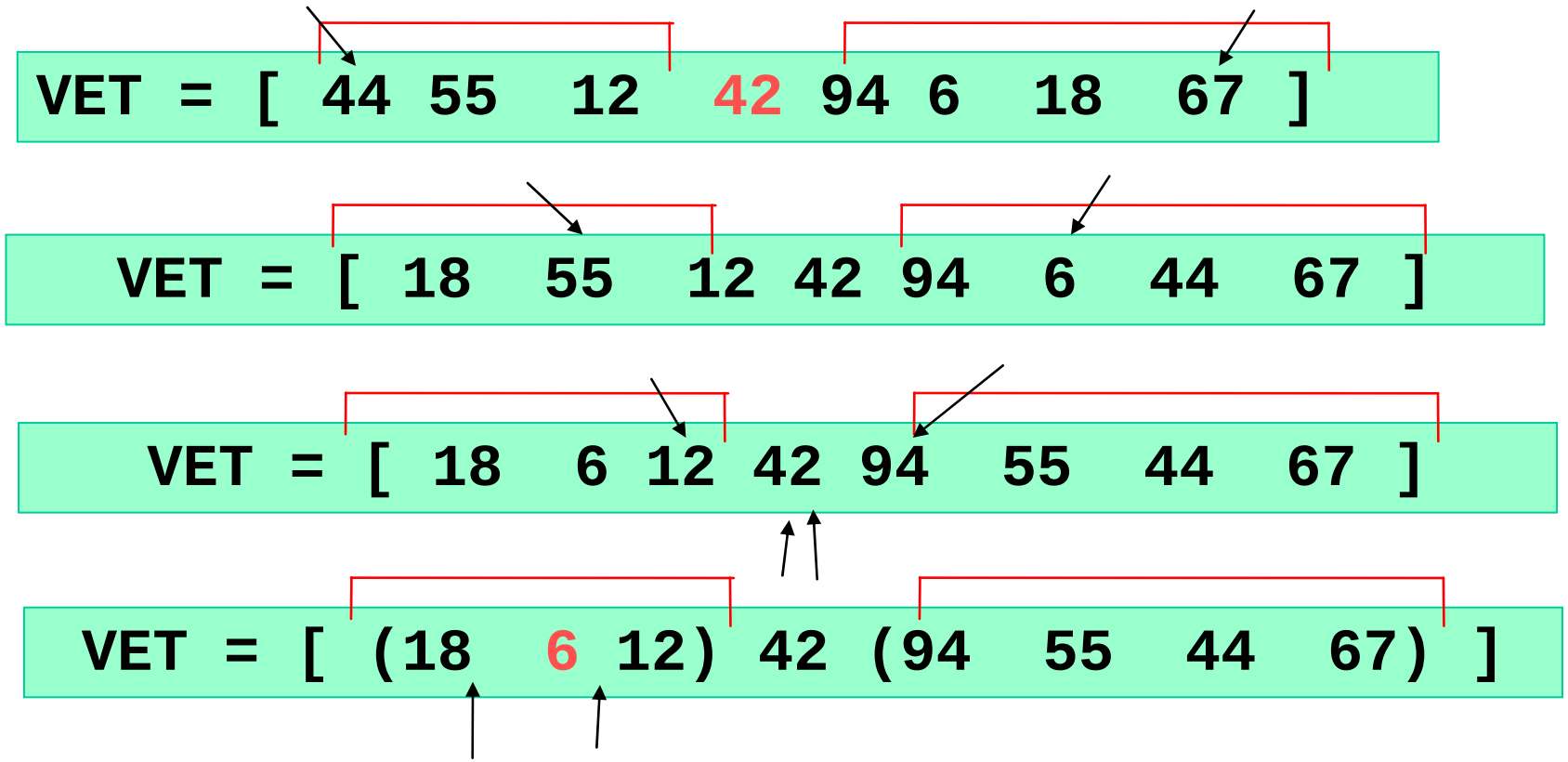
$$N_C = N (\log N)$$

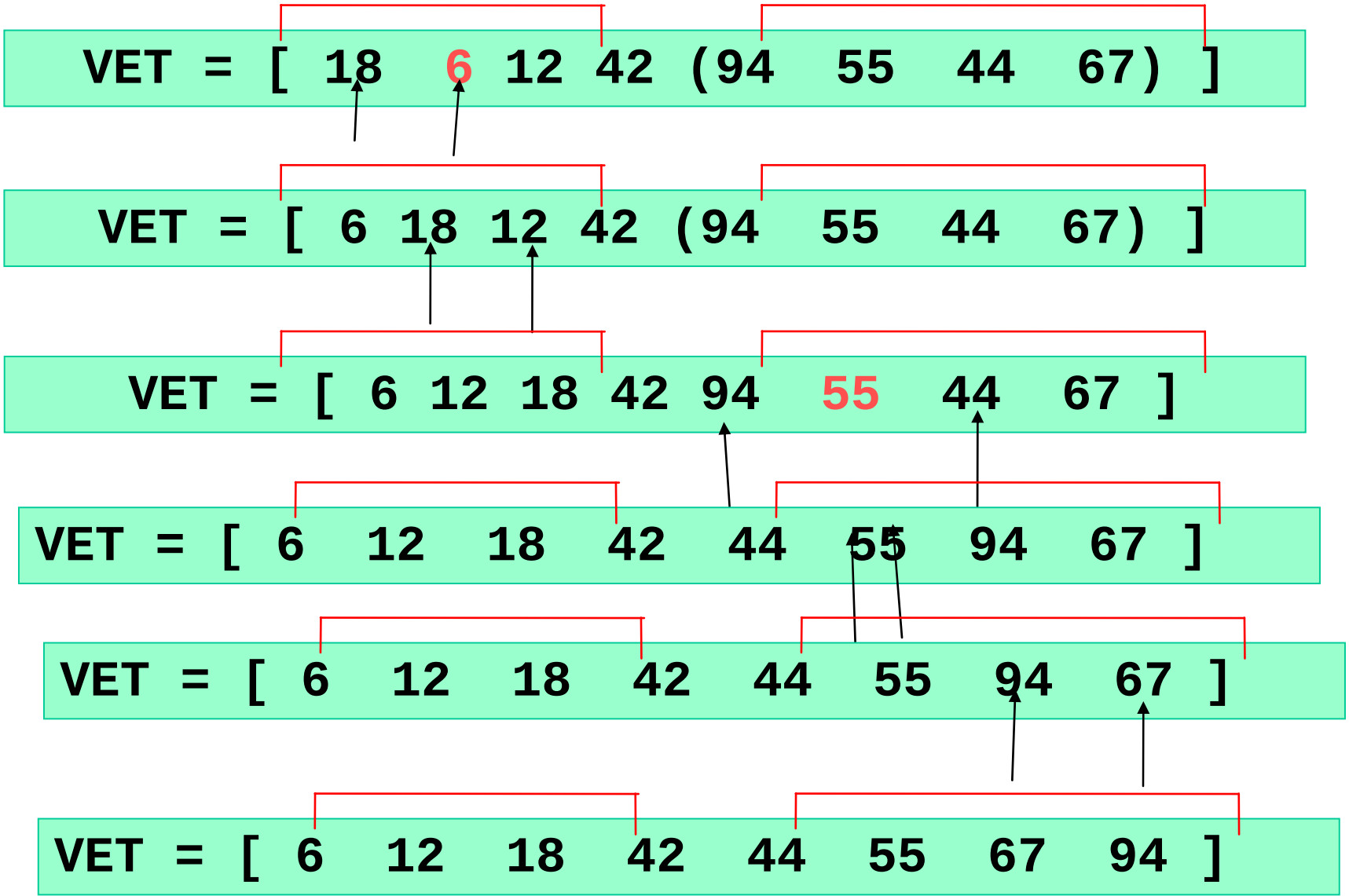
$$N_T = N (\log N)$$

**1º passo: N comparações**

**a cada passo: divide o vetor ao meio**

EXEMPLO


VET = [ 44 55 12 42 94 6 18 67 ]
VET = [ 18 55 12 42 94 6 44 67 ]
VET = [ 18 6 12 42 94 55 44 67 ]
VET = [ (18 6 12) 42 (94 55 44 67) ]


VET = [ 18 6 12 42 (94 55 44 67) ]
VET = [ 6 18 12 42 (94 55 44 67) ]
VET = [ 6 12 18 42 94 55 44 67 ]
VET = [ 6 12 18 42 44 55 94 67 ]
VET = [ 6 12 18 42 44 55 94 67 ]
VET = [ 6 12 18 42 44 55 67 94 ]

# Recursão X Iteração

- Todo processo **recursivo** pode ser **transformado** em um processo **iterativo**, bastando **simular** a pilha de recursão, caso não se conheça outro tipo de definição não-recursiva.
- A versão **não-recursiva** é, em geral, mais **eficiente** do que a recursiva.
- A escolha por uma função recursiva é feita quando **tempo/espaço** não são problemáticos, ou se a versão recursiva for mais **simples**.

Material preparado por
Profª Roseli A. Francelin Romero e
Valéria Feltrim (PAE) – 1999
Revisado em 2008 por Roseli Romero

# Fim