Aula 6

Professores:

Dante Corbucci Filho
Leandro A. F. Fernandes

Conteúdo:

- Algoritmos de Busca
  - Busca Simples
  - Busca com Sentinela
  - Busca Binária
- Busca do Menor e Maior Elementos
- Noções de Complexidade de Algoritmos

Algoritmos de Busca

- O problema de busca é caracterizado pela procura de um determinado elemento em um grupo de elementos do mesmo tipo.
  - Exemplos:
    - Encontrar o registro de um cliente entre todos os registros dos clientes de um banco, para que seja possível informar seu saldo.
    - Encontrar o registro de um aluno dentre todos os registros dos alunos de uma universidade, para que seja possível gerar o seu histórico escolar.

Algoritmos de Busca

- O problema de busca é caracterizado pela procura de um determinado elemento em um grupo de elementos do mesmo tipo.
  - Exemplos:
    - Encontrar o registro de um cliente entre todos os registros dos clientes de um banco, para que seja possível informar seu saldo.
    - Encontrar o registro de um aluno dentre todos os registros dos alunos de uma universidade, para que seja possível gerar o seu histórico escolar.

- Algoritmos de busca visam resolver o problema da busca e são, portanto, bastante utilizados em computação.

Algoritmos de Busca

- O problema de busca é caracterizado pela procura de um determinado elemento em um grupo de elementos do mesmo tipo.
  - Exemplos:
    - Encontrar o registro de um cliente entre todos os registros dos clientes de um banco, para que seja possível informar seu saldo.
    - Encontrar o registro de um aluno dentre todos os registros dos alunos de uma universidade, para que seja possível gerar o seu histórico escolar.

- Algoritmos de busca visam resolver o problema da busca e são, portanto, bastante utilizados em computação.

- Necessita-se de algoritmos eficientes de busca.
  - De forma simplificada, um algoritmo eficiente é aquele que realiza sua tarefa em tempo computacional reduzido.

Algoritmos de Busca

- O tempo de execução de um algoritmo de busca depende do tamanho do conjunto de elementos a serem consultados.
  - É mais rápido procurar um elemento em um grupo de 100 do que em um grupo de 100.000.

Algoritmos de Busca

- O tempo de execução de um algoritmo de busca depende do tamanho do conjunto de elementos a serem consultados.
  - É mais rápido procurar um elemento em um grupo de 100 do que em um grupo de 100.000.
- Além disso, a eficiência do processo de busca depende do algoritmo adotado.
  - Exploraremos o conceito de complexidade de algoritmos no sentido de tentar avaliar a eficiência dos algoritmos estudados.

Algoritmos de Busca

- Sem perda de generalidade, o problema será atacado considerando-se que:

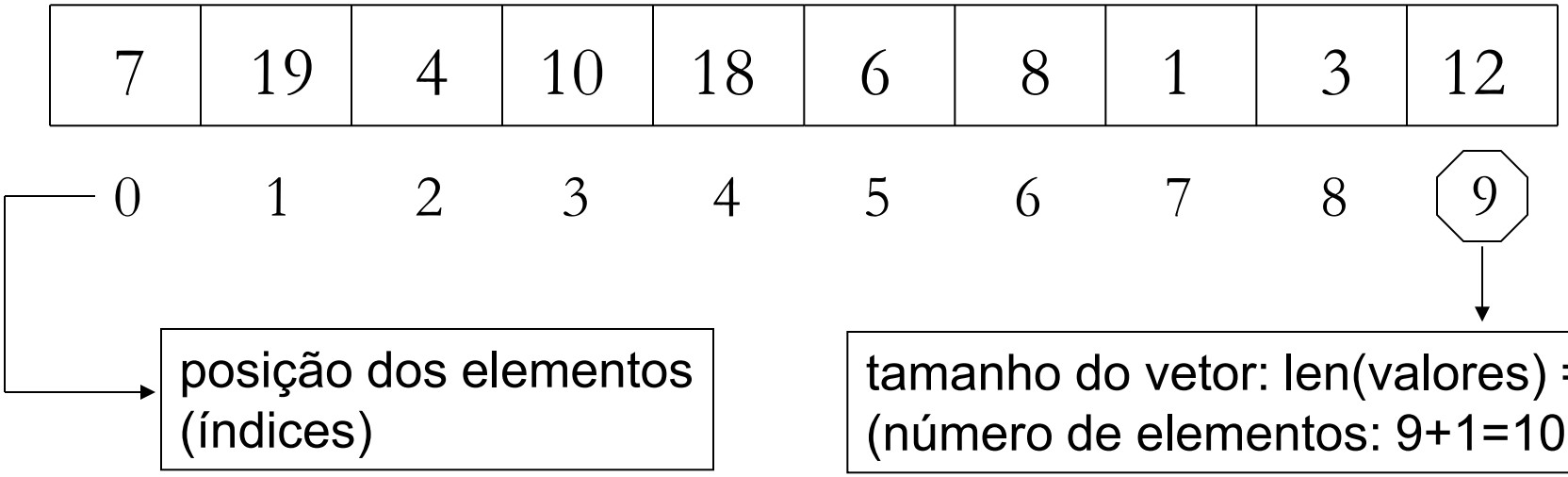
1. Os elementos a serem percorridos são numéricos, distintos e estão armazenados em um vetor;

2. O número de elementos define o tamanho do vetor;

3. O elemento procurado pode não estar no vetor (no conjunto de elementos).

Algoritmos de Busca

numeros: contém o grupo de elementos



Exemplo 1) Elemento procurado: 18 Resposta: posição 4

Exemplo 2) Elemento procurado: 5 Resposta: não encontrado

Exemplo

```
# Subprogramas

...

# Programa Principal de Busca
numeros = [0]*10      # Cria o vetor numeros zerado, com tamanho n = 10
preencher(numeros)

# dado: o elemento a ser procurado
dado = int(input("Escolha valor a ser procurado: "))

# onde: o local no vetor onde dado foi encontrado ou -1 se não encontrado
onde = buscaElemento(numeros, dado)

escreverResposta(dado, onde)
```

Exemplo (continuação)

```
# Subprogramas
def preencher(valores):
    for ind in range(len(valores)):
        valores[ind] = int(input("Elemento["+str(ind)+"]="))
    return None
def buscaElemento(valores, procurado):
    ...
def escreverResposta(valor, pos):
    if pos<0:
        print(valor,"não foi encontrado")
    else:
        print(valor, "foi encontrado na posição", pos)
    return
# Programa Principal de Busca
numeros = [0]*10
preencher(numeros)
dado = int(input("Escolha valor a ser procurado: "))
onde = buscaElemento(numeros, dado)
escreverResposta(dado, onde)
```

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: -1

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: -1
indice: 0

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: -1
indice: 1

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: -1
indice: 2

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: -1
indice: 3

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: 4
indice: 4

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: 4
indice: 5

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: 4
indice: 6

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: 4
indice: 7

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: 4
indice: 8

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: 4
indice: 9

Busca Simples (utilizando *for*)

Todas as posições do vetor são comparadas com o valor procurado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 18
local: 4
indice: 9

Resultado

```
# Subprogramas
def preencher(valores):
    ...

def buscaElemento(valores, procurado):
    local = -1
    for indice in range(len(valores)):
        if valores[indice]==procurado:
            local = indice
    return local

def escreverResposta(valor, pos):
    ...

# Programa Principal de Busca
numeros = [0]*10
preencher(numeros)
dado = int(input("Escolha valor a ser procurado: "))
onde = buscaElemento(numeros, dado)
escreverResposta(dado, onde)
```

Busca Simples (utilizando *for*)

- Todas as posições do vetor são comparadas com o valor procurado, mesmo quando este é encontrado antes do fim do vetor.

Busca Simples (utilizando *for*)

- Todas as posições do vetor são comparadas com o valor procurado, mesmo quando este é encontrado antes do fim do vetor.

- Nitidamente, esta é uma busca ineficiente.

Busca Simples (utilizando *for*)

- Todas as posições do vetor são comparadas com o valor procurado, mesmo quando este é encontrado antes do fim do vetor.

- Nitidamente, esta é uma busca ineficiente.

- O algoritmo avalia necessariamente **n** elementos, onde **n** é o tamanho do vetor. Portanto, sua complexidade é da ordem de **n**, representado por O(**n**).

Busca Simples (utilizando *for* com saída rápida)

- Utilizando-se sub-programação, não se considera uma má prática de programação se realizar um **return** dentro de uma repetição.

- O subprograma *buscaElemento* pode ter sua eficiência melhorada quando encontra o elemento no vetor, não necessitando repetir todos os índices previstos no **for**:

```
def buscaElemento(valores, procurado):
    for indice in range(len(valores)):
        if valores[indice]==procurado:
            return indice                # retorna ao encontrar o local
    return -1                            # retorna -1 se terminar sem encontrar
```

Busca Simples (utilizando *while*)

Neste algoritmo, o processo de busca é interrompido quando o elemento procurado é encontrado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 4
local: -1
indice: 0

Busca Simples (utilizando *while*)

Neste algoritmo, o processo de busca é interrompido quando o elemento procurado é encontrado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 4
local: -1
indice: 1

Busca Simples (utilizando *while*)

Neste algoritmo, o processo de busca é interrompido quando o elemento procurado é encontrado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

procurado: 4
local: -1
indice: 2

Busca Simples (utilizando *while*)

Neste algoritmo, o processo de busca é interrompido quando o elemento procurado é encontrado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

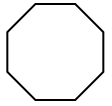
procurado: 4
local: 2
indice: 2

Busca Simples (utilizando *while*)

Neste algoritmo, o processo de busca é interrompido quando o elemento procurado é encontrado.

valores: contém o grupo de elementos

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9



procurado: 4
local: 2
indice: 10

Resultado

```
# Subprogramas
def preencher(valores):
    ...
def buscaElemento(valores, procurado):      # primeira solução
    local = -1
    indice = 0
    while indice<len(valores):
        if valores[indice]!=procurado:
            indice = indice + 1
        else:
            local = indice
            indice = len(valores)
    return local
def escreverResposta(valor, pos):
    ...
# Programa Principal de Busca
numeros = [0]*10
preencher(numeros)
dado = int(input("Escolha valor a ser procurado: "))
onde = buscaElemento(numeros, dado)
escreverResposta(dado, onde)
```

```
# Subprogramas
def preencher(valores):
    ...

def buscaElemento(valores, procurado):      # segunda solução
    indice = 0
    while indice<len(valores):
        if valores[indice]!=procurado:
            indice = indice + 1
        else:
            return indice    # retorna ao encontrar local
    return -1                # retorna -1 se terminar sem encontrar

def escreverResposta(valor, pos):
    ...
# Programa Principal de Busca
numeros = [0]*10
preencher(numeros)
dado = int(input("Escolha valor a ser procurado: "))
onde = buscaElemento(numeros, dado)
escreverResposta(dado, onde)
```

Busca Simples (utilizando *while*)

- Em média, este algoritmo executa **n**/2 vezes o corpo da repetição **while**.

Busca Simples (utilizando *while*)

- Em média, este algoritmo executa **n**/2 vezes o corpo da repetição **while**.

- Em algumas vezes, o dado será encontrado logo na primeira posição, em outras, na última posição.

Busca Simples (utilizando *while*)

- Em média, este algoritmo executa **n**/2 vezes o corpo da repetição **while**.

- Em algumas vezes, o dado será encontrado logo na primeira posição, em outras, na última posição.

- No pior caso, o algoritmo avalia **n** elementos. Portanto, sua complexidade também é da ordem de **n**, representado por O(**n**).

Busca com Sentinela (utilizando *while*)

- O algoritmo anterior poderia executar menos comparações, se não houvesse a necessidade de evitar ultrapassar o fim do vetor, caso o elemento procurado não se encontre no conjunto.

Busca com Sentinela (utilizando *while*)

- O algoritmo anterior poderia executar menos comparações, se não houvesse a necessidade de evitar ultrapassar o fim do vetor, caso o elemento procurado não se encontre no conjunto.

- Propõe-se então alocar uma posição a mais no vetor e inserir forçadamente o elemento procurado nesta posição.

Busca com Sentinela (utilizando *while*)

- O algoritmo anterior poderia executar menos comparações, se não houvesse a necessidade de evitar ultrapassar o fim do vetor, caso o elemento procurado não se encontre no conjunto.

- Propõe-se então alocar uma posição a mais no vetor e inserir forçadamente o elemento procurado nesta posição.

- Desta forma, necessariamente o elemento procurado será encontrado. Se for encontrado na posição **n**+1, significa que o elemento não pertence ao conjunto original.

Busca com Sentinela (utilizando *while*)

- O algoritmo anterior poderia executar menos comparações, se não houvesse a necessidade de evitar ultrapassar o fim do vetor, caso o elemento procurado não se encontre no conjunto.

- Propõe-se então alocar uma posição a mais no vetor e inserir forçadamente o elemento procurado nesta posição.

- Desta forma, necessariamente o elemento procurado será encontrado. Se for encontrado na posição **n**+1, significa que o elemento não pertence ao conjunto original.

- Apesar de executar menos comparações, o algoritmo avalia, no pior caso, **n**+1 elementos. Portanto, sua complexidade também é da ordem de **n**, representado por O(**n**).

```
# Subprogramas
def preencher(valores):
    ...
def buscaElemento(valores, procurado):
    indice = 0
    while valores[indice]!=procurado:
        indice = indice + 1
    if indice==len(valores)-1:
        local = -1              # local do sentinela, informa que não achou
    else:
        local = indice
    return local

def escreverResposta(valor, pos):
    ...
# Programa Principal de Busca
numeros = [0]*10
preencher(numeros)
dado = int(input("Escolha valor a ser procurado: "))
numeros.append(dado)     # coloca o procurado no final: o sentinela
onde = buscaElemento(numeros, dado)
escreverResposta(dado, onde)
```

Busca Binária

- Os algoritmos apresentados até o momento foram projetados sem levar em consideração se os elementos do vetor encontram-se ordenados ou não.

Busca Binária

- Os algoritmos apresentados até o momento foram projetados sem levar em consideração se os elementos do vetor encontram-se ordenados ou não.

- Caso os elementos se encontrem ordenados, algoritmos mais eficientes podem ser implementados.

Busca Binária

- Os algoritmos apresentados até o momento foram projetados sem levar em consideração se os elementos do vetor encontram-se ordenados ou não.

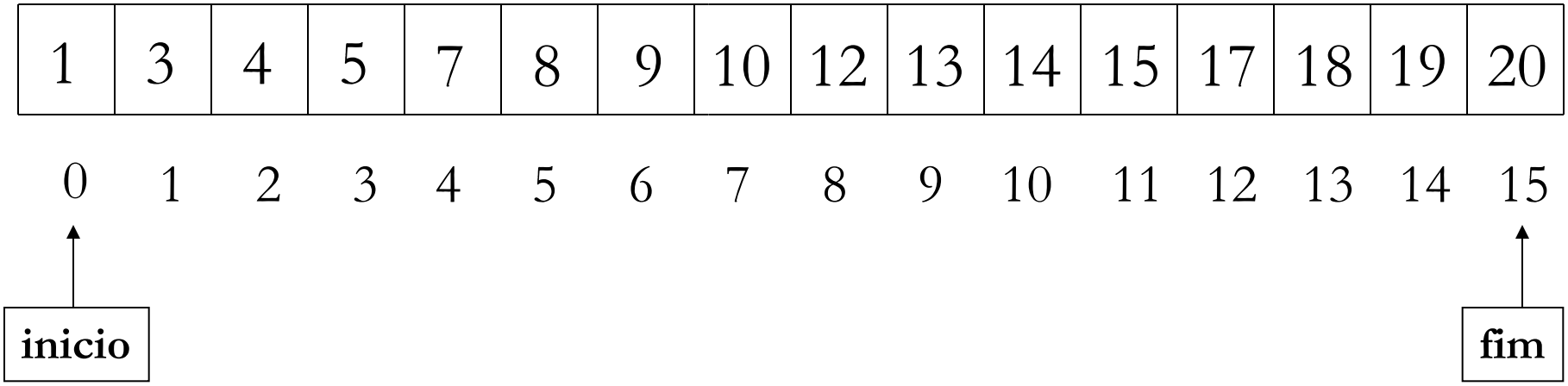
- Caso os elementos se encontrem ordenados, algoritmos mais eficientes podem ser implementados.

- No algoritmo chamado busca binária, cada passo divide o espaço de busca em dois grupos até encontrar o elemento sendo procurado.

Busca Binária

Os **n** elementos encontram-se ordenados no vetor.

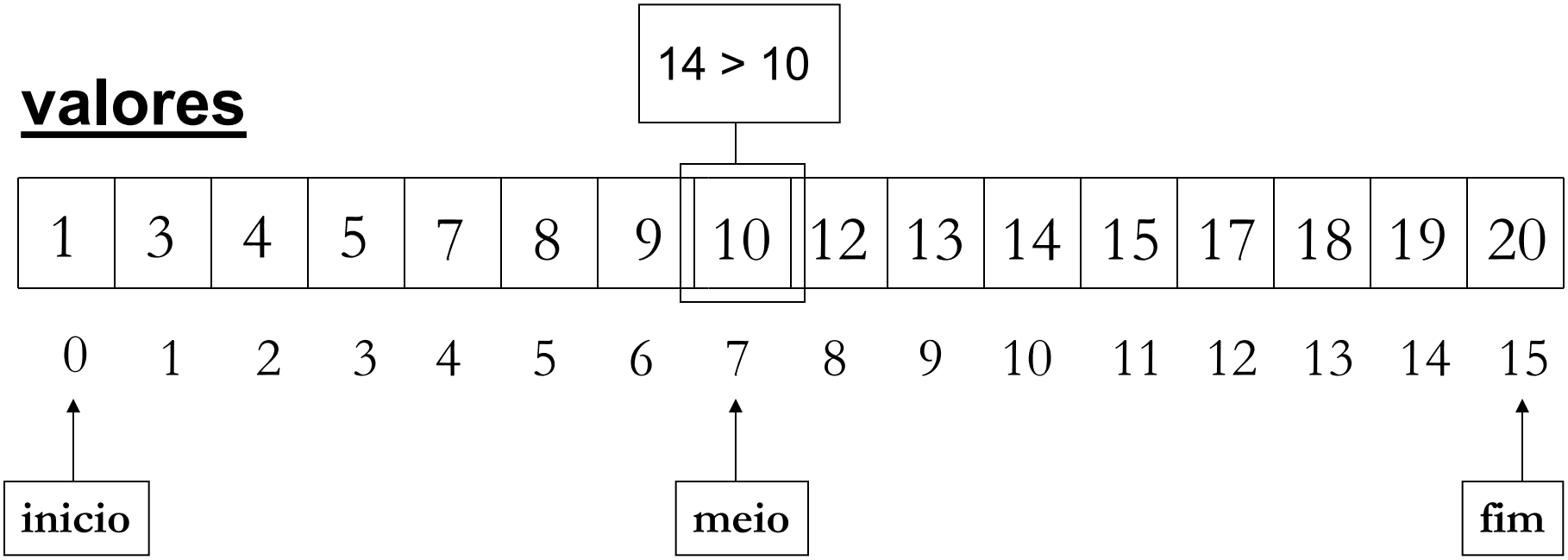
valores



procurado: 14

Busca Binária

procurado: 14

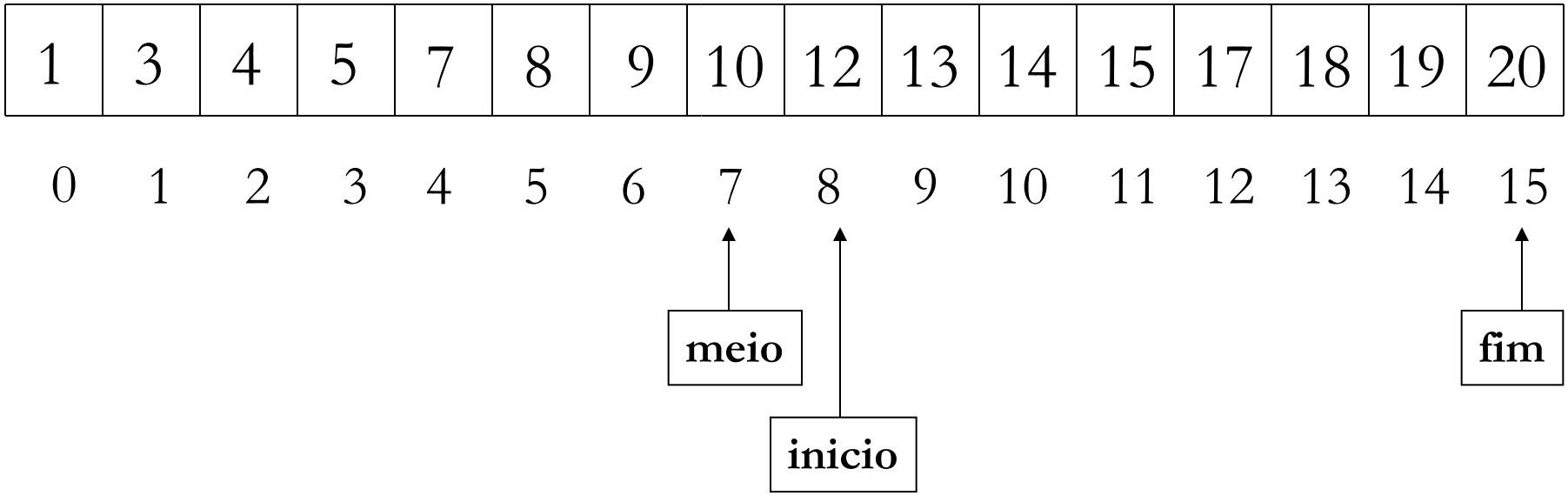


Número de elementos comparados: 1

Busca Binária

procurado: 14

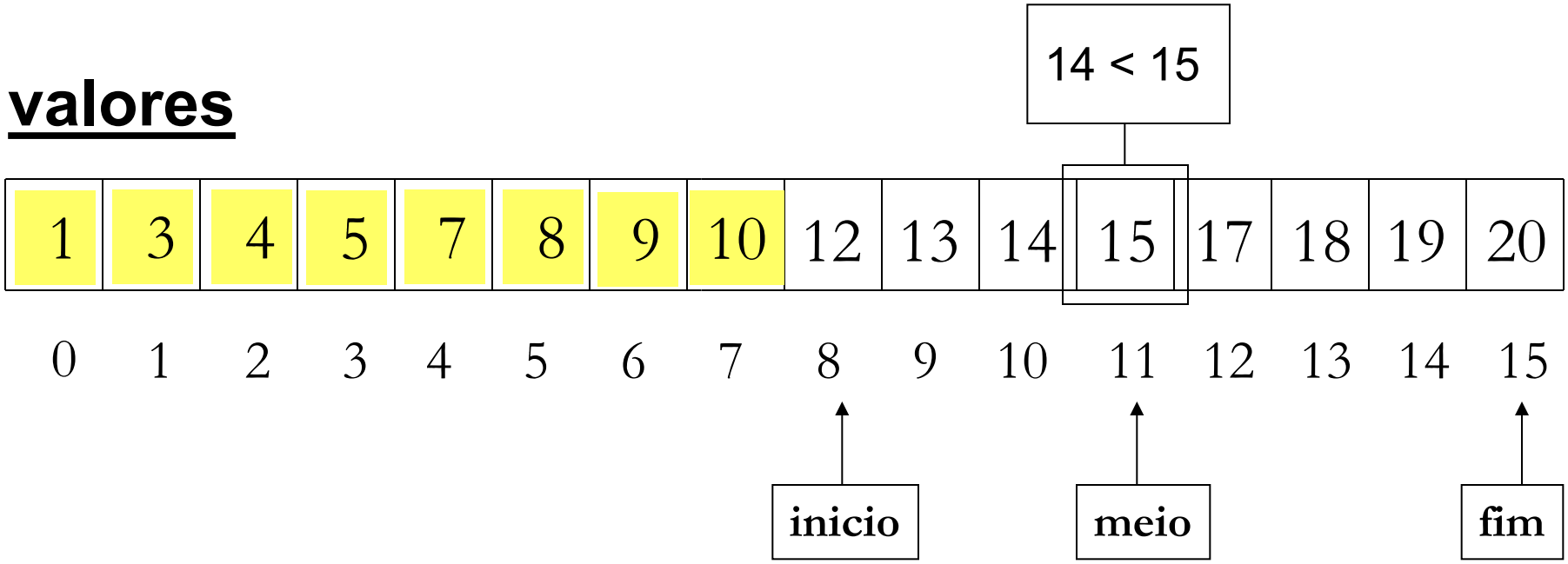
valores



Número de elementos comparados: 1

Busca Binária

procurado: 14

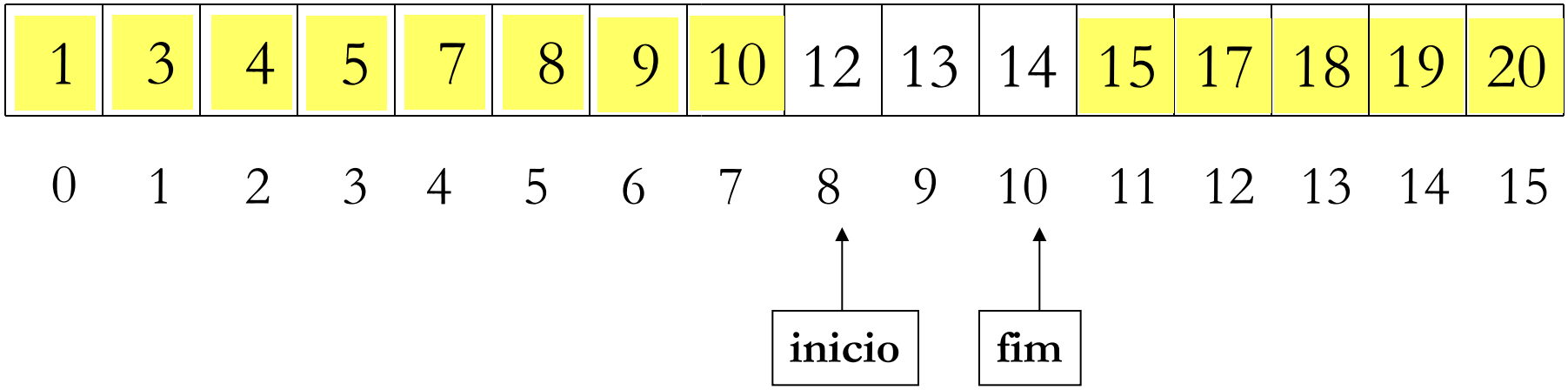


Número de elementos comparados: 2

Busca Binária

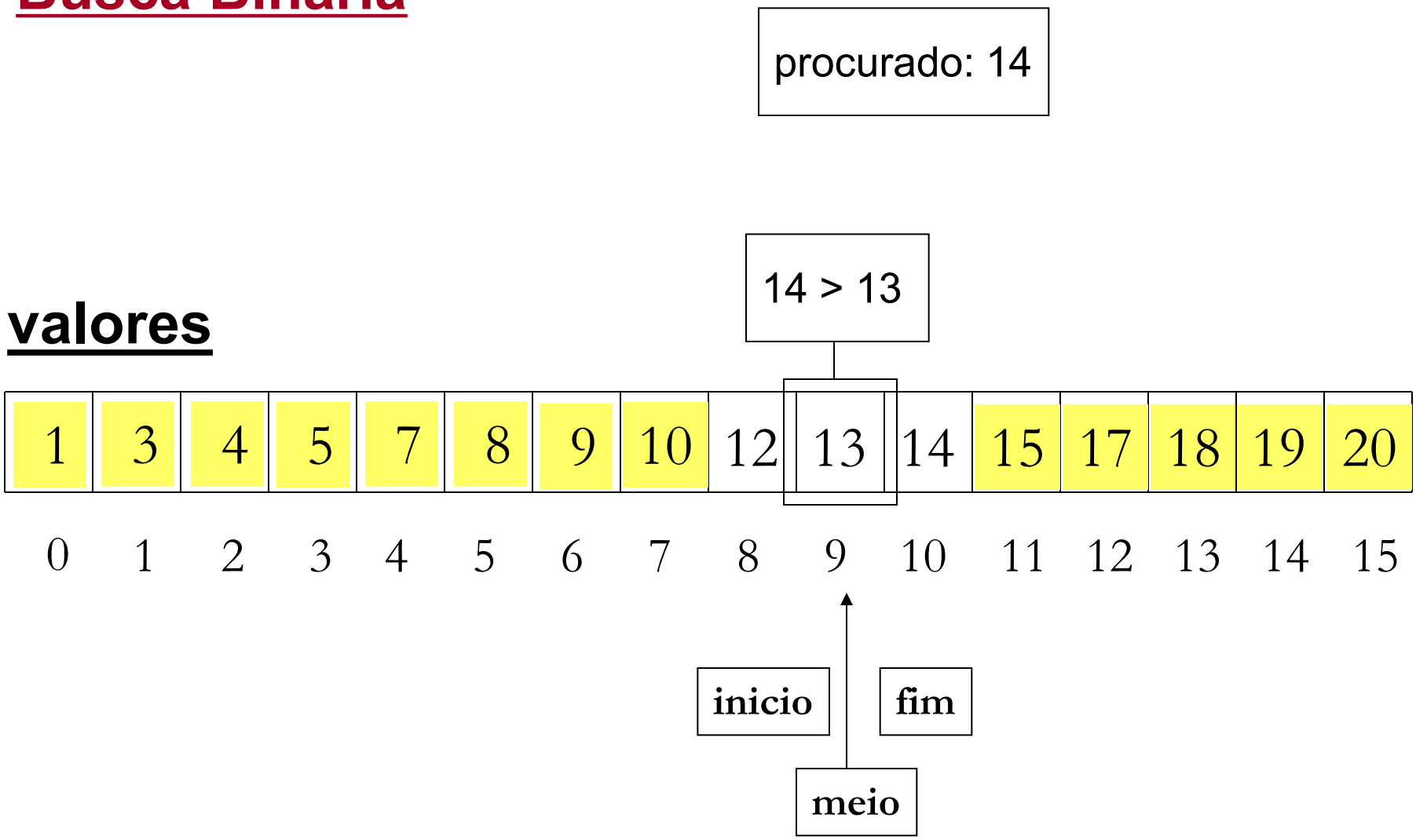
procurado: 14

valores



Número de elementos comparados: 2

Busca Binária

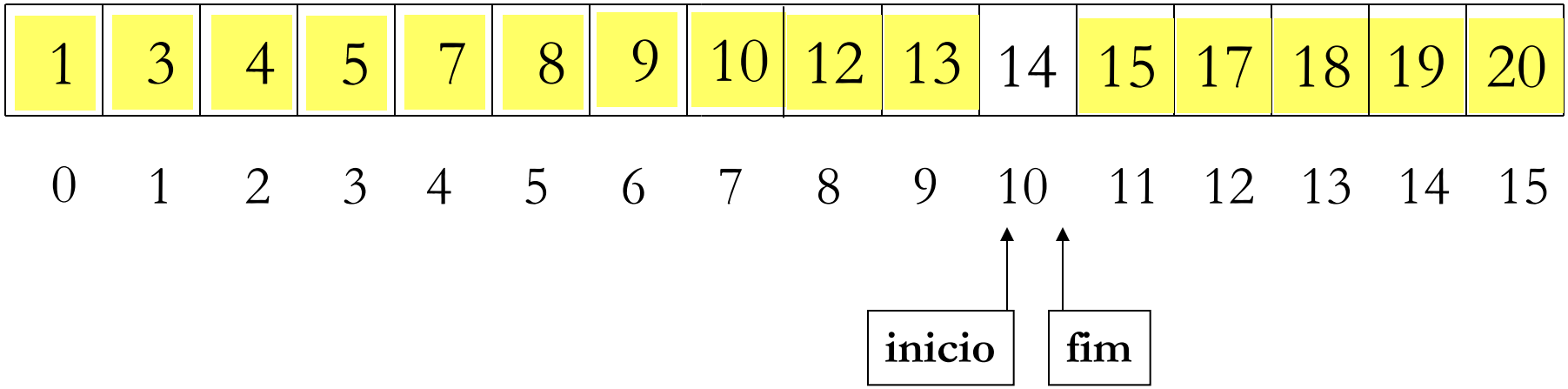


Número de elementos comparados: 3

Busca Binária

procurado: 14

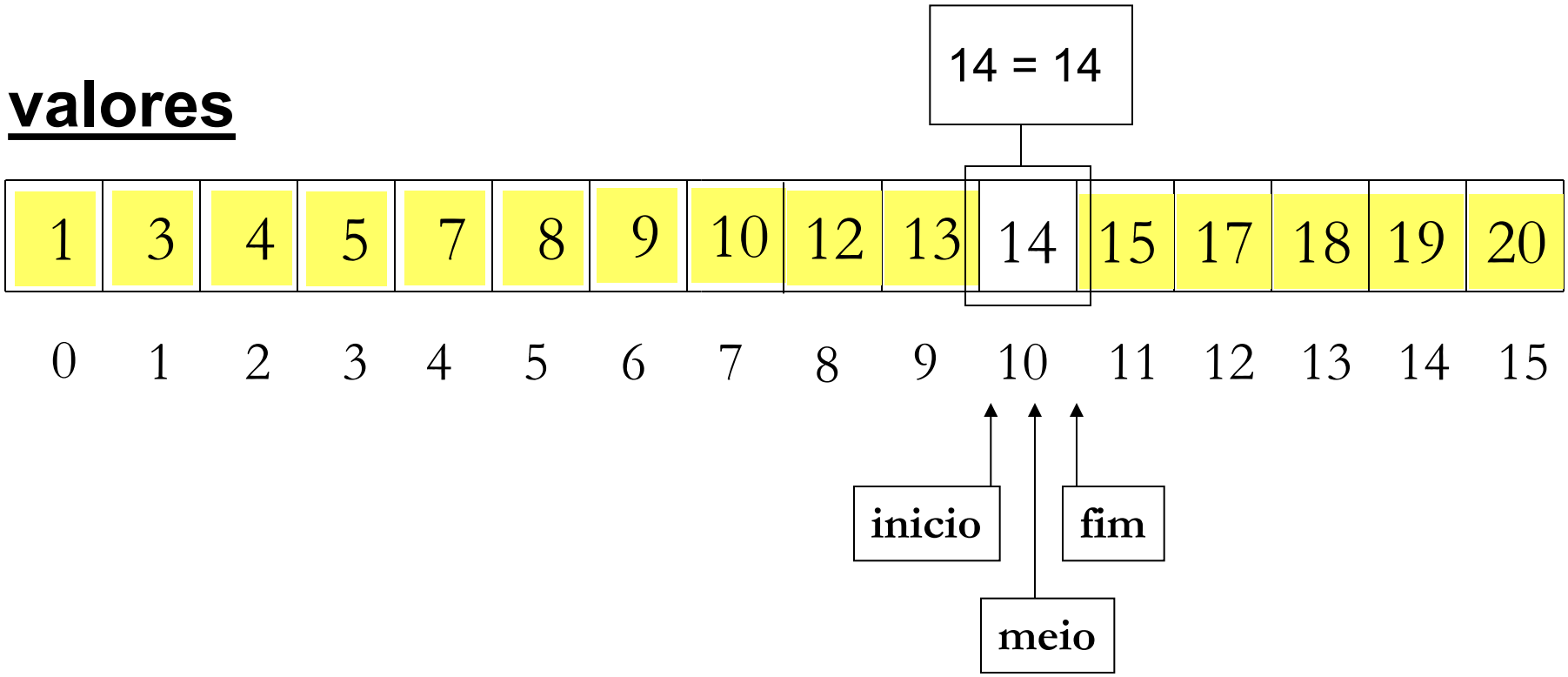
valores



Número de elementos comparados: 3

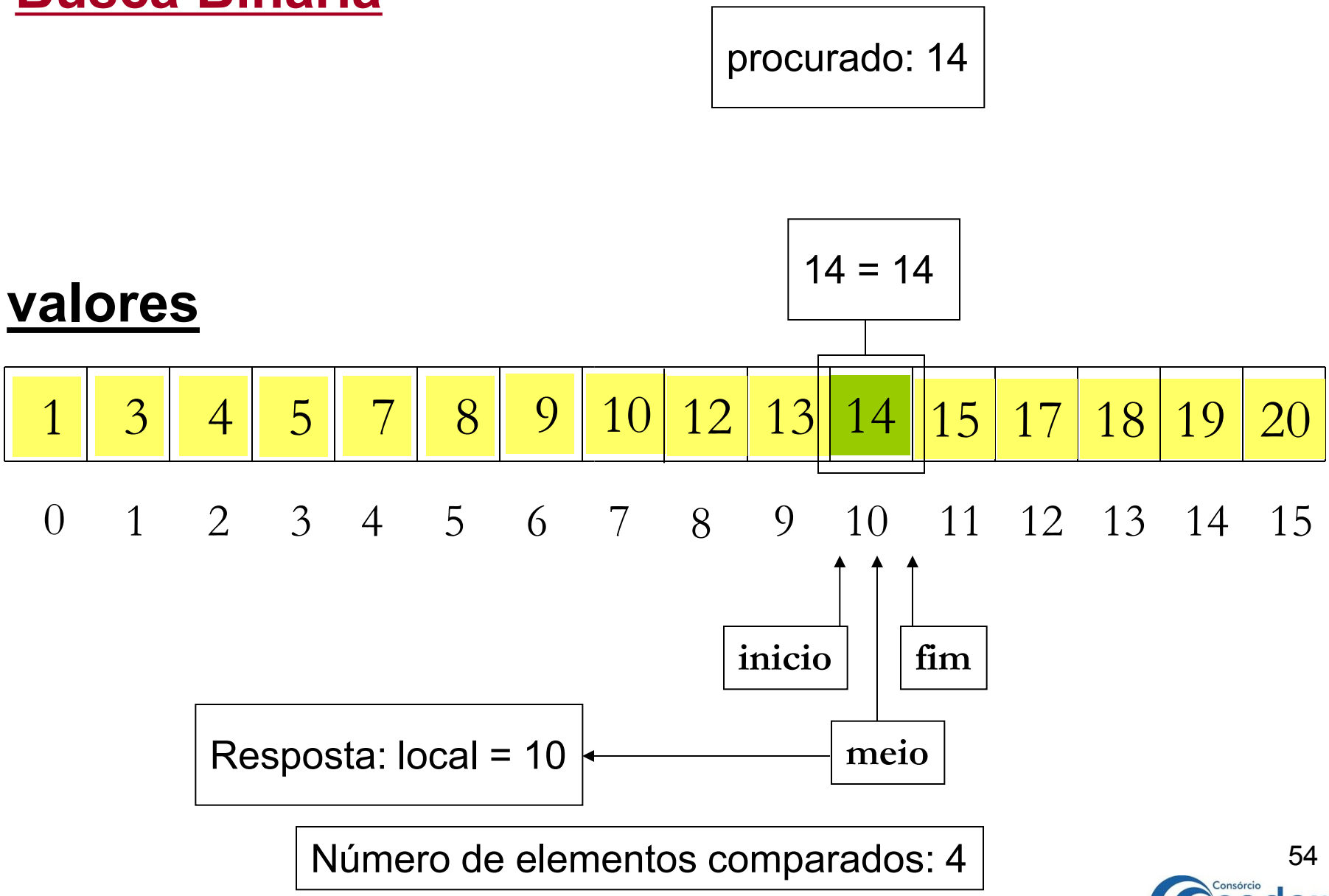
Busca Binária

procurado: 14



Número de elementos comparados: 4

Busca Binária



Busca Binária

```
def buscaElemento(valores, procurado):
    inicio = 0
    fim = len(valores)-1
    meio  = (inicio + fim) // 2
    while (inicio<fim) and (procurado!=valores[meio]):
        if procurado>valores[meio]:
            inicio = meio + 1
        else:
            fim = meio -1
         meio = (inicio + fim) // 2
    if procurado!=valores[meio]:
        local = -1
    else:
        local = meio
    return local
```

Busca Binária

- Na busca binária, a cada passo, divide-se o espaço de busca em dois até que o elemento procurado seja encontrado.

Busca Binária

- Na busca binária, a cada passo, divide-se o espaço de busca em dois até que o elemento procurado seja encontrado.
- Considerando-se um vetor de 16 posições, no máximo 4 divisões podem ser feitas. E portanto, no máximo 4 elementos do vetor serão comparados com o elemento procurado.

Busca Binária

- Na busca binária, a cada passo, divide-se o espaço de busca em dois até que o elemento procurado seja encontrado.
- Considerando-se um vetor de 16 posições, no máximo 4 divisões podem ser feitas. E portanto, no máximo 4 elementos do vetor serão comparados com o elemento procurado.
- Observe que se o vetor for de 32 posições (vetor duplicado), no máximo 5 elementos do vetor serão acessados (apenas um a mais). Ou ainda, se o vetor tiver 32.000 posições, apenas 15 avaliações serão necessárias.

Busca Binária

- Na busca binária, a cada passo, divide-se o espaço de busca em dois até que o elemento procurado seja encontrado.
- Considerando-se um vetor de 16 posições, no máximo 4 divisões podem ser feitas. E portanto, no máximo 4 elementos do vetor serão comparados com o elemento procurado.
- Observe que se o vetor for de 32 posições (vetor duplicado), no máximo 5 elementos do vetor serão acessados (apenas um a mais). Ou ainda, se o vetor tiver 32.000 posições, apenas 15 avaliações serão necessárias.
- Na prática, se o vetor tiver **n** posições, a busca binária avaliará, no pior caso, $\log_2(\mathbf{n})$ elementos. Portanto, sua complexidade é da ordem de log(**n**), representado por O(log(**n**)).

Busca do Menor e Maior Elementos

- Este problema é caracterizado pela procura do menor e do maior elementos em um grupo de elementos do mesmo tipo.

Busca do Menor e Maior Elementos

- Este problema é caracterizado pela procura do menor e do maior elementos em um grupo de elementos do mesmo tipo.

- Sem perda de generalidade, o problema será atacado considerando-se que:

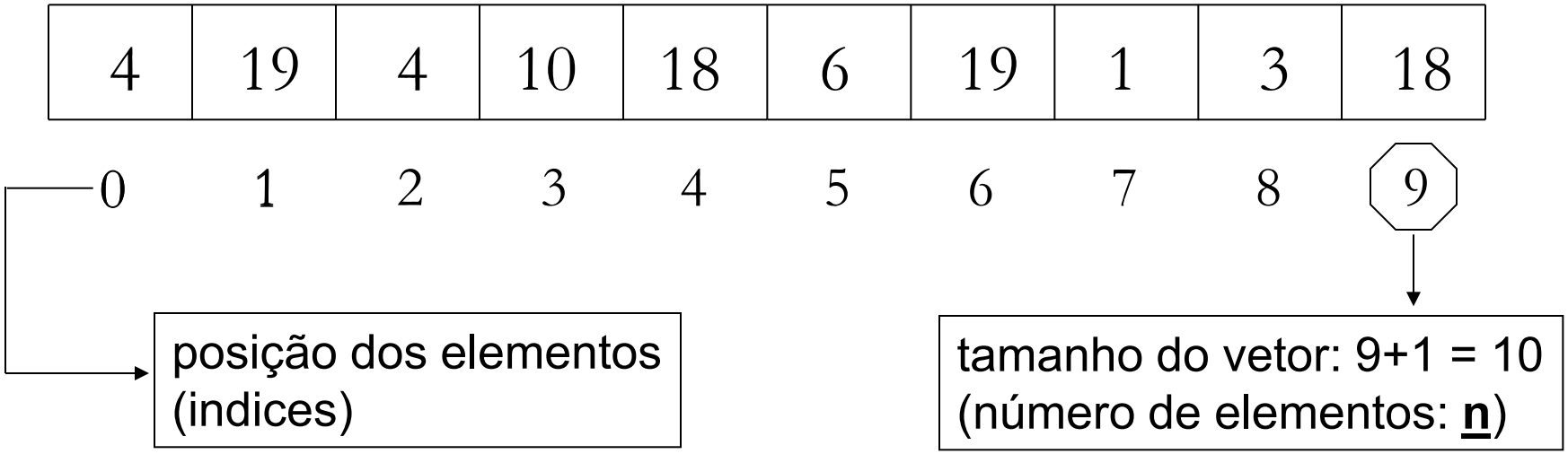
  – Os elementos a serem percorridos são numéricos e estão armazenados em um vetor;

  – Estes elementos podem não ser distintos;

  – O número de elementos define o tamanho do vetor.

Busca do Menor e Maior Elementos

valores: contém o grupo de elementos



Resposta:	Menor Elemento: 1
	Maior Elemento: 19

Busca do Menor e Maior Elementos

```
# Subprogramas
def preencher(valores):
    for ind in range(len(valores)):
        valores[ind] = int(input("Elemento["+str(ind)+"]="))
    return
def buscarMenorMaiorElementos(valores):
    ...

def escrever(infos):
    print("O menor elemento =", infos[0], "e o maior elemento =", infos[1])
    return None

# Programa Principal de Busca do Menor e do Maior Elementos
numeros = [0]*10
preencher(numeros)
extremos = buscarMenorMaiorElementos(numeros)
escrever(extremos)
```

Busca do Menor e Maior Elementos (continuação)

```
# Subprogramas
def preencher(valores):
    ...
def buscarMenorMaiorElementos(valores):
    menor = valores[0]
    maior = valores[0]
    for indice in range(1,len(valores)):
        if menor > valores[indice]:
            menor = valores[indice]
        elif maior < valores[indice]:
            maior = valores[indice]
    return [menor, maior]
def escrever(infos):
    ...
# Programa Principal de Busca do Menor e do Maior Elementos
numeros = [0]*10
preencher(numeros)
extremos = buscarMenorMaiorElementos(numeros)
escrever(extremos)
```

Busca do Menor e Maior Elementos

- O algoritmo apresentado encontra o menor e o maior elementos em um vetor de **n** elementos.

Busca do Menor e Maior Elementos

- O algoritmo apresentado encontra o menor e o maior elementos em um vetor de **n** elementos.

- Dado que cada um dos **n** elementos é avaliado necessariamente uma vez, a sua complexidade é da ordem de **n**, representado por O(**n**).

Faça os Exercícios Relacionados a essa Aula

Clique no botão para visualizar os enunciados:

Aula 6

Professores:

Dante Corbucci Filho
Leandro A. F. Fernandes

Conteúdo:

- Algoritmos de Busca
  - Busca Simples
  - Busca com Sentinela
  - Busca Binária
- Busca do Menor e Maior Elementos
- Noções de Complexidade de Algoritmos