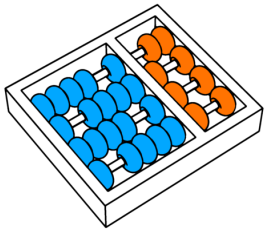


MC102 – Aula 11
Algoritmos de Busca

Algoritmos e Programação de Computadores

Zanoni Dias

2019

Instituto de Computação

Roteiro

O Problema da Busca

Busca Sequencial

Busca Binária

Análise de Eficiência

Exercícios

O Problema da Busca

O Problema da Busca

- Vamos estudar alguns algoritmos para o seguinte problema:

Definição do Problema

Dada uma chave de busca e uma coleção de elementos, onde cada elemento possui um identificador único, desejamos encontrar o elemento da coleção que possui o mesmo identificador da chave de busca ou verificar que não existe nenhum elemento na coleção com a chave fornecida.

- Nos nossos exemplos, a coleção de elementos será representada por uma lista de inteiros.
  - O identificador do elemento será o próprio valor de cada elemento.
- Apesar de usarmos inteiros, os algoritmos que estudaremos servem para buscar elementos em qualquer coleção de elementos que possuam chaves que possam ser comparadas.

O Problema da Busca

- O problema da busca é um dos mais básicos na área de Computação e possui diversas aplicações.
  - Buscar um aluno dado o seu RA.
  - Buscar um cliente dado o seu CPF.
  - Buscar uma pessoa dado o seu RG.
- Estudaremos algoritmos simples para realizar a busca assumindo que os dados estão em uma lista.
- Existem estruturas de dados e algoritmos mais complexos utilizados para armazenar e buscar elementos. Estas abordagens não serão estudadas nesta disciplina.

O Problema da Busca

- Nos vamos criar uma função `busca(lista, chave)`:
  - A função deve receber uma `lista` de números inteiros e uma `chave` para busca.
  - A função deve retornar o índice da lista que contém a chave ou o valor -1, caso a chave não esteja na lista.

O Problema da Busca

chave = 45

	0	1	2	3	4	5	6	7	8	9
lista	20	5	15	24	67	45	1	76	21	11

chave = 100

	0	1	2	3	4	5	6	7	8	9
lista	20	5	15	24	67	45	1	76	21	11

- No primeiro exemplo, a função deve retornar 5, enquanto no segundo exemplo, a função deve retornar -1.

Busca Sequencial

Busca Sequencial

- A busca sequencial é o algoritmo mais simples de busca:
  - Percorra a lista comparando a chave com os valores dos elementos em cada um das posições.
  - Se a chave for igual a algum dos elementos, retorne a posição correspondente na lista.
  - Se a lista toda foi percorrida e a chave não for encontrada, retorne o valor -1.

Busca Sequencial

```
1 def buscaSequencial(lista, chave):
2   i = 0
3   for número in lista:
4     if número == chave:
5       return i
6     i = i + 1
7   return -1
```

Busca Sequencial

```
1 def buscaSequencial(lista, chave):
2   n = len(lista)
3   for índice in range(n):
4     if lista[índice] == chave:
5       return índice
6
7   return -1
```

Busca Sequencial

- Podemos usar também a função `enumerate(lista)`, que retorna uma lista com tuplas da forma `(índice, elemento)`.

```
1 def buscaSequencial(lista, chave):
2   for (índice, número) in enumerate(lista):
3     if número == chave:
4       return índice
5
6   return -1
```

Busca Sequencial

```python
def buscaSequencial(lista, chave):
  ...

def main():
  lista = [20, 5, 15, 24, 67, 45, 1, 76, 21, 11]

  pos = buscaSequencial(lista, 45)
  if pos != -1:
    print("Posição da chave 45 na lista:", pos)
  else:
    print("A chave 45 não se encontra na lista.")

main()
# Posição da chave 45 na lista: 5
```

Busca Sequencial

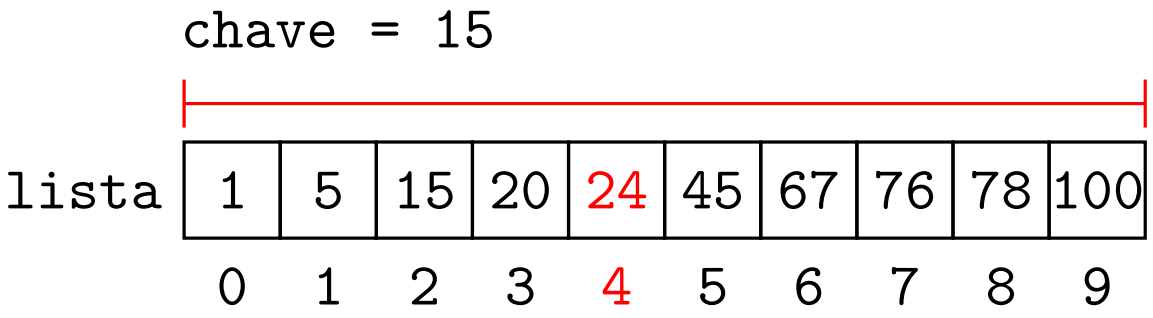
```
1  def buscaSequencial(lista, chave):
2    ...
3
4  def main():
5    lista = [20, 5, 15, 24, 67, 45, 1, 76, 21, 11]
6
7    pos = buscaSequencial(lista, 100)
8    if pos != -1:
9      print("Posição da chave 100 na lista:", pos)
10   else:
11     print("A chave 100 não se encontra na lista.")
12
13 main()
14 # A chave 100 não se encontra na lista.
```

Busca Binária

Busca Binária

- A busca binária é um algoritmo mais eficiente, entretanto, requer que a lista esteja ordenada pelos valores da chave de busca.
- A ideia do algoritmo é a seguinte (assuma que a lista está ordenada pelos valores da chave de busca):
  - Verifique se a chave de busca é igual ao valor da posição do meio da lista.
  - Caso seja igual, devolva esta posição.
  - Caso o valor desta posição seja maior que a chave, então repita o processo, mas considere uma lista reduzida, com os elementos do começo da lista até a posição anterior a do meio.
  - Caso o valor desta posição seja menor que chave, então repita o processo, mas considere uma lista reduzida, com os elementos da posição seguinte a do meio até o final da lista.

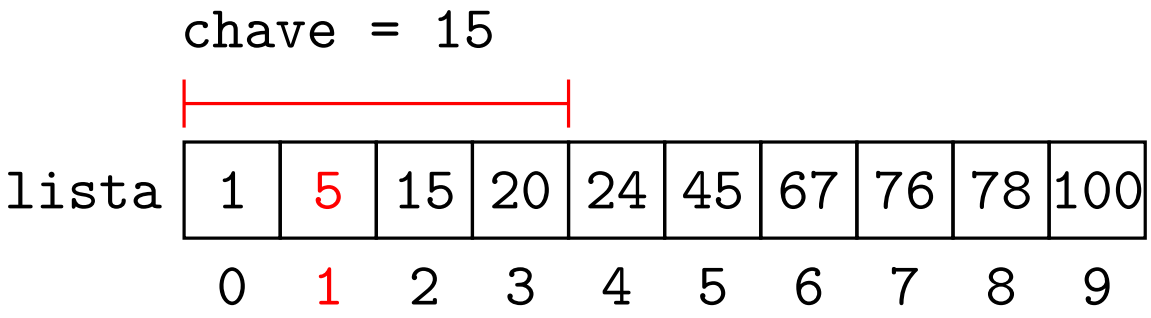
Busca Binária - Buscando a Chave 15




```
pos_ini = 0
pos_fim = 9
pos_meio = 4
```

- Como `lista[pos_meio] > chave`, devemos continuar a busca na primeira metade da região e, para isso, atualizamos a variável `pos_fim`.

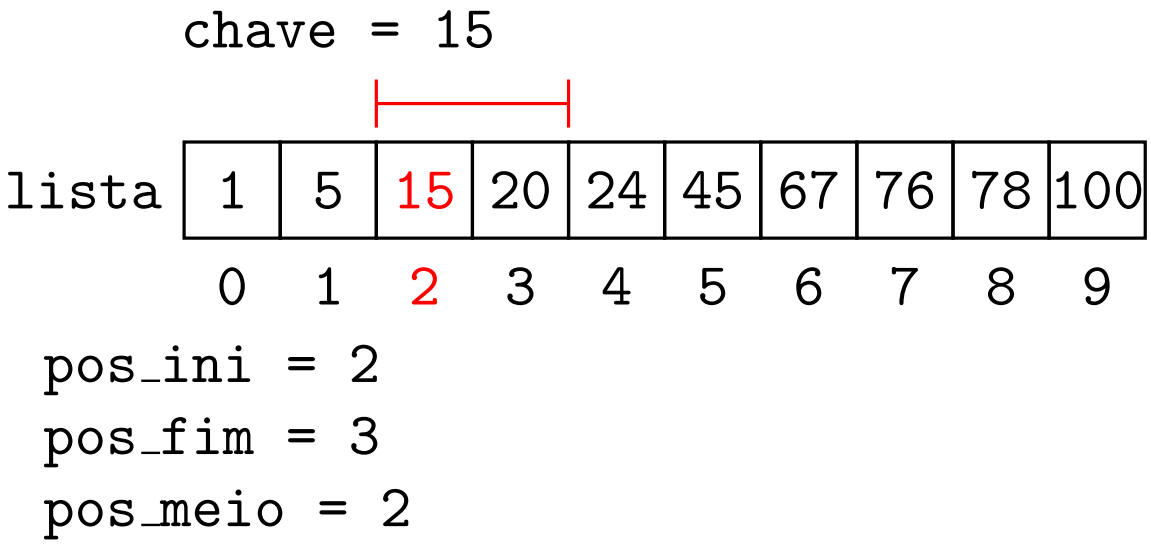
Busca Binária - Buscando a Chave 15




```
pos_ini = 0
pos_fim = 3
pos_meio = 1
```

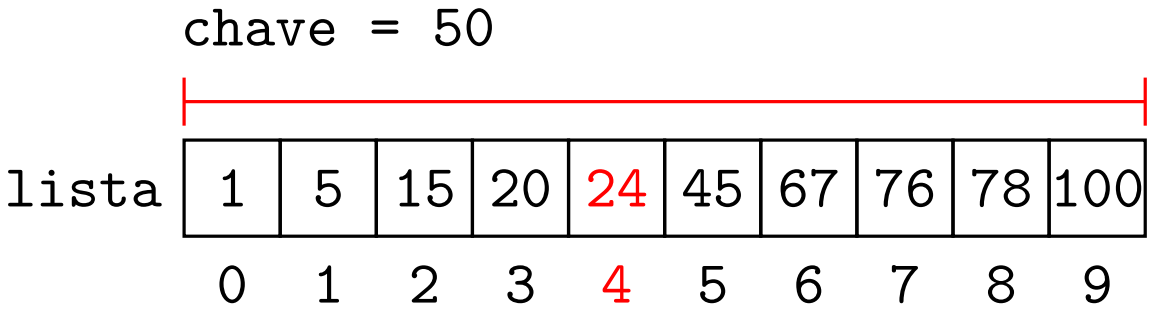
- Como `lista[pos_meio] < chave`, devemos continuar a busca na segunda metade da região e, para isso, atualizamos a variável `pos_ini`.

Busca Binária - Buscando a Chave 15



- Finalmente, encontramos a chave (`lista[pos_meio] = chave`) e, sendo assim, devolvemos a sua posição na lista (`pos_meio`).

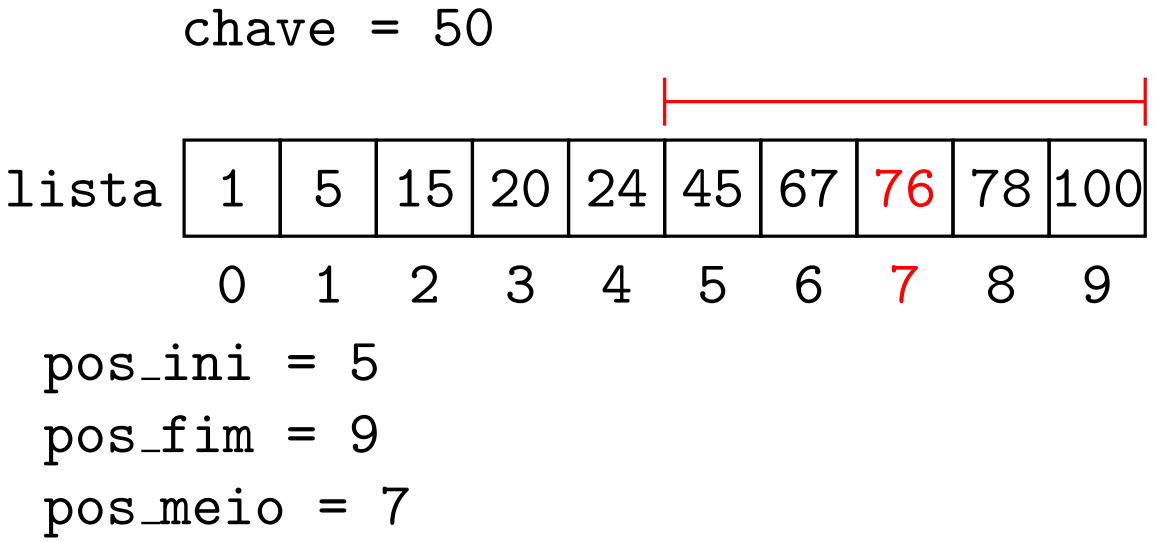
Busca Binária - Buscando a Chave 50




```
pos_ini = 0
pos_fim = 9
pos_meio = 4
```

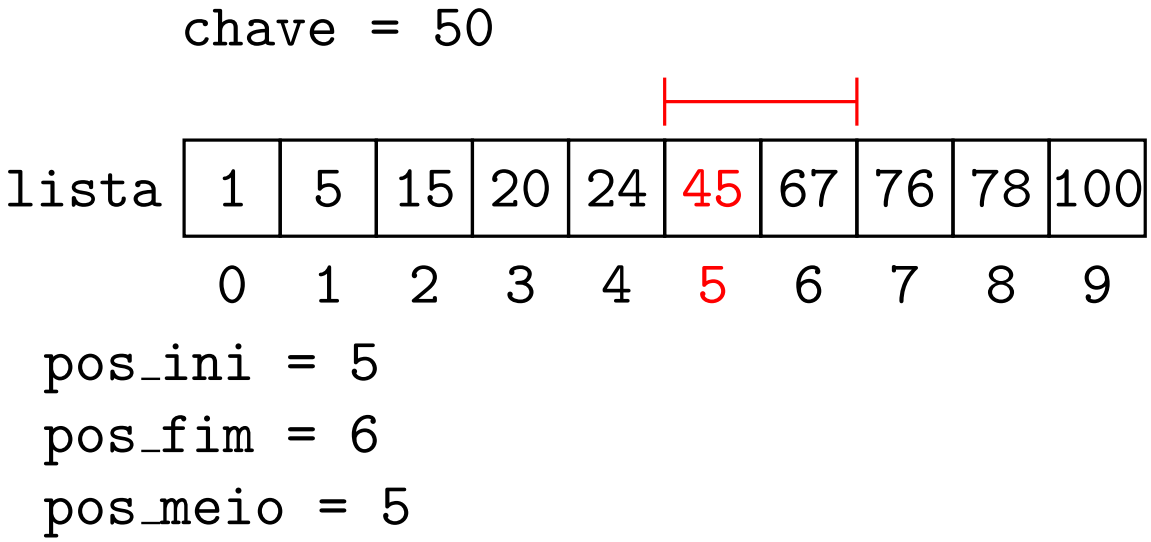
- Como `lista[pos_meio] < chave`, devemos continuar a busca na segunda metade da região e, para isso, atualizamos a variável `pos_ini`.

Busca Binária - Buscando a Chave 50



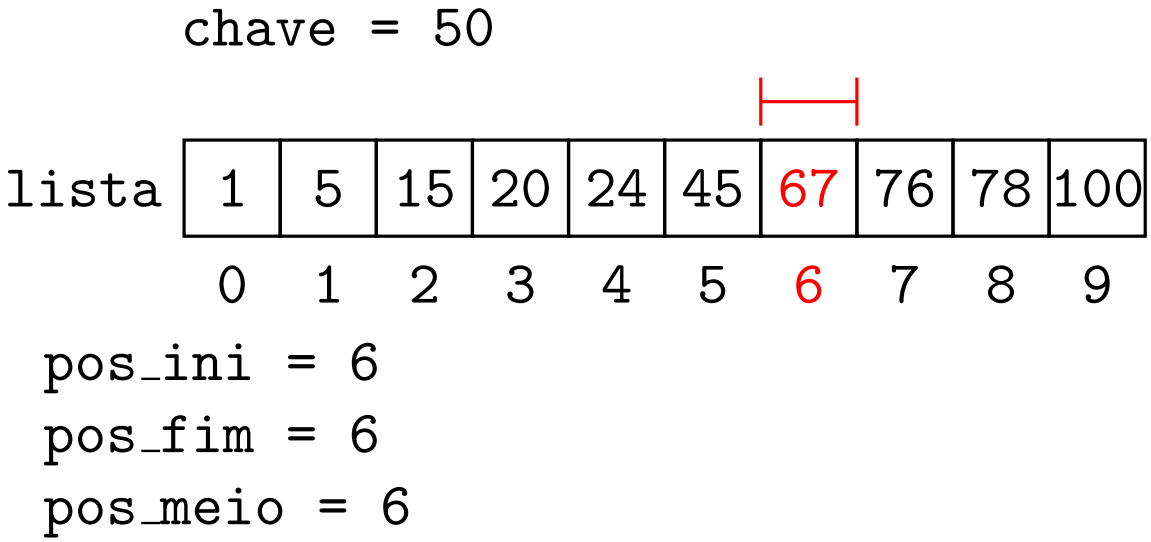
- Como `lista[pos_meio] > chave`, devemos continuar a busca na primeira metade da região e, para isso, atualizamos a variável `pos_fim`.

Busca Binária - Buscando a Chave 50



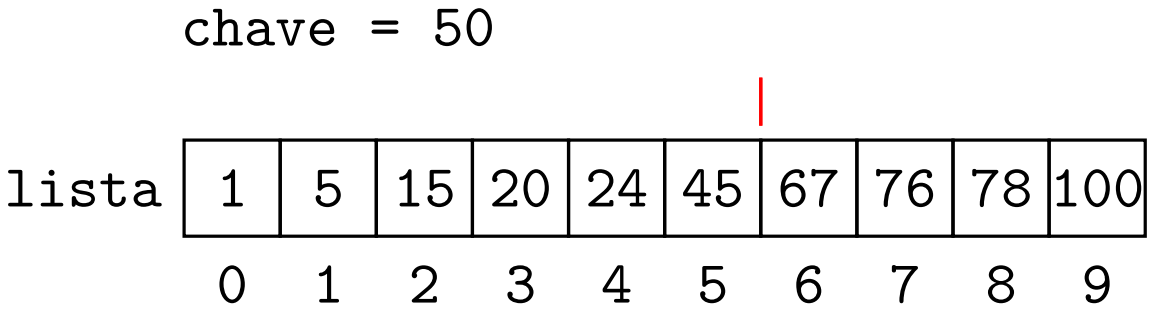
- Como `lista[pos_meio] < chave`, devemos continuar a busca na segunda metade da região e, para isso, atualizamos a variável `pos_ini`.

Busca Binária - Buscando a Chave 50



- Como `lista[pos_meio] > chave`, devemos continuar a busca na primeira metade da região e, para isso, atualizamos a variável `pos_fim`.

Busca Binária - Buscando a Chave 50



- Como `pos_ini > pos_end`, determinamos que a chave não está na lista e retornamos o valor -1.

Busca Binária

```
1  def buscaBinária(lista, chave):
2    pos_ini = 0
3    pos_fim = len(lista) - 1
4
5    while pos_ini <= pos_fim:
6      pos_meio = (pos_ini + pos_fim) // 2
7
8      if lista[pos_meio] == chave:
9        return pos_meio
10     if lista[pos_meio] > chave:
11       pos_fim = pos_meio - 1
12     if lista[pos_meio] < chave:
13       pos_ini = pos_meio + 1
14
15   return -1
```

Busca Binária

```
1  def buscaBinária(lista, chave):
2    pos_ini = 0
3    pos_fim = len(lista) - 1
4
5    while pos_ini <= pos_fim:
6      pos_meio = (pos_ini + pos_fim) // 2
7
8      if lista[pos_meio] == chave:
9        return pos_meio
10     if lista[pos_meio] > chave:
11       pos_fim = pos_meio - 1
12     else:
13       pos_ini = pos_meio + 1
14
15   return -1
```

Busca Binária

```
1  def buscaBinária(lista, chave):
2    ..
3
4  def main():
5    # Para usar a busca binária a lista deve estar ordenada
6    lista = [1, 5, 15, 20, 24, 45, 67, 76, 78, 100]
7
8    pos = buscaBinária(lista, 15)
9    if pos != -1:
10     print("Posição da chave 15 na lista:", pos)
11   else:
12     print("A chave 15 não se encontra na lista.")
13
14 main()
15 # Posição da chave 15 na lista: 2
```

Análise de Eficiência

Eficiência da Busca Sequencial

- Na melhor das hipóteses, a chave de busca estará na posição 0. Portanto, teremos um único acesso em `lista[0]`.
- Na pior das hipóteses, a chave é o último elemento ou não pertence à lista e, portanto, acessamos todos os n elementos da lista.
- É possível mostrar que, se as chaves possuirem a mesma probabilidade de serem requisitadas, o número médio de acessos nas buscas cujas chaves encontram-se na lista será igual a:

$$\frac{n+1}{2}$$

Eficiência da Busca Binária

- Na melhor das hipóteses, a chave de busca estará na posição do meio da lista. Portanto, teremos um único acesso.
- Na pior das hipóteses, teremos $(\log_2 n)$ acessos.
  - Para observar isso, note que, a cada verificação de uma posição da lista, o tamanho da lista considerada é dividido pela metade.
  - No pior caso, a busca é repetida até que a lista considerada tenha tamanho 1.
  - Assim, o número de acessos x pode ser encontrado resolvendo-se a equação:

    $$\frac{n}{2^x} = 1$$

    cuja solução é $x = \log_2 n$.
- É possível mostrar que, se as chaves possuirem a mesma probabilidade de serem requisitadas, o número médio de acessos nas buscas cujas chaves encontram-se na lista será igual a:

$$(\log_2 n) - 1$$

Eficiência dos Algoritmos

- Para se ter uma ideia da diferença de eficiência dos dois algoritmos, considere uma lista com um milhão de itens (10^6 itens).
- Com a busca sequencial, para buscar um elemento qualquer da lista necessitamos, em média, de:

$$(10^6 + 1)/2 \approx 500000 \text{ acessos.}$$

- Com a busca binária, para buscar um elemento qualquer da lista necessitamos, em média, de:

$$(\log_2 10^6) - 1 \approx 19 \text{ acessos.}$$

Eficiência dos Algoritmos

- Uma ressalva importante deve ser feita: para utilizar a busca binária, a lista precisa estar ordenada.
- Se você tiver um cadastro onde vários itens são removidos e inseridos com frequência e a busca deve ser feita de forma intercalada com essas operações, então a busca binária pode não ser a melhor opção, já que você precisará manter a lista ordenada.
- Caso o número de buscas seja muito maior que as demais operações de atualização do cadastro, então a busca binária pode ser uma boa opção.

Exercícios

Exercícios

1. Refaça as funções de busca sequencial e busca binária assumindo que a lista possui chaves que podem ocorrer múltiplas vezes na lista. Neste caso, você deve retornar uma lista com todas as posições onde a chave foi encontrada. Se a chave não for encontrada na lista, retornar uma lista vazia.

Exercícios

2. Mostre como implementar uma variação da busca binária que retorne um inteiro `k` entre `0` e `n`, tal que, ou `lista[k] = chave`, ou a chave não se encontra na lista, mas poderia ser inserida entre as posições `(k-1)` e `k` de forma a manter a lista ordenada. Note que, se `k = 0`, então a chave deveria ser inserida antes da primeira posição da lista, assim como, se `k = n`, a chave deveria ser inserida após a última posição da lista.
3. Use a função desenvolvida acima para, dada uma lista ordenada de `n` números inteiros e distintos e dois outros inteiros `X` e `Y`, retornar o número de chaves da lista que são maiores ou iguais a `X` e menores ou iguais a `Y`.

Exercício 1 - Buscando Múltiplos Elementos

```
1 def buscaSequencial(lista, chave):
2   posições = []
3   for (índice, número) in enumerate(lista):
4     if número == chave:
5       posições.append(índice)
6
7   return posições
```

Exercício 1 - Buscando Múltiplos Elementos

```
1  def buscaBinária(lista, chave):
2    pos_ini = 0
3    pos_fim = len(lista) - 1
4
5    while pos_ini <= pos_fim:
6      pos_meio = (pos_ini + pos_fim) // 2
7
8      if lista[pos_meio] == chave:
9        return encontraChaves(lista, pos_meio)
10     if lista[pos_meio] > chave:
11       pos_fim = pos_meio - 1
12     if lista[pos_meio] < chave:
13       pos_ini = pos_meio + 1
14
15   return []
```

Exercício 1 - Buscando Múltiplos Elementos

```
1  def encontraChaves(lista, pos):
2    posições = []
3    for i in range(pos, len(lista)):
4      if lista[i] > lista[pos]:
5        break
6      posições.append(i)
7    for i in range(pos - 1, -1, -1):
8      if lista[i] < lista[pos]:
9        break
10     posições.append(i)
11   return posições
12
13 def buscaBinária(lista, chave):
14   ...
15       return encontraChaves(lista, pos_meio)
16   ...
```

Exercício 2 - Onde um Elemento Deve Ser Inserido

```
1  def buscaBinária2(lista, chave):
2    pos_ini = 0
3    pos_fim = len(lista) - 1
4
5    while pos_ini <= pos_fim:
6      pos_meio = (pos_ini + pos_fim) // 2
7
8      if lista[pos_meio] == chave:
9        return pos_meio
10     if lista[pos_meio] > chave:
11       pos_fim = pos_meio - 1
12     if lista[pos_meio] < chave:
13       pos_ini = pos_meio + 1
14
15   return pos_ini
```

Exercício 3 - Número de Elementos entre X e Y

```python
def buscaBinária2(lista, chave):
  ...

def elementosEntre(lista, X, Y):
  pos_x = buscaBinária2(lista, X)
  pos_y = buscaBinária2(lista, Y)
  if pos_y < len(lista) and lista[pos_y] == Y:
    return pos_y - pos_x + 1
  else:
    return pos_y - pos_x
```