Algorítmos (1)

Aula 8

Curso: BIG863 - Basic Python Programming for Ecologists

Professora: Dra. Cecilia F. Fiorini

Supervisor: Prof. Dr. Fernando A. O. Silveira

https://meet.google.com/zdi-ueoz-nsr, 3 de maio de 2023

Roteiro*

*Conteúdo adaptado a partir de material desenvolvido pelo Prof. Zanoni Dias e disponível em https://ic.unicamp.br/ mc102.

ECMVS

O Problema da Ordenação

- Vamos estudar alguns algoritmos para o seguinte problema:

Definição do Problema

Dada uma coleção de elementos, com uma relação de ordem entre eles, ordenar os elementos da coleção de forma crescente.

- Nos nossos exemplos, a coleção de elementos será representada por uma lista de inteiros.
  - Números inteiros possuem uma relação de ordem entre eles.
- Apesar de usarmos números inteiros, os algoritmos que estudaremos servem para ordenar qualquer coleção de elementos que possam ser comparados entre si.

ECMVS

O Problema da Ordenação

- O problema da ordenação é um dos mais básicos em computação.
- Muito provavelmente este é um dos problemas com maior número de aplicações diretas ou indiretas (como parte da solução para um problema maior).
- Exemplos de aplicações diretas:
  - Criação de rankings.
  - Definição de preferências em atendimentos por prioridade.
- Exemplos de aplicações indiretas:
  - Otimização de sistemas de busca.
  - Manutenção de estruturas de bancos de dados.

ECMVS

Bubble Sort

- A ideia do algoritmo Bubble Sort é a seguinte:
- O algoritmo faz iterações repetindo os seguintes passos:
  - Se lista[0] > lista[1], troque lista[0] com lista[1].
  - Se lista[1] > lista[2], troque lista[1] com lista[2].
  - Se lista[2] > lista[3], troque lista[2] com lista[3].
  - ...
  - Se lista[n-2] > lista[n-1], troque lista[n-2] com lista[n-1].
- Após uma iteração executando os passos acima, o que podemos garantir?
  - O maior elemento estará na posição correta (a última da lista).

ECMVS

Bubble Sort

- Após a primeira iteração de trocas, o maior elemento estará na posição correta.
- Após a segunda iteração de trocas, o segundo maior elemento estará na posição correta.
- E assim sucessivamente...
- Quantas iterações são necessárias para deixar a lista completamente ordenada?

Bubble Sort

- No exemplo abaixo, os elementos sublinhados estão sendo comparados (e, eventualmente, serão trocados):
  - 57, 32, 25, 11, 90, 63
  - 32, 57, 25, 11, 90, 63
  - 32, 25, 57, 11, 90, 63
  - 32, 25, 11, 57, 90, 63
  - 32, 25, 11, 57, 90, 63
  - 32, 25, 11, 57, 63, 90
- Isto termina a primeira iteração de trocas.
- Como a lista possui 6 elementos, temos que realizar 5 iterações.
- Note que, após a primeira iteração, não precisamos mais avaliar a última posição da lista.

ECMVS

Trocando Elementos em uma Lista

- Podemos trocar os elementos das posições i e j de uma lista da seguinte forma:

```
1  lista = [1, 2, 3, 4, 5]
2  i = 0 # lista[0] = 1
3  3 j = 2 # lista[2] = 3
4
5  aux = lista[i]
6  lista[i] = lista[j]
7  lista[j] = aux
8
9  print(lista)
10 # [3, 2, 1, 4, 5]
```


ECMVS

Trocando Elementos em uma Lista

- Podemos trocar os elementos das posições i e j de uma lista da seguinte forma:

```
1 lista = [1, 2, 3, 4, 5]
2 i = 0 # lista[0] = 1
3 j = 2 # lista[2] = 3
4
5 (lista[i], lista[j]) = (lista[j], lista[i])
6
7 print(lista)
8 # [3, 2, 1, 4, 5]
```


ECMVS

Bubble Sort

- O código abaixo realiza as trocas de uma iteração do algoritmo.
- Os pares de elementos das posições 0 e 1, 1 e 2, . . ., i-1 e i são comparados e, eventualmente, trocados.
- Assumimos que, das posições i+1 até n-1, a lista já possui os maiores elementos ordenados.

```
1 for j in range(i):
2     if lista[j] > lista[j + 1]:
3         (lista[j], lista[j + 1]) = (lista[j + 1], lista[j])
```


ECMVS

Bubble Sort

```
1 def bubbleSort(lista):
2     n = len(lista)
3     for i in range(n - 1, 0, -1):
4         for j in range(i):
5             if lista[j] > lista[j + 1]:
6                 (lista[j], lista[j + 1]) = (lista[j + 1], lista[j])
```

- Note que as comparações na primeira iteração ocorrem até a última posição da lista.
- Na segunda iteração, elas ocorrem até a penúltima posição.
- E assim sucessivamente...

ECMVS

Bubble Sort - Análise de Complexidade

```
1 def bubbleSort(lista):
2     n = len(lista)
3     for i in range(n - 1, 0, -1):
4         for j in range(i):
5             if lista[j] > lista[j + 1]:
6                 (lista[j], lista[j + 1]) = (lista[j + 1], lista[j])
```

- Número máximo de comparações entre elementos da lista:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

ECMVS

Bubble Sort - Análise de Complexidade

```
1 def bubbleSort(lista):
2     n = len(lista)
3     for i in range(n - 1, 0, -1):
4         for j in range(i):
5             if lista[j] > lista[j + 1]:
6                 (lista[j], lista[j + 1]) = (lista[j + 1], lista[j])
```

- Número máximo de trocas entre elementos da lista:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

ECMVS

Bubble Sort - Análise de Complexidade

```
1 def bubbleSort(lista):
2     n = len(lista)
3     for i in range(n - 1, 0, -1):
4         for j in range(i):
5             if lista[j] > lista[j + 1]:
6                 (lista[j], lista[j + 1]) = (lista[j + 1], lista[j])
```

- Número mínimo de comparações entre elementos da lista:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

ECMVS

Bubble Sort - Análise de Complexidade

```
1 def bubbleSort(lista):
2     n = len(lista)
3     for i in range(n - 1, 0, -1):
4         for j in range(i):
5             if lista[j] > lista[j + 1]:
6                 (lista[j], lista[j + 1]) = (lista[j + 1], lista[j])
```

- Número mínimo de trocas entre elementos da lista:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 0 = 0$$

ECMVS

To bubble sort or not to bubble sort?

Bubble Sort

ECMVS

Selection Sort

- Dada uma lista contendo n números inteiros, desejamos ordenar essa lista de forma crescente.
- A ideia do algoritmo é a seguinte:
  - Encontre o menor elemento a partir da posição 0. Troque este elemento com o elemento da posição 0.
  - Encontre o menor elemento a partir da posição 1. Troque este elemento com o elemento da posição 1.
  - Encontre o menor elemento a partir da posição 2. Troque este elemento com o elemento da posição 2.
  - E assim sucessivamente...

ECMVS

Selection Sort

- No exemplo abaixo, os elementos sublinhados representam os elementos que serão trocados na iteração i do Selection Sort:
  - Iteração 0: 57, 32, 25, 11, 90, 63
  - Iteração 1: 11, 32, 25, 57, 90, 63
  - Iteração 2: 11, 25, 32, 57, 90, 63
  - Iteração 3: 11, 25, 32, 57, 90, 63
  - Iteração 4: 11, 25, 32, 57, 90, 63
  - Iteração 5: 11, 25, 32, 57, 63, 90

ECMVS

Selection Sort

- Podemos criar uma função que retorna o índice do menor elemento de uma lista (formado por n números inteiros) a partir de uma posição inicial dada:

```
1 def indiceMenor(lista, inicio):
2     minimo = inicio
3     n = len(lista)
4     for j in range(inicio + 1, n):
5         if lista[minimo] > lista[j]:
6             minimo = j
7     return minimo
```


ECMVS

Selection Sort

- Dada a função anterior, que encontra o índice do menor elemento de uma lista a partir de uma dada posição, como implementar o algoritmo de ordenação?
  - Encontre o menor elemento a partir da posição 0 e troque-o com o elemento da posição 0.
  - Encontre o menor elemento a partir da posição 1 e troque-o com o elemento da posição 1.
  - Encontre o menor elemento a partir da posição 2 e troque-o com o elemento da posição 2.
  - E assim sucessivamente...

ECMVS

Selection Sort

- Usando a função auxiliar indiceMenor podemos implementar o Selection Sort da seguinte forma:

```
1  def selectionSort(lista):
2      n = len(lista)
3      for i in range(n - 1):
4          minimo = indiceMenor(lista, i)
5          (lista[i], lista[minimo]) = (lista[minimo], lista[i])
```


ECMVS

Selection Sort - Análise de Complexidade

```
1  def selectionSort(lista):
2      n = len(lista)
3      for i in range(n - 1):
4          minimo = indiceMenor(lista, i)
5          (lista[i], lista[minimo]) = (lista[minimo], lista[i])
```

- Número máximo de comparações entre elementos da lista:

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} n - i - 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

ECMVS

Selection Sort - Análise de Complexidade

```
1 def selectionSort(lista):
2     n = len(lista)
3     for i in range(n - 1):
4         minimo = indiceMenor(lista, i)
5         (lista[i], lista[minimo]) = (lista[minimo], lista[i])
```

- Número máximo de trocas entre elementos da lista:

$$f(n) = \sum_{i=0}^{n-2} 1 = n - 1$$

ECMVS

Selection Sort - Análise de Complexidade

```
1  def selectionSort(lista):
2      n = len(lista)
3      for i in range(n - 1):
4          minimo = indiceMenor(lista, i)
5          (lista[i], lista[minimo]) = (lista[minimo], lista[i])
```

- Número mínimo de comparações entre elementos da lista:

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} n - i - 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

ECMVS

Selection Sort - Análise de Complexidade

```
1 def selectionSort(lista):
2     n = len(lista)
3     for i in range(n - 1):
4         minimo = indiceMenor(lista, i)
5         (lista[i], lista[minimo]) = (lista[minimo], lista[i])
```

- Número mínimo de trocas entre elementos da lista:

$$f(n) = \sum_{i=0}^{n-2} 1 = n - 1$$

ECMVS

Selection Sort

- É possível diminuir o número de trocas no melhor caso?
- Vale a pena testar se lista[i] é diferente de lista[minimo] antes de realizar a troca?

Insertion Sort

- A ideia do algoritmo Insertion Sort é a seguinte:
  - A cada iteração i, os elementos das posições 0 até i-1 da lista estão ordenados.
  - Então, precisamos inserir o elemento da posição i, entre as posições 0 e i, de forma a deixar a lista ordenada até a posição i.
  - Na iteração seguinte, consideramos que a lista está ordenada até a posição i e repetimos o processo até que a lista esteja completamente ordenada.

Insertion Sort

- No exemplo abaixo, o elemento sublinhado representa o elemento que será inserido na i-ésima iteração do Insertion Sort:
  - 57, 25, 32, 11, 90, 63: lista ordenada entre as posições 0 e 0.
  - 25, 57, 32, 11, 90, 63: lista ordenada entre as posições 0 e 1.
  - 25, 32, 57, 11, 90, 63: lista ordenada entre as posições 0 e 2.
  - 11, 25, 32, 57, 90, 63: lista ordenada entre as posições 0 e 3.
  - 11, 25, 32, 57, 90, 63: lista ordenada entre as posições 0 e 4.
  - 11, 25, 32, 57, 63, 90: lista ordenada entre as posições 0 e 5.

ECMVS

Insertion Sort

- Podemos criar uma função que, dados uma lista e um índice i, insere o elemento de índice i entre os elementos das posições 0 e i-1 (pré-ordenados), de forma que todos os elementos entre as posições 0 e i fiquem ordenados:

```
1  def insertion(lista, i):
2      aux = lista[i]
3      j = i - 1
4      while (j >= 0) and (lista[j] > aux):
5          lista[j + 1] = lista[j]
6          j = j - 1
7      lista[j + 1] = aux
```

Insertion Sort

- Exemplo de execução da função insertion:
  - Configuração inicial:
    - 11, 31, 54, 58, 66, 12, 47, i = 5, aux = 12
  - Iterações:
    - 11, 31, 54, 58, 66, 12, 47, j = 4
    - 11, 31, 54, 58, 66, 66, 47, j = 3
    - 11, 31, 54, 58, 58, 66, 47, j = 2
    - 11, 31, 54, 54, 58, 66, 47, j = 1
    - 11, 31, 31, 54, 58, 66, 47, j = 0
  - Neste ponto temos que lista[j] < aux, logo, o loop while é encerrado e a atribuição lista[j + 1] = aux é executada:
    - 11, 12, 31, 54, 58, 66, 47

ECMVS

Insertion Sort

- Em Python podemos implementar a função insertion de forma ainda mais simples, inserindo o elemento na posição desejada com um único comando.

```
1  def insertion(lista, i):
2      j = i - 1
3      while (j >= 0) and (lista[j] > lista[i]):
4          j = j - 1
5      lista[j + 1:i + 1] = [lista[i]] + lista[j + 1:i]
```

Insertion Sort

- Usando a função auxiliar insertion podemos implementar o Insertion Sort da seguinte forma:

```
1  def insertionSort(lista):
2      n = len(lista)
3      for i in range(1, n):
4          insertion(lista, i)
```

Insertion Sort - Análise de Complexidade

```
1 def insertionSort(lista):
2     n = len(lista)
3     for i in range(1, n):
4         insertion(lista, i)
```

- Número máximo de comparações entre elementos da lista:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

ECMVS

Insertion Sort - Análise de Complexidade

```
1 def insertionSort(lista):
2     n = len(lista)
3     for i in range(1, n):
4         insertion(lista, i)
```

- Número máximo de modificações realizadas na lista:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i} 1 = \sum_{i=1}^{n-1} (i+1) = (n-1)\frac{n+2}{2} = \frac{n^2+n}{2} - 1$$

ECMVS

Insertion Sort - Análise de Complexidade

```
1 def insertionSort(lista):
2     n = len(lista)
3     for i in range(1, n):
4         insertion(lista, i)
```

- Número mínimo de comparações entre elementos da lista:

$$f(n) = \sum_{i=1}^{n-1} 1 = n - 1$$

Insertion Sort - Análise de Complexidade

```
1 def insertionSort(lista):
2     n = len(lista)
3     for i in range(1, n):
4         insertion(lista, i)
```

- Número mínimo de modificações realizadas na lista:

$$f(n) = \sum_{i=1}^{n-1} 1 = n - 1$$

Resumo

- Não existe um algoritmo de ordenação que seja o melhor em todas as possíveis situações.
- Para escolher o algoritmo mais adequado para uma dada situação, precisamos verificar as características específicas dos elementos que devem ser ordenados.
- Por exemplo:
  - Se os elementos a serem ordenados forem grandes, por exemplo, registros acadêmicos de alunos, o Selection Sort pode ser uma boa escolha, já que ele efetuará, no pior caso, muito menos trocas que o Insertion Sort ou o Bubble Sort.
  - Se os elementos a serem ordenados estiverem quase ordenados (situação relativamente comum), o Insertion Sort realizará muito menos operações (comparações e trocas) do que o Selection Sort ou o Bubble Sort.
- Teste de tempo de execução dos algoritmos de ordenação:
  Link do Teste

ECMVS

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 identificador igual ao 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.

ECMVS

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 uma amostra dado o seu código.
  - 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.

ECMVS

O Problema da Busca

- 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

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

chave = 100

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

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

ECMVS

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 uma 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.

ECMVS

Busca Sequencial

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


ECMVS

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      return -1
```


ECMVS

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      return -1
```


ECMVS

Busca Sequencial

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


ECMVS

Busca Sequencial

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

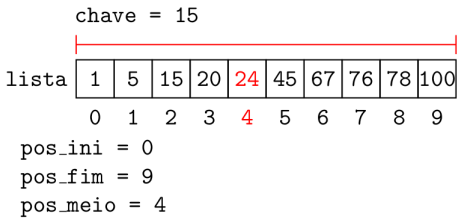

ECMVS

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.

ECMVS

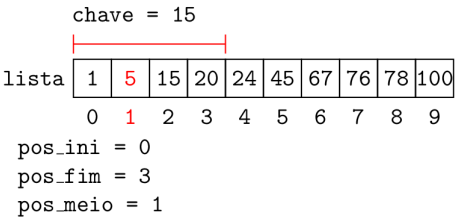
Busca Binária - Buscando a Chave 15



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

ECMVS

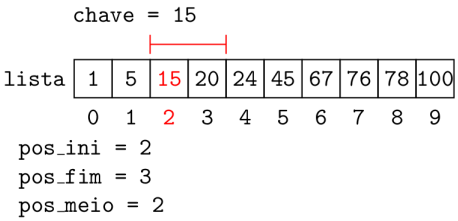
Busca Binária - Buscando a Chave 15



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

ECMVS

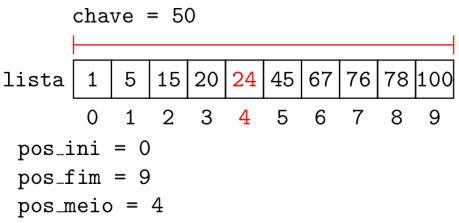
Busca Binária - Buscando a Chave 15



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

ECMVS

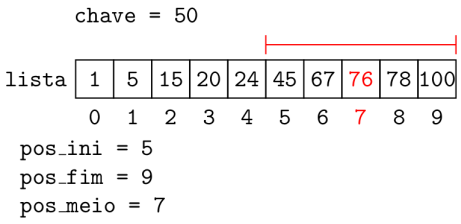
Busca Binária - Buscando a Chave 15



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

ECMVS

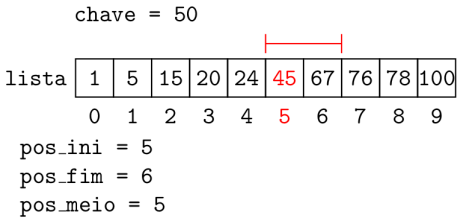
Busca Binária - Buscando a Chave 15



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

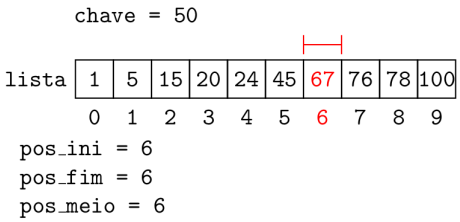
ECMVS

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.

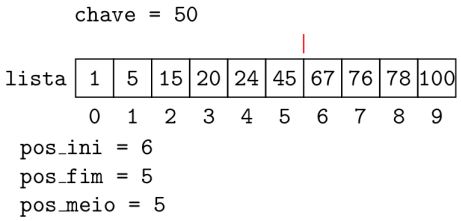
Busca Binária - Buscando a Chave 50



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

ECMVS

Busca Binária - Buscando a Chave 50



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

ECMVS

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     return -1
```


ECMVS

Busca Binária

```
1  def buscaBinária(lista, chave):
2      ...
3
4  chave = 15
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, chave)
9
10 if pos != -1:
11     print("Posição da chave", chave, "na lista:", pos)
12 else:
13     print("A chave", chave, "não se encontra na lista")
14
15 # Posição da chave 15 na lista: 2
```

Busca Binária

```
1  def buscaBinária(lista, chave):
2      ...
3
4  chave = 50
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, chave)
9
10 if pos != -1:
11     print("Posição da chave", chave, "na lista:", pos)
12 else:
13     print("A chave", chave, "não se encontra na lista")
14
15 # A chave 50 não se encontra na lista
```

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:

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

ECMVS

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, dividimos a lista até a que ela fique com um único elemento (último acesso realizado à lista).
- Note que, a cada acesso, o tamanho da lista é diminuído, pelo menos, pela metade.
- Quantas vezes um número pode ser dividido por dois antes dele se tornar igual a um?
- Esta é exatamente a definição de logaritmo na base 2.
- Ou seja, no pior caso o número de acesso é igual a log2 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:

$$(\log_2 n) - 1$$

ECMVS

Eficiência da Busca Binária

- Para se ter uma ideia da diferença de eficiência dos dois algoritmos, considere uma lista com um milhão de 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:

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

ECMVS

Eficiência da Busca Binária

- 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.

ECMVS