

MC-102 — Aula 15  
Ordenação – InsertionSort  
e Busca

Instituto de Computação – Unicamp

21 de Setembro de 2015

# Roteiro

- 1 InsertionSort
- 2 O Problema da Busca
- 3 Busca Sequencial
- 4 Busca Binária
- 5 Questões sobre eficiência
- 6 Exercícios

# Ordenação

- Continuamos com o estudo de algoritmos para o problema de ordenação:

Dado uma coleção de elementos com uma relação de ordem entre si, devemos gerar uma saída com os elementos ordenados.

- Novamente usaremos uma lista de inteiros como exemplo de coleção a ser ordenada.

# Insertion-Sort

- Seja **vet** uma lista contendo números, que devemos deixar ordenado.
- A idéia do algoritmo é a seguinte:
  - ▶ A cada passo, uma porção de 0 até  $i - 1$  da lista já está ordenada.
  - ▶ Devemos inserir o item da posição  $i$  na posição correta para deixar a lista ordenada até a posição  $i$ .
  - ▶ No passo seguinte consideramos que a lista está ordenado até  $i$ .

# Insertion-Sort

Exemplo: [5,3,2,1,90,6].

O valor sublinhado representa onde está o índice  $i$

[5, 3, 2, 1, 90, 6] : lista ordenada de 0 – 0.

[3, 5, 2, 1, 90, 6] : lista ordenada de 0 – 1.

[2, 3, 5, 1, 90, 6] : lista ordenada de 0 – 2.

[1, 2, 3, 5, 90, 6] : lista ordenada de 0 – 3.

[1, 2, 3, 5, 90, 6] : lista ordenada de 0 – 4.

[1, 2, 3, 5, 6, 90) : lista ordenado de 0 – 5.

# Insertion-Sort

- Vamos supor que a lista está ordenada de 0 até  $i - 1$ .
- Vamos inserir o elemento da posição  $i$  no lugar correto.

```
atual=vet[i]
j = i
while j>0 and vet[j-1]>atual:
    vet[j]=vet[j-1]
    j=j-1
vet[j]=atual
```

```
def insertionSort(vet):
    for i in range(1,len(vet)):
        atual=vet[i]
        j=i
        while j>0 and vet[j-1]>atual:
            vet[j]=vet[j-1]
            j = j-1
        vet[j]=atual
```

```
>>> lista=[20, 5, 15, 24, 67, 45, 1, 76]
>>> insertionSort(lista)
>>> lista
[1, 5, 15, 20, 24, 45, 67, 76]
```

# O Problema da Busca

- Vamos estudar alguns algoritmos para o seguinte problema:

Temos uma coleção de elementos, onde cada elemento possui um identificador/chave único, e recebemos uma chave para busca. Devemos encontrar o elemento da coleção que possui a mesma chave ou identificar que não existe nenhum elemento com a chave dada.

- Nos nossos exemplos usaremos uma lista de inteiros como a coleção.
  - ▶ O valor da chave será o próprio valor de cada número.
- Apesar de usarmos inteiros, os algoritmos 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 em Computação e também possui diversas aplicações.
  - ▶ Suponha que temos um cadastro com registros de motoristas.
  - ▶ Uma lista de registros é usado para armazenar as informações dos motoristas. Podemos usar como chave o número da carteira de motorista, ou o RG, ou o CPF.
- Veremos algoritmos simples para realizar a busca assumindo que dados estão em uma lista.
- Em cursos mais avançados são estudados outros algoritmos e estruturas (que não uma lista) para armazenar e buscar elementos.

# O Problema da Busca

- Nos nossos exemplos vamos criar a função:
  - ▶ **busca(vet, chave)**, que recebe uma lista e uma chave para busca.
  - ▶ A função deve retornar o índice da lista que contém a chave ou -1 caso a chave não esteja na lista.

# O Problema da Busca

Python já contém um método em listas que faz a busca `index()`:

```
>>> lista=[20, 5, 15, 24, 67, 45, 1, 76]
>>> lista.index(24)
3
```

mas esse método não funciona da forma que queremos para chaves que não estão na lista

```
>>> lista.index(100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 100 is not in list
```

# Busca Sequencial

- A busca sequencial é o algoritmo mais simples de busca:
  - ▶ Percorra toda a lista comparando a chave com o valor de cada posição.
  - ▶ Se for igual para alguma posição, então devolva esta posição.
  - ▶ Se a lista toda foi percorrida então devolva -1.
  - ▶ A busca sequencial é mais ou menos o que o *index* faz, mas em vez de retornar -1 ele dá um erro.

# Busca Sequencial

```
def buscaSequencial(vet, chave):  
    for i in range(len(vet)):  
        if vet[i] == chave:  
            return i  
    return -1
```

# Busca Sequencial

```
>>> buscaSequencial(lista,24)
3
>>> buscaSequencial(lista,100)
-1
```

# Busca Binária

- A busca binária é um algoritmo um pouco mais sofisticado.
- É mais eficiente, mas requer que a lista esteja ordenada pelos valores da chave de busca.
- A idéia do algoritmo é a seguinte (assuma que a lista está ordenada):
  - ▶ 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, então repita o processo mas considere que a lista tem metade do tamanho, indo até posição anterior a do meio.
  - ▶ Caso o valor desta posição seja menor, então repita o processo mas considere que a lista tem metade do tamanho e inicia na posição seguinte a do meio.

# Busca Binária

```
# a lista começa em posIni e termina em posFim
```

```
posIni = 0
```

```
posFim = len(vet)-1
```

```
Repita enquanto tamanho da lista considerada for >= 1
```

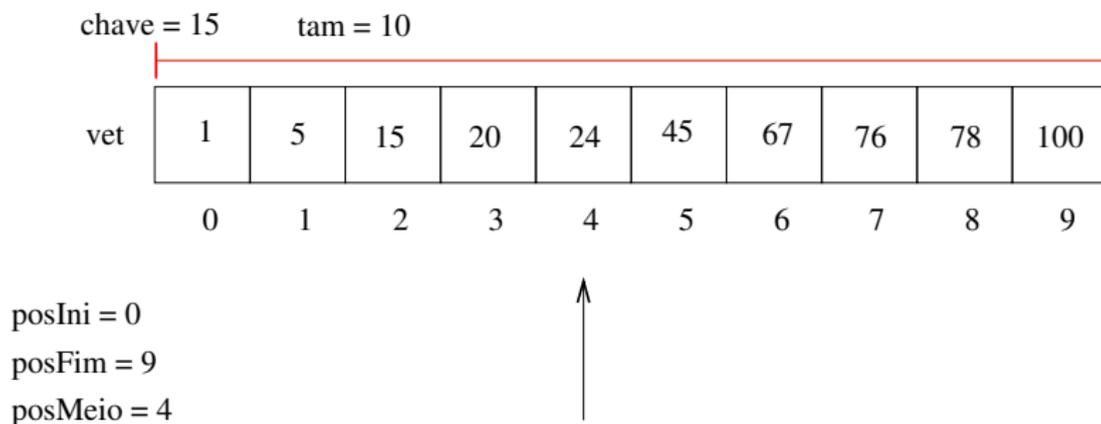
```
    posMeio = (posIni + posFim)/2 # deve retornar um inteiro
```

```
    Se vet[posMeio] == chave Então  
        devolva posMeio
```

```
    Se vet[posMeio] > chave Então  
        posFim = posMeio - 1
```

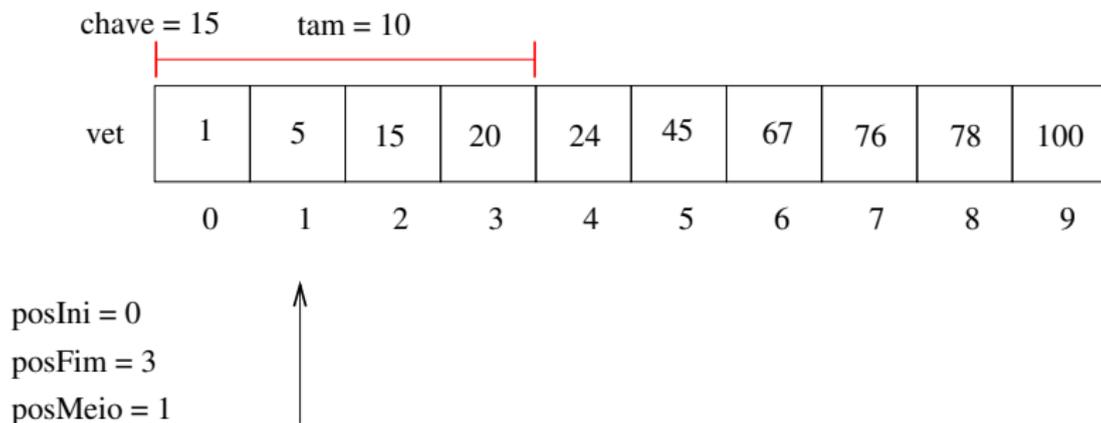
```
    Se vet[posMeio] < chave Então  
        posIni = posMeio + 1
```

# Busca Binária



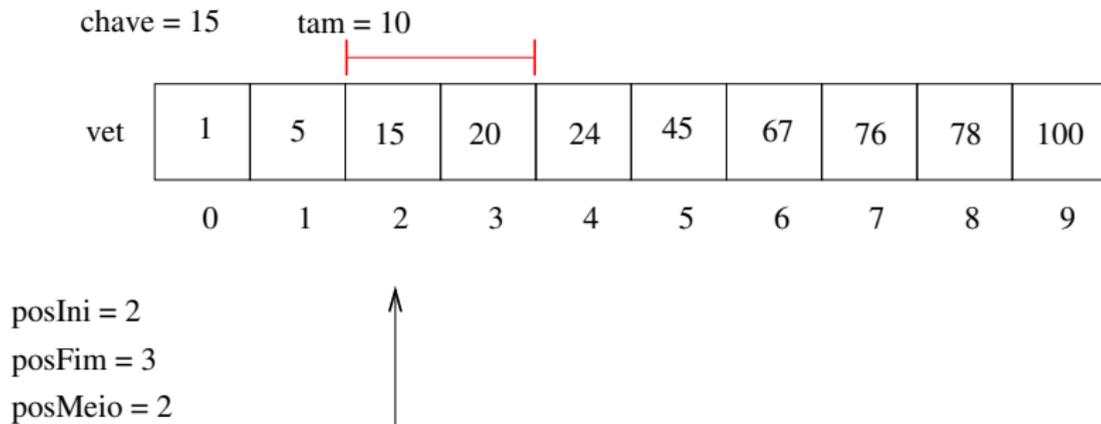
Como o valor da posição do meio é maior que a chave, atualizamos **posFim** da lista considerada.

# Busca Binária



Como o valor da posição do meio é menor que a chave, atualizamos **posIni** da lista considerada.

# Busca Binária



Finalmente encontramos a chave e podemos devolver sua posição 2.

# Busca Binária

```
def buscaBinaria(vet, chave):
    posIni=0
    posFim=len(vet)-1
    while posIni <= posFim:
        posMeio = (posIni+posFim)//2
        if vet[posMeio] == chave:
            return posMeio
        elif vet[posMeio] > chave:
            posFim = posMeio - 1
        else:
            posIni = posMeio + 1
    return -1
```

# Busca Binária

```
>>> lista=[20, 5, 15, 24, 67, 45, 1, 76]
>>> insertionSort(lista)
>>> lista
[1, 5, 15, 20, 24, 45, 67, 76]
>>> buscaBinaria(lista,24)
4
>>> buscaBinaria(lista,25)
-1
```

# Eficiência dos Algoritmos

Podemos medir a eficiência de qualquer algoritmo analisando a quantidade de recursos (tempo, memória, banda de rede, etc.) que o algoritmo usa para resolver o problema para o qual foi proposto.

- A forma mais simples é medir a eficiência em relação ao tempo. Para isso, analisamos quantas instruções um algoritmo usa para resolver o problema.
- Podemos fazer uma análise simplificada dos algoritmos de busca analisando a quantidade de vezes que os algoritmos **acessam** uma posição da lista.

# Eficiência dos Algoritmos

No caso da busca sequencial existem três possibilidades:

- Na melhor das hipóteses a chave de busca estará na posição 0. Portanto teremos um único acesso em **vet[0]**.
- Na pior das hipóteses, a chave é o último elemento ou não pertence a lista, e portanto acessaremos todas as posições da lista.
- É possível mostrar que se uma chave qualquer pode ser requisitada com a mesma probabilidade, então o número de acessos será

$$(tam + 1)/2$$

na média.

# Eficiência dos Algoritmos

No caso da busca binária temos as três possibilidades:

- Na melhor das hipóteses a chave de busca estará na posição do meio. Portanto teremos um único acesso.
- Na pior das hipóteses, teremos  $(\log \text{tam})$  acessos.
  - ▶ Para ver isso note que a cada verificação de uma posição da lista, o tamanho da lista considerado é dividido pela metade. No pior caso repetimos a busca até a lista considerada ter tamanho 1. Se você pensar um pouco, o número de acessos  $x$  pode ser encontrado resolvendo-se a equação:

$$\frac{\text{tam}}{2^x} = 1$$

cuja solução é  $x = (\log_2 \text{tam})$ .

- É possível mostrar que se uma chave qualquer pode ser requisitada com a mesma probabilidade, então o número de acessos será

$$(\log_2 \text{tam}) - 1$$

na média.

# Eficiência dos Algoritmos

Para se ter uma idéia da diferença de eficiência dos dois, considere que temos um cadastro com  $10^6$  (um milhão) de itens.

- Com a busca sequencial, a procura de um item qualquer gastará na média

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

- Com a busca binária teremos

$$(\log_2 10^6) - 1 \approx 20 \text{ acessos.}$$

# Eficiência dos Algoritmos

Mas uma ressalva 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 intercalado com estas operações, então a busca binária pode não ser a melhor opção, já que você precisará ficar mantendo a lista ordenada.
- Caso o número de buscas feitas seja muito maior, quando comparado com outras operações, então a busca binária é uma boa opção.

# Exercícios

- Altere o código do algoritmo insertionSort para que este ordene uma lista de inteiros em ordem decrescente.

# Exercícios

- Refaça as funções de busca sequencial e busca binária assumindo que a lista possui chaves que podem aparecer repetidas. Neste caso, você deve retornar uma lista com todas as posições onde a chave foi encontrada. Se a chave só aparece uma vez, a lista conterá apenas um índice. E se a chave não aparece, as funções devem retornar a lista vazia.

## Informações extras

O Python já tem um método que ordena listas `sort()`:

```
>>> lista
[20, 5, 15, 24, 67, 45, 1, 76]
>>> lista.sort()
>>> lista
[1, 5, 15, 20, 24, 45, 67, 76]
```

e uma função que cria uma lista nova, ordenada, dado o parametro, `sorted(lista)`:

```
>>> lista=[20, 5, 15, 24, 67, 45, 1, 76]
>>> x=sorted(lista)
>>> lista
[20, 5, 15, 24, 67, 45, 1, 76]
>>> x
[1, 5, 15, 20, 24, 45, 67, 76]
```

## Informações extras

E você pode usar esse método e a função em vez de implementar as 3 sorts que vimos: `selectionSort`, `bubleSort` e `insertionSort`.

Então por que aprender os sorts? O objetivo dessa disciplina não é apenas ensinar o Python, mas ensinar a programar. Os 3 algoritmos vistos são algoritmos mais complexos (do que os vistos até agora) e vai prepara-los para outros algoritmos mais complexos que trabalham com listas.