MC-102 — Aula 20
Ordenação – Insertion Sort e Busca em Vetores

Eduardo C. Xavier

Instituto de Computação – Unicamp

25 de Novembro de 2020

Roteiro

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 um vetor de inteiros como exemplo de coleção a ser ordenada.

Insertion Sort

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

Insertion Sort

Exemplo: (5,3,2,1,90,6).
O valor sublinhado representa onde está o índice i

$(5, \underline{3}, 2, 1, 90, 6)$: vetor ordenado de $0 - 0$.
$(3, 5, \underline{2}, 1, 90, 6)$: vetor ordenado de $0 - 1$.
$(2, 3, 5, \underline{1}, 90, 6)$: vetor ordenado de $0 - 2$.
$(1, 2, 3, 5, \underline{90}, 6)$: vetor ordenado de $0 - 3$.
$(1, 2, 3, 5, 90, \underline{6})$: vetor ordenado de $0 - 4$.
$(1, 2, 3, 5, 6, 90)$: vetor ordenado de $0 - 5$.

Insertion Sort

Exemplo: (5,3,2,1,90,6).
O valor sublinhado representa onde está o índice i
$(5, \underline{3}, 2, 1, 90, 6)$: vetor ordenado de $0 - 0$.
$(3, 5, \underline{2}, 1, 90, 6)$: vetor ordenado de $0 - 1$.
$(2, 3, 5, \underline{1}, 90, 6)$: vetor ordenado de $0 - 2$.
$(1, 2, 3, 5, \underline{90}, 6)$: vetor ordenado de $0 - 3$.
$(1, 2, 3, 5, 90, \underline{6})$: vetor ordenado de $0 - 4$.
$(1, 2, 3, 5, 6, 90)$: vetor ordenado de $0 - 5$.

Insertion Sort

Exemplo: (5,3,2,1,90,6).
O valor sublinhado representa onde está o índice i
$(5, \underline{3}, 2, 1, 90, 6)$: vetor ordenado de $0 - 0$.
$(3, 5, \underline{2}, 1, 90, 6)$: vetor ordenado de $0 - 1$.
$(2, 3, 5, \underline{1}, 90, 6)$: vetor ordenado de $0 - 2$.
$(1, 2, 3, 5, \underline{90}, 6)$: vetor ordenado de $0 - 3$.
$(1, 2, 3, 5, 90, \underline{6})$: vetor ordenado de $0 - 4$.
$(1, 2, 3, 5, 6, 90)$: vetor ordenado de $0 - 5$.

Insertion Sort

Exemplo: (5,3,2,1,90,6).
O valor sublinhado representa onde está o índice i
$(5, \underline{3}, 2, 1, 90, 6)$: vetor ordenado de $0 - 0$.
$(3, 5, \underline{2}, 1, 90, 6)$: vetor ordenado de $0 - 1$.
$(2, 3, 5, \underline{1}, 90, 6)$: vetor ordenado de $0 - 2$.
$(1, 2, 3, 5, \underline{90}, 6)$: vetor ordenado de $0 - 3$.
$(1, 2, 3, 5, 90, \underline{6})$: vetor ordenado de $0 - 4$.
$(1, 2, 3, 5, 6, 90)$: vetor ordenado de $0 - 5$.

Insertion Sort

Exemplo: (5,3,2,1,90,6).
O valor sublinhado representa onde está o índice i
$(5, \underline{3}, 2, 1, 90, 6)$: vetor ordenado de $0 - 0$.
$(3, 5, \underline{2}, 1, 90, 6)$: vetor ordenado de $0 - 1$.
$(2, 3, 5, \underline{1}, 90, 6)$: vetor ordenado de $0 - 2$.
$(1, 2, 3, 5, \underline{90}, 6)$: vetor ordenado de $0 - 3$.
$(1, 2, 3, 5, 90, \underline{6})$: vetor ordenado de $0 - 4$.
$(1, 2, 3, 5, 6, 90)$: vetor ordenado de $0 - 5$.

Insertion Sort

Exemplo: (5,3,2,1,90,6).
O valor sublinhado representa onde está o índice i

$(5, \underline{3}, 2, 1, 90, 6)$: vetor ordenado de $0 - 0$.
$(3, 5, \underline{2}, 1, 90, 6)$: vetor ordenado de $0 - 1$.
$(2, 3, 5, \underline{1}, 90, 6)$: vetor ordenado de $0 - 2$.
$(1, 2, 3, 5, \underline{90}, 6)$: vetor ordenado de $0 - 3$.
$(1, 2, 3, 5, 90, \underline{6})$: vetor ordenado de $0 - 4$.
$(1, 2, 3, 5, 6, 90)$: vetor ordenado de $0 - 5$.

Insertion Sort

Exemplo: (5,3,2,1,90,6).
O valor sublinhado representa onde está o índice i

$(5, \underline{3}, 2, 1, 90, 6)$: vetor ordenado de $0 - 0$.
$(3, 5, \underline{2}, 1, 90, 6)$: vetor ordenado de $0 - 1$.
$(2, 3, 5, \underline{1}, 90, 6)$: vetor ordenado de $0 - 2$.
$(1, 2, 3, 5, \underline{90}, 6)$: vetor ordenado de $0 - 3$.
$(1, 2, 3, 5, 90, \underline{6})$: vetor ordenado de $0 - 4$.
$(1, 2, 3, 5, 6, 90)$: vetor ordenado de $0 - 5$.

Insertion Sort

- Usaremos dois métodos de listas: **insert** e **pop**.
- O método **pop** remove (e devolve) o elemento de uma posição específica na lista.

```
>>> l = [10, 20, 30, 40, 50]
>>> aux = l.pop(2)
>>> l
[10, 20, 40, 50]
>>> aux
30
```

- O método **insert** insere em uma posição específica na lista um determinado objeto.

```
>>> l
[10, 20, 40, 50]
>>> l.insert(2,25)
>>> l
[10, 20, 25, 40, 50]
```

Insertion Sort

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

```
aux = v.pop(i) #Colocar elemento v[i] na pos. correta
j = i - 1
while j>=0 and v[j] > aux:
    j = j - 1
```

- Quando o laço termina a execução??

Insertion Sort

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

```
aux = v.pop(i) #Colocar elemento v[i] na pos. correta
j = i - 1
while j>=0 and v[j] > aux:
    j = j - 1
```

- Quando o laço termina a execução??

Insertion Sort

- Quando o laço termina a execução??
- Ou **j == -1** Ou **v[j]** <= **aux**.
- Logo a posição correta para **aux** será **j+1**.

```
aux = v.pop(i) #Colocar elemento v[i] na pos. correta
j = i - 1
while j>=0 and v[j] > aux:
    j = j - 1
#Quando terminar o laço:
#Ou j=-1 ou v[j]<= aux
v.insert(j+1, aux)
```

Insertion Sort

Exemplo $[1, 3, 5, 10, 20, 2^*, 4]$ com $i = 5$; $aux = 2$.

$[1, 3, 5, 10, \underline{20}, 4]$: $j = 4$; e $v[j] > aux$

$(1, 3, 5, \underline{10}, 20, 4)$: $j = 3$; e $v[j] > aux$

$(1, 3, \underline{5}, 10, 20, 4)$: $j = 2$; e $v[j] > aux$

$(1, \underline{3}, 5, 10, 20, 4)$: $j = 1$; e $v[j] > aux$

$(\underline{1}, 3, 5, 10, 20, 4)$: $j = 0$;

Aqui temos que $v[j] <= aux$ logo fazemos $v.insert(j + 1, aux)$

$[1, 2, 3, 5, 10, 20, 4]$

Insertion Sort

Exemplo $[1, 3, 5, 10, 20, 2^*, 4]$ com $i = 5$; $aux = 2$.
$[1, 3, 5, 10, \underline{20}, 4]$: $j = 4$; e $v[j] > aux$
$(1, 3, 5, \underline{10}, 20, 4)$: $j = 3$; e $v[j] > aux$
$(1, 3, \underline{5}, 10, 20, 4)$: $j = 2$; e $v[j] > aux$
$(1, \underline{3}, 5, 10, 20, 4)$: $j = 1$; e $v[j] > aux$
$(\underline{1}, 3, 5, 10, 20, 4)$: $j = 0$;
Aqui temos que $v[j] <= aux$ logo fazemos $v.insert(j + 1, aux)$
$[1, 2, 3, 5, 10, 20, 4]$

Insertion Sort

Exemplo $[1, 3, 5, 10, 20, 2^*, 4]$ com $i = 5$; $aux = 2$.
$[1, 3, 5, 10, \underline{20}, 4]$: $j = 4$; e $v[j] > aux$
$(1, 3, 5, \underline{10}, 20, 4)$: $j = 3$; e $v[j] > aux$
$(1, 3, \underline{5}, 10, 20, 4)$: $j = 2$; e $v[j] > aux$
$(1, \underline{3}, 5, 10, 20, 4)$: $j = 1$; e $v[j] > aux$
$(\underline{1}, 3, 5, 10, 20, 4)$: $j = 0$;
Aqui temos que $v[j] <= aux$ logo fazemos $v.insert(j + 1, aux)$
$[1, 2, 3, 5, 10, 20, 4]$

Insertion Sort

Exemplo $[1, 3, 5, 10, 20, 2^*, 4]$ com $i = 5$; $aux = 2$.
$[1, 3, 5, 10, \underline{20}, 4]$: $j = 4$; e $v[j] > aux$
$(1, 3, 5, \underline{10}, 20, 4)$: $j = 3$; e $v[j] > aux$
$(1, 3, \underline{5}, 10, 20, 4)$: $j = 2$; e $v[j] > aux$
$(1, \underline{3}, 5, 10, 20, 4)$: $j = 1$; e $v[j] > aux$
$(\underline{1}, 3, 5, 10, 20, 4)$: $j = 0$;
Aqui temos que $v[j] <= aux$ logo fazemos $v.insert(j + 1, aux)$
$[1, 2, 3, 5, 10, 20, 4]$

Insertion Sort

Exemplo $[1, 3, 5, 10, 20, 2^*, 4]$ com $i = 5$; $aux = 2$.
$[1, 3, 5, 10, \underline{20}, 4]$: $j = 4$; e $v[j] > aux$
$(1, 3, 5, \underline{10}, 20, 4)$: $j = 3$; e $v[j] > aux$
$(1, 3, \underline{5}, 10, 20, 4)$: $j = 2$; e $v[j] > aux$
$(1, \underline{3}, 5, 10, 20, 4)$: $j = 1$; e $v[j] > aux$
$(\underline{1}, 3, 5, 10, 20, 4)$: $j = 0$;
Aqui temos que $v[j] <= aux$ logo fazemos $v.insert(j + 1, aux)$
$[1, 2, 3, 5, 10, 20, 4]$

Insertion Sort

Exemplo $[1, 3, 5, 10, 20, 2^*, 4]$ com $i = 5$; $aux = 2$.
$[1, 3, 5, 10, \underline{20}, 4]$: $j = 4$; e $v[j] > aux$
$(1, 3, 5, \underline{10}, 20, 4)$: $j = 3$; e $v[j] > aux$
$(1, 3, \underline{5}, 10, 20, 4)$: $j = 2$; e $v[j] > aux$
$(1, \underline{3}, 5, 10, 20, 4)$: $j = 1$; e $v[j] > aux$
$(\underline{1}, 3, 5, 10, 20, 4)$: $j = 0$;
Aqui temos que $v[j] <= aux$ logo fazemos $v.insert(j + 1, aux)$
$[1, 2, 3, 5, 10, 20, 4]$

Insertion Sort

Exemplo $[1, 3, 5, 10, 20, 2^*, 4]$ com $i = 5$; $aux = 2$.
$[1, 3, 5, 10, \underline{20}, 4]$: $j = 4$; e $v[j] > aux$
$(1, 3, 5, \underline{10}, 20, 4)$: $j = 3$; e $v[j] > aux$
$(1, 3, \underline{5}, 10, 20, 4)$: $j = 2$; e $v[j] > aux$
$(1, \underline{3}, 5, 10, 20, 4)$: $j = 1$; e $v[j] > aux$
$(\underline{1}, 3, 5, 10, 20, 4)$: $j = 0$;
Aqui temos que $v[j] <= aux$ logo fazemos $v.insert(j + 1, aux)$
$[1, 2, 3, 5, 10, 20, 4]$

Insertion Sort

Código completo:

```
def insertionSort(v):
    for i in range(1, len(v)):
        aux = v.pop(i) #Colocar elemento v[i] na pos. correta

        j = i - 1
        while j>=0 and v[j] > aux:
            j = j - 1
        #Quando terminar o laço:
        #Ou j=-1 ou v[j]<= aux
        v.insert(j+1, aux)
```

Insertion Sort

Código completo:

```
def insertionSort(v):
    for i in range(1, len(v)):
        aux = v.pop(i) #Colocar elemento v[i] na pos. correta

        j = i - 1
        while j>=0 and v[j] > aux:
            j = j - 1
        #Quando terminar o laço:
        #Ou j=-1 ou v[j]<= aux
        v.insert(j+1, aux)
```

Insertion Sort

Código completo.

```
import random

def insertionSort(v):
    for i in range(1, len(v)):
        aux = v.pop(i) #Colocar elemento v[i] na pos. correta
        j = i - 1
        while j>=0 and v[j] > aux:
            j = j - 1
        #Quando terminar o laço:
        #Ou j=-1 ou v[j]<= aux
        v.insert(j+1, aux)

def main():
    v = [ random.randint(0, 100) for i in range(10)]
    print(v)
    insertionSort(v)
    print(v)

main()
```

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 tuplas como a coleção.
  - Cada tupla tem dois campos: um representando o RG de uma pessoa e outro o nome da pessoa.
- Os algoritmos servem para buscar elementos em qualquer coleção de elementos que possuam chaves que possam ser comparadas, como registros com algum campo de identificação único (RA, ou RG, ou CPF, etc.).

O Problema da Busca

- Nos nossos exemplos vamos criar a função:
  - **def busca(v, key):** que recebe uma lista, e uma chave para busca.
  - A função deve retornar o índice da lista que contém a chave ou **None** caso a chave não esteja na lista.

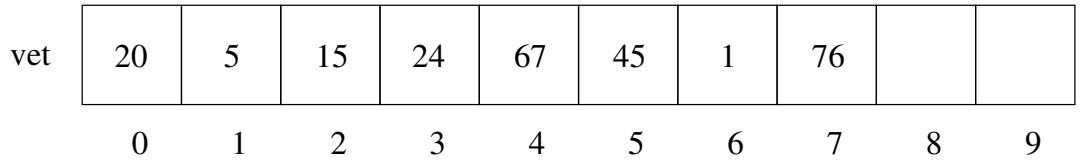
O Problema da Busca

- Lembre-se que a lista **v** no nosso exemplo armazena tuplas onde o primeiro campo de cada tupla é a chave (RG):

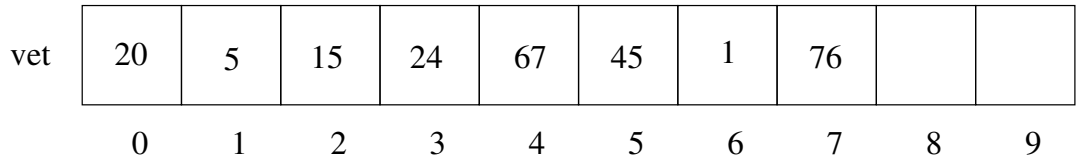
```
>l = [(102, 'Ana'), (103, 'Beto'), (500, 'Joao'), (202, 'Julia')]
>l[0]
(102, 'Ana')
>l[2][0]
500
```

O Problema da Busca

chave = 45 tam = 8



chave = 100 tam = 8



No exemplo mais acima, a função deve retornar 5, enquanto no exemplo mais abaixo a função deve retornar **None**.

Busca Sequencial

- A busca sequencial é o algoritmo mais simples de busca:
  - Percorra toda a lista comparando a chave com o valor de chave em cada posição.
  - Se for igual para alguma posição, então devolva esta posição.
  - Se a lista toda foi percorrida então devolva **None**.

Busca Sequencial

```
def buscaSequencial(v, key):
    for i in range(len(v)):
        if v[i][0] == key:
            return i
    return None
```

Busca Sequencial

Para testar a busca, geramos uma lista de chaves aleatórias:

```
def gera_chaves(n):
    '''Gera uma lista com n números aleatórios únicos'''
    v = []
    i = 0
    while i<n:
        k = random.randint(0,100*n)
        if k not in v:
            v.append(k)
            i = i + 1
    return v
```

Tal lista pode ser usada para construir uma lista **v** de tuplas:

```
k = gera_chaves(10)
v = [(k[i],i) for i in range(len(k))]
```

Busca Sequencial

```
import random

def main2():
    k = gera_chaves(10)
    v = [(k[i],i) for i in range(len(k))]
    print(v)

    ale = random.randint(0,len(v)-1)
    i = busca_sequencial(v, v[ale][0])
    print(i == ale, i, ale)

def busca_sequencial(v, key):
    for i in range(len(v)):
        if v[i][0] == key:
            return i
    return None

main2()
```

Busca Binária

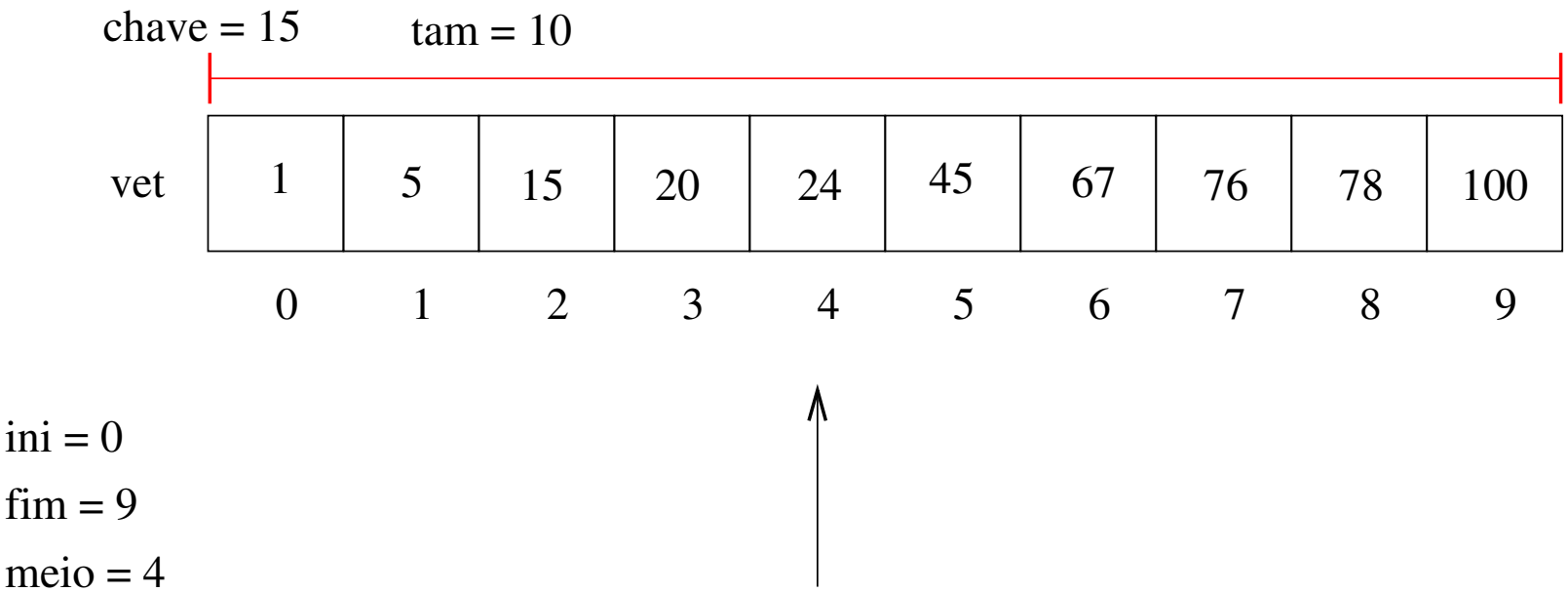
- A busca binária é um algoritmo um pouco mais sofisticado.
- É mais eficiente, mas requer que o vetor esteja ordenado pelos valores da chave de busca.
- A idéia do algoritmo é a seguinte (assuma que o vetor está ordenado):
  - Verifique se a chave de busca é igual a chave na posição do meio do vetor.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

Busca Binária

Pseudo-Código:

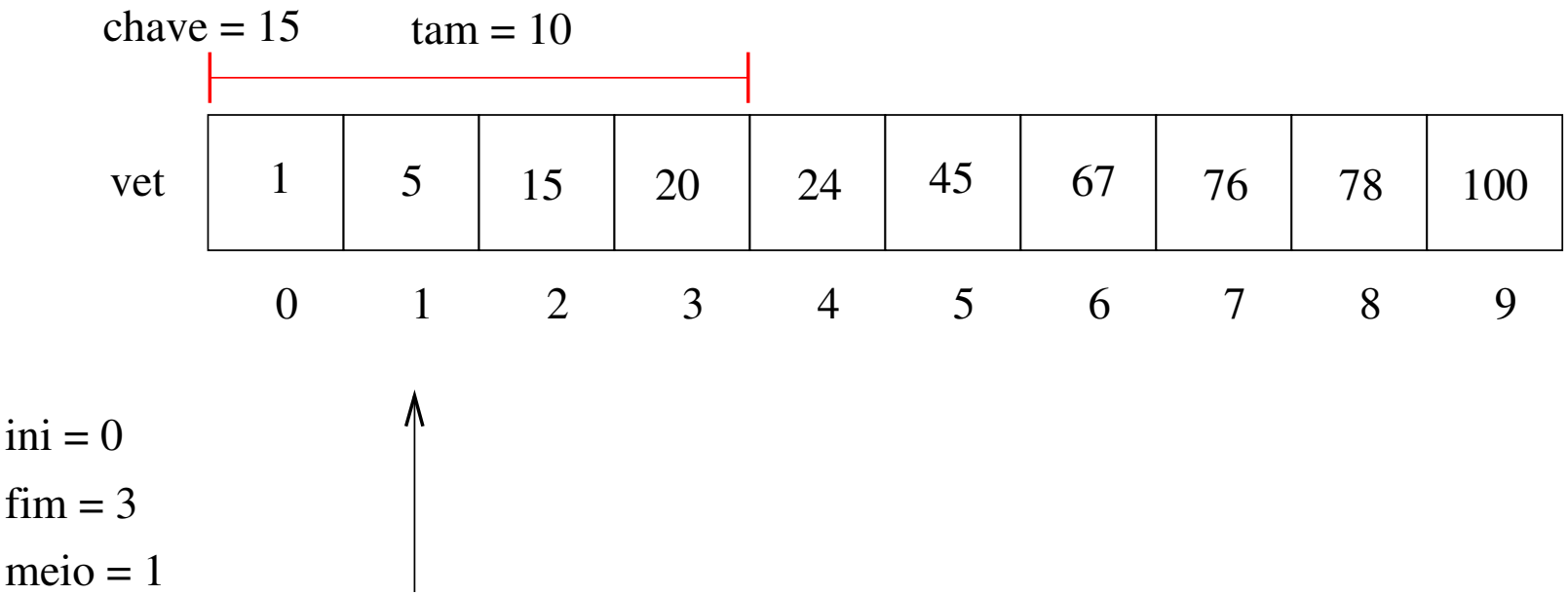
```
#vetor considerando começo em ini e final em fim
ini = 0
fim = tam-1

Repita enquanto tamanho do vetor considerado for >= 1
        meio = (ini + fim)//2

        Se v[meio] == chave Então
                devolva meio

        Se v[meio] > chave Então
                fim = meio - 1

        Se v[meio] < chave Então
                ini = meio + 1
```

Busca Binária



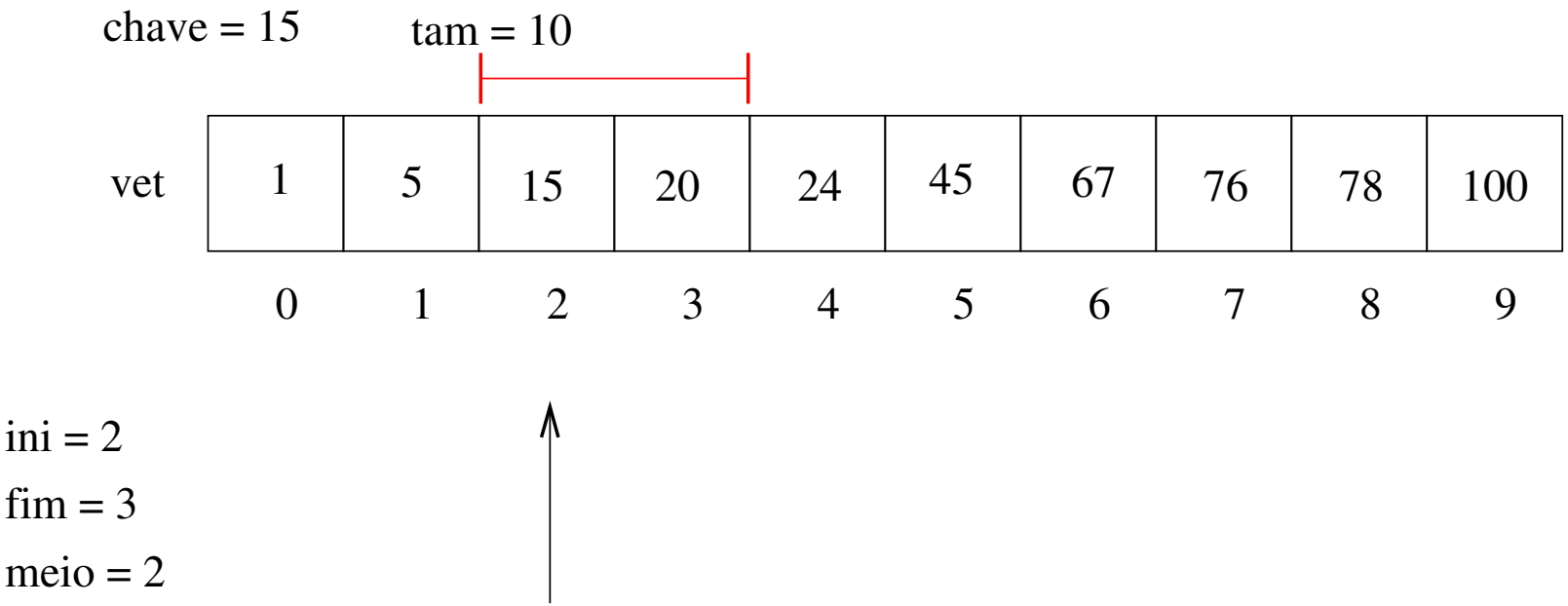
Como o valor da posição do meio é maior que a chave, atualizamos **fim** do vetor considerado.

Busca Binária



Como o valor da posição do meio é menor que a chave, atualizamos **ini** do vetor considerado.

Busca Binária



Finalmente encontramos a chave e podemos devolver sua posição 2.

Busca Binária

Código completo:

```
def busca_binaria(v, key):
    ini = 0
    fim = len(v)-1
    while ini <= fim:
        #enquanto o vetor tiver pelo menos 1 elemento
        #continua com busca
        meio = (ini + fim)//2
        if v[meio][0] == key:
            return meio
        elif v[meio][0] > key:
            fim = meio - 1
        else:
            ini = meio + 1
    return None
```

Busca Binária

Exemplo de uso:

```
import random

def main():
    k = gera_chaves(10)
    v = [(k[i],i) for i in range(len(k))]
    print(v)
    insertionSort(v, 0)
    print(v)

    ale = random.randint(0,len(v)-1)
    i = busca_binaria(v, v[ale][0])
    print(i == ale) #Deve imprimir True já que procuramos pela chave que está em ale

    print(busca_binaria(v, -100))#Deve imprimir None, chaves >= 0


def buscaBinaria(v, key):
    ini = 0
    fim = len(v)-1
    while ini <= fim:
        #enquanto o vetor tiver pelo menos 1 elemento
        #continua com busca
        meio = (ini + fim)//2
        if v[meio] == key:
            return meio
        elif v[meio] > key:
            fim = meio - 1
        else:
            ini = meio + 1
    return None

main()
```

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.

- É comum 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 do vetor.

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 **v[0]**.
- Na pior das hipóteses, a chave é o último elemento ou não pertence ao vetor, e portanto acessaremos todas as **n** posições do vetor.
- É possível mostrar que se uma chave qualquer pode ser requisitada com a mesma probabilidade, então o número de acessos será

$$(n+1)/2$$

na média, onde n é o tamanho do vetor.

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 $(\log_2 \mathbf{n})$ acessos.
  - Para ver isso note que a cada verificação de uma posição do vetor, o tamanho do vetor considerado é dividido pela metade. No pior caso repetimos a busca até o vetor considerado ter tamanho 1. Se você pensar um pouco, 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 uma chave qualquer pode ser requisitada com a mesma probabilidade, então o número de acessos será

  $$(\log_2 n) - 1$$

  na média.

Eficiência dos Algoritmos

Para se ter uma idéia da diferença de eficiência dos dois algoritmos, 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, o vetor precisa estar ordenado!

- Se você tiver um cadastro onde vários itens são removidos e inseridos com frequência, e a busca deve ser feita intercalada com estas operações, então a busca binária pode não ser a melhor opção, já que você precisará ficar mantendo o vetor ordenado.
- 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 um vetor de inteiros em ordem decrescente.

Exercícios

- Refaça as funções de busca sequencial e busca binária assumindo que o vetor possui chaves que podem aparecer repetidas. Neste caso, você deve retornar em uma lista todas as posições onde a chave foi encontrada.
  Protótipo: **def busca(v, key)**