



Algoritmos e Programação de Computadores

Revisão: Prova 2

Profa. Sandra Avila

Instituto de Computação (IC/Unicamp)

MC102, 15 Junho, 2018

Conteúdo da Prova 2

- Prova 1: Variáveis, Tipos, Comandos condicionais, Comandos repetitivos, Listas
- Tuplas
- Dicionários
- Funções
- Matrizes e listas multidimensionais
- Algoritmos de busca: Sequencial e Binária
- Recursão
- Algoritmos de ordenação: Bubble, Insertion, Selection, Quick e Merge
- Noções básicas: Expressões regulares & Arquivos

Revisão do Conteúdo

Tuplas

- Tuplas são uma sequência de elementos separados por vírgulas, representados ou não entre parênteses.
- **Tuplas são imutáveis.**
- `(18, "abril", 9.5, 1)` é uma tupla de 4 elementos.



Tuplas

- Mais exemplos de tuplas.

```
tupla1 = ('abril', 18, 4, 2018)
tupla2 = (1, 2, 3, 4, 5, 6, 7)
tupla3 = "a", "b", "c", "d"
tupla4 = ("MC102", )
tupla5 = ()
```

tupla4 representa uma tupla com um único elemento. A vírgula após o elemento é necessária para diferenciar de uma expressão entre parênteses.

Tuplas

- Como strings, tuplas são **imutáveis**.

```
a = (18, "abril", 9.5, 1)
```

```
a[2] = 9.0
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

Tuplas: Empacotamento e Desempacotamento

- Os elementos de uma tupla podem ser acessados de uma forma implícita na atribuição (conhecido como desempacotamento).

```
x, y = (18, 20)
```

```
x
```

```
18
```

```
y
```

```
20
```

Tuplas: Empacotamento e Desempacotamento

- A tupla também pode ser implicitamente criada apenas separando os elementos por vírgula (conhecido como empacotamento).

```
18, 20  
(18, 20)
```

```
"abril", 9.5  
( 'abril', 9.5)
```


Dicionários


- Dicionários são estruturas de dados que associam uma chave com um valor.
- **As chaves só podem ser dados de tipos imutáveis.**
- ```
ra = {"Liz": 229874,
 "Hugo": 215793,
 "Sofia": 199745}
```



# Dicionários

- O valor associado a uma chave pode ser modificado, ou uma nova chave (e seu valor) podem ser incluídos no dicionário.

```
ra = {"Liz": 229874, "Hugo": 215793, "Sofia": 199745}
ra
{'Hugo': 215793, 'Liz': 229874, 'Sofia': 199745}
```



**Um dicionário é uma coleção não ordenada de pares chave-valor.**

# Operações em Dicionários

- O laço **for** aplicado a um dicionário faz a variável do laço passar por todas as **chaves** do dicionário.

```
ra = {"Liz": 229874, "Hugo": 215793, "Sofia": 199745}
```

```
for x in ra:
 print(x)
```

```
Liz
```

```
Hugo
```

```
Sofia
```

# Métodos em Dicionários

- `items()` retorna todos os pares chave/conteúdo do dicionário.
- `keys()` retorna todas as chaves do dicionário.
- `values()` retorna todos os valores do dicionário.

# Métodos em Dicionários

- `items()` retorna todos os pares chave/conteúdo do dicionário.
- `keys()` retorna todas as chaves do dicionário.
- `values()` retorna todos os valores do dicionário.

```
ra = {"Liz": 229874, "Hugo": 215793, "Sofia": 199745}
ra.items()
dict_items([('Liz', 229874), ('Hugo', 215793), ('Sofia',
199745)])
ra.keys()
dict_keys(['Liz', 'Hugo', 'Sofia'])
ra.values()
dict_values([229874, 215793, 199745])
```

# Iterando em Dicionários

- Ao fazer uma iteração sobre dicionários, a chave e o valor correspondente podem ser recuperados ao mesmo tempo usando o método `items()`:

```
ra = {"Liz": 229874, "Hugo": 215793, "Sofia": 199745}
```

```
for nome, numero in ra.items():
 print(nome, numero, sep=' ')
```

```
Liz 229874
```

```
Hugo 215793
```

```
Sofia 199745
```

# Funções

- Funções são estruturas que **agrupam um conjunto de comandos**, que são executados quando a função é chamada.

```
def quadrado (x) :
 return x * x
```



# Por que definir uma função?

- Evitar que os blocos do programa fiquem grandes demais e, por consequência, mais difíceis de ler e entender.
- Permitir o reaproveitamento de código já construído (por você ou por outros programadores).
- Evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa, minimizando erros e facilitando alterações.



# Definindo Funções

- Uma função é definida da seguinte forma:

```
def nome (parâmetro1, parâmetro2, ..., parâmetroN):
 comandos
 return valor do retorno
```

- Os **parâmetros** são variáveis, que são inicializadas com valores indicados durante a chamada/invocação da função.
- O comando **return** devolve para o invocador da função o resultado da execução desta.

# Exemplo de uma função

- A lista de parâmetros de uma função pode ser vazia.



```
def leNumeroInt():
 numero = input("Digite um número inteiro: ")
 return int(numero)
```

```
r = leNumeroInt()
print("Número digitado:", r)
```

# Definindo funções depois do seu uso

- O programa será organizado da seguinte forma:

```
def main () :
 comandos

def função1 (parâmetros) :
 comandos

def função2 (parâmetros) :
 comandos
...

main ()
```

# Variáveis Locais e Variáveis Globais

- Uma variável é chamada **local** se ela é criada ou alterada **dentro de uma função**.
- Nesse caso, ela existe somente dentro daquela função, e após o término da execução da mesma a variável deixa de existir.
- Variáveis parâmetros também são variáveis locais.

# Variáveis Locais e Variáveis Globais

- Uma variável é chamada **global** se ela for criada **fora de qualquer função**.
- Essa variável pode ser visível por todas as funções.
- Qualquer função pode alterá-la.

# Organização de um Programa

```
variáveis globais
```

```
def main():
 variáveis locais
 comandos
```

```
def função1(parâmetros):
 variáveis locais
 comandos
```

```
def função2(parâmetros):
 variáveis locais
 comandos
```

```
...
```

```
main()
```

# Escopo de Variáveis

- O **escopo** de uma variável determina de quais partes do código ela pode ser acessada, ou seja, de quais partes do código a variável é visível.
- A regra de escopo em Python é bem simples:
  - As variáveis **globais** são **visíveis por todas as funções**.
  - As variáveis **locais** são **visíveis apenas na função onde foram criadas**.

# Variáveis Locais e Variáveis Globais

```
def f1(a):
 x = 10
 print(a+x)

def f2(a):
 c = 10
 print(a+x+c)
```

```
x = 4
f1(3)
f2(3)
print(x)
```

```
13
17
4
```

Neste outro exemplo **f1** cria uma variável local `x` com valor 10.

O valor de `x` global permanece com 4.



# Matrizes e listas multidimensionais

- Em Python, uma matriz pode ser representada como uma **lista de listas**, onde um elemento da lista contém uma linha da matriz, que por sua vez corresponde a uma lista com os elementos da coluna da matriz.



# Declarando uma Matriz com Listas

- Para criar uma matriz de dimensões  $l \times c$  inicialmente vazia podemos utilizar listas.
- Exemplo de uma matriz  $3 \times 4$  inicialmente vazia:

```
mat = [[] for i in range(3)]
#dentro da lista externa cria-se vazia 3 listas []
mat
[[], [], []]
```

- Note que cada lista interna representa uma linha da matriz, e seu tamanho pode ser 4 ou qualquer outro valor.

# Exemplo de Declaração de Matriz

- Criar matriz 3 x 4 onde cada posição  $(i, j)$  contém o valor de  $i * j$ .

```
mat = []
for i in range(3): # para cada linha de 0 ate 2
 l = [] # linha começa vazia
 for j in range(4): # para cada coluna de 0 ate 3
 l.append(i*j) # preenche colunas da linha i
 mat.append(l) # adiciona linha na matriz
print(mat)
```

```
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6]]
```

# Exemplo de Declaração de Matriz

- Obtendo o mesmo resultado utilizando **compreensão de listas**:

```
mat = [[i*j for j in range(4)] for i in range(3)]
print(mat)
```

```
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6]]
```

# Acessando os Dados da Matriz

```
nome_da_matriz[linha][coluna]
```

- Ex: `matriz[1][10]`: refere-se a variável na 2ª linha e na 11ª coluna da matriz.
- Lembre-se que, como a matriz está implementada com listas, a primeira posição em uma determinada dimensão começa no índice 0.
- O acesso a posições inválidas causa um erro de execução.

# Algoritmos de Busca

- Dada uma coleção de elementos, queremos **encontrar o elemento da coleção que possui a mesma chave** ou identificar que não existe nenhum elemento com a chave dada.
- Busca Sequencial & Binária



# Busca

- Nos nossos exemplos vamos criar a função:
  - **busca(lista, 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.

# 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  $-1$ .



# Busca Sequencial

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

```
lista = [20, 5, 15, 24, 67, 45, 1, 76]
buscaSequencial(lista, 24)
buscaSequencial(lista, 100)
```

```
3
-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.

# Busca Binária

- A ideia 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

```
def buscaBinaria(lista, chave):
 inicio = 0
 fim = len(lista)-1
 while inicio <= fim:
 meio = (inicio + fim)//2
 if lista[meio] == chave:
 return meio
 elif lista[meio] > chave:
 fim = meio - 1
 else:
 inicio = meio + 1
 return -1
```

# Algoritmos de Ordenação

- Dado uma coleção de elementos com **uma relação de ordem entre si**, devemos gerar uma saída com os elementos ordenados.
- **Selection, Insertion, Bubble, Quick & Merge Sort**



# Selection Sort

## (Ordenação por Seleção)

# Selection Sort (Ordenação por Seleção)

- A ideia do algoritmo é a seguinte:
  - Ache o menor elemento a partir da posição 0. Troque então este elemento com o elemento da posição 0.
  - Ache o menor elemento a partir da posição 1. Troque então este elemento com o elemento da posição 1.
  - Ache o menor elemento a partir da posição 2. Troque então este elemento com o elemento da posição 2.
  - E assim sucessivamente...

# Selection Sort (Ordenação por Seleção)

- Passo a passo para [9, 6, 3, 5, 1, 2, 0, 4, 7, 8]:

[0, 6, 3, 5, 1, 2, 9, 4, 7, 8]

[0, 1, 3, 5, 6, 2, 9, 4, 7, 8]

[0, 1, 2, 5, 6, 3, 9, 4, 7, 8]

[0, 1, 2, 3, 6, 5, 9, 4, 7, 8]

[0, 1, 2, 3, 4, 5, 9, 6, 7, 8]

[0, 1, 2, 3, 4, 5, 6, 9, 7, 8]

[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]



# Selection Sort (Ordenação por Seleção)

```
def selectionSort(vet):
 for i in range(len(vet)-1):
 #Acha o menor elemento a partir da posição i
 menor = i
 for j in range(i, len(vet)):
 if vet[menor] > vet[j]:
 menor = j
 #Troca com o elemento da posição i
 vet[i], vet[menor] = vet[menor], vet[i]
```

# Bubble Sort

## (Ordenação por Bolha)

# Bubble Sort (Ordenação por Bolha)

- A ideia do algoritmo é a seguinte:
  - Compare `vet[0]` com `vet[1]` e troque-os se `vet[0] > vet[1]`.
  - Compare `vet[1]` com `vet[2]` e troque-os se `vet[1] > vet[2]`.
  - Compare `vet[2]` com `vet[3]` e troque-os se `vet[2] > vet[3]`.
  - ...
  - Compare `vet[tam-2]` com `vet[tam-1]` e troque-os se `vet[tam-2] > vet[tam-1]`.
  - E assim sucessivamente ...

# Bubble Sort (Ordenação por Bolha)

- Após uma iteração repetindo estes passos o que podemos garantir?
  - O maior elemento estará na posição correta!
- Após outra iteração de trocas, o segundo maior elemento estará na posição correta.
- E assim sucessivamente.
- Quantas iterações destas trocas precisamos para deixar a lista ordenada?

# Bubble Sort (Ordenação por Bolha)

- Exemplo: [5, 3, 2, 1, 90, 6].
  - Iteração 1. [5, 3, 2, 1, 90, 6] Faz troca: [3, 5, 2, 1, 90, 6]  
[3, 5, 2, 1, 90, 6] Faz troca: [3, 2, 5, 1, 90, 6]  
[3, 2, 5, 1, 90, 6] Faz troca: [3, 2, 1, 5, 90, 6]  
[3, 2, 1, 5, 90, 6]  
[3, 2, 1, 5, 90, 6] Faz troca: [3, 2, 1, 5, 6, 90]
- Isto termina a primeira iteração de trocas. Temos que repetir todo o processo mais 4 vezes!

# Bubble Sort (Ordenação por Bolha)

- Passo a passo para [9, 6, 3, 5, 1, 2, 0, 4, 7, 8]:

[9, 6, 3, 5, 1, 2, 0, 4, 7, 8]

[6, 9, 3, 5, 1, 2, 0, 4, 7, 8]

[6, 3, 9, 5, 1, 2, 0, 4, 7, 8]

[6, 3, 5, 9, 1, 2, 0, 4, 7, 8]

[6, 3, 5, 1, 9, 2, 0, 4, 7, 8]

[6, 3, 5, 1, 2, 9, 0, 4, 7, 8]

[6, 3, 5, 1, 2, 0, 9, 4, 7, 8]

[6, 3, 5, 1, 2, 0, 4, 9, 7, 8]

[6, 3, 5, 1, 2, 0, 4, 7, 9, 8]

...

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Bubble Sort (Ordenação por Bolha)

```
def bubbleSort(vet):
 #Índices i em ordem decrescente
 for i in range(len(vet)-1,0,-1):
 #Troca com o elemento da posição i
 for j in range(i):
 if vet[j] > vet[j+1]:
 vet[j], vet[j+1] = vet[j+1], vet[j]
```

# Insertion Sort

## (Ordenação por Inserção)



# Insertion Sort (Ordenação por Inserção)

- A ideia 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 (Ordenação por Inserção)

- Exemplo: [5, 3, 2, 1, 90, 6].
  - [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 ordenada de 0-5.

# Insertion Sort (Ordenação por Inserção)

```
def insertionSort(vet):
 for i in range(1, len(vet)):
 aux = vet[i]
 j = i - 1
 while (j >= 0 and vet[j] > aux): #põe elementos vet[j] > vet[i]
 vet[j + 1] = vet[j] #para frente
 j = j - 1
 vet[j + 1] = aux #põe v[i] na posição correta
```

# Insertion Sort (Ordenação por Inserção)

- Passo a passo para [9, 6, 3, 5, 1, 2, 0, 4, 7, 8]:

[9, 6, 3, 5, 1, 2, 0, 4, 7, 8]

[6, 9, 3, 5, 1, 2, 0, 4, 7, 8]

[3, 6, 9, 5, 1, 2, 0, 4, 7, 8]

[3, 5, 6, 9, 1, 2, 0, 4, 7, 8]

[1, 3, 5, 6, 9, 2, 0, 4, 7, 8]

[1, 2, 3, 5, 6, 9, 0, 4, 7, 8]

[0, 1, 2, 3, 5, 6, 9, 4, 7, 8]

[0, 1, 2, 3, 4, 5, 6, 9, 7, 8]

[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Recursão

- Uma função é chamada de recursiva **se o corpo da função chama a própria função**, direta ou indiretamente.

```
def fatorial(n):
 if n == 1:
 return 1
 else:
 return n * fatorial(n-1)
```



# Soma de Números

```
def somaLista(numeros):
 if len(numeros) == 1:
 return numeros[0]
 else:
 return numeros[0] + somaLista(numeros[1:])
```

**Recursão**



```
def somaLista(numeros):
 soma = 0
 for i in numeros:
 soma = soma + i
 return soma
```

# Fatorial

```
def fatorial(n):
 if n == 1:
 return 1
 else:
 return n * fatorial(n-1)
```

**Recursão**



```
def fatorial (n):
 total = 1
 k = 1
 while k <= n:
 total = total * k
 k = k + 1
 return total
```

# Números de Fibonacci

```
def fibonacci(n):
 if n == 0:
 return 0
 elif n == 1:
 return 1
 else:
 return (fibonacci(n-1) + fibonacci(n-2))
```

**Recursão**





# Algoritmos de Ordenação

- Selection, Insertion, Bubble, **Quick** & **Merge** Sort.
- Vamos usar a técnica de **recursão** para resolver o problema de **ordenação**.



# Dividir e Conquistar

- Temos que resolver um problema  $P$  de tamanho  $n$ .
- **Dividir**: Quebramos  $P$  em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **Conquistar**: Unimos as soluções dos sub-problemas para obter solução do problema maior  $P$ .

# Quick Sort

# Quick Sort

- Vamos supor que devemos ordenar uma lista de uma posição `inicial` até `fim`.
- **Dividir:**
  - Escolha em elemento especial da lista chamado `pivô`.
  - Particione a lista em uma posição `pos` tal que todos elementos de `inicial` até `pos - 1` são menores ou iguais do que o `pivô`, e todos elementos de `pos` até `fim` são maiores ou iguais ao `pivô`.

# Quick Sort

- Resolvemos o problema de ordenação de forma recursiva para estas duas sub-listas (uma de `inicial` até `pos-1` e a outra de `pos` até `fim`).
- **Conquistar**: Nada a fazer já que a lista estará ordenada devido à fase de divisão.

# Quick Sort: Particionamento

Dado um valor  $p$  como  $pivô$ , como fazer o particionamento?

- Podemos “varrer” a lista do início para o fim até encontrarmos um elemento maior que o  $pivô$ .
- “Varremos” o vetor do fim para o início até encontrarmos um elemento menor ou igual ao  $pivô$ .
- Trocamos então estes elementos de posições e continuamos com o processo até termos verificado todas as posições do vetor.

# Quick Sort: Particionamento

A função retorna a posição de partição. Ela considera o último elemento como o pivô.

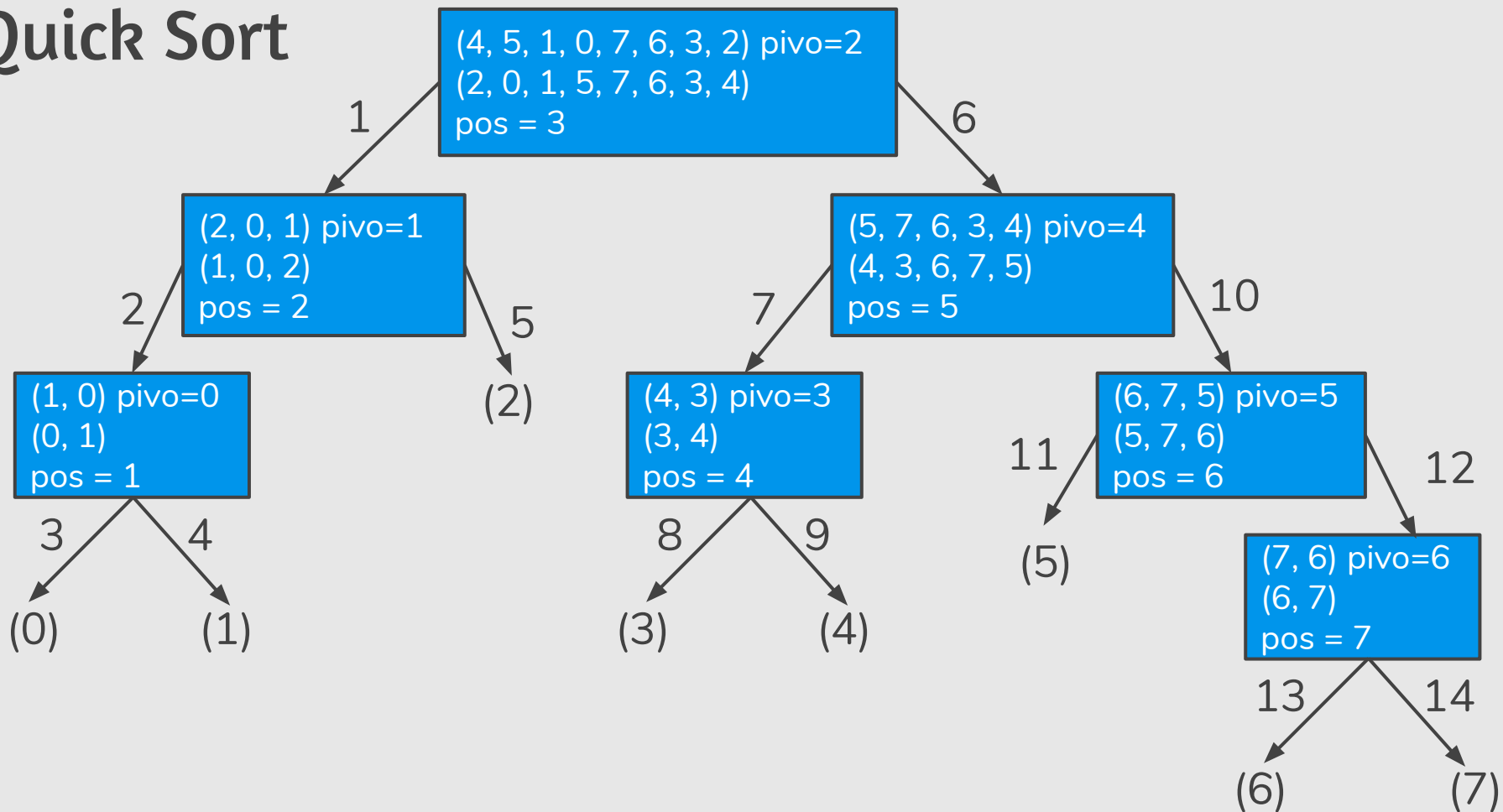
```
def particiona (v, inicio, fim):
 pivo = v[fim]
 while (inicio < fim):
 # o laço para quando inicio == fim => checamos o vetor inteiro
 while (inicio < fim) and (v[inicio] <= pivo):
 # acha posição de elemento maior que pivo
 inicio = inicio + 1
 while (inicio < fim) and (v[fim] > pivo) :
 # acha posição de elemento menor ou igual que pivo
 fim = fim - 1
 v[inicio], v[fim] = v[fim], v[inicio] # troca elementos de posição
 return inicio
```

# Quick Sort

```
def quickSort(v, inicio, fim):
 if (inicio < fim):
 # tem pelo menos 2 elementos a serem ordenados
 pos = particiona(v, inicio, fim)
 quickSort(v, inicio, pos-1)
 quickSort(v, pos, fim)
```



# Quick Sort



# Merge Sort

# Merge Sort: Ordenação por Intercalação

- O Merge Sort é um algoritmo baseado na técnica **dividir e conquistar**.
- Neste caso temos que ordenar uma lista de tamanho  $n$ .
  - **Dividir**: Dividimos a lista de tamanho  $n$  em duas sub-listas de tamanho aproximadamente iguais (de tamanho  $n/2$ ).
  - Resolvemos o problema de ordenação de forma recursiva para estas duas sub-listas.
  - **Conquistar**: Com as duas sub-listas ordenadas, construímos uma lista ordenada de tamanho  $n$  ordenado.

# Merge: Fusão

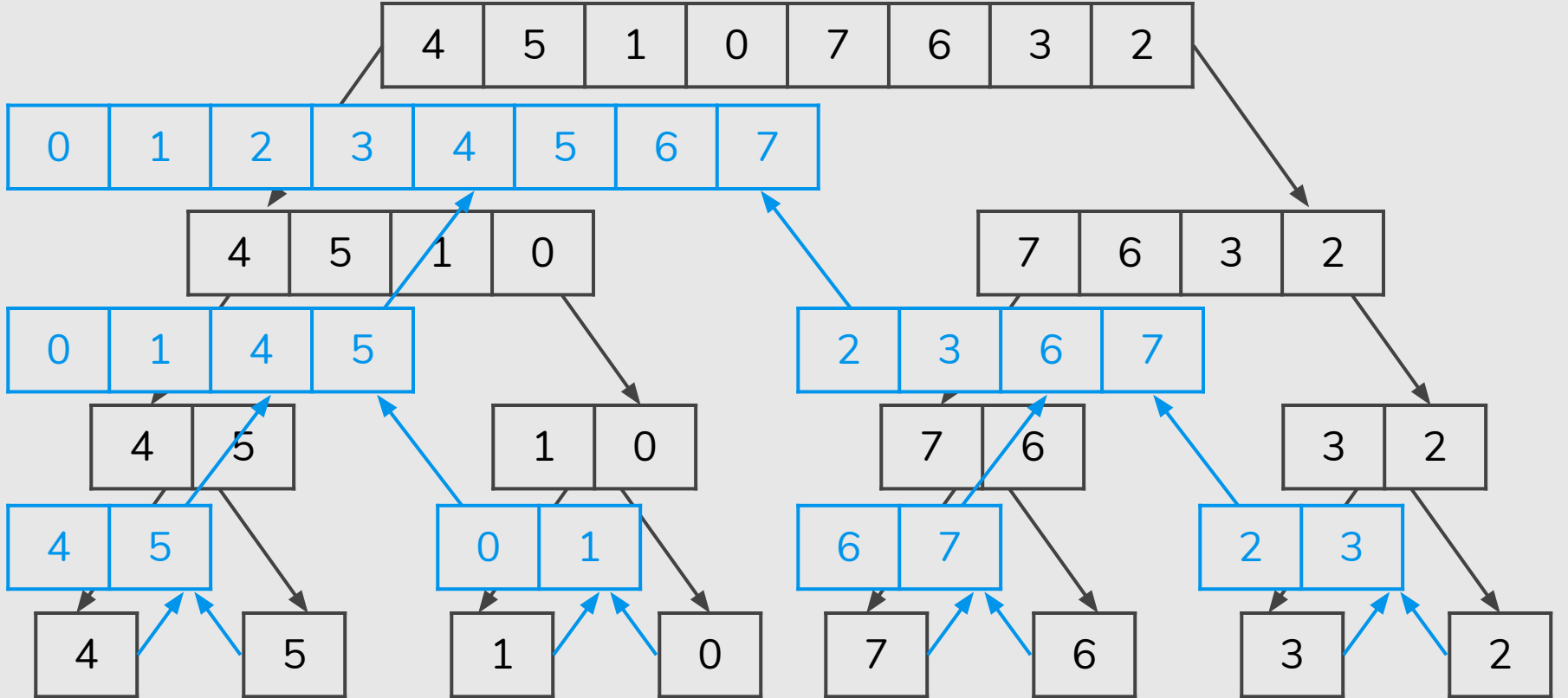
- A ideia é executar um laço que testa em cada iteração quem é o menor elemento dentre  $v1[i]$  e  $v2[j]$ , e copia este elemento para uma nova lista.
- Durante a execução deste laço podemos chegar em uma situação onde todos os elementos de uma das listas ( $v1$  ou  $v2$ ) foram todos avaliados. Neste caso terminamos o laço e copiamos os elementos restantes da outra lista.

```
def merge (v, inicio, meio, fim, aux):
 i = inicio; j = meio+1; k = 0; # indices da metade inf, sup e aux respc.
 while (i <= meio and j <= fim): # enquanto não avaliou completamente um dos
 if (v[i] <= v[j]): # vetores, copia menor elemento para aux
 aux[k] = v[i]
 k = k + 1
 i = i + 1
 else:
 aux[k] = v[j]
 k = k + 1
 j = j + 1
 while (i <= meio): # copia resto da primeira sub-lista
 aux[k] = v[i]
 k = k + 1
 i = i + 1
 while (j <= fim): # copia resto da segunda sub-lista
 aux[k] = v[j]
 k = k + 1
 j = j + 1
 i = inicio; k = 0;
 while (i <= fim): # copia lista ordenada aux para v
 v[i] = aux[k]
 i = i + 1
 k = k + 1
```

# Merge Sort

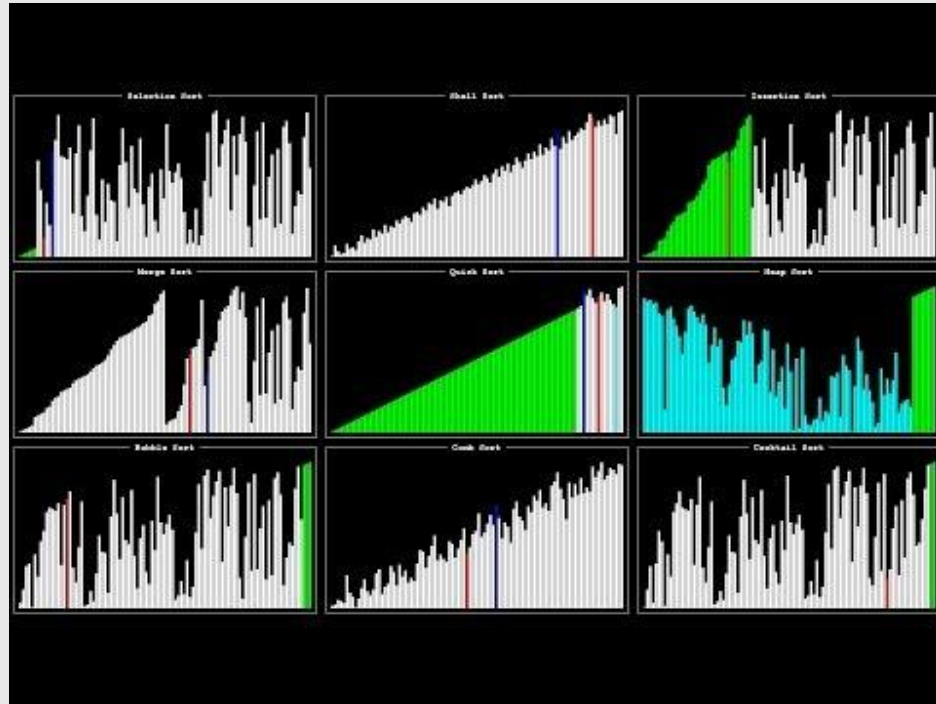
```
def mergeSort(v, inicio, fim, aux):
 meio = (fim + inicio) // 2
 if (inicio < fim): # lista tem pelo menos 2 elementos
 # para ordenar
 mergeSort(v, inicio, meio, aux)
 mergeSort(v, meio+1, fim, aux)
 merge(v, inicio, meio, fim, aux)
```

# Merge Sort



# Visualization and Comparison of Sorting Algorithms

<https://www.youtube.com/watch?v=ZZuD6iUe3Pc>





# Arquivos

- Arquivos Textos & Binários.

```
arquivo = open("tarefas.txt", "r")
```

```
arquivo = open("tarefas.bin",
"rb")
```

---

# Arquivos Textos

- O primeiro passo então é abrir o arquivo com o comando `open`:

```
variavel_arquivo = open("nome do arquivo", "modo")
```

- O `"nome do arquivo"` pode ser relativo ou absoluto.
- O `"modo"` pode ser `"r"` (leitura), `"r+"` (leitura e escrita), `"w"` (escrita), `"a"` (append).

# Abrindo um Arquivo Texto para Leitura

```
arquivo = open("tarefas.txt", "r")
```

- O primeiro parâmetro para `open` é uma string com o nome do arquivo
  - Pode ser absoluto, por exemplo: `"/home/sandra/tarefas.txt"`
  - Pode ser relativo como no exemplo acima: `"tarefas.txt"`
- O segundo parâmetro é uma string informando como o arquivo será aberto. Se para leitura ou gravação de dados, ou ambos.
  - No nosso exemplo o `"r"` significa que abrimos um arquivo texto para leitura.

# Abrindo um Arquivo Texto para Leitura

- Ao se trabalhar com arquivos é bom colocar a abertura do arquivo no bloco `try`, e o tratamento da exceção no bloco `except`.

```
try:
 arquivo = open("tarefas.txt", "r")
 print("Abri arquivo com sucesso.")
except:
 print("Não foi possível abrir o arquivo.")
```

# Lendo Dados de um Arquivo Texto

- O programa pode ser alterado para ler todo o arquivo de uma vez.

```
try:
 arquivo = open("tarefas.txt", "r")
 while True:
 s = arquivo.readline()
 print(s, end="")
 if (s == ""):
 break
 arquivo.close()
except:
 print("Não foi possível abrir o arquivo.")
```

# Resumindo ... Arquivo Texto

```
arquivo = open("nome do arquivo", "modo")
```

| <b>modo</b> | <b>operador</b>   | <b>indicador de posição</b> |
|-------------|-------------------|-----------------------------|
| r           | leitura           | início do arquivo           |
| r+          | leitura e escrita | início do arquivo           |
| w           | escrita           | início do arquivo           |
| a           | (append) escrita  | final do arquivo            |

# Arquivos Binários

- A motivação principal é que objetos (como inteiros, listas, dicionários) na sua representação em binário, **ocupam pouco espaço na memória**, quando comparado com sua representação em formato texto.

# Abrindo um Arquivo Binário

```
arquivo = open("nome do arquivo", "modo")
```

| <b>modo</b> | <b>operador</b>   |
|-------------|-------------------|
| rb          | leitura           |
| wb          | escrita           |
| r+b         | leitura e escrita |



# Lendo e Escrevendo no Arquivo Binário

- Utilizaremos o módulo `pickle` para ler e escrever objetos para um arquivo.
- Primeiramente deve-se importar este módulo:

```
import pickle
```

- Após isso podemos escrever um objeto em arquivo com o método `pickle.dump` e podemos ler um objeto em arquivo com o método `pickle.load`.

# Lendo e **Escrevendo** no Arquivo Binário

- Para **escrever um objeto em um arquivo binário** usamos o método `pickle.dump`.

```
pickle.dump(objeto, arquivo)
```

- `objeto`: este é o objeto a ser salvo em arquivo.
- `arquivo`: esta é a variável associada a um arquivo previamente aberto em modo binário.

# Lendo e **Escrevendo** no Arquivo Binário

- Exemplo: O programa abaixo salva uma lista em arquivo.

```
import pickle
try:
 arquivo = open("teste.bin", "wb")
 lista = [1, 2, 3]
 pickle.dump(lista, arquivo)
 arquivo.close()
except:
 print("Problemas com o arquivo.")
```

# Lendo e Escrevendo no Arquivo Binário

- Para **ler um objeto de um arquivo binário** usamos o método `pickle.load`.

```
objeto = pickle.load(arquivo)
```

- `arquivo`: esta é a variável associada a um arquivo previamente aberto em modo binário.
- O método automaticamente reconhece o tipo de objeto salvo em arquivo, carrega este para a memória e atribui para a variável `objeto`.

# Lendo e Escrevendo no Arquivo Binário

- Exemplo: O programa abaixo lê a lista previamente salva em arquivo.

```
import pickle
try:
 arquivo = open("teste.bin", "rb")
 l = pickle.load(arquivo)
 print(l)
 arquivo.close()
except:
 print("Problemas com o arquivo.")
```

# Expressões Regulares

- Expressões regulares são formas concisas de **descrever um conjunto de strings** que satisfazem um determinado padrão.
- São uma mini-linguagem que permite especificar as **regras de construção de um conjunto de strings**.



# Exemplo de Expressão Regular

- Um exemplo de expressão regular é: `'\d+'`
- Essa RE representa uma sequência de 1 ou mais dígitos.

```
import re
digitos = re.search(r'\d+', 'Turma de MC102')
digitos
```

```
<_sre.SRE_Match object; span=(11, 14), match='102'>
```

# Expressão Regular

- Python fornece algumas classes pré-definidas que são bastante úteis:
  - `\d`: Qualquer dígito decimal, isto é, `[0-9]`.
  - `\D`: É o complemento de `\d`, equivalente a `[^0-9]`, i.e, faz o match com um caractere que não seja dígito.
  - `\s`: Faz match com caracteres *whitespace*, i.e., equivalente a `[\t\n\r\f\v]`.
  - `\S`: O complemento de `\s`.
  - `\w`: Faz o match com um caractere alfanumérico, i.e., equivalente a `[a-zA-Z0-9]`.
  - `\W`: O complemento de `\w`.



# Exemplo: Buscando e-mail

- Vamos construir uma RE para buscar um email:
  - O usuário é uma sequência de caracteres alfanuméricos `\w+` separado por `@`.
  - O host é uma sequência de caracteres alfanuméricos `\w+`.

```
re.search(r'\w+@\w+.\w+(\.\w+)?', 'Meu e-mail é login@ic.unicamp.br')
```

```
<_sre.SRE_Match object; span=(13, 32), match='login@ic.unicamp.br'>
```

# Exemplo: Buscando e-mail

- Vamos construir uma RE para buscar um email:
  - O usuário é uma sequência de caracteres alfanuméricos `\w+` separado por `@`.
  - O host é uma sequência de caracteres alfanuméricos `\w+`.

```
re.search(r'\w+@\w+.\w+(\.\w+)?', 'Meu e-mail é login@ic.unicamp.br')
```

```
<_sre.SRE_Match object; span=(13, 32), match='login@ic.unicamp.br'>
```

# Exercícios

A

B

C

D

1. O que será impresso pelo programa?

```
tupla1 = (10, 3, "mc102", 2.5)
tupla1[0] = 2
tupla1[1] = 8
tupla2 = tupla1[0:2]
print(tupla2)
```

a. Não irá compilar.

b. (2, 8)

c. (2, 8, "mc102")

d. (10, 3, "mc102")

2. Qual é o valor impresso ao final da seguinte sequência de comandos?

```
tupla = (3, 1, 2, 5)
a, b, c, d = tupla
print(a+d)
```

- a. Nada é impresso, ocorre um erro de execução.
- b. 4
- c. 3
- d. 8**

### 3. O que será impresso pelo programa?

```
tupla1 = (3, 1, 2, 5)
tupla2 = tupla1[1:4]
tupla3 = tupla1 + tupla2
print(tupla3)
```

- a. (6, 2, 4, 10)
- b. (3, 1, 2, 5)
- c. (3, 1, 2, 5, 1, 2, 5)**
- d. (1, 2, 5)

4. O que imprime o seguinte comando?

```
animais = {"gato":12, "cachorro":6, "elefante":23}
animais["rato"] = animais["gato"] + animais["cachorro"]
print(animais["rato"])
```

- a. Não irá compilar.
- b. 12
- c. 6
- d. 18**

5. O que será impresso pelo programa a seguir?

```
animais = {"gato":12,"cachorro":6,"rato":18,"elefante":23}
for i in ["cachorro","rato","elefante","gato"]:
 if i == "gato" or i == "rato":
 print("a",end=" ")
 else:
 print(animais[i],end=" ")
```

a. a a a a

b. a 6 a 18

c. 6 a 23 a

d. 6 18 23 12



6. Qual dos seguintes é um cabeçalho válido de uma função (a primeira linha da definição da função)?

a. `def desenhaCirculo(t) :`

b. `def desenhaCirculo:`

c. `desenhaCirculo(t, tam) :`

d. `def desenhaCirculo(t, tam)`

7. Quais são os parâmetros da seguinte função?

```
def potencia (base, expoente) :
 resultado = 1
 for numero in range (1, expoente+1) :
 # base ** expoente = base * base (expoente vezes)
 resultado = resultado * base
 return resultado
```

- a. resultado
- b. numero, resultado
- c. **base, expoente**
- d. numero, resultado, base, expoente

8. O que o seguinte código imprime?

```
def pot(b, p):
 y = b ** p
 return y
def quadrado(x):
 a = pot(x, 2)
 return a

n = 5
resultado = quadrado(n)
print(resultado)
```

a. 25

b. 5

c. 125

d. 32

## 9. O que o seguinte código imprime?

```
def adicionaNumero(lista, elem):
 lista.append(elem)

lista = [5]
adicionaNumero(lista, 10)
adicionaNumero(lista, 5)
print(lista)
```

- a. Não irá compilar.
- b. [5]
- c. [5, 10]
- d. [5, 10, 5]

## 10. O que o seguinte código imprime?

```
def adicionaNumero(lista, elem):
 lista.append(elem)

lista1 = [1, 2, 3, 4, 5]
lista2 = [10, 20]
lista1 = lista2
adicionaNumero(lista1, 30)
print(lista1)
```

- a. Não irá compilar.
- b. [1, 2, 3, 4, 5, 30]
- c. [10, 20, 30]**
- d. [1, 2, 3, 4, 5, 10, 20, 30]

11. O que é impresso pelos seguintes comandos?

```
def func():
 a = b + 10
 return a

a = 10
b = 20
c = func()
print("c = ", a + b + c)
```

- a. 30
- b. 40
- c. 60**
- d. Erro.

12. O que o seguinte código imprime?

```
def func():
 global a
 b = 6
 a = b + 30

func()
a = 10
b = 20
print("c = ", a + b)
```

- a. 30
- b. 56
- c. 86
- d. 36

13. O que será exibido pelo programa?

```
def func(p):
 global a
 a = b + 30
 print("res = ", p + a)
```

```
a = 10
func(a)
b = 20
```

- a. res = 60
- b. res = 40
- c. res = 30
- d. Não irá compilar.**



## 14. O que será exibido pelo programa?

```
mat = []
n = 2
for i in range(n):
 lista = []
 for j in range(n):
 lista.append(1*i)
 mat.append(lista)
print(mat)
```

- a. `[[1, 1], [2, 2]]`
- b. `[[0, 0], [1, 1]]`**
- c. `[]`
- d. `[[0, 1], [0, 1]]`

15. Quantas comparações são feitas na Busca Sequencial e na Busca Binária até o valor da posição onde se encontra a chave a ser retornada?

```
lista = [2, 5, 6, 7, 9, 10]
chave = 9
```

- a. Sequencial: 6 comparações & Binária: 3 comparações
- b. Sequencial: 3 comparações & Binária: 6 comparações
- c. Sequencial: 5 comparações & Binária: 2 comparações**
- d. Sequencial: 2 comparações & Binária: 5 comparações

16. Para a lista `[3, 5, 1, 2, 0, 4]`, qual foi o algoritmo de ordenação aplicado?

```
[3, 5, 1, 2, 0, 4]
[3, 1, 5, 2, 0, 4]
[3, 1, 2, 5, 0, 4]
[3, 1, 2, 0, 5, 4]
[3, 1, 2, 0, 4, 5]
...
[0, 1, 2, 3, 4, 5]
```

- a. `selectionSort(lista)`
- b. `bubbleSort(lista)`**
- c. `insertionSort(lista)`
- d. Nenhuma das opções acima.

17. Para a lista `[3, 5, 1, 2, 0, 4]`, qual foi o algoritmo de ordenação aplicado?

```
[0, 5, 1, 2, 3, 4]
```

```
[0, 1, 5, 2, 3, 4]
```

```
[0, 1, 2, 5, 3, 4]
```

```
[0, 1, 2, 3, 5, 4]
```

```
[0, 1, 2, 3, 4, 5]
```

```
[0, 1, 2, 3, 4, 5]
```

- a. `selectionSort(lista)`
- b. `bubbleSort(lista)`
- c. `insertionSort(lista)`
- d. Nenhuma das opções acima.

18. Para a lista  $[3, 5, 1, 2, 0, 4]$ , qual foi o algoritmo de ordenação aplicado?

$[3, 5, 1, 2, 0, 4]$

$[1, 3, 5, 2, 0, 4]$

$[1, 2, 3, 5, 0, 4]$

$[0, 1, 2, 3, 5, 4]$

$[0, 1, 2, 3, 4, 5]$

- a. `selectionSort(lista)`
- b. `bubbleSort(lista)`
- c. **`insertionSort(lista)`**
- d. Nenhuma das opções acima.

19. O que é impresso pelo trecho de código a seguir?

```
def rec(n):
 if n == 10:
 return 1
 return 1 + rec(n+1)

n = 6
print(rec(n))
```

- a. 7
- b. 5**
- c. 6
- d. 1

# Exercícios: Atividades Conceituais

- Tuplas: 10
- Dicionários: 10
- Funções: 14, 15, 16, 17, 20
- Matrizes e listas multidimensionais: 18
- Recursão: 23
- Algoritmos de busca: 22
- Algoritmos de ordenação: 21
- Expressões regulares: 19

**Boa  
Prova!**

AMO  
PROGRAMAR



! ! !  
D - - -  
X - - -  
T - - -  
# - - -  
o - - -  
n - - -  
! ! !