MAC 115 — Introdução à Ciência da Computação

Aula 21

Nelson Lago

IF noturno – 2023

Previously on MAC 115...

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
matriz[1][1] = 3
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
matriz[1][1] = 3
print(matriz)
```

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
matriz[1][1] = 3
print(matriz)
```

[[0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0]]

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
matriz[1][1] = 3
print(matriz)
```

[[0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0]]

Matrizes

E para criar uma matriz de zeros, digamos, 5x3?

```
linha = [0]*3
matriz = [linha]*5
matriz[1][1] = 3
print(matriz)
```

[[0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0], [0, 3, 0]]

Algoritmos

Programar envolve

Algoritmos

Programar envolve

1. **Compreender um problema em termos computacionais**
   - Cálculos?
   - Armazenamento e recuperação de dados?
   - Processamento de multimídia?
   - ...

Algoritmos

Programar envolve

1. **Compreender um problema em termos computacionais**
   - Cálculos?
   - Armazenamento e recuperação de dados?
   - Processamento de multimídia?
   - ...
2. **Definir como esse problema pode ser solucionado (*algoritmo*)**
   - O algoritmo é *abstrato* (como a planta de um prédio ou uma receita de bolo)

Algoritmos

Programar envolve

1. **Compreender um problema em termos computacionais**
   - Cálculos?
   - Armazenamento e recuperação de dados?
   - Processamento de multimídia?
   - ...
2. **Definir como esse problema pode ser solucionado (*algoritmo*)**
   - O algoritmo é *abstrato* (como a planta de um prédio ou uma receita de bolo)
3. ***Implementar* o algoritmo em uma linguagem de programação**
   - Gerando um *programa* que pode ser *executado* para solucionar o problema

Algoritmos

Programar envolve

1. **Compreender um problema em termos computacionais**
   - Cálculos?
   - Armazenamento e recuperação de dados?
   - Processamento de multimídia?
   - ...
2. **Definir como esse problema pode ser solucionado (*algoritmo*)**
   - O algoritmo é *abstrato* (como a planta de um prédio ou uma receita de bolo)
3. ***Implementar* o algoritmo em uma linguagem de programação**
   - Gerando um *programa* que pode ser *executado* para solucionar o problema
     - » *Uma receita de bolo em alemão é útil? Para quem?*

Algoritmos

Programar envolve

1. **Compreender um problema em termos computacionais**
   - Cálculos?
   - Armazenamento e recuperação de dados?
   - Processamento de multimídia?
   - ...
2. **Definir como esse problema pode ser solucionado (*algoritmo*)**
   - O algoritmo é *abstrato* (como a planta de um prédio ou uma receita de bolo)
3. ***Implementar* o algoritmo em uma linguagem de programação**
   - Gerando um *programa* que pode ser *executado* para solucionar o problema
     - » *Uma receita de bolo em alemão é útil? Para quem?*
4. **Testar o programa**

Mas como criar o algoritmo “certo”?

Algoritmos

Algoritmos

1. **Conhecendo técnicas comuns**
   (ou seja, estudo e prática)

Algoritmos

1. **Conhecendo técnicas comuns (ou seja, estudo e prática)**

Algoritmos

1. Conhecendo técnicas comuns (ou seja, estudo e prática)
2. Usando um algoritmo que já existe

Busca linear

Escreva uma função que recebe uma lista de números `L` e um número `n` e informa se o número está ou não na lista

Busca linear

Escreva uma função que recebe uma lista de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
```

Busca linear

Escreva uma função que recebe uma lista de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    for i in L:
```

Busca linear

Escreva uma função que recebe uma lista de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    for i in L:
        if i == n:
            return True
```

Busca linear

Escreva uma função que recebe uma lista de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    for i in L:
        if i == n:
            return True
    return False
```

Mas é assim que procuramos uma palavra em um dicionário?!?!

Busca binária

Quando procuramos em um dicionário:

Busca binária

Quando procuramos em um dicionário:

- **Abrimos o dicionário "em algum lugar perto do meio"**

Busca binária

Quando procuramos em um dicionário:

- **Abrimos o dicionário "em algum lugar perto do meio"**
  - Ih, é antes → abre "em algum lugar do meio" entre o começo e a página atual

Busca binária

Quando procuramos em um dicionário:

- **Abrimos o dicionário "em algum lugar perto do meio"**
  - Ih, é antes → abre "em algum lugar do meio" entre o começo e a página atual
  - Ih, é depois → abre "em algum lugar do meio" entre a página atual e o fim

Busca binária

Quando procuramos em um dicionário:

- **Abrimos o dicionário "em algum lugar perto do meio"**
  - Ih, é antes → abre "em algum lugar do meio" entre o começo e a página atual
  - Ih, é depois → abre "em algum lugar do meio" entre a página atual e o fim
- **No caso do dicionário, sabemos o mínimo ("a") e o máximo ("z") da lista, então não abrimos "no meio", mas tentamos "chutar" uma página próxima**

Busca binária

Quando procuramos em um dicionário:

- **Abrimos o dicionário "em algum lugar perto do meio"**
  - Ih, é antes → abre "em algum lugar do meio" entre o começo e a página atual
  - Ih, é depois → abre "em algum lugar do meio" entre a página atual e o fim
- **No caso do dicionário, sabemos o mínimo ("a") e o máximo ("z") da lista, então não abrimos "no meio", mas tentamos "chutar" uma página próxima**
- **Numa lista de números genérica, não temos essa "dica", então o melhor é olhar o "meio" mesmo**

Busca binária

Vamos procurar pelo número 23

Busca binária

Vamos procurar pelo número 23

1	3	7	14	21	22	23	26	27	30	31

Busca binária

Vamos procurar pelo número 23

Busca binária

Vamos procurar pelo número 23

Busca binária

Vamos procurar pelo número 23

Busca binária

Vamos procurar pelo número 23

Busca binária

Vamos procurar pelo número 23

Busca binária

Vamos procurar pelo número 23

Busca binária

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

Busca binária

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
```

Busca binária

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):









    return False
```

Busca binária

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    i = 0
    j = len(L) -1









    return False
```

Busca binária

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    i = 0
    j = len(L) -1
    while i <= j:









    return False
```

Busca binária

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    i = 0
    j = len(L) -1
    while i <= j:
        meio = (i + j) // 2









    return False
```

Busca binária

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    i = 0
    j = len(L) -1
    while i <= j:
        meio = (i + j) // 2
        if n == L[meio]:
            return True




    return False
```

Busca binária

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    i = 0
    j = len(L) -1
    while i <= j:
        meio = (i + j) // 2
        if n == L[meio]:
            return True
        elif n > L[meio]:
            i = meio +1


    return False
```

Busca binária

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    i = 0
    j = len(L) -1
    while i <= j:
        meio = (i + j) // 2
        if n == L[meio]:
            return True
        elif n > L[meio]:
            i = meio +1
        else:
            j = meio -1
    return False
```

Busca binária

Escreva uma função que recebe uma lista de números e informa se ela está ordenada

Busca binária

Escreva uma função que recebe uma lista de números e informa se ela está ordenada

```
def ordenada(L):
```

Busca binária

Escreva uma função que recebe uma lista de números e informa se ela está ordenada

```
def ordenada(L):



    return True
```

Busca binária

Escreva uma função que recebe uma lista de números e informa se ela está ordenada

```
def ordenada(L):

        if L[i] < L[i-1]:
            return False
    return True
```

Busca binária

Escreva uma função que recebe uma lista de números e informa se ela está ordenada

```
def ordenada(L):
    for i in range( , len(L)):
        if L[i] < L[i-1]:
            return False
    return True
```

Busca binária

Escreva uma função que recebe uma lista de números e informa se ela está ordenada

```
def ordenada(L):
    for i in range(1, len(L)):
        if L[i] < L[i-1]:
            return False
    return True
```

Ordenação

E como ordenar uma lista?

Ordenação

E como ordenar uma lista?

`lista.sort()`! 😜

Ordenação – ideias

Para ordenar uma lista A:

Ordenação – ideias

Para ordenar uma lista A:

- **“Encontra o próximo e coloca no fim da lista”**

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"**
  1. Cria uma lista `B` vazia

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"**
  1. Cria uma lista `B` vazia
  2. Encontra o menor elemento da lista `A` e move esse elemento para o final da lista `B`

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"**
  1. Cria uma lista `B` vazia
  2. Encontra o menor elemento da lista `A` e move esse elemento para o final da lista `B`
  3. Repete até a lista `A` estar vazia

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"**
  1. Cria uma lista `B` vazia
  2. Encontra o menor elemento da lista `A` e move esse elemento para o final da lista `B`
  3. Repete até a lista `A` estar vazia
- **"Pega qualquer um e encontra o lugar dele na lista"**

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"**
  1. Cria uma lista `B` vazia
  2. Encontra o menor elemento da lista `A` e move esse elemento para o final da lista `B`
  3. Repete até a lista `A` estar vazia
- **"Pega qualquer um e encontra o lugar dele na lista"**
  1. Cria uma lista `B` vazia

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"**
  1. Cria uma lista `B` vazia
  2. Encontra o menor elemento da lista `A` e move esse elemento para o final da lista `B`
  3. Repete até a lista `A` estar vazia
- **"Pega qualquer um e encontra o lugar dele na lista"**
  1. Cria uma lista `B` vazia
  2. Escolhe um elemento qualquer da lista `A` e move esse elemento para a lista `B`

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"**
  1. Cria uma lista `B` vazia
  2. Encontra o menor elemento da lista `A` e move esse elemento para o final da lista `B`
  3. Repete até a lista `A` estar vazia
- **"Pega qualquer um e encontra o lugar dele na lista"**
  1. Cria uma lista `B` vazia
  2. Escolhe um elemento qualquer da lista `A` e move esse elemento para a lista `B`
     » *como a lista agora só tem um elemento, ela está "naturalmente" ordenada*

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"**
  1. Cria uma lista `B` vazia
  2. Encontra o menor elemento da lista `A` e move esse elemento para o final da lista `B`
  3. Repete até a lista `A` estar vazia
- **"Pega qualquer um e encontra o lugar dele na lista"**
  1. Cria uma lista `B` vazia
  2. Escolhe um elemento qualquer da lista `A` e move esse elemento para a lista `B`
     » *como a lista agora só tem um elemento, ela está "naturalmente" ordenada*
  3. Escolhe um elemento qualquer da lista `A` e move esse elemento para a lista `B` de maneira a manter essa lista ordenada

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"**
  1. Cria uma lista `B` vazia
  2. Encontra o menor elemento da lista `A` e move esse elemento para o final da lista `B`
  3. Repete até a lista `A` estar vazia
- **"Pega qualquer um e encontra o lugar dele na lista"**
  1. Cria uma lista `B` vazia
  2. Escolhe um elemento qualquer da lista `A` e move esse elemento para a lista `B`
     » *como a lista agora só tem um elemento, ela está "naturalmente" ordenada*
  3. Escolhe um elemento qualquer da lista `A` e move esse elemento para a lista `B` de maneira a manter essa lista ordenada
     » *(é preciso mover os elementos da lista `B` para "abrir espaço" para o novo elemento)*

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):









    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):

    for i in range(len(L)):









    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
    idxmenor = 0
    for i in range(len(L)):









    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
    idxmenor = 0
    for i in range(len(L)):
        if L[i] < L[idxmenor]:
            idxmenor = i




    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
    idxmenor = 0
    for i in range(len(L)):
        if L[i] < L[idxmenor]:
            idxmenor = i
    menor = L[idxmenor]




    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
    idxmenor = 0
    for i in range(len(L)):
        if L[i] < L[idxmenor]:
            idxmenor = i
    menor = L[idxmenor]


    del L[-1]
    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
    idxmenor = 0
    for i in range(len(L)):
        if L[i] < L[idxmenor]:
            idxmenor = i
    menor = L[idxmenor]
    for i in range(idxmenor, len(L)   ):
        L[i] = L[i+1]
    del L[-1]
    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
    idxmenor = 0
    for i in range(len(L)):
        if L[i] < L[idxmenor]:
            idxmenor = i
    menor = L[idxmenor]
    for i in range(idxmenor, len(L) -1):
        L[i] = L[i+1]
    del L[-1]
    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):




    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
    for i in range(len(L)):




    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
    for i in range(len(L)):



    menor = L[-1]
    del L[-1]
    return menor
```

Exercício

Escreva uma função que recebe uma lista, encontra o menor elemento, remove esse elemento da lista e o devolve

```
def encontra_menor(L):
    for i in range(len(L)):
        if L[i] < L[-1]:
            L[i], L[-1] = L[-1], L[i]
    menor = L[-1]
    del L[-1]
    return menor
```

Exercício

Escreva uma função que recebe uma lista ordenada e um inteiro e insere esse inteiro na lista mantendo a lista ordenada (é preciso mover os elementos da lista para “abrir espaço” para o novo elemento)

Exercício

Escreva uma função que recebe uma lista ordenada e um inteiro e insere esse inteiro na lista mantendo a lista ordenada (é preciso mover os elementos da lista para “abrir espaço” para o novo elemento)

```
def encaixa(L, n):
```

Exercício

Escreva uma função que recebe uma lista ordenada e um inteiro e insere esse inteiro na lista mantendo a lista ordenada (é preciso mover os elementos da lista para “abrir espaço” para o novo elemento)

```
def encaixa(L, n):
    L.append(n)
```

Exercício

Escreva uma função que recebe uma lista ordenada e um inteiro e insere esse inteiro na lista mantendo a lista ordenada (é preciso mover os elementos da lista para “abrir espaço” para o novo elemento)

```
def encaixa(L, n):
    L.append(n)
    i = len(L) -1
    while i > 0                  :


        i -= 1
```

Exercício

Escreva uma função que recebe uma lista ordenada e um inteiro e insere esse inteiro na lista mantendo a lista ordenada (é preciso mover os elementos da lista para “abrir espaço” para o novo elemento)

```
def encaixa(L, n):
    L.append(n)
    i = len(L) -1
    while i > 0 and L[i-1] > L[i]:

        i -= 1
```

Exercício

Escreva uma função que recebe uma lista ordenada e um inteiro e insere esse inteiro na lista mantendo a lista ordenada (é preciso mover os elementos da lista para “abrir espaço” para o novo elemento)

```
def encaixa(L, n):
    L.append(n)
    i = len(L) -1
    while i > 0 and L[i-1] > L[i]:
        L[i-1], L[i] = L[i], L[i-1]
        i -= 1
```

Ordenação – ideias

- **Mas nem precisa de duas listas!**

Ordenação – ideias

- Mas nem precisa de duas listas!
- Basta criar uma divisão imaginária na lista

Ordenação – ideias

- **Mas nem precisa de duas listas!**
- **Basta criar uma divisão imaginária na lista**
  - O pedaço à esquerda da divisão é a lista “nova”, que está em ordem

Ordenação – ideias

- **Mas nem precisa de duas listas!**
- **Basta criar uma divisão imaginária na lista**
  - O pedaço à esquerda da divisão é a lista “nova”, que está em ordem
  - O pedaço à direta é a lista “velha”, que não está em ordem

Ordenação – ideias

- **Mas nem precisa de duas listas!**
- **Basta criar uma divisão imaginária na lista**
  - O pedaço à esquerda da divisão é a lista “nova”, que está em ordem
  - O pedaço à direta é a lista “velha”, que não está em ordem
  - A linha da divisão imaginária vai avançando a cada passo, de maneira que os elementos “saem” da lista desordenada e “entram” na lista ordenada

Ordenação – ideias

Para ordenar uma lista A:

Ordenação – ideias

Para ordenar uma lista A:

- **"Encontra o próximo e coloca no fim da lista"** – *selection sort*

Ordenação – ideias

Para ordenar uma lista A:

- **"Encontra o próximo e coloca no fim da lista"** – ***selection sort***
  1. Encontra o menor elemento da lista e troca esse elemento com o primeiro

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"** – *selection sort*
  1. Encontra o menor elemento da lista e troca esse elemento com o primeiro
  2. Encontra o menor elemento dos que sobraram e troca esse elemento com o segundo; etc

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"** – *selection sort*
  1. Encontra o menor elemento da lista e troca esse elemento com o primeiro
  2. Encontra o menor elemento dos que sobraram e troca esse elemento com o segundo; etc
- **"Pega qualquer um e encontra o lugar dele na lista"** – *insertion sort*

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"** – *selection sort*
  1. Encontra o menor elemento da lista e troca esse elemento com o primeiro
  2. Encontra o menor elemento dos que sobraram e troca esse elemento com o segundo; etc
- **"Pega qualquer um e encontra o lugar dele na lista"** – *insertion sort*
  1. A lista dos elementos até o elemento zero está ordenada

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"** – ***selection sort***
  1. Encontra o menor elemento da lista e troca esse elemento com o primeiro
  2. Encontra o menor elemento dos que sobraram e troca esse elemento com o segundo; etc
- **"Pega qualquer um e encontra o lugar dele na lista"** – ***insertion sort***
  1. A lista dos elementos até o elemento zero está ordenada
  2. Acrescenta o elemento um a essa lista (pode ser na posição um ou na posição zero) de maneira que a lista dos elementos até a posição um esteja ordenada

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"** – *selection sort*
  1. Encontra o menor elemento da lista e troca esse elemento com o primeiro
  2. Encontra o menor elemento dos que sobraram e troca esse elemento com o segundo; etc
- **"Pega qualquer um e encontra o lugar dele na lista"** – *insertion sort*
  1. A lista dos elementos até o elemento zero está ordenada
  2. Acrescenta o elemento um a essa lista (pode ser na posição um ou na posição zero) de maneira que a lista dos elementos até a posição um esteja ordenada
  3. Acrescenta o elemento dois a essa lista (pode ser nas posições zero, um ou dois) de maneira que a lista dos elementos até a posição dois esteja ordenada; etc

Ordenação – ideias

Para ordenar uma lista `A`:

- **"Encontra o próximo e coloca no fim da lista"** – ***selection sort***
  1. Encontra o menor elemento da lista e troca esse elemento com o primeiro
  2. Encontra o menor elemento dos que sobraram e troca esse elemento com o segundo; etc
- **"Pega qualquer um e encontra o lugar dele na lista"** – ***insertion sort***
  1. A lista dos elementos até o elemento zero está ordenada
  2. Acrescenta o elemento um a essa lista (pode ser na posição um ou na posição zero) de maneira que a lista dos elementos até a posição um esteja ordenada
  3. Acrescenta o elemento dois a essa lista (pode ser nas posições zero, um ou dois) de maneira que a lista dos elementos até a posição dois esteja ordenada; etc
     - » *(é preciso mover alguns dos elementos para "abrir espaço" para o novo elemento)*

“encontrar o próximo”

(quem?)

(select)

“encontrar o próximo”

(quem?)

(select)

vs

“encontrar o próximo”

(quem?)

(select)

vs

“encontrar o lugar do próximo”

(onde?)

(insert)

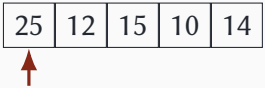
Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.

25	12	15	10	14

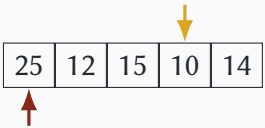
Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.

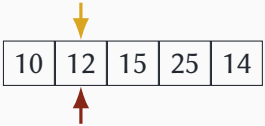


Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.

Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.

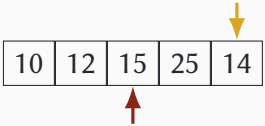


Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.

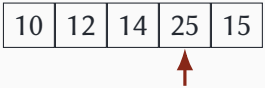
Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



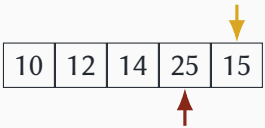
Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



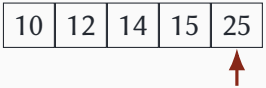
Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.



Selection sort

Selection sort: Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.

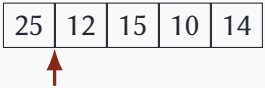
Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta

25	12	15	10	14

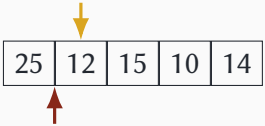
Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta



Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta

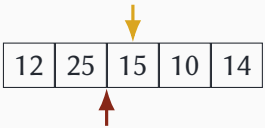


Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta

Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta

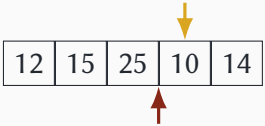


Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta

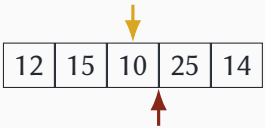
Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta



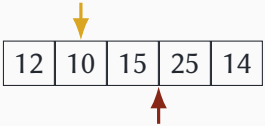
Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta



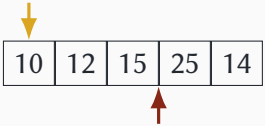
Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta



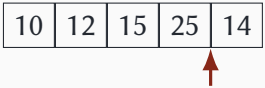
Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta



Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta

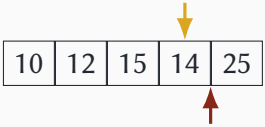


Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta

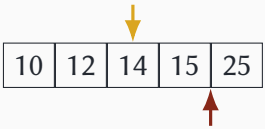
Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta



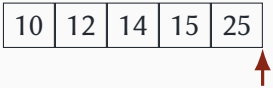
Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta



Insertion sort

Insertion sort: Considera que o “lado esquerdo” da lista está em ordem; a cada iteração, passa um elemento do “lado direito” para o “lado esquerdo”, garantindo que a ordem do “lado esquerdo” continue correta



Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): “Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”

Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): “Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”

```
def selectionSort(L):
```

Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): “Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”

```
def selectionSort(L):




def idxmenor(L, início):
```

Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): "Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc."

```
def selectionSort(L):
    for i in range(len(L)):



def idxmenor(L, início):
```

Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): “Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc.”

```
def selectionSort(L):
    for i in range(len(L)):
        certo = idxmenor(L, i)


def idxmenor(L, início):
```

Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): "Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc."

```
def selectionSort(L):
    for i in range(len(L)):
        certo = idxmenor(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmenor(L, início):
```

Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): "Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc."

```
def selectionSort(L):
    for i in range(len(L)):
        certo = idxmenor(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmenor(L, início):




    return min
```

Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): "Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc."

```
def selectionSort(L):
    for i in range(len(L)):
        certo = idxmenor(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmenor(L, início):
    min = início




    return min
```

Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): "Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc."

```
def selectionSort(L):
    for i in range(len(L)):
        certo = idxmenor(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmenor(L, início):
    min = início
    for i in range(      , len(L)):



    return min
```

Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): "Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc."

```
def selectionSort(L):
    for i in range(len(L)):
        certo = idxmenor(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmenor(L, início):
    min = início
    for i in range(      , len(L)):
        if L[i] < L[min]:
            min = i
    return min
```

Exercício – selection sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por seleção (*selection sort*): "Procura o menor de todos e coloca na primeira posição; procura o menor de todos entre os que sobraram e coloca na segunda posição; etc."

```
def selectionSort(L):
    for i in range(len(L)):
        certo = idxmenor(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmenor(L, início):
    min = início
    for i in range(início, len(L)):
        if L[i] < L[min]:
            min = i
    return min
```

Exercício – insertion sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por inserção (*insertion sort*): “Considera que o ‘lado esquerdo’ da lista está em ordem; a cada iteração, passa um elemento do lado direito para o lado esquerdo, garantindo que a ordem do lado esquerdo continue correta”

Exercício – insertion sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por inserção (*insertion sort*): “Considera que o ‘lado esquerdo’ da lista está em ordem; a cada iteração, passa um elemento do lado direito para o lado esquerdo, garantindo que a ordem do lado esquerdo continue correta”

```
def insertionSort(L):
```

Exercício – insertion sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por inserção (*insertion sort*): “Considera que o ‘lado esquerdo’ da lista está em ordem; a cada iteração, passa um elemento do lado direito para o lado esquerdo, garantindo que a ordem do lado esquerdo continue correta”

```
def insertionSort(L):
    for i in range(  len(L)):
        encaixa(L, i)
```

Exercício – insertion sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por inserção (*insertion sort*): "Considera que o 'lado esquerdo' da lista está em ordem; a cada iteração, passa um elemento do lado direito para o lado esquerdo, garantindo que a ordem do lado esquerdo continue correta"

```
def insertionSort(L):
    for i in range(1,len(L)):
        encaixa(L, i)
```

Exercício – insertion sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por inserção (*insertion sort*): “Considera que o ‘lado esquerdo’ da lista está em ordem; a cada iteração, passa um elemento do lado direito para o lado esquerdo, garantindo que a ordem do lado esquerdo continue correta”

```
def insertionSort(L):
    for i in range(1,len(L)):
        encaixa(L, i)
def encaixa(L, i):
```

Exercício – insertion sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por inserção (*insertion sort*): “Considera que o ‘lado esquerdo’ da lista está em ordem; a cada iteração, passa um elemento do lado direito para o lado esquerdo, garantindo que a ordem do lado esquerdo continue correta”

```
def insertionSort(L):
    for i in range(1,len(L)):
        encaixa(L, i)
def encaixa(L, i):
      while i > 0                  :


        i -= 1
```

Exercício – insertion sort

Escreva uma função que recebe uma lista de inteiros e ordena essa lista usando o algoritmo de ordenação por inserção (*insertion sort*): “Considera que o ‘lado esquerdo’ da lista está em ordem; a cada iteração, passa um elemento do lado direito para o lado esquerdo, garantindo que a ordem do lado esquerdo continue correta”

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

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir do menor para o maior, ir do maior para o menor

```
def selectionSort(L):
    for i in range(len(L)):
        certo = idxmenor(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmenor(L, início):
    min = início
    for i in range(início, len(L)):
        if L[i] < L[min]:
            min = i
    return min
```

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir do menor para o maior, ir do maior para o menor

```
def selectionSort(L):
    for i in range(len(L) -1, -1, -1):
        certo = idxmenor(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmenor(L, início):
    min = início
    for i in range(início, len(L)):
        if L[i] < L[min]:
            min = i
    return min
```

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir do menor para o maior, ir do maior para o menor

```
def selectionSort(L):
    for i in range(len(L) -1, -1, -1):
        certo = idxmaior(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmenor(L, início):
    min = início
    for i in range(início, len(L)):
        if L[i] < L[min]:
            min = i
    return min
```

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir do menor para o maior, ir do maior para o menor

```
def selectionSort(L):
    for i in range(len(L) -1, -1, -1):
        certo = idxmaior(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmaior(L, final):
    min = início
    for i in range(início, len(L)):
        if L[i] < L[min]:
            min = i
    return min
```

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir do menor para o maior, ir do maior para o menor

```
def selectionSort(L):
    for i in range(len(L) -1, -1, -1):
        certo = idxmaior(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmaior(L, final):
    max = início
    for i in range(início, len(L)):
        if L[i] < L[max]:
            max = i
    return max
```

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir do menor para o maior, ir do maior para o menor

```
def selectionSort(L):
    for i in range(len(L) -1, -1, -1):
        certo = idxmaior(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmaior(L, final):
    max = início
    for i in range(início, len(L)):
        if L[i] > L[max]:
            max = i
    return max
```

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir do menor para o maior, ir do maior para o menor

```
def selectionSort(L):
    for i in range(len(L) -1, -1, -1):
        certo = idxmaior(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmaior(L, final):
    max = final
    for i in range(final, -1, -1):
        if L[i] > L[max]:
            max = i
    return max
```

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir do menor para o maior, ir do maior para o menor

```
def selectionSort(L):
    for i in range(len(L) -1, -1, -1):
        certo = idxmaior(L, i)
        L[i], L[certo] = L[certo], L[i]

def idxmaior(L, final):
    max = 0
    for i in range(final +1):
        if L[i] > L[max]:
            max = i
    return max
```

Exercício – insertion sort

Modifique a implementação anterior de *insertion sort* para, ao invés de manter o segmento ordenado da lista à esquerda, mantê-lo à direita.

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

Exercício – insertion sort

Modifique a implementação anterior de *insertion sort* para, ao invés de manter o segmento ordenado da lista à esquerda, mantê-lo à direita.

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

Exercício – insertion sort

Modifique a implementação anterior de *insertion sort* para, ao invés de manter o segmento ordenado da lista à esquerda, mantê-lo à direita.

```
def insertionSort(L):
    for i in range(len(L) -1, 0, -1):
        encaixa(L, i)
def encaixa(L, i):
      while i < len(L) -1 and L[i-1] > L[i]:
        L[i-1], L[i] = L[i], L[i-1]
        i += 1
```

Exercício – insertion sort

Modifique a implementação anterior de *insertion sort* para, ao invés de manter o segmento ordenado da lista à esquerda, mantê-lo à direita.

```
def insertionSort(L):
    for i in range(len(L) -1, 0, -1):
        encaixa(L, i)
def encaixa(L, i):
      while i < len(L) -1 and L[i+1] < L[i]:
        L[i+1], L[i] = L[i], L[i+1]
        i += 1
```

Mas como criar o algoritmo “certo”?

Algoritmos

Algoritmos

1. **Conhecendo técnicas comuns**
   (ou seja, estudo e prática)

Algoritmos

1. **Conhecendo técnicas comuns (ou seja, estudo e prática)**

Algoritmos

1. Conhecendo técnicas comuns (ou seja, estudo e prática)
2. Usando um algoritmo que já existe

Divisão e conquista

Busca binária – divisão e conquista

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    i = 0
    j = len(L) -1
    while i <= j:
        meio = (i + j) // 2
        if n == L[meio]:
            return True
        elif n > L[meio]:
            i = meio +1
        else:
            j = meio -1
    return False
```

Busca binária – divisão e conquista

Escreva uma função que recebe duas listas ordenadas de números `La`, `Lb` e um número `n` e informa se o número está ou não em uma delas

Busca binária – divisão e conquista

Escreva uma função que recebe duas listas ordenadas de números `La`, `Lb` e um número `n` e informa se o número está ou não em uma delas

```
def pertences(La, Lb, n):
```

Busca binária – divisão e conquista

Escreva uma função que recebe duas listas ordenadas de números `La`, `Lb` e um número `n` e informa se o número está ou não em uma delas

```
def pertences(La, Lb, n):
    return pertence(La, n) or pertence(Lb, n)
```

Busca binária – divisão e conquista

Mas uma lista com mais de um elemento é igual a duas sub-listas!

```
pertence(L, n)
```

$\Leftrightarrow$

```
pertence(L[:len(L)//2], n) or pertence(L[len(L)//2:]), n)
```

Busca binária – divisão e conquista

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    i = 0
    j = len(L) -1
    while i <= j:
        meio = (i + j) // 2
        if n == L[meio]:
            return True
        elif n > L[meio]:
            i = meio +1
        else:
            j = meio -1
    return False
```

Busca binária – divisão e conquista

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
```

Busca binária – divisão e conquista

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):




    meio = len(L) // 2
    return pertence(L[:meio], n) or pertence(L[meio:], n)
```

Busca binária – divisão e conquista

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):



    if len(L) == 1:
        return L[0] == n
    meio = len(L) // 2
    return pertence(L[:meio], n) or pertence(L[meio:], n)
```

Busca binária – divisão e conquista

Escreva uma função que recebe uma lista ordenada de números `L` e um número `n` e informa se o número está ou não na lista

```
def pertence(L, n):
    if len(L) == 0:
        return False
    if len(L) == 1:
        return L[0] == n
    meio = len(L) // 2
    return pertence(L[:meio], n) or pertence(L[meio:], n)
```

Recursão

- **Recursão é útil quando você**

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$1 + 2 + 3 + 4 =$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 =$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 =$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 = (3 + 3) + 4 =$$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 = (3 + 3) + 4 = 6 + 4 =$$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 = (3 + 3) + 4 = 6 + 4 = 10$$

Recursão

- **Recursão é útil quando você**
  - Sabe como resolver o problema quando ele é “pequeno”
  - Sabe como dividir o problema “grande” em partes menores
    - » *(Até ele ficar “pequeno o suficiente”)*

Você sabe somar mais de dois números de uma vez?

$$1 + 2 + 3 + 4 = (1 + 2) + 3 + 4 = 3 + 3 + 4 = (3 + 3) + 4 = 6 + 4 = 10$$

- **Recursões são inspiradas nas demonstrações por indução da matemática**

Exercício – somatória

Escreva uma função que recebe uma lista de inteiros e devolve a soma dos elementos da lista usando o operador + apenas com 2 operandos.

Exercício – somatória

Escreva uma função que recebe uma lista de inteiros e devolve a soma dos elementos da lista usando o operador + apenas com 2 operandos.

```
def somatória(L):
```

Exercício – somatória

Escreva uma função que recebe uma lista de inteiros e devolve a soma dos elementos da lista usando o operador + apenas com 2 operandos.

```
def somatória(L):
    if len(L) == 2:
        return L[0] + L[1]
```

Exercício – somatória

Escreva uma função que recebe uma lista de inteiros e devolve a soma dos elementos da lista usando o operador + apenas com 2 operandos.

```
def somatória(L):
    if len(L) == 2:
        return L[0] + L[1]


    return L[0] + somatória(L[1:])
```

Exercício – somatória

Escreva uma função que recebe uma lista de inteiros e devolve a soma dos elementos da lista usando o operador + apenas com 2 operandos.

```
def somatória(L):
    if len(L) == 2:
        return L[0] + L[1]
    if len(L) == 1:
        return L[0]
    return L[0] + somatória(L[1:])
```

Exercício – somatória

Escreva uma função que recebe uma lista de inteiros e devolve a soma dos elementos da lista usando o operador + apenas com 2 operandos.

```
def somatória(L):
    if len(L) == 2:
        return L[0] + L[1]
    if len(L) == 1:
        return L[0]
    return L[0] + somatória(L[1:])
```

Exercício – reversão de string

Escreva uma função (recursiva) que recebe uma string e devolve a string na ordem inversa

Exercício – reversão de string

Escreva uma função (recursiva) que recebe uma string e devolve a string na ordem inversa

```
def reverte(L):
```

Exercício – reversão de string

Escreva uma função (recursiva) que recebe uma string e devolve a string na ordem inversa

```
def reverte(L):
    if len(L) <= 1:
        return L
```

Exercício – reversão de string

Escreva uma função (recursiva) que recebe uma string e devolve a string na ordem inversa

```
def reverte(L):
    if len(L) <= 1:
        return L
    return L[-1]
```

Exercício – reversão de string

Escreva uma função (recursiva) que recebe uma string e devolve a string na ordem inversa

```
def reverte(L):
    if len(L) <= 1:
        return L
    return L[-1] + reverte(      )
```

Exercício – reversão de string

Escreva uma função (recursiva) que recebe uma string e devolve a string na ordem inversa

```
def reverte(L):
    if len(L) <= 1:
        return L
    return L[-1] + reverte(L[:-1])
```

Recursão – mergesort

Um algoritmo de ordenação mais eficiente que os anteriores é o *mergesort*, que é um algoritmo recursivo:

Recursão – mergesort

Um algoritmo de ordenação mais eficiente que os anteriores é o *mergesort*, que é um algoritmo recursivo:

- **Uma lista com zero ou um elementos sempre está ordenada**

Recursão – mergesort

Um algoritmo de ordenação mais eficiente que os anteriores é o *mergesort*, que é um algoritmo recursivo:

- **Uma lista com zero ou um elementos sempre está ordenada**
- **Para ordenar uma lista maior, basta**

Recursão – mergesort

Um algoritmo de ordenação mais eficiente que os anteriores é o *mergesort*, que é um algoritmo recursivo:

- **Uma lista com zero ou um elementos sempre está ordenada**
- **Para ordenar uma lista maior, basta**
  1. Dividir a lista na "metade"

Recursão – mergesort

Um algoritmo de ordenação mais eficiente que os anteriores é o *mergesort*, que é um algoritmo recursivo:

- **Uma lista com zero ou um elementos sempre está ordenada**
- **Para ordenar uma lista maior, basta**
  1. Dividir a lista na "metade"
  2. Ordenar cada uma das partes

Recursão – mergesort

Um algoritmo de ordenação mais eficiente que os anteriores é o *mergesort*, que é um algoritmo recursivo:

- **Uma lista com zero ou um elementos sempre está ordenada**
- **Para ordenar uma lista maior, basta**
  1. Dividir a lista na "metade"
  2. Ordenar cada uma das partes
  3. Juntar as duas listas mantendo a ordem

Recursão – mergesort

Um algoritmo de ordenação mais eficiente que os anteriores é o *mergesort*, que é um algoritmo recursivo:

- **Uma lista com zero ou um elementos sempre está ordenada**
- **Para ordenar uma lista maior, basta**
  1. Dividir a lista na "metade"
  2. Ordenar cada uma das partes
  3. Juntar as duas listas mantendo a ordem

(a desvantagem do *mergesort* é que ele precisa criar uma lista auxiliar durante o processamento, ocupando mais memória)

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
    if len(L) <= 1:
        return L
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
    if len(L) <= 1:
        return L
    meio = len(L) // 2
    esqd = L[:meio]
    drta = L[meio:]
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
    if len(L) <= 1:
        return L
    meio = len(L) // 2
    esqd = L[:meio]
    drta = L[meio:]
    mergesort(esqd)
    mergesort(drta)
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
    if len(L) <= 1:
        return L
    meio = len(L) // 2
    esqd = L[:meio]
    drta = L[meio:]
    mergesort(esqd)
    mergesort(drta)
```

```
    juntas = []
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
    if len(L) <= 1:
        return L
    meio = len(L) // 2
    esqd = L[:meio]
    drta = L[meio:]
    mergesort(esqd)
    mergesort(drta)
```

```
    juntas = []
    for i in range(len(L)):
        L[i] = juntas[i]
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
    if len(L) <= 1:
        return L
    meio = len(L) // 2
    esqd = L[:meio]
    drta = L[meio:]
    mergesort(esqd)
    mergesort(drta)
```

```
    juntas = []
    i, j = 0, 0









    for i in range(len(L)):
        L[i] = juntas[i]
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
    if len(L) <= 1:
        return L
    meio = len(L) // 2
    esqd = L[:meio]
    drta = L[meio:]
    mergesort(esqd)
    mergesort(drta)
```

```
    juntas = []
    i, j = 0, 0
    while i < len(esqd) and j < len(drta):









    for i in range(len(L)):
        L[i] = juntas[i]
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
    if len(L) <= 1:
        return L
    meio = len(L) // 2
    esqd = L[:meio]
    drta = L[meio:]
    mergesort(esqd)
    mergesort(drta)
```

```
    juntas = []
    i, j = 0, 0
    while i < len(esqd) and j < len(drta):
        if esqd[i] < drta[j]:
            juntas.append(esqd[i])
            i += 1
        else:
            juntas.append(drta[j])
            j += 1









    for i in range(len(L)):
        L[i] = juntas[i]
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
    if len(L) <= 1:
        return L
    meio = len(L) // 2
    esqd = L[:meio]
    drta = L[meio:]
    mergesort(esqd)
    mergesort(drta)
```

```
    juntas = []
    i, j = 0, 0
    while i < len(esqd) and j < len(drta):
        if esqd[i] < drta[j]:
            juntas.append(esqd[i])
            i += 1
        else:
            juntas.append(drta[j])
            j += 1
    while i < len(esqd):
        juntas.append(esqd[i])
        i += 1




    for i in range(len(L)):
        L[i] = juntas[i]
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mergesort(L):
    if len(L) <= 1:
        return L
    meio = len(L) // 2
    esqd = L[:meio]
    drta = L[meio:]
    mergesort(esqd)
    mergesort(drta)
```

```
    juntas = []
    i, j = 0, 0
    while i < len(esqd) and j < len(drta):
        if esqd[i] < drta[j]:
            juntas.append(esqd[i])
            i += 1
        else:
            juntas.append(drta[j])
            j += 1
    while i < len(esqd):
        juntas.append(esqd[i])
        i += 1
    while j < len(drta):
        juntas.append(drta[j])
        j += 1

    for i in range(len(L)):
        L[i] = juntas[i]
```