MAC 110 — Introdução à Ciência da Computação

Aula 23

Nelson Lago

BMAC – 2024

CC BY

Previously on MAC110...

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

1	3	7	14	21	22	23	26	27	30	31

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

1	3	7	14	21	22	23	26	27	30	31

Busca binária

Busca binária

Vamos procurar pelo número 23

1	3	7	14	21	22	23	26	27	30	31

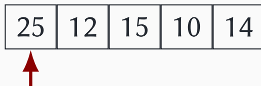
Selection sort

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

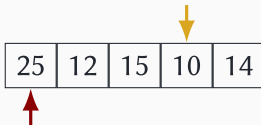
Selection sort

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.



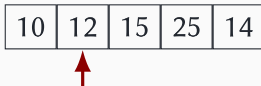
Selection sort

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.



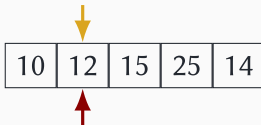
Selection sort

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.



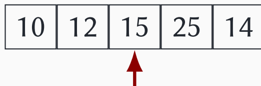
Selection sort

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.



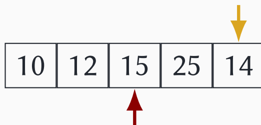
Selection sort

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.



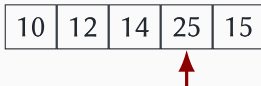
Selection sort

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.



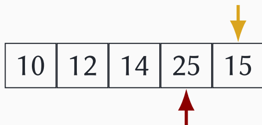
Selection sort

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.



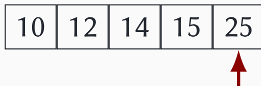
Selection sort

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.

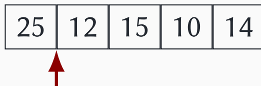
Insertion sort

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

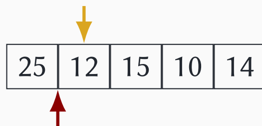
Insertion sort

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



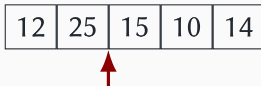
Insertion sort

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



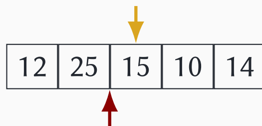
Insertion sort

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



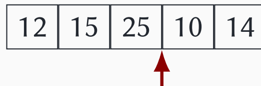
Insertion sort

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



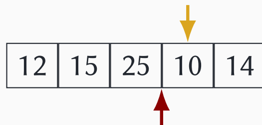
Insertion sort

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



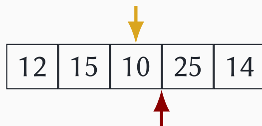
Insertion sort

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



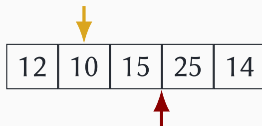
Insertion sort

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



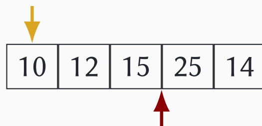
Insertion sort

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



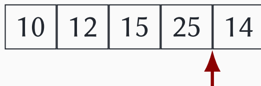
Insertion sort

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

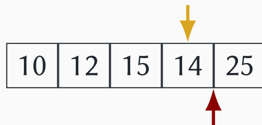


Insertion sort

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

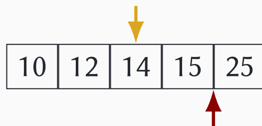
Insertion sort

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



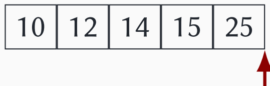
Insertion sort

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 menor
```

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):
    menor = início




    return menor
```

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):
    menor = início
    for i in range(      , len(L)):



    return menor
```

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

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

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”

```python
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(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(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(len(L)):
        encaixa(L, i)
def encaixa(L, i):
      while i > 0 and L[i-1] > L[i]:

        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(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 colocando os elementos do menor ao maior a partir da esquerda, ir colocando os elementos do maior ao menor a partir da direita

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

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir colocando os elementos do menor ao maior a partir da esquerda, ir colocando os elementos do maior ao menor a partir da direita

```python
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):
    menor = início
    for i in range(início, len(L)):
        if L[i] < L[menor]:
            menor = i
    return menor
```

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir colocando os elementos do menor ao maior a partir da esquerda, ir colocando os elementos do maior ao menor a partir da direita

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

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir colocando os elementos do menor ao maior a partir da esquerda, ir colocando os elementos do maior ao menor a partir da direita

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

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir colocando os elementos do menor ao maior a partir da esquerda, ir colocando os elementos do maior ao menor a partir da direita

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

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir colocando os elementos do menor ao maior a partir da esquerda, ir colocando os elementos do maior ao menor a partir da direita

```
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):
    maior = início
    for i in range(início, len(L)):
        if L[i] > L[maior]:
            maior = i
    return maior
```

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir colocando os elementos do menor ao maior a partir da esquerda, ir colocando os elementos do maior ao menor a partir da direita

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

Exercício – selection sort

Modifique a implementação anterior de *selection sort* para, ao invés de ir colocando os elementos do menor ao maior a partir da esquerda, ir colocando os elementos do maior ao menor a partir da direita

```
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):
    maior = 0
    for i in range(final +1):
        if L[i] > L[maior]:
            maior = i
    return maior
```

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)):
        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, -1, -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, -1, -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, -1, -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)**

Doctor Strange (2016)
dir. Scott Derrickson

Algoritmos

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

Doctor Strange (2016)
dir. Scott Derrickson

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 recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

```
def somatória(L):
```

Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

```
def somatória(L):
    if len(L) == 0:
        return 0
```

Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

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

Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

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

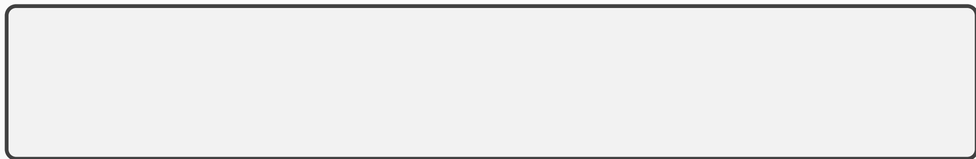
Exercício – somatória

Escreva uma função recursiva que recebe uma lista de inteiros e devolve a soma dos elementos da lista

```
def somatória(L):
    if len(L) == 0:
        return 0
    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***

Recursão – mergesort

- **Um algoritmo de ordenação mais eficiente que os anteriores é o *mergesort***
  - (a desvantagem do *mergesort* é que ele precisa criar uma lista auxiliar durante o processamento, ocupando mais memória)

Recursão – mergesort

- **Um algoritmo de ordenação mais eficiente que os anteriores é o *mergesort***
  - (a desvantagem do *mergesort* é que ele precisa criar uma lista auxiliar durante o processamento, ocupando mais memória)
- **Uma lista com zero ou um elementos sempre está ordenada**

Recursão – mergesort

- **Um algoritmo de ordenação mais eficiente que os anteriores é o *mergesort***
  - (a desvantagem do *mergesort* é que ele precisa criar uma lista auxiliar durante o processamento, ocupando mais memória)

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

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

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

- **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 partes mantendo a ordem

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

```
def mescla_listas(l1, l2):
    lista = []
    i, j = 0, 0
    while i < len(l1) and j < len(l2):
        if l1[i] < l2[j]:
            lista.append(l1[i])
            i += 1
        else:
            lista.append(l2[j])
            j += 1
    while i < len(l1):
        lista.append(l1[i])
        i += 1
    while j < len(l2):
        lista.append(l2[j])
        j += 1
    return lista
```

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
```

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

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

Exercício – mergesort

Escreva uma função que implementa o algoritmo *mergesort*

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

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

Recursão vs laços

- **Maneiras diferentes de pensar repetições**

Recursão vs laços

- Maneiras diferentes de pensar repetições
- Muitas vezes “dá na mesma”

Recursão vs laços

- Maneiras diferentes de pensar repetições
- Muitas vezes “dá na mesma”
- Às vezes, uma é mais simples de entender que a outra