MAC2166 - Introdução à Computação (1º Semestre de 2025)

Grande áreas Civil, Mecânica, Química e Petróleo

Guia de Estudos - Semana 12

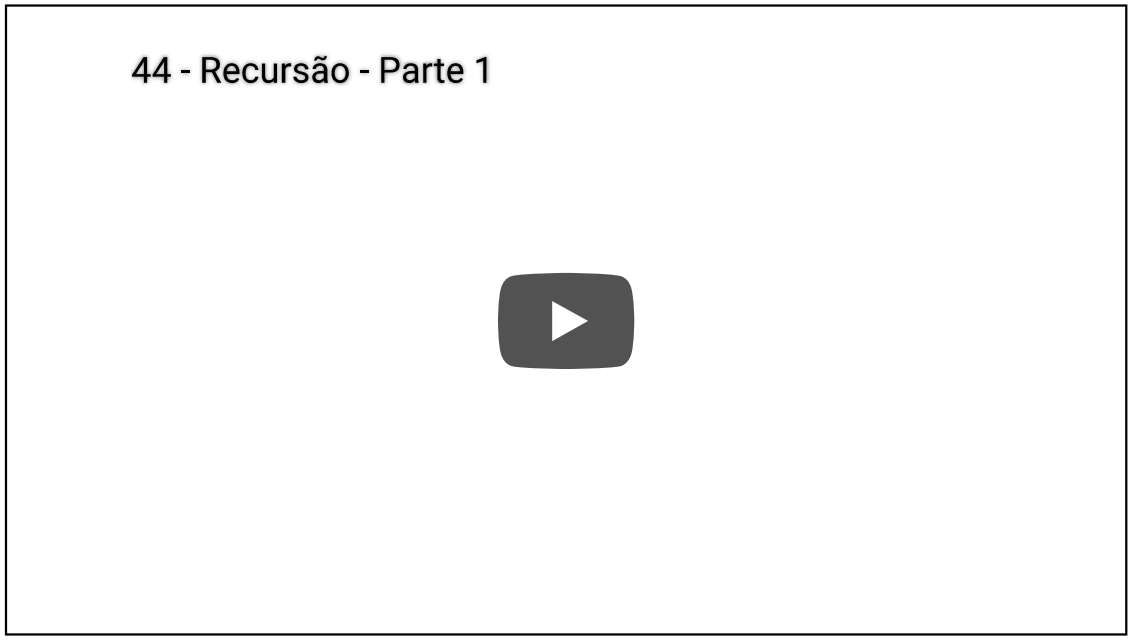
Tópicos

- Funções Recursivas
  - fatorial recursivo.
  - cálculo da série de Fibonacci.
  - função recursiva para cálculo de potência.
  - exemplo de listas com recursão.
- Algoritmos de Ordenação (Parte 2)
  - algoritmo de ordenação da bolha (*Bubble Sort*)
  - algoritmo de ordenação por intercalação (*Merge Sort*)

Antes de sua aula síncrona desta semana, estude os conteúdos abaixo e os materiais recomendados e tente resolver os exercícios. Os assuntos e exercícios serão discutidos pelo(a) professor(a) na aula.

Funções Recursivas

Veja a videoaula do prof. Fabio Kon (IME-USP) sobre Recursão:



Uma função é dita **recursiva** quando dentro dela é feita uma ou mais chamadas a ela mesma.

A ideia é dividir um problema original um subproblemas menores de mesma natureza (divisão) e depois combinar as soluções obtidas para gerar a solução do problema original de tamanho maior (conquista). Os subproblemas são resolvidos recursivamente do mesmo modo em função de instâncias menores, até se tornarem problemas triviais que são resolvidos de forma direta, interrompendo a recursão.

Exemplo 1:

Calcular o fatorial de um número.

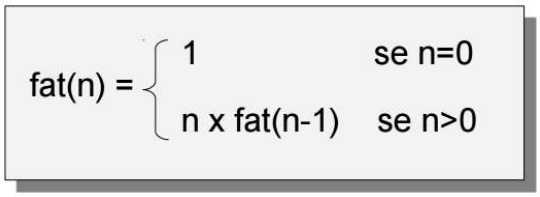
Solução 1:

Solução não recursiva.

```
def fatorial(n):
 fat = 1
 while n > 1:
    fat *= n
    n -= 1
 return fat
```

Solução 2:

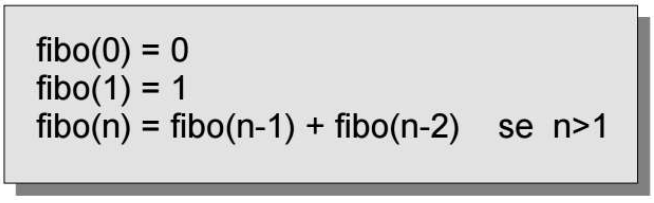
Solução recursiva: n! = n.(n-1)!




```
def fatorial(n):
  if n == 0: #Caso trivial
    return 1 #Solução direta
  else:
    return n*fatorial(n-1) #Chamada recursiva
```

Exemplo 2: Cálculo da série de Fibonacci

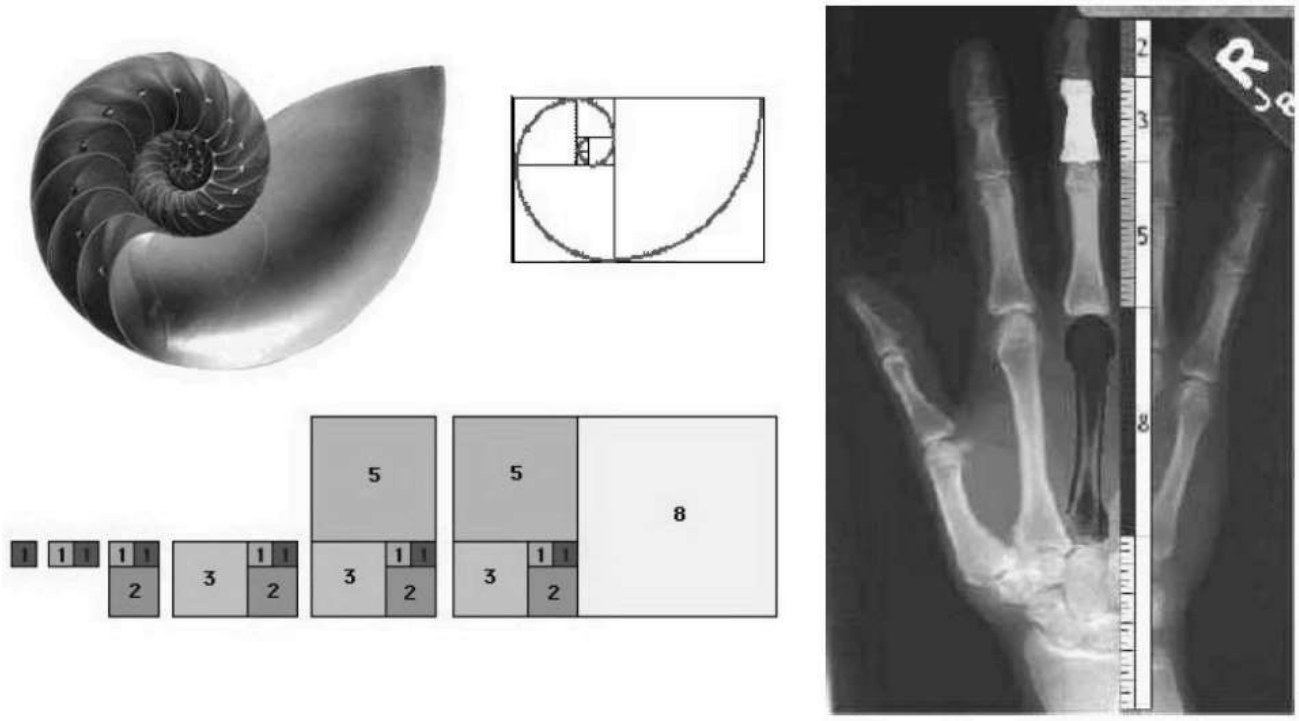
A sequência de Fibonacci consiste em uma série de números, tais que, definindo seus dois primeiros números como sendo 0 e 1, os números seguintes são obtidos através da soma dos seus dois antecessores.



Exemplo da sequência:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...
```

A ocorrência da sucessão de Fibonacci na natureza é tão frequente que é difícil acreditar que seja acidental (ex: flores, conchas, mão humana).



Solução 1:

Versão recursiva:

```
def fibo(n):
  if n <= 1:
    return n
  else:
    return fibo(n-1) + fibo(n-2)
```

Solução 2:

Versão não recursiva:

```
def fibo(n):
  f0 = 0
  f1 = 1
  for k in range(1,n+1):
    f2 = f0 + f1
    f0 = f1
    f1 = f2
  return f0
```

Análise de eficiência das funções Fibonacci calculando o número de operações de soma S(n) realizadas:

- **Versão recursiva**

$$S(n) = \begin{cases} 0 & \text{se } n \le 1 \\ S(n-1) + S(n-2) + 1 & \text{se } n > 1 \end{cases}$$

$$S(n) = fibo(n+1) - 1$$

$$S(n) \approx 1.6^n/1.4$$

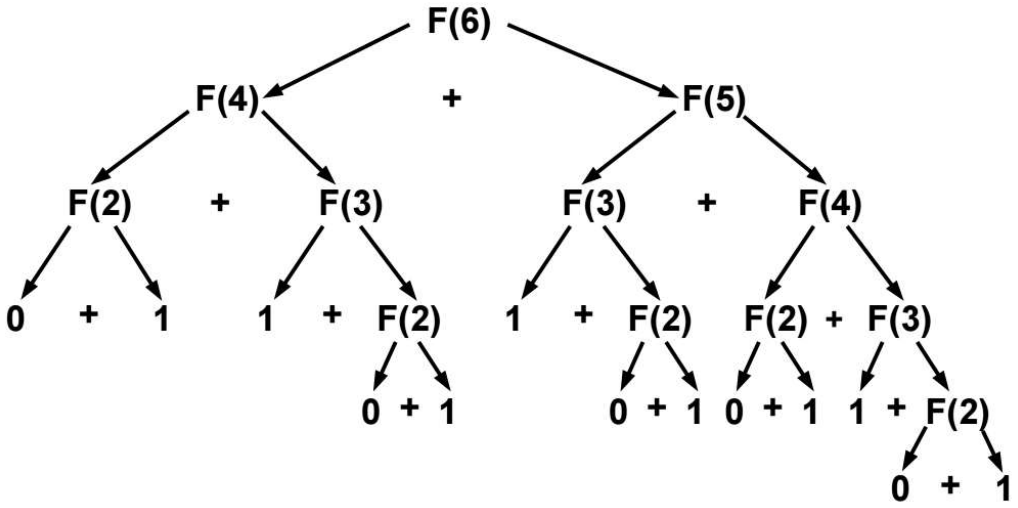
- **Versão não recursiva** - $S(n) = n$

Neste caso não faz sentido utilizar a versão recursiva.

Árvore de Recorrência:

A solução recursiva é ineficiente, pois recalcula várias vezes a solução para valores intermediários:

Por exemplo, para F(6)=fibo(6) = 8, temos 2xF(4), 3xF(3), 5xF(2).



Problema 1:

Escreva uma função recursiva `potencia(x, n)`, que calcule `x` elevado a `n`, sendo `x` um número real e `n` um inteiro positivo.

Solução 1:

Solução não recursiva.

```
def potencia(x, n):
  pot = 1.0
  while n > 0:
    pot *= x
    n -= 1
  return pot
```

Solução 2:

Solução recursiva: x^n = x . x^(n-1)

```
def potencia(x, n):
  if n == 0:    #Caso trivial
    return 1.0  #Solução direta
  else:
    return x*potencia(x, n-1) #Chamada recursiva
```

Solução 3:

Solução recursiva: x^n = x^(n/2). x^(n/2) = (x^(n/2))^2

```
def potencia(x, n):
  if n == 0: #Caso trivial
    return 1.0
  elif n%2 == 0: #Se n é par...
    pot = potencia(x, n//2)
    return pot*pot
  else: #Se n é ímpar...
    pot = potencia(x, n//2)
    return pot*pot*x
```

Problema 2

Escreva uma **função recursiva** que implemente o algoritmo da Busca Binária. A sua função deve receber como parâmetros um número real `x`, uma lista ordenada `L`, uma posição `inicio` e uma posição `fim`. A função deve devolver True quando `x` ocorre na sublista `L[inicio:fim+1]` ou False caso `x` não esteja na lista.

Solução:

```
def pertence(x, L, inic, fim):
  meio = (inic+fim)//2
  if inic > fim:
    return False
  elif x == L[meio]:
    return True
  elif x > L[meio]:
    return pertence_aux(x, L, meio+1, fim)
  else:
```

```
10      return pertence_aux(x, L, inic, meio-1)
```

Mais ordenação

Algoritmo de Ordenação da Bolha (*Bubble Sort*)

O algoritmo da bolha utiliza a seguinte estratégia para ordenar uma lista *seq* com *N* elementos:

- Repita *N-1* vezes:
    - varra a lista com *pos* variando de *1* até *N-1*:
        - se *seq[pos]* for menor que o elemento anterior então:
            - troca o elemento em *pos* com o seu anterior

A ideia é que, a cada varrida, elementos “grandes” são arrastados para a direita e elementos “pequenos” são arrastados para a esquerda da lista. Repetindo-se essas trocas por um número suficiente de vezes, obtemos a lista ordenado. No entanto, em muitos casos, é possível que a lista fique ordenada após algumas poucas trocas. Nesse caso, é possível melhorar esse algoritmo para que pare assim que se descubra que ele se tornou ordenado.

- houve_troca = True
- enquanto houve_troca:
    - houve_troca = False
    - varra a lista com *pos* variando de *1* até *N-1*:
        - se *seq[pos]* for menor que o elemento anterior então:
            - troca o elemento em *pos* com o seu anterior
            - houve_troca = True

Para entender esse algoritmo assista a animação desse método com dança folclórica húngara ou o vídeo Bubble Sort.

Problema 3

Escreva a função `ordenacao_bolha(seq)` que recebe como parâmetro uma lista `seq` e a ordena usando o algoritmo de ordenação por bolha.
Você deve rearranjar os elementos da lista de entrada ao invés de criar uma nova lista ordenada.

Solução 1:

O laço mais interno percorre a lista em `j` comparando elementos consecutivos da lista (`seq[j]` e `seq[j+1]`) e trocando a sua ordem sempre que eles estiverem fora de ordem, isto é `seq[j] > seq[j+1]`. Após a primeira varredura, o maior elemento da lista vai ocupar a última posição da lista, ou seja sua posição final definitiva na lista ordenada. A medida que os maiores valores são movidos para o final da lista, temos a correspondente redução do subtrecho remanescente a ser ordenado, até que todo a lista esteja ordenada.

```
1  def ordenacao_bolha(seq):
2    n = len(seq)
3    for i in range(n-1, 0, -1):
4      for j in range(0,i):
5        if seq[j] > seq[j+1]:
6          troca(seq, j, j+1)
```

Solução 2:

Uma segunda versão do algoritmo **bubble sort** é apresentada abaixo. Essa variante possui um segundo critério de parada do laço. Sempre que a lista é percorrida no laço mais interno, sem que haja trocas de elementos, podemos concluir que a lista já se encontra ordenada e podemos interromper o processo.

```
1  def ordenacao_bolha(seq):
2    n = len(seq)
3    for i in range(n-1,0,-1):
4      houvetroca = False
5      for j in range(0,i):
6        if seq[j] > seq[j+1]:
7          troca(seq, j, j+1)
8          houvetroca = True
9      if not houvetroca:
10       break
```

Observação:

A método `sort()` da classe `list` disponível no Python faz ordenação:

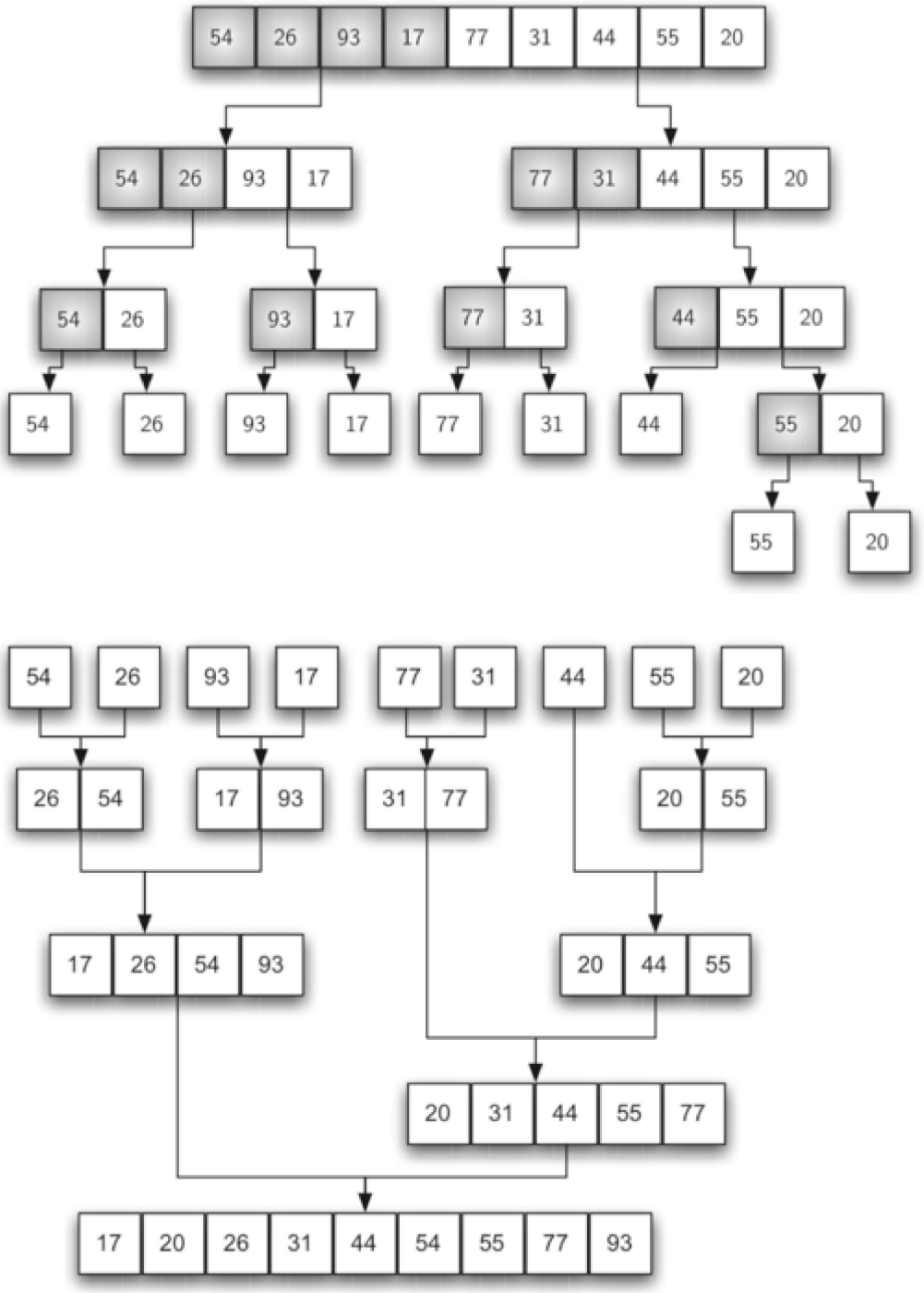
```
>>> L = [8,3,7,1,2]
>>> L.sort()
>>> print(L)
[1, 2, 3, 7, 8]
```

Algoritmo de Ordenação por Intercalação (*Merge Sort*)

O *Merge Sort* é um algoritmo de ordenação recursivo que usa a estratégia “dividir para conquistar”. O algoritmo funciona do seguinte modo:

- Se a lista estiver vazia ou tiver um único item, ela está ordenada por definição (esse é o caso base).
- Se a lista tiver mais de um item, dividir a lista em duas e chamar recursivamente o merge sort em ambas as metades.
- Depois que as metades estiverem ordenadas, intercalá-las (como feito no problema 3 da semana 7) para obter a lista ordenada.

A figura abaixo mostra uma lista sendo dividida em listas menores pelo *Merge Sort*. A figura seguinte mostra como as listas menores ordenadas são intercaladas.



Fonte das imagens: Resolução de Problemas com Algoritmos e Estruturas de Dados usando Python

Para entender o algoritmo assista a animação dele com dança folclórica da Transilvânia.

Problema 4:

Faça uma função recursiva para a ordenar uma lista de números com o algoritmo *Merge Sort*.

Solução:

Dividimos ao meio a lista e ordenamos as duas metades via duas chamadas recursivas. A ordenação final é então obtida via intercalação dos valores das duas metades ordenadas.

```
def mergeSortAux(L, inic, fim):
    if fim - inic + 1 <= 1:
        return
    m = (inic + fim)//2
    mergeSortAux(L, inic, m)
    mergeSortAux(L, m+1, fim)
    A = []
    i = inic
    j = m+1
    while i <= m or j <= fim:
        if i > m:
            A.append(L[j])
            j += 1
        elif j > fim:
            A.append(L[i])
            i += 1
        elif L[i] < L[j]:
            A.append(L[i])
            i += 1
        else:
            A.append(L[j])
            j += 1
    for k in range(inic, fim+1):
        L[k] = A[k-inic]


def mergeSort(L):
    n = len(L)
    mergeSortAux(L, 0, n-1)
```