BCM0505-22 – Processamento da Informação

Vetores - Parte 3

Maycon Sambinelli
m.sambinelli@ufabc.edu.br
http://professor.ufabc.edu.br/~m.sambinelli/

Outline

Introdução

Vetor contido em outro

Ordenação

Hierarquia Corporativa

Busca Binária

Exercícios

Introdução

Mais vetores

Nessa aula vamos ver exemplos com usos mais elaborados de vetores

Vetor contido em outro

Verificando se um vetor está contido em outro

Faça um programa que receba dois conjuntos A e B e verifica se $A \subseteq B$.

Verificando se um vetor está contido em outro

Faça um programa que receba dois conjuntos A e B e verifica se $A \subseteq B$.

```
conjA = le_vetor()
conjB = le_vetor()

contido = True
for i in range(len(conjA)):
    elem_existe = False
    for j in range(len(conjB)):
        if conjA[i] == conjB[j]:
            elem_existe = True
    if not elem_existe:
        contido = False

resp = ""
if not contido:
    resp = "não"
print("O conjunto A {resp} está contido no conjunto B")
```

Verificando se um vetor está contido em outro (v2)

```
1  conjA = le_vetor()
2  conjB = le_vetor()
3
4  contido = True
5  for i in range(len(conjA)):
6      if busca(conjB, conjA[i]) == -1:
7          contido = False
8
9  resp = ""
10 if not contido:
11     resp = "não"
12 print("O conjunto A {resp} está contido no conjunto B")
```

Verificando se um vetor está contido em outro (v3)

```
1  conjA = le_vetor()
2  conjB = le_vetor()
3
4  contido = True
5  for elem in conjA:
6      if elem not in conjB:
7          contido = False
8
9  resp = ""
10 if not contido:
11     resp = "não"
12 print("O conjunto A {resp} está contido no conjunto B")
```

Ordenação

Ordenação

Um problema de programação recorrente é o de ordenar uma coleção de elementos segundo algum critério.

Dizemos que uma sequência de números $a_1, a_2, \ldots, a_n$ está em ordem:

- crescente se $a_1 < a_2 < a_3 < \cdots < a_n$

Ordenação

Um problema de programação recorrente é o de ordenar uma coleção de elementos segundo algum critério.

Dizemos que uma sequência de números $a_1, a_2, \ldots, a_n$ está em ordem:

- crescente se $a_1 < a_2 < a_3 < \cdots < a_n$
- não decrescente se $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n$

Ordenação

Um problema de programação recorrente é o de ordenar uma coleção de elementos segundo algum critério.

Dizemos que uma sequência de números $a_1, a_2, \ldots, a_n$ está em ordem:

- crescente se $a_1 < a_2 < a_3 < \cdots < a_n$
- não decrescente se $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n$
- decrescente se $a_1 > a_2 > a_3 > \cdots > a_n$

Ordenação

Um problema de programação recorrente é o de ordenar uma coleção de elementos segundo algum critério.

Dizemos que uma sequência de números $a_1, a_2, \ldots, a_n$ está em ordem:

- crescente se $a_1 < a_2 < a_3 < \cdots < a_n$
- não decrescente se $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n$
- decrescente se $a_1 > a_2 > a_3 > \cdots > a_n$
- não crescente se $a_1 \geq a_2 \geq a_3 \geq \cdots \geq a_n$

Ordenação

Um problema de programação recorrente é o de ordenar uma coleção de elementos segundo algum critério.

Dizemos que uma sequência de números $a_1, a_2, \ldots, a_n$ está em ordem:

- crescente se $a_1 < a_2 < a_3 < \cdots < a_n$
- não decrescente se $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n$
- decrescente se $a_1 > a_2 > a_3 > \cdots > a_n$
- não crescente se $a_1 \geq a_2 \geq a_3 \geq \cdots \geq a_n$

Ordenação

Um problema de programação recorrente é o de ordenar uma coleção de elementos segundo algum critério.

Dizemos que uma sequência de números $a_1, a_2, \ldots, a_n$ está em ordem:

- crescente se $a_1 < a_2 < a_3 < \cdots < a_n$
- não decrescente se $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n$
- decrescente se $a_1 > a_2 > a_3 > \cdots > a_n$
- não crescente se $a_1 \geq a_2 \geq a_3 \geq \cdots \geq a_n$

Exercício: Escreva uma função `ordena(v)` que recebe um vetor `v` de números reais e orderne `v` em ordem não decrescente.
A ordenação deve ocorrer no próprio vetor `v`, ou seja, nenhum vetor novo deve ser criado.

Ordenação

Ideia:

- Encontrar o menor elemento do vetor `v[0:]` e colocá-lo na entrada `v[0]`

Este algoritmo é chamado de *Selection Sort*

Ordenação

Ideia:

- Encontrar o menor elemento do vetor `v[0:]` e colocá-lo na entrada `v[0]`
  - *Vamos usar a notação slice apenas para referênciar o subvetor de interesse. O slice do Python cria um vetor novo com os elementos do intervalo especificado, o que é uma operação custosa e desnecessária para o nosso objetivo.*

Este algoritmo é chamado de *Selection Sort*

Ordenação

Ideia:

- Encontrar o menor elemento do vetor `v[0:]` e colocá-lo na entrada `v[0]`
  - *Vamos usar a notação slice apenas para referênciar o subvetor de interesse. O slice do Python cria um vetor novo com os elementos do intervalo especificado, o que é uma operação custosa e desnecessária para o nosso objetivo.*
- Na sequência, encontramos o menor elemento do subvetor `v[1:]` e o colocamos na posição na entrada `v[1]`

Este algoritmo é chamado de *Selection Sort*

Ordenação

Ideia:

- Encontrar o menor elemento do vetor `v[0:]` e colocá-lo na entrada `v[0]`
  - *Vamos usar a notação slice apenas para referênciar o subvetor de interesse. O slice do Python cria um vetor novo com os elementos do intervalo especificado, o que é uma operação custosa e desnecessária para o nosso objetivo.*
- Na sequência, encontramos o menor elemento do subvetor `v[1:]` e o colocamos na posição na entrada `v[1]`
- Na sequência, encontramos o menor elemento do subvetor `v[2:]` e o colocamos na posição na entrada `v[2]`

Este algoritmo é chamado de *Selection Sort*

Ordenação

Ideia:

- Encontrar o menor elemento do vetor `v[0:]` e colocá-lo na entrada `v[0]`
  - *Vamos usar a notação slice apenas para referênciar o subvetor de interesse. O slice do Python cria um vetor novo com os elementos do intervalo especificado, o que é uma operação custosa e desnecessária para o nosso objetivo.*
- Na sequência, encontramos o menor elemento do subvetor `v[1:]` e o colocamos na posição na entrada `v[1]`
- Na sequência, encontramos o menor elemento do subvetor `v[2:]` e o colocamos na posição na entrada `v[2]`
- ...

Este algoritmo é chamado de *Selection Sort*

Ordenação

Ideia:

- Encontrar o menor elemento do vetor `v[0:]` e colocá-lo na entrada `v[0]`
  - *Vamos usar a notação slice apenas para referênciar o subvetor de interesse. O slice do Python cria um vetor novo com os elementos do intervalo especificado, o que é uma operação custosa e desnecessária para o nosso objetivo.*
- Na sequência, encontramos o menor elemento do subvetor `v[1:]` e o colocamos na posição na entrada `v[1]`
- Na sequência, encontramos o menor elemento do subvetor `v[2:]` e o colocamos na posição na entrada `v[2]`
- ...
- Na sequência, encontramos o menor elemento do subvetor `v[i:]` e o colocamos na posição na entrada `v[i]`

Este algoritmo é chamado de *Selection Sort*

Selection Sort

```python
def selection_sort(vetor):
    for i in range(len(vetor) - 1):
        indice_menor = i
        for j in range(i + 1, len(vetor)):
            if vetor[j] < vetor[indice_menor]:
                indice_menor = j
        tmp = vetor[i]
        vetor[i] = vetor[indice_menor]
        vetor[indice_menor] = tmp

vetor = [1, 8, 1, 10, 0, 2, 4, 2]
selection_sort(vetor)
print(vetor)
```

Hierarquia Corporativa

Senta que lá vem história....

Na vastidão de um prédio espelhado com mais de cinquenta andares, funcionava a ABCorp. Lá dentro, entre corredores frios e salas de reuniões intermináveis, era comum que pequenos conflitos surgissem: desde brigas sobre o controle do ar-condicionado até disputas mais sérias sobre decisões de projeto.

Senta que lá vem história....

Na vastidão de um prédio espelhado com mais de cinquenta andares, funcionava a ABCorp. Lá dentro, entre corredores frios e salas de reuniões intermináveis, era comum que pequenos conflitos surgissem: desde brigas sobre o controle do ar-condicionado até disputas mais sérias sobre decisões de projeto.
Normalmente, esses conflitos eram resolvidos com uma boa conversa entre os envolvidos. Mas, quando a tensão subia e o impasse persistia, restava apenas uma saída: acionar o **Árbitro Hierárquico**.

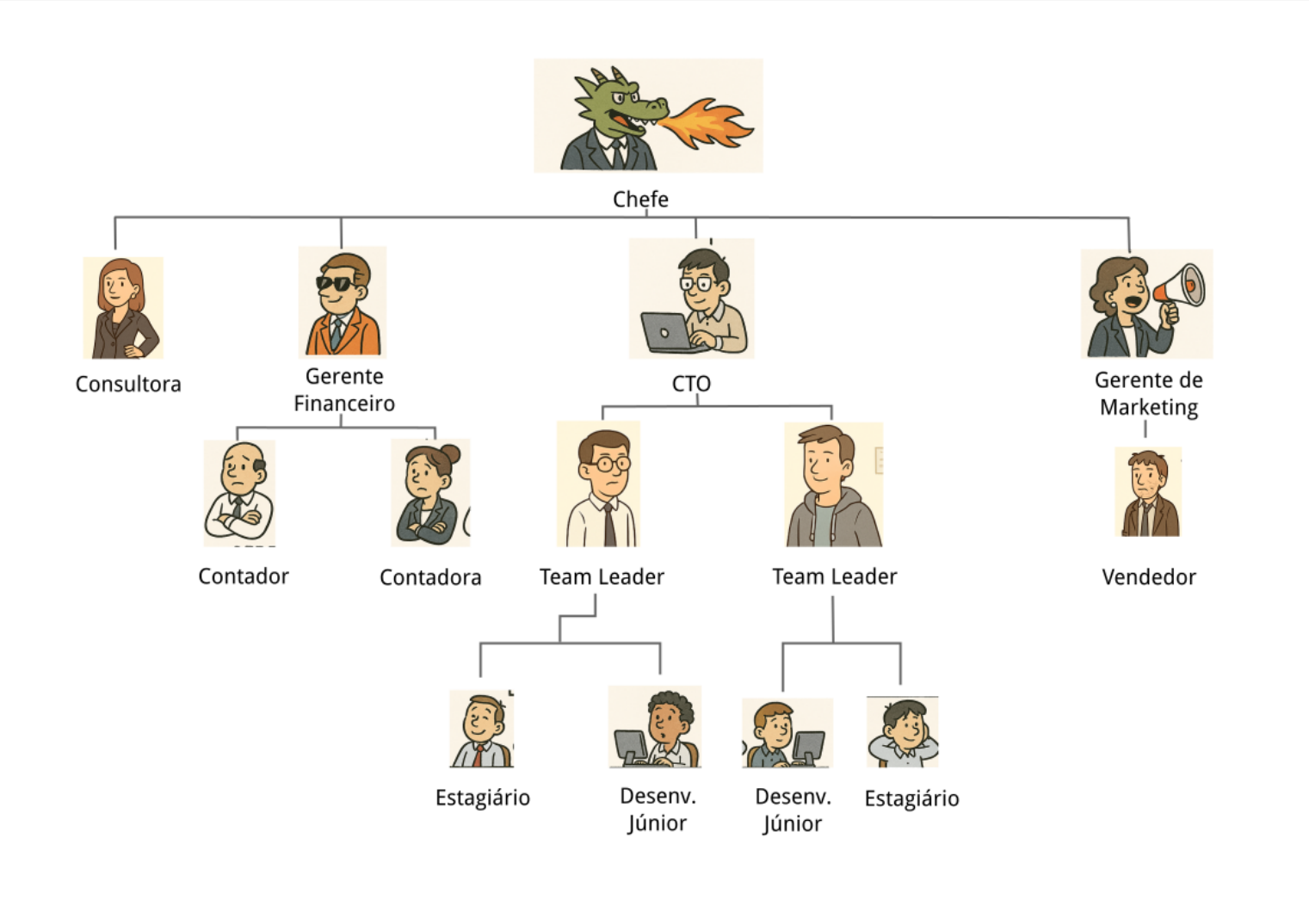
Senta que lá vem história....

Na vastidão de um prédio espelhado com mais de cinquenta andares, funcionava a ABCorp. Lá dentro, entre corredores frios e salas de reuniões intermináveis, era comum que pequenos conflitos surgissem: desde brigas sobre o controle do ar-condicionado até disputas mais sérias sobre decisões de projeto.
Normalmente, esses conflitos eram resolvidos com uma boa conversa entre os envolvidos. Mas, quando a tensão subia e o impasse persistia, restava apenas uma saída: acionar o **Árbitro Hierárquico**.
Na ABCorp, o **Árbitro Hierárquico** de um grupo de funcionários é o superior mais próximo com autoridade sobre todos os envolvidos. Em outras palavras, o menor cargo na hierarquia que ainda estivesse acima de todos os envolvidos no conflito.

Senta que lá vem história....

Na vastidão de um prédio espelhado com mais de cinquenta andares, funcionava a ABCorp. Lá dentro, entre corredores frios e salas de reuniões intermináveis, era comum que pequenos conflitos surgissem: desde brigas sobre o controle do ar-condicionado até disputas mais sérias sobre decisões de projeto.
Normalmente, esses conflitos eram resolvidos com uma boa conversa entre os envolvidos. Mas, quando a tensão subia e o impasse persistia, restava apenas uma saída: acionar o **Árbitro Hierárquico**.
Na ABCorp, o **Árbitro Hierárquico** de um grupo de funcionários é o superior mais próximo com autoridade sobre todos os envolvidos. Em outras palavras, o menor cargo na hierarquia que ainda estivesse acima de todos os envolvidos no conflito.
Em uma empresa com centenas de funcionários, o setor de RH já não dava conta de designar o Árbitro Hierárquico. Por isso, cabe a você, o novo estagiário da ABCorp, criar um programa que, dado um par de funcionários em conflito, identifique corretamente o Árbitro Hierárquico.

Explicação com figurinha...



Solucionando o Problema

Como representar a entrada?

Solucionando o Problema

Como representar a entrada?

- Vamos assumir que a empresa tem n funcionários e que eles são unicamente identificados por números inteiros no intervalo $[0, n-1]$

Solucionando o Problema

Como representar a entrada?

- Vamos assumir que a empresa tem n funcionários e que eles são unicamente identificados por números inteiros no intervalo $[0, n-1]$

Como representar a hierarquia?

Solucionando o Problema

Como representar a entrada?

- Vamos assumir que a empresa tem n funcionários e que eles são unicamente identificados por números inteiros no intervalo $[0, n-1]$

Como representar a hierarquia?

- Vetor!

Solucionando o Problema

Como representar a entrada?

- Vamos assumir que a empresa tem n funcionários e que eles são unicamente identificados por números inteiros no intervalo $[0, n-1]$

Como representar a hierarquia?

- Vetor!
- `organograma[i] = j` se e somente se o superior imediado do funcionário de número `i` for `j`

Solucionando o Problema

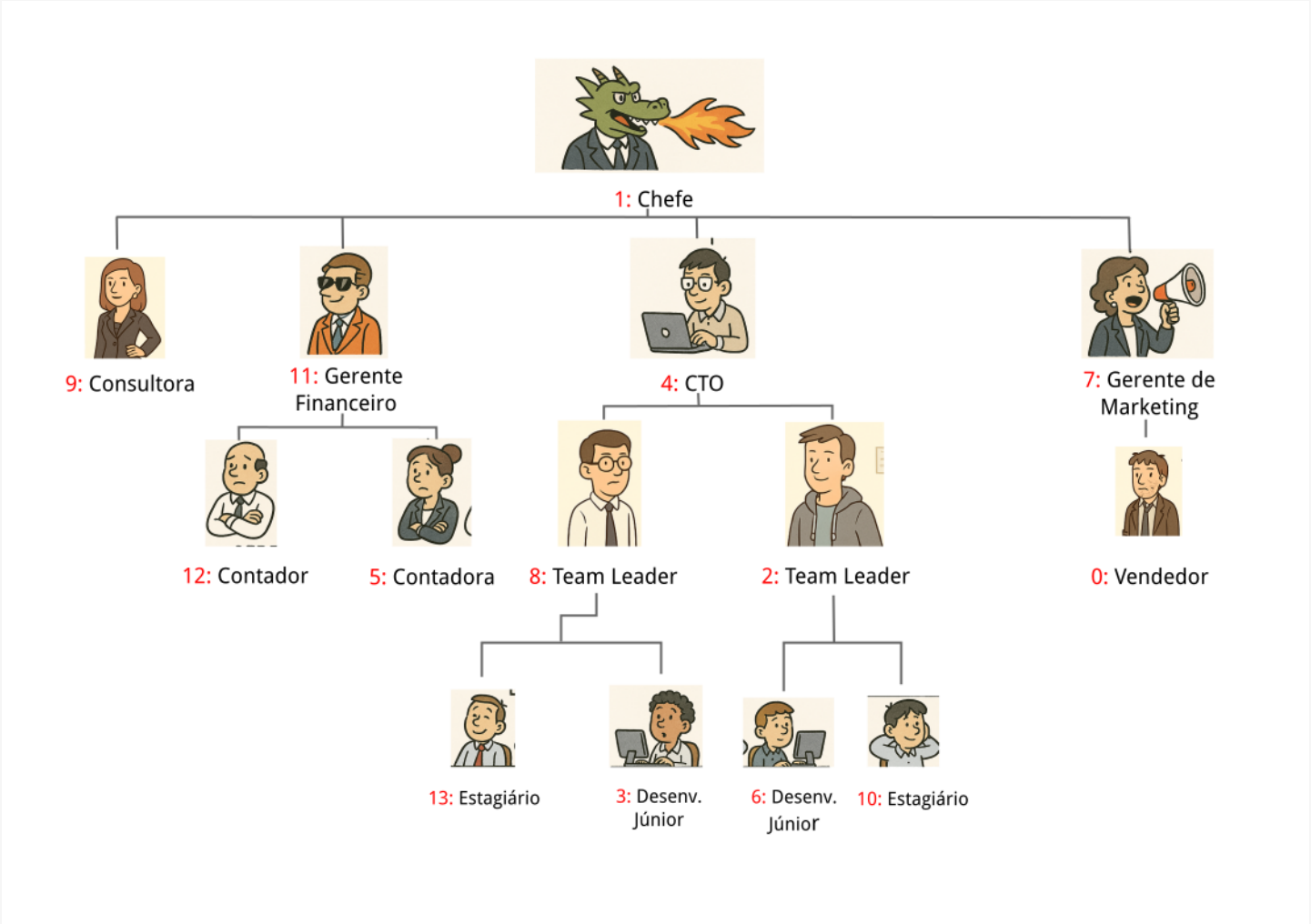
Como representar a entrada?

- Vamos assumir que a empresa tem n funcionários e que eles são unicamente identificados por números inteiros no intervalo $[0, n-1]$

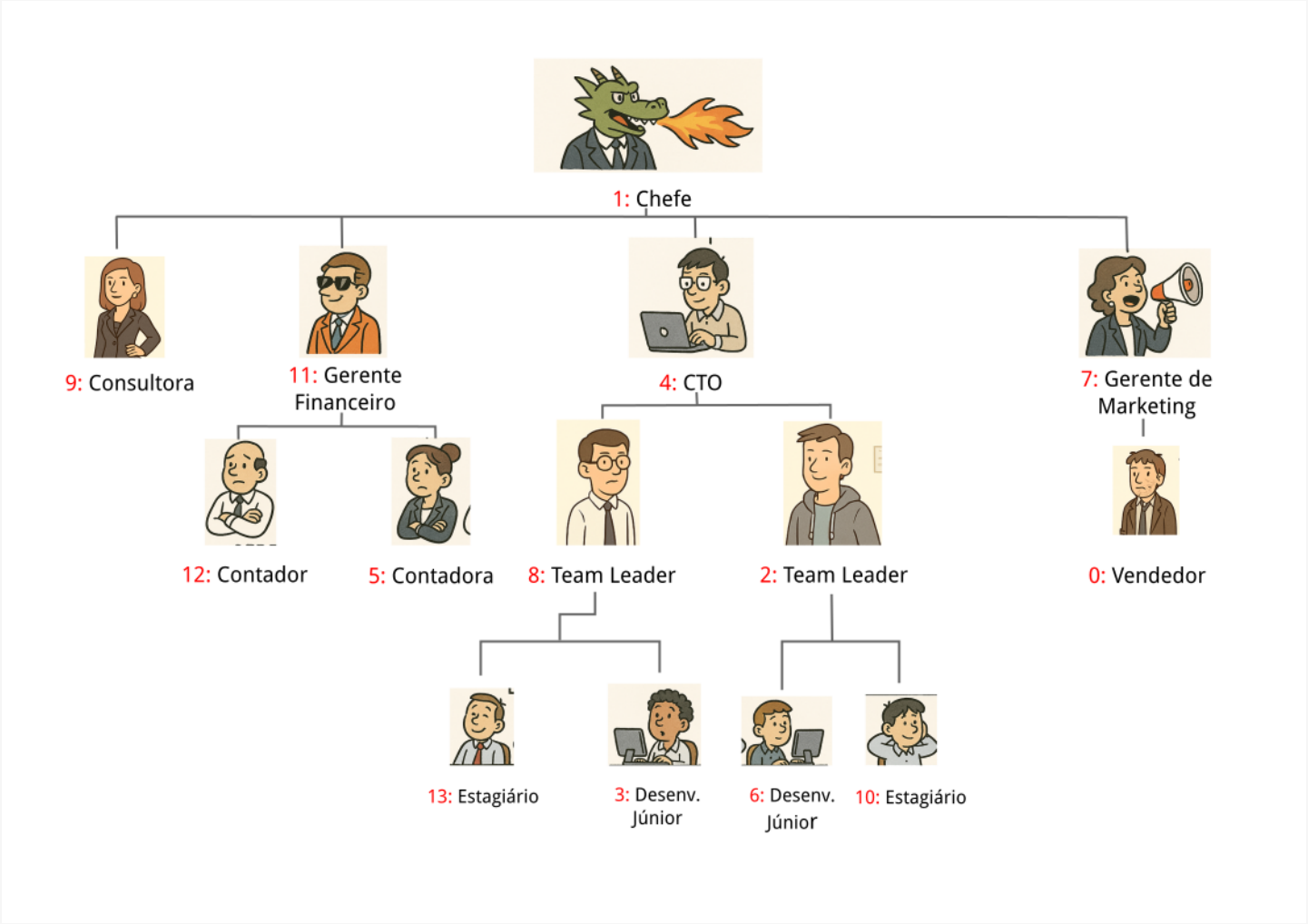
Como representar a hierarquia?

- Vetor!
- `organograma[i] = j` se e somente se o superior imediado do funcionário de número `i` for `j`
- `organograma[i] = i` se e somente se `i` for o chefão (the BOSS)

Explicação com figurinha...



Explicação com figurinha...



0	1	2	3	4	5	6	7	8	9	10	11	12	13
7	1	4	8	1	11	2	1	4	1	2	1	11	8

Solução do problema

```
1  def linha_hierarquica(organograma, id):
2      hierarquia = [id]
3      while organograma[id] != id:
4          hierarquia.append(organograma[id])
5          id = organograma[id]
6      return hierarquia
7
8  def arbitro_hierarquico(organograma, id1, id2):
9      assert(id1 != id2)
10     h1 = linha_hierarquica(organograma, id1)
11     h2 = linha_hierarquica(organograma, id2)
12     i = 0
13     while h1[len(h1)-1 - i]  == h2[len(h2)-1 - i]:
14         i += 1
15     return h1[len(h1) - i]
16
17             #  0  1  2  3  4  5  6  7  8  9  10 11 12
18 organograma = [0, 0, 0, 0, 1, 1, 5, 2, 2, 2, 7, 7, 3]
19
20 print(arbitro_hierarquico(organograma, 9, 10))
```

Busca Binária

Busca

Além do problema de ordenação, outro problema recorrente em computação é o problema de busca. Em um problema de busca, buscamos um dado elemento em uma coleção de elementos.

Busca

Além do problema de ordenação, outro problema recorrente em computação é o problema de busca. Em um problema de busca, buscamos um dado elemento em uma coleção de elementos.

Anteriormente neste curso já resolvemos o problema de busca de um inteiro em um vetor.

```
1  def busca(vet, x):
2      for i in range(len(vet)):
3          if vet[i] == x:
4              return i
5      return -1
```

Busca

Além do problema de ordenação, outro problema recorrente em computação é o problema de busca. Em um problema de busca, buscamos um dado elemento em uma coleção de elementos.

Anteriormente neste curso já resolvemos o problema de busca de um inteiro em um vetor.

```
1  def busca(vet, x):
2      for i in range(len(vet)):
3          if vet[i] == x:
4              return i
5      return -1
```

Note que a função busca pode acabar tendo que acessar todas as n entradas do vetor para determinar se `x` $\in$ `vet`

- Isso quer dizer que o algoritmo pode levar n “passos” para executar

Busca

Além do problema de ordenação, outro problema recorrente em computação é o problema de busca. Em um problema de busca, buscamos um dado elemento em uma coleção de elementos.

Anteriormente neste curso já resolvemos o problema de busca de um inteiro em um vetor.

```
1  def busca(vet, x):
2      for i in range(len(vet)):
3          if vet[i] == x:
4              return i
5      return -1
```

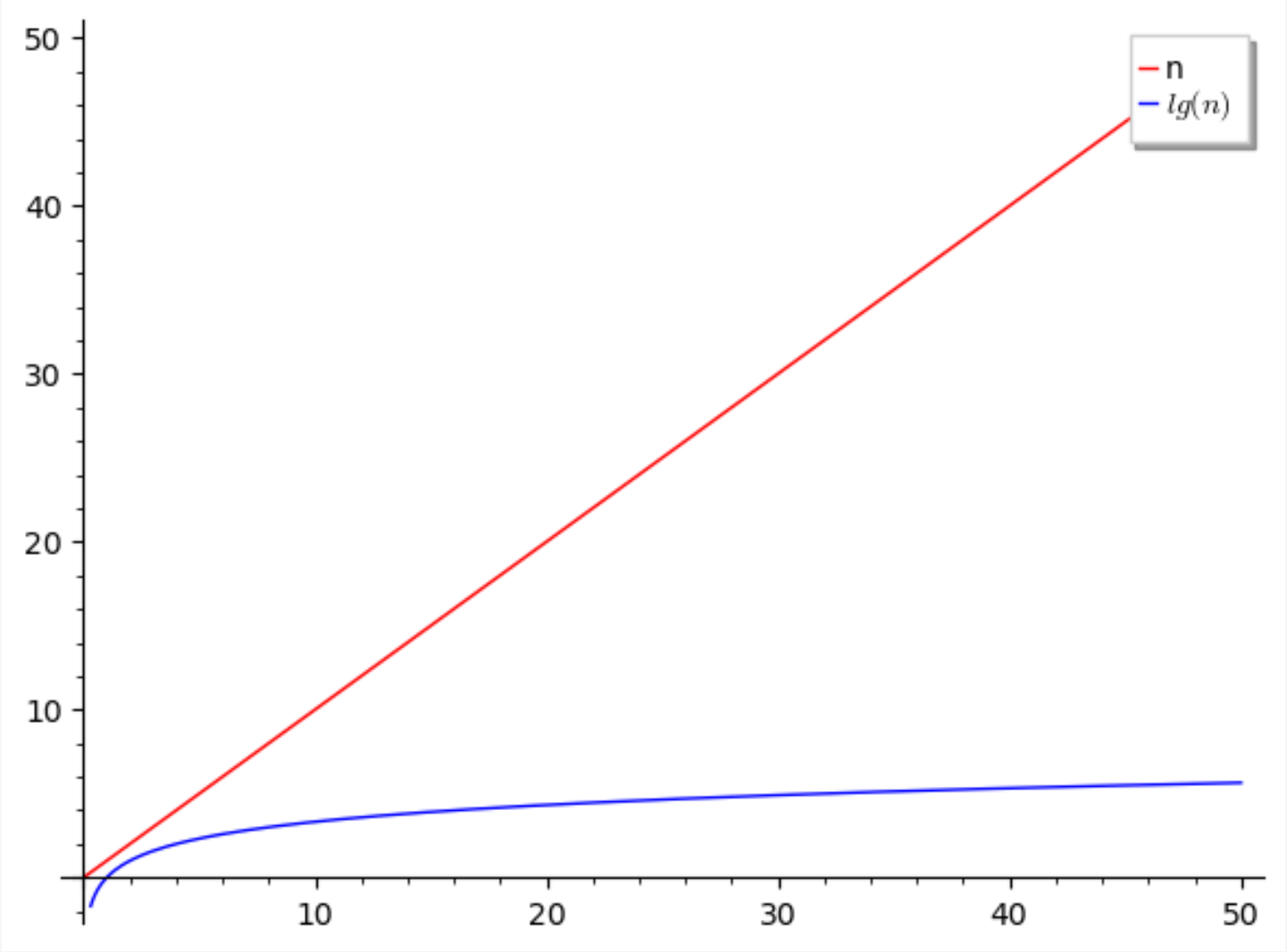
Note que a função busca pode acabar tendo que acessar todas as n entradas do vetor para determinar se `x` $\in$ `vet`

- Isso quer dizer que o algoritmo pode levar n "passos" para executar
- Em análise de algoritmos, diremos que esse código é $O(n)$

Busca

No entanto, se os dados estiverem ordenados, podemos fazer um código muito mais esperto e que pode requerer uma quantidade de "passos" de no máximo $\lg n$

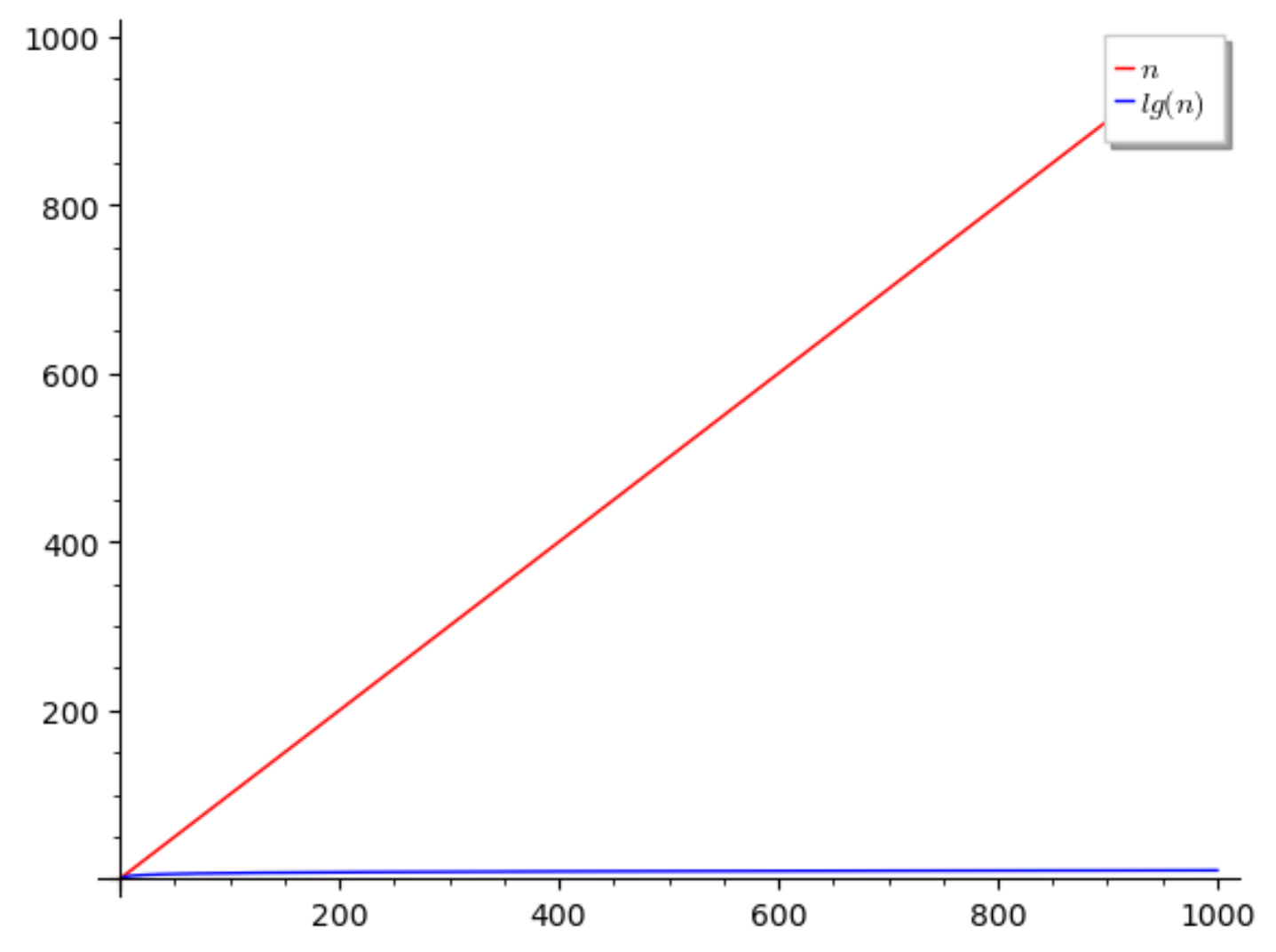
- No jargão de análise de algoritmos: $O(\lg n)$



Busca

No entanto, se os dados estiverem ordenados, podemos fazer um código muito mais esperto e que pode requerer uma quantidade de "passos" de no máximo $\lg n$

- No jargão de análise de algoritmos: $O(\lg n)$



Busca Binária

Ideia:
Vamos manter uma "janela de busca" `A[ini:fim + 1]` de onde *x* pode estar

- Inicialmente:
  - `ini = 0`
  - `fim = n - 1`

Busca Binária

Ideia:
Vamos manter uma "janela de busca" `A[ini:fim + 1]` de onde x pode estar

- Inicialmente:
  - `ini = 0`
  - `fim = n - 1`

Em cada passo, calculamos o meio da janela de busca e analisamos o elemento armazenado em `A[meio]`

- Se `x <= A[meio]`, então se `x` $\in$ `A`, temos que `x` deve pertencer ao `A[0:meio]`

Busca Binária

Ideia:
Vamos manter uma "janela de busca" `A[ini:fim + 1]` de onde *x* pode estar

- Inicialmente:
  - `ini = 0`
  - `fim = n - 1`

Em cada passo, calculamos o meio da janela de busca e analisamos o elemento armazenado em `A[meio]`

- Se `x <= A[meio]`, então se `x` $\in$ `A`, temos que `x` deve pertencer ao `A[0:meio]`
- Se `x > A[meio]`, então se `x` $\in$ `A`, então `x` deve pertencer ao intervalo `A[meio+1:]`

Busca Binária

Ideia:
Vamos manter uma "janela de busca" `A[ini:fim + 1]` de onde *x* pode estar

- Inicialmente:
  - `ini = 0`
  - `fim = n - 1`

Em cada passo, calculamos o meio da janela de busca e analisamos o elemento armazenado em `A[meio]`

- Se `x <= A[meio]`, então se `x` $\in$ `A`, temos que `x` deve pertencer ao `A[0:meio]`
- Se `x > A[meio]`, então se `x` $\in$ `A`, então `x` deve pertencer ao intervalo `A[meio+1:]`
- Note que estamos diminuindo a janela de busca pela metade em cada iteração

Busca Binária

Exemplo:

0	1	2	3	4	5	6	7	8	9	10
1	2	7	10	15	20	21	22	27	32	35

Código

```python
def busca_binaria(A, x):
    ini = 0
    fim = len(A) - 1

    while ini < fim:
        meio = (ini + fim) // 2
        if x <= A[meio]:
            fim = meio
        else:
            ini = meio + 1

    if fim >= 0 and A[ini] == x:
        return ini
    else:
        return -1

S = [1, 2, 7, 10, 15, 20, 21, 22]
print(busca_binaria(S, 7))
```

Exercícios

Exercícios

1. Adapte o código do Selection Sort para ordenar os elementos em ordem não crescente.
2. Escreva uma função que recebe um vetor que já está *ordenado* e um elemento, e insira-o no vetor, mantendo-o ordenado (pode haver repetições).
3. Escreva uma função que, dado o organograma da ABCorp e um id de um funcionário, devolve o número de subordinads do funcionário cujo id foi fornecido.
4. O nível de um funcionário dentro da ABCorp é definido como o número de supervisores. Escreva uma função que, dado o organograma da ABCorpo e um id de um funcionário, devolve o nível do funcionário cujo id foi fornecido.