TRYBE
Modulo IV – Ciência da Computação

Bloco 37: Algoritmos e estrutura de dados

1) Estrutura de Dados I - Arrays

Tipo de dado VS. estrutura de dado.

Tipo de dado: conjunto de valores que uma variável ou constante pode assumir.
Estrutura de dado (ED / DS): coleção tanto de **valores** quanto de **operações**. Implementações de tipos abstratos de dados.
Tipos abstratos de dados (TAD / ADT): é a ideia do dado, ou seja a forma de **classificar estruturas de dados** com base em **usos e comportamentos** que fornecem. Eles não especificam como a estrutura de dados deve ser implementada, mas simplesmente fornecem uma interface mínima esperada e um conjunto de comportamentos. *(Lembra classes abstratas e interface de OO).*

Por que arrays?

Array é um **TAD que possui uma coleção de elementos que são acessados através do índice**.

Implementação estática VS. dinâmica: define um valor fixo de tamanho e não pode ser modificada durante a execução do programa / permite que cresça à medida que elementos são inseridos.

Entendendo a estrutura

Falando aqui da ED Array (ou seja implementação do TAD).

Entender **como cresce**:
"Quando há um crescimento, um novo endereço na memória é reservado para um nova lista. Em seguida, os elementos são copiados da lista original para a nova, o novo elemento é adicionado já nesse espaço de memória" e finalmente *"o nome original é atribuído a nova lista"*.

Como **elemento é adicionado segundo o posicionamento**:
"Quando adicionamos em uma posição intermediária, todos os elementos com índices posteriores ao inserido serão movidos em uma posição." Indice +1. O mesmo acontece removindo.

Arrays multidimensionais

Pode ser útil para modelagens de matrizes, tabuleiros em jogos, tabelas de dados.

Arrays no Pythonverso

→ Module array – vem na linguagem Python

```
import sys
from array import array

# define um array vazio de inteiros sem sinal
myarray = array("I")

# podemos adicionar alguns valores
myarray.insert(0, 5)  # na posição 0 o valor 5
myarray.insert(1, 3)
myarray.insert(2, 5)
print("Após adicionar alguns valores: ", myarray)

# adicionar em uma posição intermediária
myarray.insert(1, 4)
print("Após inserção em índice intermediário: ", myarray)

# remover um valor através do índice
myarray.pop(0)
print("Após remover um valor:", myarray)

# compare o tamanho entre ua lista e um array
elements = list(range(100)) # definimos uma lista de 100 números
print("Tamanho da lista:", sys.getsizeof(elements))
print("Tamanho do array", sys.getsizeof(array("I", elements)))
```

→ Module numpy para n-dimensional - *python3 -m pip install numpy*

Usado para analisar dados (Pandas) e machine learning (scikit-learn)

```
import numpy as np

# define um array vazio de inteiros
myarray = np.array([], dtype=int)

# podemos adicionar alguns valores
myarray = np.insert(myarray, 0, 5)  # na posição 0 o valor 5
myarray = np.insert(myarray, 1, 3)
myarray = np.insert(myarray, 2, 5)
print("Após adicionar alguns valores: ", myarray)

# adicionar em uma posição intermediária
myarray = np.insert(myarray, 1, 4)
print("Após inserção em índice intermediário: ", myarray)

# remover um valor através do índice
myarray = np.delete(myarray, 0)
print("Após remover um valor:", myarray)
```

Dicas diversas

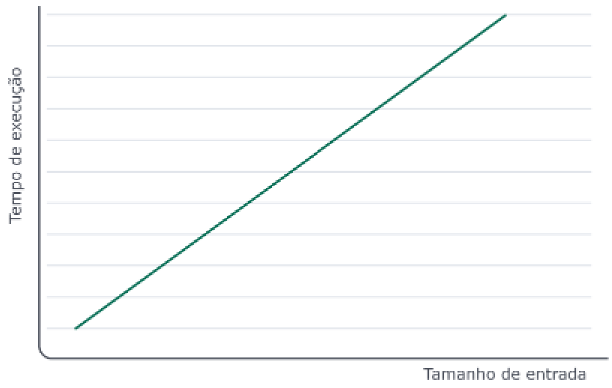
→ Em python não existem tipos primitivos, tudo é objeto.

----

2) Estrutura de Dados I - Complexidade de Algoritmos

Ordem de complexidade

O **tempo de execução** do algortimo é **proporcional ao tamanho do dado** entrado, ou seja varia na medida em que a entrada cresce.



Na medida em que o tamanho da entrada cresce, o tempo de execução cresce na mesma proporção

Função matemática que representa uma relação linear: ***f(n) = n*** *.*
Sequential search*.*

Complexidade de tempo e de espaço

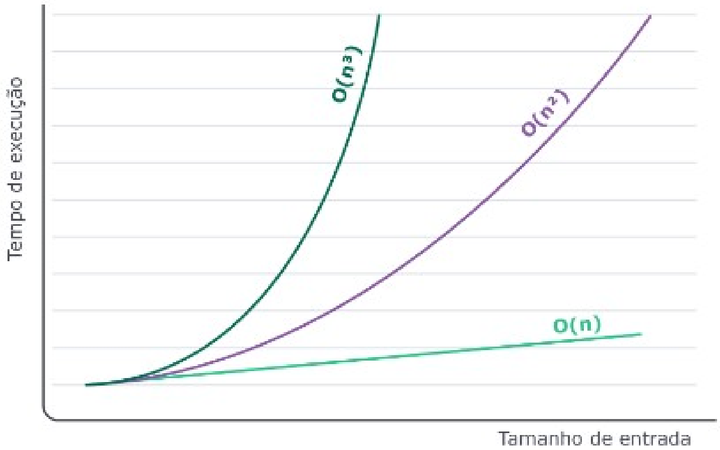
Complexidade de **tempo** é sobre tempo para **executar**.
Complexidade de **espaço** é sobre **memória ocupada**.
Exemplos: é de O(n) se for proporcional que nem ordem de complexidade, ou 0(1) se algoritmo usa mesma unidade única de memória independentemente do tamanho da entrada.

Complexidade quadrática

Caso de **O(n^2).**

```
# Os arrays tem sempre o mesmo tamanho
def multiply_arrays(array1, array2):
    result = []
    for number1 in array1:
        for number2 in array2:
            result.append(number1 + number2)

    return result

# def multiply_array(numbers):
  # ...

sum_array(array_com_dois_mil_numeros)
# O tempo de execução deste algoritmo foi 0.45s

sum_array(array_com_quatro_mil_numeros)
# Já esse teve tempo de execução de 1.8s, quatro vezes maior → 2x mais dados e 2x arrays
```

Comparando complexidades



Observe no gráfico a relação entre as ordens de complexidade linear, quadrática e cúbica

$O(n^3)$ executa em mais tempo com entrada de tamanho menor.

Complexidade logarítmica

O(log n) - *número de vezes que n deve ser dividido por 2 para obter 1.*
Um algoritmo logarítmico geralmente reduz pela metade o tamanho do input a cada iteração.
Binary search (exemplo de procurar nome em dictionário).

Complexidade exponencial e fatorial

Algoritmos que, para aumentos pequenos no tamanho da entrada, aumentam enormemente o seu tempo de execução. Usados apenas na **ausência de outras possibilidades**.

Exponencial: 2^n ;
Fatorial: n! .

Exemplo: algoritmo de "força bruta" (explorar todas cenários).
Categoria de na computação chamada **problemas NP Completos.**

Analisando algoritmos com várias estruturas de repetição

Casos onde o algoritmo tem algoritmos de **diversos graus de complexidade aninhados**. Consideramos que a **complexidade menor é desprezível**, então a omitimos. Privilegiar de analisar o algoritmo com complexidade maior.

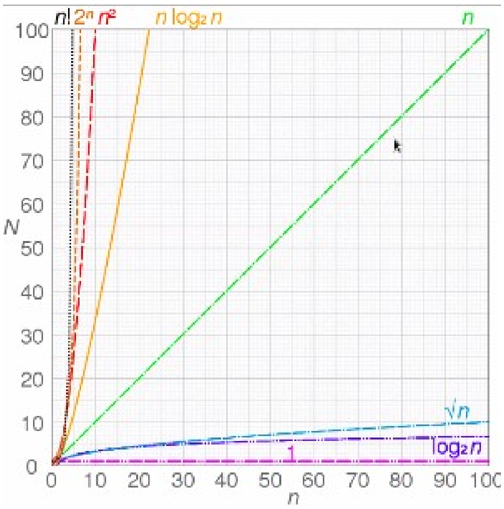
Melhor caso, pior caso e caso médio

Quando um algoritmo pode ser O(1), O(n * ½) ou O(n).

```
def linear_search(numbers, n):
    for index, number in enumerate(numbers):
        if(number == n): return index

    return -1

print(linear_search([1, 2, 3, 4, 5], 4))
```

Em suma

Ordens de complexidade – combinações.



- **Constantes:** `O(1)` ;
- **Logarítmicos:** `O(log n)` ;
- **Linear:** `O(n)` ;
- **Quadráticos:** `O(n²)` ;
- **Cúbicos:** `O(n³)` ;
- **Exponencial:** `O(2ⁿ)` ;
- **Fatorial:** `O(n!)` .

<u>Dicas diversas</u>

→ [Virtual DOM](), promovido pelo React. Ideia de comparar primeiro e depois chegar no front.

---

3) Recursividade e Estratégias para solução de problemas

Recursividade

Para resolver **problemas grandes** segundo um esquema de bonecas russas.

Leis da recursão

Um algoritmo recursivo deve obedecer três regras:
1/ Ter caso base: condição de parada do algoritmo recursivo e menor subproblema do problema;
2/ Mudar o seu estado e se aproximar do case base;
3/ Chamar a si mesmo recursivamente.

Praticar recursividade

Nome da função e parâmetro:
Condição de parada
Chamada de si mesma

```
def countdown(n):  # nome da função e parâmetro
    if n == 0:  # condição de parada
        print('FIM!')
    else:
        print(n)
        countdown(n - 1)  # chamada de si mesma com um novo valor


countdown(5)
```

Pilha de execução envolvida:

- Toda vez que chamamos uma função em um programa, o sistema operacional reserva memória para as variáveis e parâmetros da função;
- Sempre quando uma função é executada, ela é guardada na pilha;
- Quando a função termina de ser executada, ela sai da pilha.

Iteratividade x Recursividade

```
def iterative_countdown(n):
  while n > 0:
     print(n)
     n = n - 1
  print("FIM!")

  return n


iterative_countdown(5)
```

Em certos casos o code recursivo é menos complexo que iterar, ganhando legibilidade e elegância. Beneficio sobre o **desempenho da pessoa programadora > o desempenho do programa**.

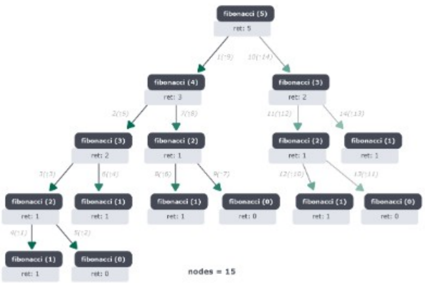
Análise de algoritmos recursivos

Equações de recorrência

Expõem o **custo de resolver um problema** qualquer, de tamanho n , em subproblemas menores. Foco na **eficiência (tempo e espaço)**.

<u>4 métodos:</u>

1/ Árvore de recursão



Fluxograma Fibonacci Recursivo

Estima custo. Sendo que cada **nó** representa o custo da solução de um subproblema.

2/ Método iterativo *(método da substituição iterativa)*

Expande o problema para encontrar um padrão, e a partir disso, encontrar um **somatório**.

3/ Método de substituição *(método da substituição indutiva)*

Mais confiável. Dois passos: estimar **hipótese** de solução, provar com **indução matemática**.

4/ Teorema mestre

Calcula recursos necessários para executar um algoritmo recursivo.
Ramificações e **big O**.

T(n) = aT(n / b) + f(n)

// n é tamanho original do problema.
// a e b constantes, a >= 1 pq tem pelo menos um subproblema, b >1 para dividirmos o problema.
// n / b é o tamanho de cada um dos subproblemas.
// função f(n) representa o custo no tempo, de cada chamada recursiva do algoritmo analisado.

Principais cuidados ao usar recursão

Risco de *stack overflow*, ou estouro de pilha em português: ficar sem memória para continuar com a execução do programa. Sempre implementar a **condição de parada** na função.

Estratégias para solução de problemas

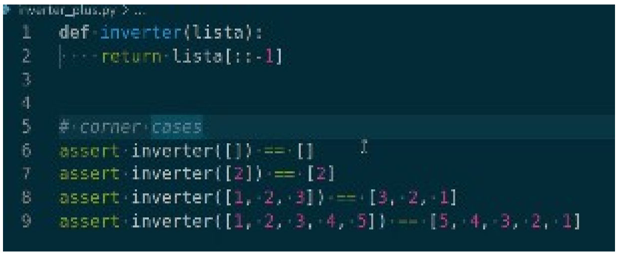
Iterativa
Repetição de uma operação, para resolver via **aproximações sucessivas** – o resultado anterior servindo de entrada.

Força bruta
Enumerar **todas as combinações** possíveis para uma solução.
Cada tentativa avalia se a solução é viável e possui valor melhor que a anterior.

Dividir e conquistar
Dividir o problema em partes menores e combinar as soluções. Ou seja: **divisão, conquista, combinação**.
Técnica particularmente boa para algoritmos paralelos, programa modularizado.

<u>Dicas diversas</u>

→ QA com corner cases



```python
def inverter(lista):
    return lista[::-1]


# corner cases
assert inverter([]) == []
assert inverter([2]) == [2]
assert inverter([1, 2, 3]) == [3, 2, 1]
assert inverter([1, 2, 3, 4, 5]) == [5, 4, 3, 2, 1]
```

---

4) Algoritmos de ordenação e busca

Algoritmos de Ordenação

Buscam colocar **elementos de uma sequência em uma determinada ordem definida**. Esta ordem pode ser **numérica, lexicográfica** ou por qualquer outra característica.

1/ Usando força bruta

Selection Sort
Divide o **array em duas partes**, uma já ordenada e outra de itens a serem ordenados. Ordena um pouco melhor a cada iteração até ter todos elementos ordenados.

Insertion Sort
Insere um elemento de cada vez em sua posição correta.

Ambos têm *complexidade de O(n²)* e complexidade de *tempo constante*, pois sem array auxiliar.

2/ Iterando

Os dois acima também iteravam. (e podem portanto ser reescritos para ser recursivos).

Bubble Sort

A cada iteração, **cada item** se desloca como uma bolha **para a posição** a qual pertence.

3/ Usando o Dividir e Conquistar

Merge Sort

A ordenação por mistura vai **dividindo a coleção em porções menores**, até uma coleção mínima.

Quick Sort

Determina um **elemento pivô** (nome dado ao elemento que divide o array em porções menores). Em seguida, todos os elementos maiores que o pivô serão colocados a direita e os menores a esquerda.

Complexidade de *O(n log n)*. Ou pior caso, $O(n^2)$.

Algoritmos de Busca

Buscam um **item com uma determinada propriedade** dentro de uma coleção.

Busca linear

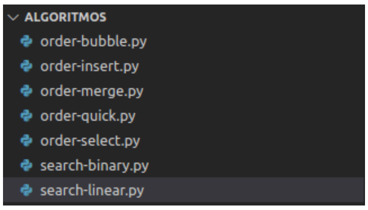
Busca sequencial, percorrendo a estrutura em busca do valor. Simples mas nem sempre mais eficiente.

Busca binária

Supõe que nossa coleção está **ordenada**. Divide a lista em **duas partes** e **verifica se o elemento do meio** é o procurado, até encontrar ele.

Dicas diversas

→ *Enquanto tiver acesso ao course Trybe*, ver gifs animados de cada algoritmo.
Assim como um exemplo em Python de cada algoritmo →



---

4) Projeto - Algoritmos

Dicas diversas do projeto:
TimSort algorithm
