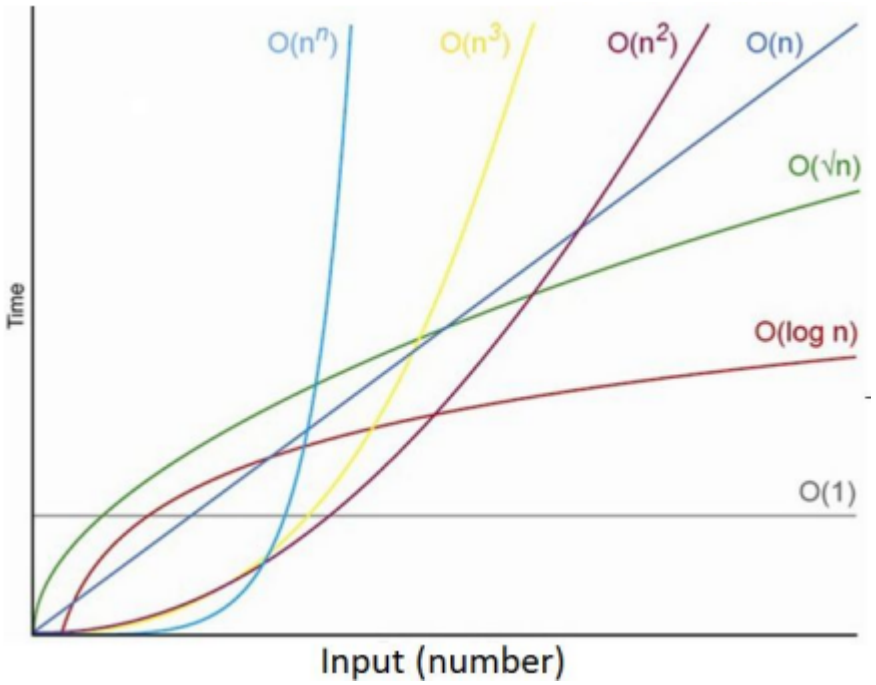
Big O notation

A complexidade algorítmica classifica a utilização de recursos na resolução de tarefas, se preocupando em quão rápido um algoritmo executa uma ação.

Esta definição é dada como uma função **T(n)** sendo **T** o tempo da execução e **n** o tamanho do dado de entrada



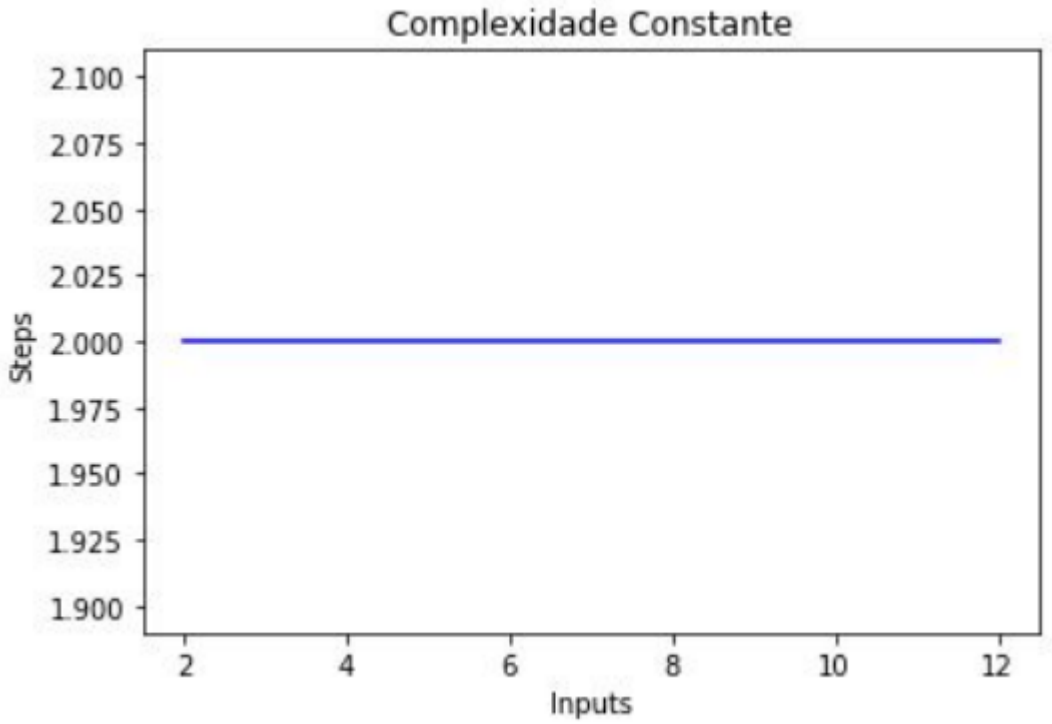
Qual a importância

A importância se dá pois envolve fatores como Performance, Tempo de execução e Dinheiro .

Tipos de complexidade

Complexidade constante (O(c))

1. Acessando um array (int x = Array[0])
2. Inserindo um nó em uma lista
3. Inserindo e retirando nós de pilhas/listas



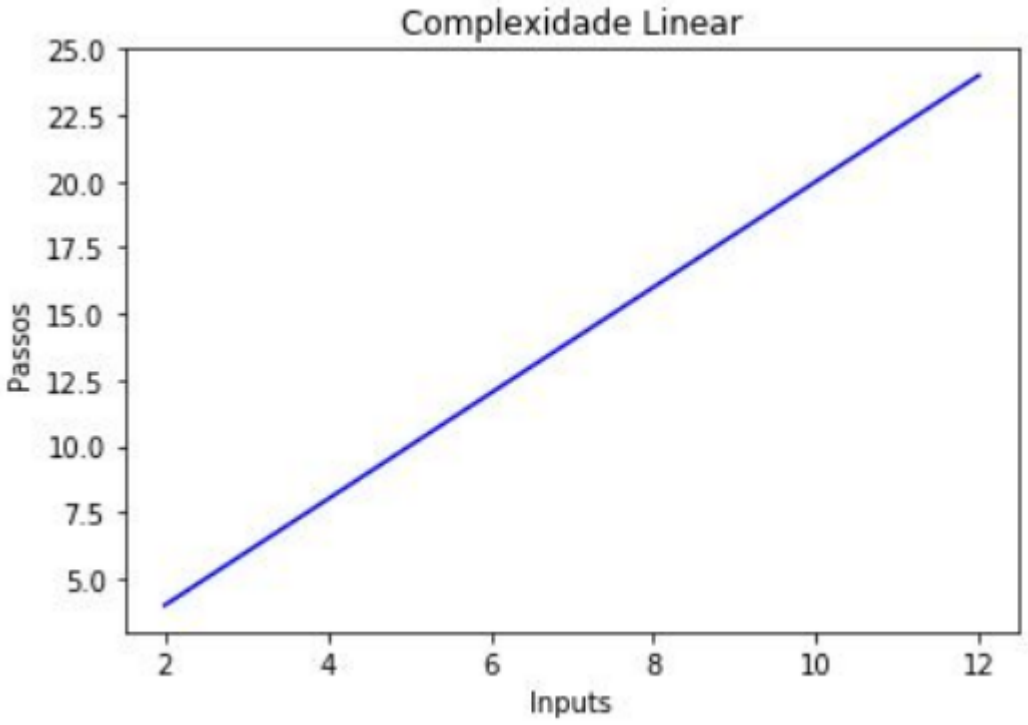
Exemplo

```
def exemplo_complexidade_constante(lista):
    # Acessar o primeiro elemento da lista, independentemente
    primeiro_elemento = lista[0]
    return primeiro_elemento

# Exemplo de uso:
lista = [1, 2, 3, 4, 5]
resultado = exemplo_complexidade_constante(lista)
print("Resultado:", resultado)
```

Complexidade Linear (O(n))

1. Pesquisa Linear(for)
2. Deleção de elemento específico em uma lista encadeada(não ordenada)
3. Comparação de duas strings



Exemplo

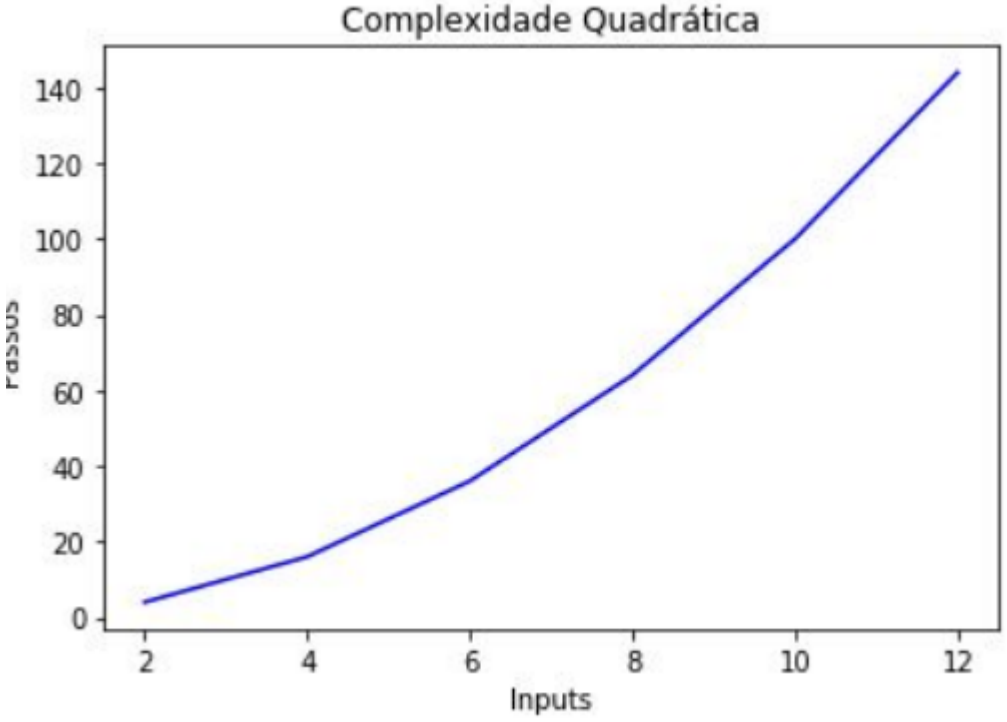
```
def exemplo_complexidade_linear(lista):
    # Percorrer cada elemento da lista uma vez
    for elemento in lista:
        print(elemento)

# Exemplo de uso:
lista = [1, 2, 3, 4, 5]
exemplo_complexidade_linear(lista
```

Complexidade quadrática (O(n^2))

1. Insertion sort

2. Selection sort
3. Bubble sort
4. Quicksort



Exemplo

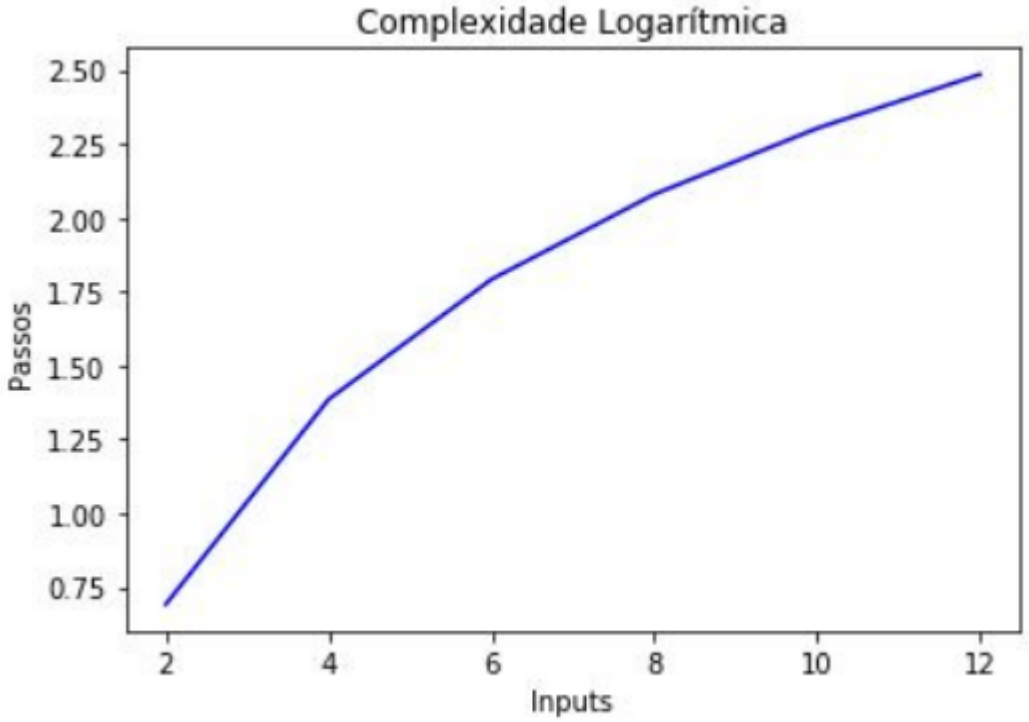
```
def exemplo_complexidade_quadratica(lista):
    # Percorrer cada par de elementos da lista usando dois lo
    for i in range(len(lista)):
        for j in range(len(lista)):
            print(lista[i], lista[j])

# Exemplo de uso:
lista = [1, 2, 3, 4, 5]
exemplo_complexidade_quadratica(lista
```

Complexidade logarítmica (O(log(n)))

1. Busca binária
2. Encontrando maior/menor em uma árvore de busca binária

3. Alguns algoritmos de dividir e conquistar baseados em funcionalidade linear
4. Calculando Fibonacci



Exemplo

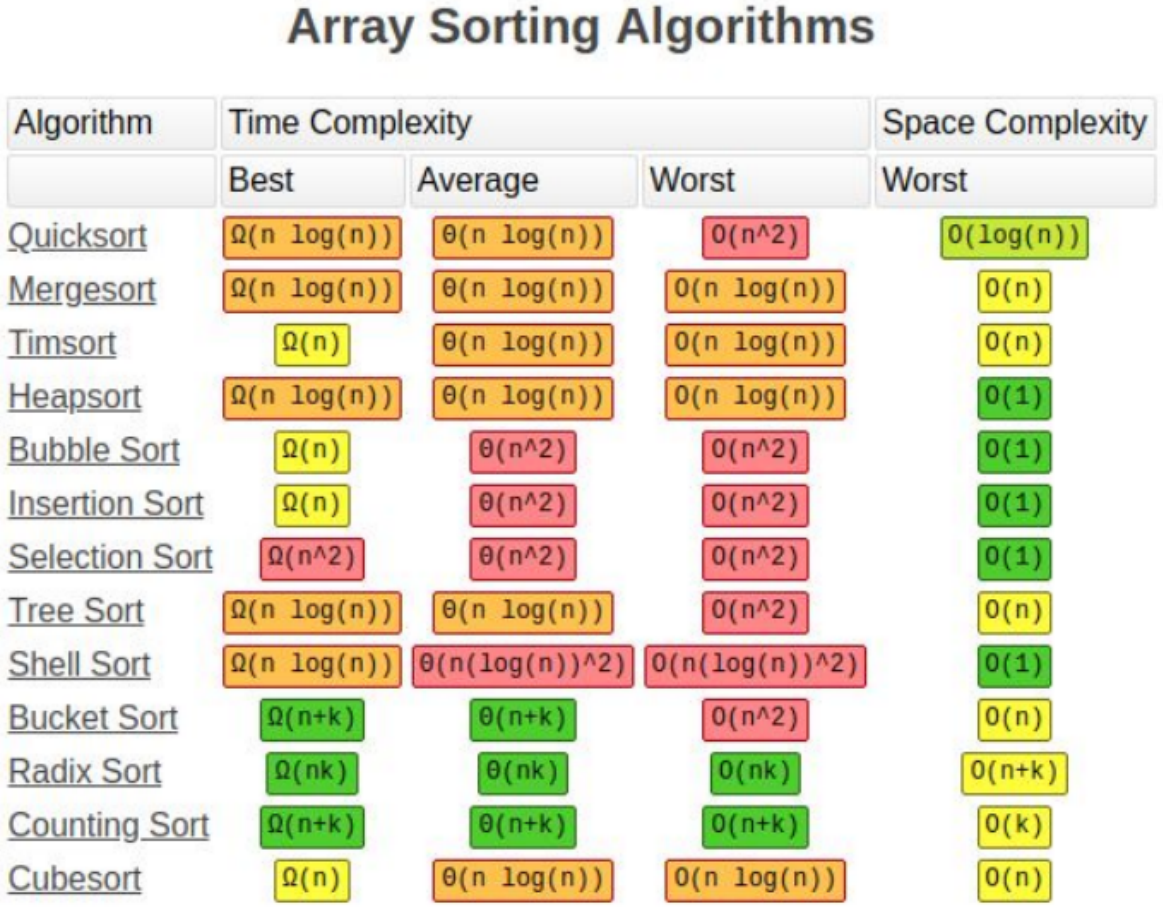
```
def busca_binaria(lista, alvo):
    # Definir os índices iniciais e finais
    inicio, fim = 0, len(lista) - 1

    while inicio <= fim:
        meio = (inicio + fim) // 2  # Calcular o índice do me

        if lista[meio] == alvo:
            return meio  # Elemento encontrado, retorna o índ
        elif lista[meio] < alvo:
            inicio = meio + 1  # Procurar na metade direita
        else:
            fim = meio - 1  # Procurar na metade esquerda

    return -1  # Elemento não encontrado
```

```
# Exemplo de uso:
lista_ordernada = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
alvo = 7
indice = busca_binaria(lista_ordernada, alvo)
print("Índice do alvo:", indice)
```



Fato interessante

Alguns dos algoritmos de busca comuns que podem ser usados em um site de e-commerce incluem:

1. **Busca Sequencial:** Este é um método simples em que os itens são verificados um por um até encontrar o desejado. Pode ser eficaz para

conjuntos de dados pequenos, mas não é tão eficiente para grandes conjuntos de dados.

2. **Busca Binária:** Se os dados estiverem ordenados, a busca binária pode ser uma escolha eficiente. No entanto, a busca binária exige que os dados estejam ordenados.
3. **Índices e Estruturas de Dados Otimizadas:** Em sistemas mais complexos, é comum usar índices e estruturas de dados otimizadas, como árvores de busca binária, árvores B ou tabelas de hash, dependendo dos requisitos específicos.
4. **Motores de Busca em Texto Completo:** Para buscas mais avançadas, especialmente em descrições de produtos e conteúdos textuais, motores de busca em texto completo (como Elasticsearch ou Solr) podem ser utilizados.
5. **Algoritmos de Recomendação:** Em alguns casos, a busca pode ser personalizada usando algoritmos de recomendação que levam em consideração o histórico de navegação e as preferências do usuário.