Análise de Algoritmos

Estruturas de Dados

Universidade de Brasília
Departamento de Ciência da Computação

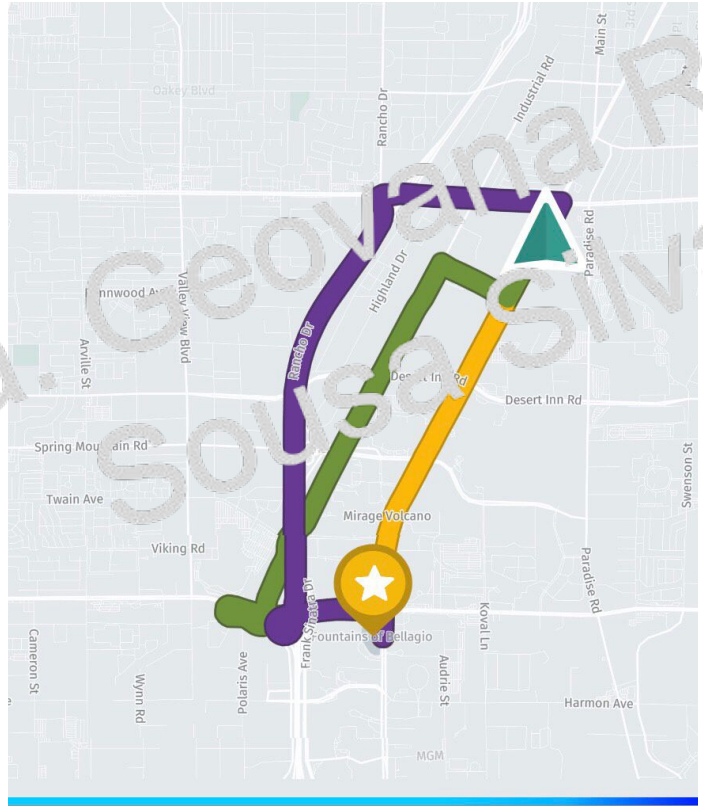
Algoritmo

Um algoritmo são instruções não ambíguas passo a passo para resolver um determinado problema.

Análise de Algoritmos

Comparar algoritmos (ou soluções) principalmente em termos de tempo de execução, mas também em termos de outros fatores (por exemplo, memória, esforço do desenvolvedor, etc.)

Análise de Algoritmos



Análise de Tempo de Execução

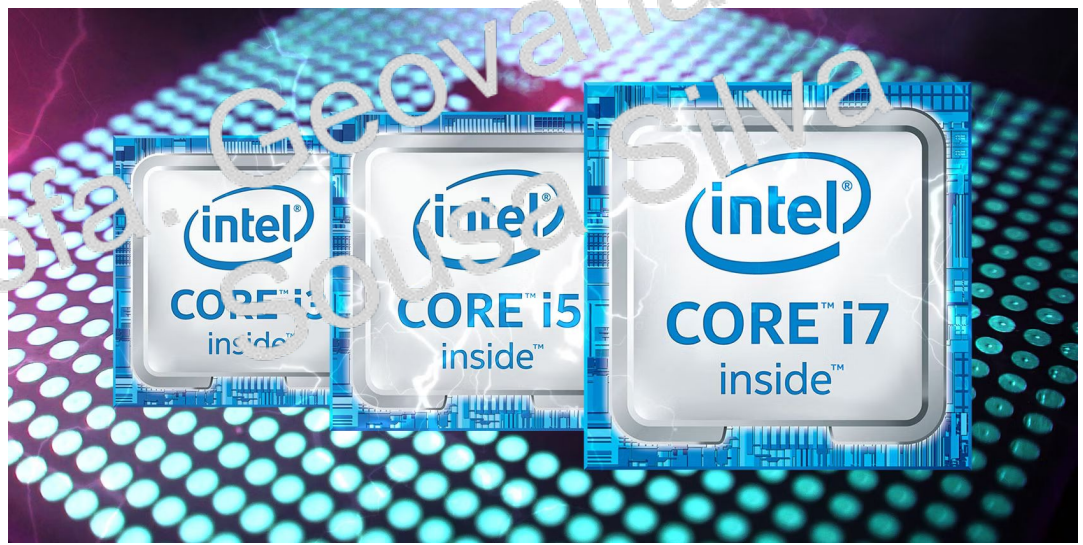
Determinar como o tempo de processamento cresce à medida que o tamanho da entrada aumenta.

Parâmetros de Análise

- “Tempo” de execução?
- Número de instruções?

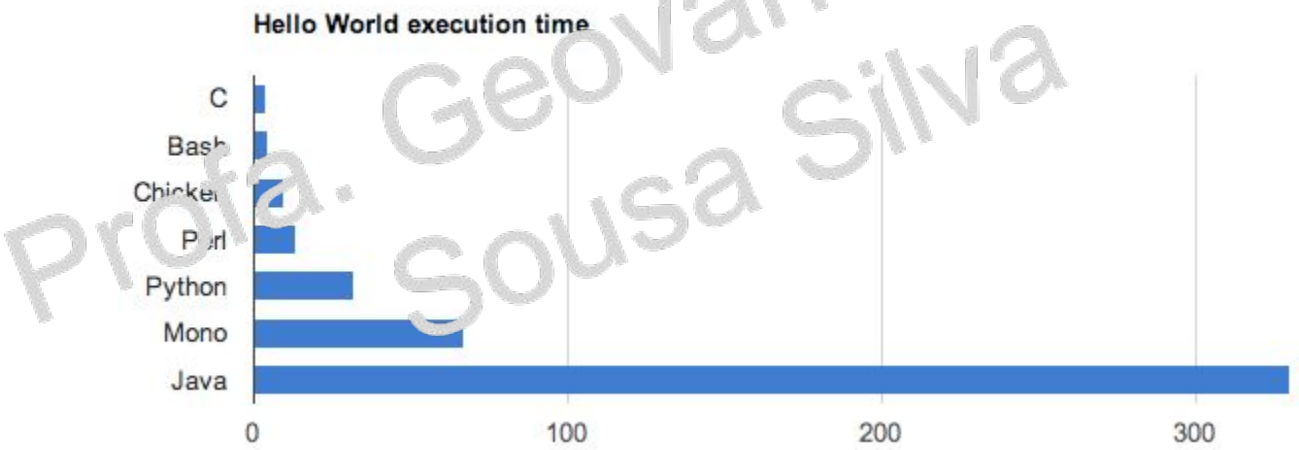
Parâmetros de Análise

- “Tempo” de execução?



Parâmetros de Análise

- “Tempo” de execução?



Parâmetros de Análise

- Número de instruções?

```
def eh_primo_simples(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

```
def eh_primo_otimizado(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
```

Parâmetros de Análise

- Número de instruções?

Mais rápido

```
def eh_primo_simples(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

```
def eh_primo_otimizado(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
```

Parâmetros de Análise

Então, qual o parâmetro ideal?

Definir o algoritmo como uma função f(*n*) onde *n* é o tamanho da entrada. Ex:

$$f(n) = n^2 + 500 \text{ para o pior caso}$$

$$f(n) = n + 100n + 500 \text{ para o melhor caso}$$

Parâmetros de Análise

Qual o melhor e pior caso?

```
def eh_primo_otimizado(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
```

Notação Big O

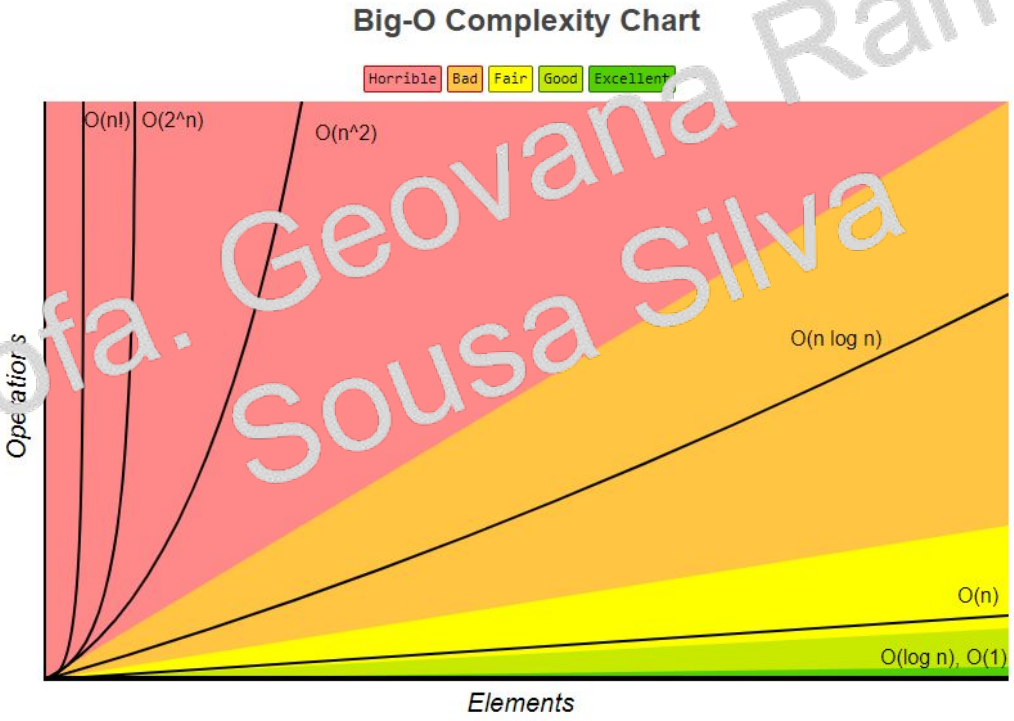
Esta notação utiliza a **ordem de magnitude** da função **do pior caso** para expressar a complexidade do algoritmo.

$f(n) = 5n + 15 \rightarrow O(n)$

$f(n) = n^2 + 100 \rightarrow O(n^2)$

$f(n) = n^4 + 100n^2 + 10n + 50 \rightarrow O(n^4)$

Notação Big O



Notação Big O

Diretrizes para cálculo de complexidade

- Estruturas sequenciais
- Iterações
- Iterações encadeadas
- Instruções consecutivas
- Condicionais
- …

Estruturas Sequenciais

A complexidade é constante. Isso ocorre pois o tempo sempre será o mesmo para qualquer valor de entrada.

```
a, b = input().split()          # O(1)

soma = int(a) + int(b)          # O(1)
multiplicacao = a * b           # O(1)

print(soma, multiplicacao)      # O(1)

# Complexidade: O(4) = O(1)
```

Iterações

A complexidade depende de *n,* pois ele determina o número de iterações.

```
n = int(input())

soma = 0
for i in range(n):
    soma += i

print(soma)

# Complexidade: O(n)
```

Iterações

Se as iterações forem fixas, a complexidade é constante.

```
soma = 0
for i in range(10):
    soma += i

print(soma)

# Complexidade: O(10) = O(1)
```

Iterações Encadeadas

A complexidade é o produto do tamanho de cada iteração do encadeamento.

```
n_1 = int(input())
n_2 = int(input())

contador = 0
for i in range(n_1):
    for j in range(n_2):
        contador += 1

print(contador)

# Complexidade: O(n_1 * n_2)
```

Condicionais

Se teste simples, a complexidade é igual a pior complexidade dentre todas as condições.

```
n_1 = int(input())
n_2 = int(input())

if n_1 > 5:
    contador = 0
    for i in range(n_1):
        for j in range(n_2):
            contador += 1

    print(contador)
else:
    print("Só um print")

# Complexidade: O(n_1 * n_2)
```

Exemplo

```
def eh_primo_otimizado(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
```

Exemplo

```
def eh_primo_otimizado(n):
    if n <= 1:                          # ______
        return False                    #
    if n <= 3:                          #
        return True                     # O(1)
    if n % 2 == 0 or n % 3 == 0:        #
        return False                    #
    i = 5                               # ______
    while i * i <= n:                           # O(√n/6 * 1)
        if n % i == 0 or n % (i + 2) == 0: # _______
            return False                    # O(1)
        i += 6                              # _______
    return True


# Complexidade = O(1 + √n/6)
# Complexidade = O(√n * 1/6) = O(√n)
```

Exemplo

```
def eh_primo_otimizado(n):
    if n <= 1:                          # ______
        return False                    #
    if n <= 3:                          #
        return True                     # O(1)
    if n % 2 == 0 or n % 3 == 0:        #
        return False                    #
    i = 5                               # ______
    while i * i <= n:                           # O(√n/6 * 1)
        if n % i == 0 or n % (i + 2) == 0: # _______
            return False                    # O(1)
        i += 6                              # _______
    return True


# Melhor(es) caso(s) → n < 4 → O(1)
```

Para casa

Calcular a complexidade dos algoritmos abaixo

- Busca linear
- Busca binária
- Bubble sort
- Selection sort
- Insertion sort

Resumo

Algoritmos podem ser analisados por meio de sua taxa de crescimento expressa por meio de função.

A notação Big O expressa a complexidade do algoritmo considerando seu pior caso.

A complexidade é calculada em função da entrada.

Bibliografia