Estrutura de Dados

Aula 02

Prof. Luiz Antonio Schalata Pacheco, Dr. Eng.

Instituto Federal de Santa Catarina
Câmpus Garopaba
Curso Superior de Tecnologia em Sistemas para Internet

schalata@ifsc.edu.br

16/02/2023

Notação Big(O)

Complexidade de Algoritmos

- Como comparar dois algoritmos?
- Comparação objetiva entre algoritmos: independente de diferenças entre poder de processamento, sistema operacional e linguagem de programação
- O quanto a "complexidade" do algoritmo aumenta de acordo com as entradas?

Exemplificando

- Programe, em Python, uma função para fazer o somatório de *n* valores. Um número deve ser repassado como parâmetro e a função deve realizar o somatório de 1 até *n*. Por exemplo, se a entrada for 10, vai somar $1 + 2 + 3...10$ e o resultado será 55.

Versão 1

```python
# Usando for
def soma1(n):
    soma = 0
    for i in range(n + 1):
        soma += i

    return soma
```

Medindo o tempo de execução

```python
import timeit

# Usando for
def soma1(n):
  soma = 0
  for i in range(n + 1):
      soma += i

  return soma

# Verificando os tempos de execucao com timeit
tempo_inicial = timeit.default_timer()
soma1(10)
tempo_final = timeit.default_timer()
print(f'Tempo soma1: {tempo_final - tempo_inicial}')
```

Aumentando o valor do parâmetro de entrada

- Execute novamente o algoritmo com n = 1.000.000.
- Depois execute novamente n = 100.000.000.
- Quais as conclusões?
- Esse algoritmo é eficiente?
- É possível acelerar a execução?

Versão 2: Usando a matemática

```python
# Usando equacao matematica
def soma2(n):
  return (n * (n + 1)) / 2
```

Medindo os tempos

```python
import timeit

# Usando equacao matematica
def soma2(n):
  return (n * (n + 1)) / 2

  return soma

# Verificando os tempos de execucao com timeit
tempo_inicial = timeit.default_timer()
soma1(10)
tempo_final = timeit.default_timer()
print(f'Tempo soma2: {tempo_final - tempo_inicial}')
```

Análises

- Observe que podemos comparar porque estamos executando no mesmo hardware e mesmo SO.
- A função soma1 executa $n + 1$ passos.
- A função soma 2 tem uma multiplicação, um somatório e uma divisão. São somente três passos, independente de n.
- Diz-se que a função soma1 é uma função $O(n)$.
- A função soma2 tem $O(3)$.

Resumindo

- A análise Big(O) independe de hardware ou de SO.
- Os tempos medidos, com certeza seriam diferentes em hardwares distintos. Com a análise através da notação Big(O) passa-se a ser objetivo.
- Soma1 tem um desempenho ruim, pois depende de n. Quanto maior n, maior o tempo de execução.
- A complexidade do algoritmo soma1 aumenta de acordo com as entradas.
- A função soma1 uma pode não ser "escalável" quando n é muito grande.
- Essa é a ideia básica de trabalhar com Big(O) para fazer comparativos entre algoritmos.

Exemplo usando listas

- Criar uma lista com 1000 elementos (do 0 ao 999)
- Medir o tempo de criação da lista

Versão 1: De forma manual

```python
def lista1():
  lista = []
  for i in range(1000):
      lista += [i]
  return lista

print(lista1())
```

Versão 2: Usando funções pré-definidas

```python
def lista2():
  return range(1000)

print(lista2())

l = lista2()
for i in l:
  print(i)
```

Análises

- A função lista1() é mais lenta, pois a inserção dos valores é feita de forma manual.
- Na função lista2(), a função range já faz esse processo automaticamente.
- Big(O) da função lista1() é O(n) ou O(1000), no caso.
- Big(O) da função lista2() não temos como saber em detalhes, só acessando a implementação da função range e analisando.
- Pelo tempo de execução, se percebe que lista2() é mais eficiente.
- Funções pré-definidas, normalmente já foram otimizadas para que o código seja mais eficiente.

Funções Big(O)

- Constante: O(1)
- Logarítmica: O(log(n))
- Linear: O(n)
- Logarítmica Linear: O(nlog(n))
- Quadrática: $O(n^2)$
- Cúbica: $O(n^3)$
- Exponencial: $O(2^n)$

Gerando gráfico das funções Big(O)

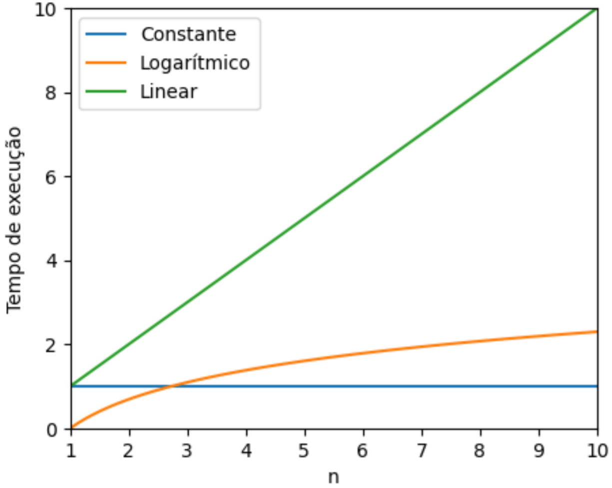
```python
import numpy as np # Biblioteca para computacao numerica
import matplotlib.pyplot as plt # Geracao de graficos

n = np.linspace(1, 10, 100) # 100 numeros espacados
labels = ['Constante', 'Logaritmico', 'Linear']
big_o = [np.ones(n.shape), np.log(n), n]  # np.ones(n.
    shape) gera 100 numeros "1".

# Definindo exibicao do grafico
plt.figure(figsize=(5,4)) # Tamanho da figura
plt.ylim(0,10) # Limite do y

for i in range(len(big_o)):
 plt.plot(n, big_o[i], label = labels[i])
plt.legend()
plt.ylabel('Tempo de execucao')
plt.xlabel('n')
plt.show()
```

Gráfico das funções Big(O)



Análises

- Função constante é a ideal. O tempo de execução (runtime) independe de n.
- Na logarítmica o runtime aumenta pouco com o aumento de n.
- Na função linear não foram feitas modificações nos dados nessa função. Runtime cresce linearmente.

Atividade de implementação...

Simule o gráfico das outras funções Big(O) citadas, exibindo todas no mesmo gráfico:

- Logarítmica Linear: O(nlog(n))
- Quadrática: $O(n^2)$
- Cubica: $O(n^3)$
- Exponencial: $O(2^n)$

Gráfico das funções Big(O)

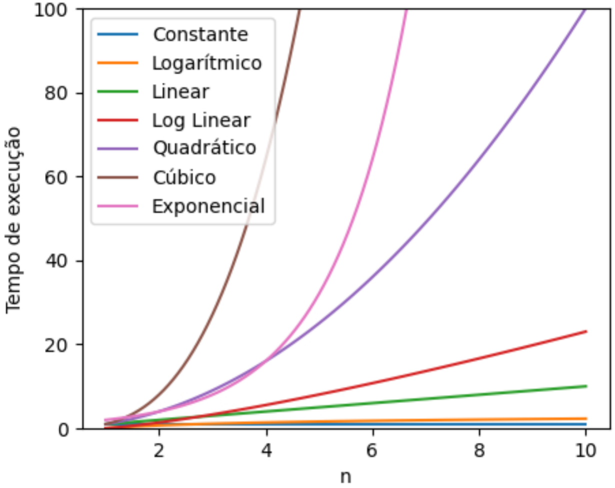
```python
import numpy as np # Biblioteca para computacao numerica
import matplotlib.pyplot as plt # Geracao de graficos

n = np.linspace(1, 10, 100) # 100 numeros espacados
labels = ['Constante', 'Logaritmico', 'Linear']
big_o = [np.ones(n.shape), np.log(n), n]  # np.ones(n.
    shape) gera 100 numeros "1".

# Definindo exibicao do grafico
plt.figure(figsize=(5,4)) # Tamanho da figura
plt.ylim(0,10) # Limite do y

for i in range(len(big_o)):
 plt.plot(n, big_o[i], label = labels[i])
plt.legend()
plt.ylabel('Tempo de execucao')
plt.xlabel('n')
plt.show()
```

Gráfico das funções Big(O)



Análises

- LogLinear apresenta um aumento no runtime em relação as anteriores.
- Os piores casos são as funções quadráticas, cúbicas (polinomiais) e a Exponencial.

Exemplo: Big(O) constante

```python
# Funcao constante
lista = [1, 2, 3, 4, 5]

# O(1) - Imprimir o primeiro valor de uma lista
def constante(n):
  print(n[0])

# O(2) - Imprimir os dois primeiros valores
def constante2(n):
  print(n[0])
  print(n[1])

constante(lista)
constante2(lista)
```

Exemplo: Big(O) linear

```python
# Funcao linear
lista = [1, 2, 3, 4, 5]

# O(n) - Imprimir todos os valores de uma lista
def linear(n):
  for i in n:
    print(i)

linear(lista)
```

Exemplo: Big(O) polinomial quadrática

```python
# Funcao quadratica
# Exemplo: Imprimir todos os n valores de uma lista

def quadratica(n):
  for i in n:
    for j in n:
      print(i, j)

quadratica(lista)
```

Exemplo: Big(O) de funções combinadas

```python
def combinacao(n):
  print(n[1])

  for i in range(5):
    print(f'teste {i}')

  for i in n:
    print(i)

  for i in n:
    print(i)

  print('teste')
  print('teste')
  print('teste')

combinacao(lista)
```

Exemplo: Big(O) de funções combinadas

```python
def combinacao(n):
  print(n[1]) # O(1)

  for i in range(5): # O(5)
    print(f'teste {i}')

  for i in n: # O(n)
    print(i)

  for i in n: # O(n)
    print(i)

  print('teste') # O(1)
  print('teste') # O(1)
  print('teste') # O(1)
```

$$O(1) + O(5) + O(n) + O(n) + O(3) = O(9) + O(2n) = O(n)$$

Questões

- Para que é utilizada a notação Big-O?
- Qual o Big(O) dos códigos abaixo:

```
for a in range(n):
  for b in range(n):
    print('teste')
```

```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for i in range(l):
  print(l[0])
  break
```

```
for i in range(5):
  print(i)
```