Lógica de Programação Orientada a Objetos

Bem-vindos!

Objetivos

- ✓ Análise de Complexidade

Por que analisar a complexidade dos algoritmos?

- A preocupação com a complexidade de algoritmos é fundamental para projetar algoritmos eficientes.
- Podemos desenvolver um algoritmo e depois analisar a sua complexidade para verificar a sua eficiência.

Eficiência ou complexidade de algoritmos

- Dados 2 algoritmos que multiplicam duas matrizes quadradas $n \times n$ (cada uma com n^2 elementos), como podemos comparar os dois algoritmos para escolher o melhor?

- Precisamos definir alguma medida que expresse a eficiência. Costuma-se medir a complexidade de um algoritmo em termos de tempo de execução ou o espaço (ou memória) usado.

- Medir o tempo absoluto não é interessante pois depende do poder computacional da máquina.

- Em Análise de Algoritmos conta-se o número de operações consideradas relevantes realizadas pelo algoritmo e expressa-se esse número como uma função de n. Essas operações podem ser comparações, operações aritméticas, movimento de dados, etc.

Pior caso, melhor caso, caso médio

- O número de operações realizadas por um determinado algoritmo pode depender da particular instância da entrada. Em geral interessa-nos o pior caso, o maior número de operações usadas para qualquer entrada de tamanho n.

- Análises também podem ser feitas para o melhor caso e o caso médio.

- Exemplo: Organizar em ordem crescente uma lista de 100 números, em que o pior caso é quando a lista está em ordem decrescente, o melhor caso é quando a lista já está organizada e o caso médio quando cerca de metade da lista está organizada.

Exercício 1

Suponha que existem 2 listas de tamanho n que devem ser ordenadas, uma de $n = 10$ e outra de $n = 100$ e 2 opções de algoritmos para ordená-las, cujo número de operações de cada um são diretamente proporcionais ao tamanho da lista de entrada:

Algoritmo 1: $2n^2 + 5n$ operações
Algoritmo 2: $500n + 4000$ operações

Qual algoritmo é mais eficiente para ordenar cada uma das listas?

Comportamento assintótico

Algoritmo 1: $f1(n) = 2n^2 + 5n$ operações
Algoritmo 2: $f2(n) = 500n + 4000$ operações

- Um caso particular interesse é quando n tem valor muito grande ($n \to \infty$), denominado comportamento assintótico.

- Os termos inferiores e as constantes multiplicativas contribuem pouco na comparação e podem ser descartados.

- O importante é observar que $f1(n)$ cresce com n^2 ao passo que $f2(n)$ cresce com n. Um crescimento quadrático é considerado pior que um crescimento linear. Assim, vamos preferir o Algoritmo 2 ao Algoritmo 1.

Dados os 2 algoritmos abaixo para calcular uma soma, pergunta-se: Qual deles é mais rápido?

```
def algoritmo_a(n):
    s = 0
    for i in range(1, n + 1):
        for j in range(i, n + 1):
            s = s + 1
    return s

resultado = algoritmo_a(10)
print("A soma dos 10 primeiros números inteiros é ", resultado)

def algoritmo_b(n):
    s = 0
    for i in range(1, n + 1):
        s = s + i
    return s

resultado = algoritmo_b(10)
print("A soma dos 10 primeiros números inteiros é ", resultado)
```

Regras para Análise de Complexidade

- Sentenças simples: Possuem complexidade constante, ou seja, O(1).

```
# Sentenças simples
s = "Brasil"
i = 42
i += 1
```

Regras para Análise de Complexidade

- Laços simples: Possuem complexidade linear no tamanho da entrada, ou seja, O(n), em que n é o tamanho da entrada.

```
# Laços simples
for i in range(n):
    # Sentenças simples
```

Regras para Análise de Complexidade

- Laços aninhados: Possuem complexidade quadrática no tamanho da entrada, ou seja, $O(n^2)$, em que n é o tamanho da entrada.

```
# Laços aninhados
for i in range(n):
    for j in range(n):
        # Sentenças simples
```

Exercício 2

- Qual é a complexidade da função abaixo?

```
def f(n, condicao):
    a = "Ordem e Progresso"
    if condicao:
      for i in range(n):
        # Sentenças simples
    else:
      for i in range(n):
        for j in range(n):
          # Sentenças simples
```

Exercício 3

- Qual é a complexidade do seguinte trecho de código?

```
a = 0;
for i in range(n):
  for j in range(i + 1, n):
    a = a + i + j;
print(a)
```

Exercício 4

Qual é a complexidade do seguinte trecho de código?

```
a = 0;
for i in range(n):
  for j in range(n):
    for k in range(n):
      a = a + i + j + k;
print(a)
```

O(log(n)) - Subpolinomial

```
i = n
while i > 1:
  i = i / 2
  print(i)
```

Perceba que a variável i é decrementada. A cada iteração do while, o valor de i é dividido por dois. Assim, para n = 64, o trecho de código acima imprime 32, 16, 8, 4, 2, 1. O laço foi executado uma vez para cada um dos números impressos, ou seja, para n = 64, o laço executou 6 vezes. Se fizermos alguns testes executando o algoritmo com valores diferentes de n, veremos que o laço sempre executa log(n) vezes. Logaritmos são comuns em análise de complexidade, e é importante darmos atenção especial às aparições deles. Em geral, logaritmos aparecem sempre que estamos repetidamente dividindo coisas pela metade ou dobrando o tamanho de coisas.

Exercício 5

Qual é a complexidade do seguinte trecho de código?

```
for i in range(n):
  i = 1
  while i < n:
    i = i * 2
    print(i)
```

O(2^n) - Exponencial

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

A função fib calcula, de forma extremamente ineficiente, o n-ésimo número da sequência de Fibonacci. Apesar dessa função possuir alguma semelhança em sua estrutura com a função fatorial, a complexidade da função fib é muito maior. Sua complexidade é O(2^n). Isso ocorre porque são feitas muitas chamadas repetidas à função fib.

Exercício 6

- O que quer dizer a sentença: O algoritmo X é assintoticamente mais eficiente que o algoritmo Y?

Exercício 7

- Ordene os laços abaixo do mais eficiente (o que executa mais rapidamente) para o menos eficiente (executa mais lentamente), assumindo que o valor de n é positivo.

  # Laço a:
  i = 0
  while i < n:
    i++

  # Laço b:
  j = 0
  while j < n:
    j += 2

  # Laço c:
  k = 0
  while k < n:
    k *= 2

Exercício 8

- Ordene as funções abaixo em ordem crescente de complexidade assintótica.

  f(n)=2^n.
  g(n)=n^3/2.
  h(n)=n*log(n).
  s(n)=n^log(n).

Exercício 9

- Dado um array de tamanho n, em que n = 20 e o array tem valores entre 0 a 19, construa um algoritmo de complexidade O(log(n)) para remover todos os valores divisíveis por 3.

Exercício 10

- Dado um array de tamanho n, em que n = 10 e o array tem valores 2, 5, 1, 3, 4, 6, 9, 8, 7, 0, nesta sequência, ordene os valores em ordem crescente utilizando um algoritmo de complexidade $O(n^2)$.