Universidade Federal de Pernambuco
Centro de Tecnologia Geociências (CTG/UFPE)

Disciplina: IF264 - Métodos Computacionais (2023.1)
Professor: Paulo Freitas
Estudante: Romário Jonas de Oliveira Veloso

Lista 9

Questão 1) Analise os trechos de código a seguir e determine a sua complexidade com notação Big O justificando as suas respostas.

a)

```
def example1(items):
    for item in items:
        print(item)
    for item in items:
        print(item)
```

Cada loop tem uma complexidade de O(n), onde n é o número de elementos em items. Ou seja, o primeiro loop 'for item in items' temos complexidade O(n), o mesmo se repete no segundo loop. Logo, quando combinamos as duas operações, a complexidade total é as oma das complexidades de cada operação individual. Portanto, a complexidade total é

$O(n) + O(n) = 2O(n)$;

b)

```
def example2(items):
    for item in items:
        for item2 in items:
            print(item, ' ' ,item2)
```

O código tem dois loops, externo e o interno. O loop externo itera n vezes dando um elemento ao loop interno que novamente faz um loop n vezes, assim, o número total de operações é proporcional ao quadrado do tamanho da lista 'items', isso nos dá uma complexidade quadrática, representada por $O(n^2)$;

c)

```
def example3(lst):
    print(lst[0])
    midpoint = len(lst)/2
    for val in lst[:midpoint]:
        print(val)
    for x in range(10):
        print("number")
```

Note que, neste caso, as operações print(lst[0]), midpoint = len(lst)/2 e o loop “for x in range(10): print(“number”)” apresentam complexidade O(1), ressaltando neste ultimo caso, de que o loop é executado 10 vezes, independentemente do tamanho da lista ‘lst’. Portanto, sua complexidade é cosntante, O(1).

No entanto, para o loop “for val in lst[:midpoint]: print(val)” este loop percorre metade da lista, deste modo o loop será executado ‘n/2’ vezes. No entanto, as constantes multiplicativas são ignoradas, então a complexidade do loop é O(n).

Cujo total de complexidade é: O(1) + O(1) + O(n) + O(1) = O(n)

As operações constantes O(1) não afetam a taxa de crescimento geral, então a complexidade dominante é a do loop que percorre metade da lista, que é O(n);

d)

```
def example4(items):
    for i in range(5):
        print("Python is awesome")
    for item in items:
        print(item)
    for item in items:
        print(item)
    print("Big O")
    print("Big O")
    print("Big O")
```

Caso semelhante ao item anterior onde o loop inicial “for i in range(5): print(“Python is awesome”) a complexidade será do tipo constante, O(1); Já nos dois loops seguintes notemos que os loops percorrem a lista ‘items’ uma vez. Se a lista ‘items’ tiver o tamanho ‘n’, então este loop será executado ‘n’ vezes. Suas respectivas complexidades são O(n). Por fim, as três instruções de ‘print(“Big O”) cada uma dessas instruções é uma operação constante com complexidade O(1).

Portanto, ao combinar todas as operações, a complexidade total é:

O(1) + O(n) + O(n) + O(1) + O(1) + O(1) = 2O(n) + O(1)

Mas, os termos adicionais de ordem inferior são omitidos e estamos interessados na complexidade dominante que é a dos dois loops que percorrem a lista “items”, que é O(n).

e)

```
def example5(num, items):
    for item in items:
        if item == num:
            return True
        else:
            pass
```

Neste problema temos que para o primeiro loop: “for item in items” a complexidade é de O(n), pelos motivos ressaltados no problema anterior. E que “if item == num: “ cuja complexidade da operação será constante (ou seja, O(1)) visto que esta operação verifica se o item atual é igual ao número ‘num’. No entanto, como esta dentro do loop, toda a complexidade desta operação será O(n);

f)

```
def example6(number):
    if number <= 1:
        return number
    return (example6(number-1) + example6(number-2))
```

Neste problema, há uma comparação de valores de maneira binária, criando uma árvore de camadas onde cada nó tem dois filhos. Por exemplo:
para ‘example6(n)’, temos camadas para ‘example6(n-1)’ e ‘example6(n-2)’;

Deste modo, o número total de nós (ou camadas da função) é a exponencial em relação a ‘n’. Logo a complexidade de tempo desta implementação será representada por O(log2 n) ;