PUC Minas

Lista 1 – Função de complexidade

INFORMAÇÕES DOCENTE						
CURSO: ENGENHARIA DE SOFTWARE	**DISCIPLINA:** FUNDAMENTOS DE PROJETO E ANÁLISE DE ALGORITMOS	**TURNO**	MANHÃ	TARDE	NOITE	**PERÍODO/SALA:** 5º
					x	
PROFESSOR (A): João Paulo Carneiro Aramuni						

Lista 1 - Gabarito

Função de complexidade

1) O algoritmo abaixo implementa uma função que encontra o maior valor dentro de uma lista.

```
1  def max(lista):
2      temp = lista[0]  # Inicializa temp com o primeiro elemento da lista
3      for i in range(1, len(lista)):  # Percorre os elementos do índice 1 até o final
4          if temp < lista[i]:  # Se encontrar um elemento maior que temp
5              temp = lista[i]  # Atualiza temp com esse valor
6      return temp  # Retorna o maior valor encontrado
```

a) Quais são as operações mais relevantes?
As operações mais significativas são aquelas que dependem de n, pois dominam o crescimento da função conforme n aumenta:
As operações que mais impactam o crescimento do tempo de execução são:
1. Comparações dentro do if → 1 operação
2. Atribuições dentro do if (no pior caso) → 1 operação
3. Iterações do for → (n - 1) vezes

Essas operações contribuem para a complexidade assintótica.

b) Quanto tempo se gasta em relação ao tamanho n da lista? (Qual a função de complexidade?)

```
1  def max(lista):
2      temp = lista[0] # (1) Atribuição inicial
3      for i in range(1, len(lista)):  # (n - 1) iterações
4          if temp < lista[i]:  # (1) Comparação
5              temp = lista[i]  # (1) Atribuição
6      return temp # (1) Retorno do valor máximo
```

Contagem de operações:
1. Inicialização: temp = lista[0] → 1 operação.
2. Laço for: O loop roda de i = 1 até i = n - 1, ou seja, (n - 1) iterações.
3. Comparação dentro do if: Acontece 1 vez por iteração.
4. Atribuição dentro do if (pior caso - lista em ordem crescente): Pode ocorrer no máximo 1 vez por iteração.
5. Retorno do valor máximo: 1 operação.

Soma das operações:
Total de operações:
$1 + (n - 1) * (1 + 1) + 1 =$
$1 + (n - 1) * (2) + 1 =$
$1 + 2(n - 1) + 1 =$
$1 + 2n - 2 + 1 =$
$2n$

Ou seja: $f(n) = 2n$

Isso confirma que a complexidade assintótica é O(n), pois descartamos constantes ao analisar a notação Big-O.

2) O algoritmo abaixo retorna o maior e o menor valor dentro de uma lista de números.

```
1  def max_min(lista):
2      temp = [lista[0], lista[0]]  # Inicializa temp com o primeiro elemento como maior e menor
3      for i in range(1, len(lista)):  # Percorre a lista do índice 1 até o final
4          if temp[0] < lista[i]:  # Se encontrar um número maior que temp[0]
5              temp[0] = lista[i]  # Atualiza temp[0] com esse valor (maior número)
6          elif temp[1] > lista[i]:  # Se encontrar um número menor que temp[1]
7              temp[1] = lista[i]  # Atualiza temp[1] com esse valor (menor número)
8      return temp  # Retorna uma lista [maior_valor, menor_valor]
```

a) Quais são as operações mais relevantes?

As operações mais significativas são aquelas que dependem de n, pois dominam o crescimento da função conforme n aumenta:
As operações que mais impactam o crescimento do tempo de execução são:
1. Comparações no if e elif
2. Atribuições dentro do if e elif
3. Execução do loop for

Essas operações contribuem para a complexidade assintótica.

b) Quanto tempo se gasta em relação ao tamanho n da lista? (Qual a função de complexidade?)

```python
def max_min(lista):
    temp = [lista[0], lista[0]]  # (1) Inicializa temp com o primeiro elemento como maior e menor
    for i in range(1, len(lista)):  # (n - 1) iterações do loop
        if temp[0] < lista[i]:  # (1) Comparação
            temp[0] = lista[i]  # (1) Atribuição (executa apenas se a comparação for verdadeira)
        elif temp[1] > lista[i]:  # (1) Comparação (só avaliada se o if for falso)
            temp[1] = lista[i]  # (1) Atribuição (executa apenas se o elif for verdadeiro)

    return temp  # (1) Retorno da lista [maior_valor, menor_valor]
```

Contagem de operações:
- Cada iteração sempre executa pelo menos uma comparação (a do if).
- Se a comparação do if for verdadeira, executamos uma atribuição e pulamos o elif.
- Se a comparação do if for falsa, então executamos a comparação do elif.
    - Se o elif for verdadeiro, fazemos uma atribuição.
    - Se o elif for falso, não há atribuição.

Portanto, para cada iteração do loop, o número de operações possíveis é:
- 1 comparação sempre (if temp[0] < lista[i])
- No máximo 1 atribuição no if (se entrar no bloco if)
- Se não entrar no if, faz 1 comparação no elif
- Se entrar no elif, faz 1 atribuição no elif

Ou seja, em cada iteração, sempre há:
- 1 ou 2 comparações.
- No máximo 1 atribuição (nunca ambas ao mesmo tempo).

Soma das operações:
Total de operações:
No pior caso, podemos assumir que o algoritmo percorre um conjunto de números em que metade das vezes ele entra no if e metade das vezes entra no elif (por exemplo, números aleatórios).

Assim, para cada iteração do loop:
- Duas comparações no pior caso, if e elif (2)
- Uma atribuição no pior caso (1)

$1 + (n - 1) * (\mathbf{2 + 1}) + 1 =$
$1 + (n - 1) * (3) + 1 =$
$1 + 3(n - 1) + 1 =$
$1 + 3n - 3 + 1 =$
$3n - 1$

Ou seja: $f(n) = 3n - 1$

O total de operações no pior caso é $3n - 1$.
A complexidade assintótica é $O(n)$, pois descartamos a constante.
