Técnicas e Análise de Algoritmos

Notação Assintótica - Parte 03

Professor: **Jeremias Moreira Gomes**

E-mail: jeremias.gomes@idp.edu.br

Introdução

Recapitulação

- Notação Assintótica Big-O
- Notação Assintótica Big-Ω
- Notação Assintótica Big-Θ

Introdução

- É imprescindível buscar sempre uma cota justa para a função de custo de pior caso de um algoritmo, isto é:
  - $O(g(n)) \downarrow$
  - $\Omega(g(n)) \uparrow$
  - $\Theta(g(n))$
- Superestimar cotas superiores ou subestimar cotas inferiores levam uma estimativa inadequada dos recursos necessários para a execução do algoritmo

Pior Caso, Melhor Caso e Caso Médio

Pior Caso, Melhor Caso e Caso Médio

- A análise de algoritmos considera três cenários possíveis, os quais relacionam a entrada com o número de iterações do algoritmo
    - O pior caso acontece quando o algoritmo executa o número máximo de iterações possível
    - No melhor caso o algoritmo executa o número mínimo de iterações possível
    - O caso médio representa o cenário esperado quando as entradas possuem determinada distribuição de probabilidade de ocorrência

Pior Caso, Melhor Caso e Caso Médio

- Em termos formais, a complexidade do caso médio é dada por

$$C_M = \sum_i p(entrada_i) passos(entrada_i)$$

$$\text{com } p(entrada_i) \geq 0 \text{ e } \sum_i p(entrada_i) = 1$$

- A fórmula para o caso médio coincide com a definição básica probabilística de valor esperado

Pior Caso, Melhor Caso e Caso Médio

- O melhor caso tem interesse meramente teórico, não sendo levado em consideração na maior parte das análises
- A maioria das análises se concentram no pior caso, pois ele é uma estimativa de como o algoritmo efetivamente vai se comportar

Pior Caso, Melhor Caso e Caso Médio

- Embora o caso médio seja mais próximo da realidade, sua análise é mais técnica e depende de conceitos elaborados de matemática e probabilidade
- Além disso, o caso médio “tende” a ser idêntico ao pior caso no contexto da complexidade assintótica
- A notação mais utilizada é a notação Big-O, seguida pela notação Big-Θ

Exemplos de análise de complexidade assintótica

Exemplo 01

- Implementar uma função que torne maiúscula a primeira letra da string dada como parâmetro

Exemplo 01

- Implementar uma função que torne maiúscula a primeira letra da string dada como parâmetro

```
void capitalize(char *s)
{
    s[0] = toupper(s[0]);
}
```

Exemplo 01

- Implementar uma função que torne maiúscula a primeira letra da string dada como parâmetro

```
void capitalize(char *s)
{
    s[0] = toupper(s[0]);
}
```

- Análise: A implementação da função faz uma única atribuição, de modo que, se a string tem n caracteres, então f(n) = 1, para qualquer tamanho de n. Assim, o algoritmo tem complexidade O(1).

Exemplo 02

- Implementar uma função que **retorne** a string dada, com a primeira letra em maiúsculo

Exemplo 02

- Implementar uma função que **retorne** a string dada, com a primeira letra em maiúsculo

```
string capitalize(string s)
{
    s[0] = toupper(s[0]);
    return s;
}
```

Esta solução, está em C++ para o uso do tipo string

Exemplo 02

- Implementar uma função que **retorne** a string dada, com a primeira letra em maiúsculo

```
string capitalize(string s)
{
    s[0] = toupper(s[0]);
    return s;
}
```

- Análise: Observe que nesse caso, os n caracteres da string s devem ser copiados para o retorno, além da atribuição feita. Assim:
  - $f(n) = n + 1$, de modo que o algoritmo tem complexidade $O(n)$

Esta solução, está em C++ para o uso do tipo string

Exemplo 03

- Implementar uma função que retorne o produto cartesiano dos conjuntos de números inteiros A e B
  - Os conjuntos são listas
  - O tamanho dessas listas é retornado em tempo constante
  - A estrutura resultado tem inserção em tempo constante

Exemplo 03 - Solução

```
Lista_par produto_cartesiano(Lista_int A, Lista_int B)
{
    Lista_par P;
    int n = A.size();
    int m = B.size();
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            P.insere({A[i], B[j]});
        }
    }
    return P;
}
```

Exemplo 03 - Análise (1/3)

- Duas atribuições para receber o tamanho das listas
- Uma atribuição de inicialização da lista no laço mais externo
- Para cada iteração do laço mais externo, são feitas duas atribuições:
  - Inicialização do j
  - Incremento do i
- Do item anterior, então, o laço externo executa 2n vezes

Exemplo 03 - Análise (2/3)

- No laço interno, são feitas duas operações
  - Incremento do j
  - Inserção do par na lista
- Do item anterior, a inserção do par possui tempo constante (O(1)), que será chamada de c. Então, o item é iniciado n vezes e executa m vezes c + 1 operações, e fica da seguinte forma:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (c+1) = (c+1)(n-1)(m-1)$$

Exemplo 03 - Análise (3/3)

- Combinando todas essas operações, a função que representa o comportamento deste algoritmo é a seguinte:

$$f(n, m) = 3 + 2n + (c + 1)(n - 1)(m - 1)$$

- Simplificando e aplicando as propriedades corretas, chega-se à complexidade O(mn), onde no pior caso, m possui o mesmo tamanho de n, cujo a complexidade é $O(n^2)$

Exemplo 04

- Implementar uma função que retorne todos os pares de naturais (a, b) tais que $a < b \leq N$, para um inteiro positivo N dado

Exemplo 04 - Solução

- Implementar uma função que retorne todos os pares de naturais (a, b) tais que a < b ≤ N , para um inteiro positivo N dado

```
Lista_pares pares_naturais(int N)
{
    Lista_pares P;
    for (int a = 1; a <= N; a++)
    {
        for (int b = a + 1; b <= N; b++)
        {
            P.insere({a, b});
        }
    }
    return P;
}
```

Exemplo 04 - Análise (1/4)

- Uma atribuição no início do laço externo
- O laço possui N iterações, e em cada iteração são feitas duas atribuições
  - Incremento de a
  - Inicialização de b

Exemplo 04 - Análise (2/4)

- O laço interno possui uma estrutura mais complexa
  - Executa N - a vezes
  - Cada iteração possui duas operações
    - Incremento de b
    - Inserção do par na lista
      - Para simplificar, consideraremos equivalente a uma atribuição

$$\sum_{a=1}^{N} = 2(N-a) = 2\sum_{a=1}^{N}(N-a)$$

Exemplo 04 - Análise (3/4)

- O somatório anterior (sem o 2), é o equivalente ao seguinte:

$$\sum_{a=1}^{N}(N-a) = (N-1) + (N-2) + \cdots + 1 + 0$$

- Reescrevendo os termos em ordem crescente, e temos

$$\underbrace{1 + \cdots + (N-2) + (N-1)}_{k} = \sum_{k=1}^{N-1} k$$

Exemplo 04 - Análise (3/4)

- Substituindo a simplificação, temos:

$$2\sum_{k=1}^{(N-1)} k = 2\left[\frac{N(N-1)}{2}\right] = N(N-1)$$

- Assim, podemos combinar todas as operações, de f(N), com o seguinte:

$$f(N) = 1 + 2N + N(N-1) = O(N^2)$$

Exemplo 05

- Implemente uma função que compute o produto da matriz A pela matriz B, onde:
  - $A_{n \times m}$ e $B_{m \times p}$

Exemplo 05 - Solução

```
Matriz operator*(Matriz A, Matriz B)
{
    int n = A.rows(), m = A.cols(), p = B.cols();
    Matriz C(n, p);      // Cria uma matriz n x p
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            for (int k = 0; k < m; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return C;
}
```

Exemplo 05 - Análise

- Nessa análise, consideraremos apenas o termo dominante
- A cada execução, o laço externo, o laço intermediário e o laço interno iteram n, p e m vezes, respectivamente
    - Em cada execução de um destes laços, o número de atribuições associadas a ele é constante
- Assim, o algoritmo terá complexidade $O(nmp)$
- A título de curiosidade, a função f (n, m, p) é dada por

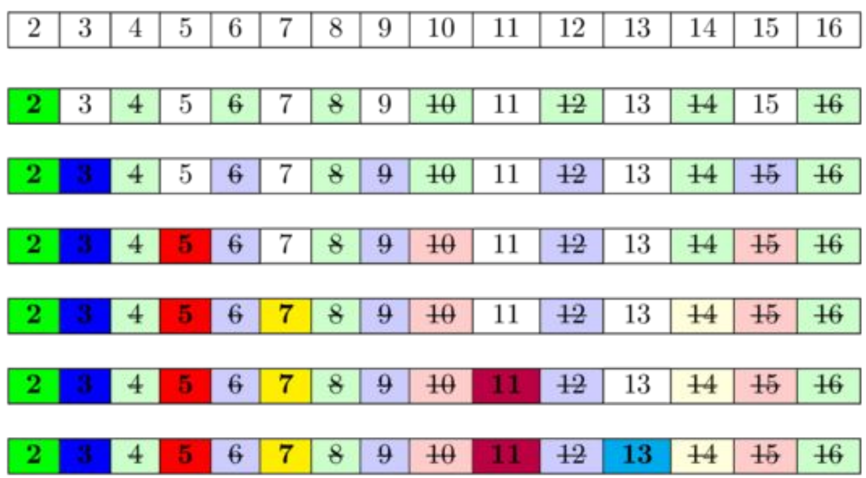
$$f(n, m, p) = 4 + 2n + 3np + 2nmp$$

Exemplo 06.0

- Responder se um número é primo ou não
    - Um número primo é um número natural maior que 1 que não pode ser formado pela multiplicação de outros dois naturais menores
        - 2, 3, 5, 7, 11, 13, 17, 19, 23, ..., ???

Exemplo 06

- Computar um vetor de inteiros V tal que a posição i = 1, se i é primo, e V[i] = 0, caso contrário, para i ≤ N
  - Também conhecido como crivo de Eratóstenes



Exemplo 06 - Solução

- Computar um vetor de inteiros V tal que a posição i = 1, se i é primo, e V[i] = 0, caso contrário, para i ≤ N
  - Também conhecido como crivo de Eratóstenes

```
void crivo(int *V, int N)
{
    preenche(V, N);
    V[0] = V[1] = 0;
    for (int i = 2; i < N; i++)
        if (V[i])
            for (int j = 2 * i; j < N; j += i)
                V[j] = 0;
}
```

Exemplo 06 - Análise (1/5)

- A função preenche, percorre o vetor de N + 1 posições, inicializando o vetor com o valor 1 (O(n))
- As posições 0 e 1 do vetor, são zeradas (não são primos)
- O laço externo inicia com uma atribuição (i = 2) e itera N − 1 vezes, incrementando a variável i em cada uma destas iterações
- Já a inicialização da variável j depende do valor de V[i]
  - Só acontecerá quando V[i] = 1, ou seja, **quando i for primo**

Exemplo 06 - Análise (2/5)

- Assim, para determinar f(N) com precisão, é necessário determinar o número exato de primos no intervalo [1, N]
  - Esse número é calculado por π(N)
  - Existem boas aproximações para esta quantidade

$$\pi(N) \approx \frac{N}{\log N}$$

  - Porém quanto mais precisa a aproximação, mas sofisticada é a função

Exemplo 06 - Análise (3/5)

- Como Big-O é uma cota superior, o número de execuções do laço interno pode ser majorado, pois $\pi(N) < N$
  - Nem todos os números do intervalo são primos
- Assim, a cada execução do laço interno itera $\left\lfloor \frac{N}{i} \right\rfloor$ vezes, fazendo duas atribuições
  - Incremento j += i
  - Atribuição V[j] = 0

Exemplo 06 - Análise (4/5)

- Assim, o número total de atribuições associadas ao laço interno são

$$\sum_{p \text{ primo}}^{N} 2 \left\lfloor \frac{N}{p} \right\rfloor$$

Exemplo 06 - Análise (4/5)

- Assim, o número total de atribuições associadas ao laço interno são

$$\sum_{p \text{ primo}}^{N} 2 \left\lfloor \frac{N}{p} \right\rfloor$$

$$\sum_{p \text{ primo}}^{N} 2 \left\lfloor \frac{N}{p} \right\rfloor \leq \sum_{i=1}^{N} 2 \left\lfloor \frac{N}{i} \right\rfloor$$

Exemplo 06 - Análise (4/5)

- Assim, o número total de atribuições associadas ao laço interno são

$$\sum_{p \text{ primo}}^{N} 2 \left\lfloor \frac{N}{p} \right\rfloor$$

$$\sum_{p \text{ primo}}^{N} 2 \left\lfloor \frac{N}{p} \right\rfloor \leq \sum_{i=1}^{N} 2 \left\lfloor \frac{N}{i} \right\rfloor \leq 2 \sum_{i=1}^{N} \frac{N}{i}$$

Exemplo 06 - Análise (4/5)

- Assim, o número total de atribuições associadas ao laço interno são

$$\sum_{p \text{ primo}}^{N} 2\left\lfloor \frac{N}{p} \right\rfloor$$

$$\sum_{p \text{ primo}}^{N} 2\left\lfloor \frac{N}{p} \right\rfloor \leq \sum_{i=1}^{N} 2\left\lfloor \frac{N}{i} \right\rfloor \leq 2\sum_{i=1}^{N} \frac{N}{i} \leq 2N\sum_{i=1}^{N} \frac{1}{i}$$

Exemplo 06 - Análise (4/5)

- Assim, o número total de atribuições associadas ao laço interno são

$$2N\sum_{i=1}^{N}\frac{1}{i} < 2N\int_{1}^{N}\frac{1}{i}di$$

$$= 2N\log N$$

Exemplo 06 - Análise (5/5)

- A sequência de aproximações utilizada fornece uma cota superior para o número de atribuições desejado
- Embora efetivamente ele seja menor do que a aproximação, ao menos há a garantia de que o número total de atribuições feitas pelo laço interno seja sempre menor do que a cota estabelecida.
- Assim, a função f (n) pode ser aproximada e tem a complexidade dada por

$$f(n) = 3 + (N+1) + 2(N+1) + 2N \log N = O(N \log N)$$

Exercícios para Reforçar o Aprendizado

- Ordene as funções abaixo em ordem de taxa de crescimento

$$\log_6 x, \quad x^5, \quad 2^x, \quad x^3, \quad log_{15} x, \quad 100x^4, \quad 64x + 1000, \quad \sqrt{x}, \quad 4$$

Exercícios para Reforçar o Aprendizado

- Descreva qual é o termo dominante das seguintes funções:

Função	Termo Dominante	Complexidade Big-O
$7n + 4$	$7n$	$O(n)$
$100n + 0.01n^2$		
$500n + n^{1.5} + 25$		
$2n + 3log_2 n$		
$5n^{\frac{1}{3}} + \sqrt{n}$		
$3^x + x^4 + 7$		

Conclusão