Análise de Complexidade Temporal e Espacial

Prof. Me. Lucas Bruzzone

Aula 02

Por que Analisar Complexidade?

Pergunta Central

Como o tempo/espaço cresce com o tamanho da entrada?

Exemplo Prático - Motivação

Processando uma lista:

- ✓ Lista com **10 elementos**: 0,001 segundos
- ? Lista com **100 elementos**: ? segundos
- ? Lista com **1.000.000 elementos**: ? segundos

Precisamos prever esse crescimento matematicamente!

Algoritmo Linear - Exemplo

Padrão mais comum:

```
Entrada: array com n elementos
for i = 1 to n:
    print(array[i])     # operacao basica
```

Comportamento observado:

- n = 10 → executa 10 vezes
- n = 100 → executa 100 vezes
- n = k → executa k vezes

Por que FOR = O(n)? - Intuição

Raciocínio simples:

n elementos → 1 operação por elemento → n operações total

Função matemática: $T(n) = n \times c$

- c = tempo constante da operação básica
- Crescimento é **linear** em relação a n

Demonstração Matemática - Parte 1

Teorema: Um loop que executa n vezes tem complexidade O(n)

Expressão matemática:

$$T(n) = \sum_{i=1}^{n} c \tag{1}$$

O que significa:

- Somamos o tempo c para cada iteração
- De $i = 1$ até $i = n$
- Cada iteração custa tempo constante c

Demonstração Matemática - Parte 2

Expandindo o somatório:

$$T(n) = \sum_{i=1}^{n} c \tag{2}$$

$$= c + c + c + \ldots + c \text{ (n vezes)} \tag{3}$$

$$= n \times c \tag{4}$$

$$= c \cdot n \tag{5}$$

Conclusão: $T(n) = c \cdot n$, portanto $T(n) = O(n)$

Variações de Loops Lineares

Diferentes formas, mesma complexidade:

- `for i = 1 to n/2`: $T(n) = \frac{n}{2} \times c = \frac{c \cdot n}{2}$
- `for i = 1 to 3n`: $T(n) = 3n \times c = 3c \cdot n$
- `for i = 1 to 2n`: $T(n) = 2n \times c = 2c \cdot n$

Todas têm crescimento linear → O(n)

Regra importante: Constantes multiplicativas não mudam a ordem de crescimento

Padrão: dividir para conquistar

```
Entrada: array ordenado com n elementos
while inicio <= fim:
    meio = (inicio + fim) / 2
    if array[meio] == x: return meio
    if x < array[meio]: fim = meio - 1
    else: inicio = meio + 1
```

Comportamento chave:

- A cada iteração: espaço de busca reduz pela **metade**
- Progressão: n $\rightarrow$ n/2 $\rightarrow$ n/4 $\rightarrow$ n/8 $\rightarrow$ ... $\rightarrow$ 1

Sequência de Divisões

Quantas vezes podemos dividir n por 2?

Sequência observada:

$$n_0 = n \tag{6}$$
$$n_1 = \frac{n}{2} \tag{7}$$
$$n_2 = \frac{n}{4} = \frac{n}{2^2} \tag{8}$$
$$n_3 = \frac{n}{8} = \frac{n}{2^3} \tag{9}$$
$$\vdots \tag{10}$$
$$n_k = \frac{n}{2^k} \tag{11}$$

Padrão: A cada passo k, temos $\frac{n}{2^k}$ elementos restantes.

Resolução - Encontrando k

Condição de parada: $n_k = 1$

Equação a resolver:

$$\frac{n}{2^k} = 1$$

Isolando k:

$$\frac{n}{2^k} = 1 \tag{12}$$

$$n = 2^k \tag{13}$$

$$k = \log_2(n) \tag{14}$$

Resultado Final - Complexidade Logarítmica

Portanto: $T(n) = k = \log_2(n)$

Conclusão Geral

Algoritmos que **dividem o problema pela metade** a cada passo têm complexidade $O(\log n)$.

Exemplos:

- Busca binária
- Inserção em árvore binária balanceada
- Alguns algoritmos divide-and-conquer

Exemplo Numérico - Verificação

Lista com 1000 elementos ordenados

Iteração	**Tamanho Restante**	**Operações**
0	1000	1
1	500	1
2	250	1
3	125	1
4	62	1
5	31	1
Total	-	**10 operações**

Verificação: $\log_2(1000) \approx 10$
Comparação: Busca linear $=$ até 1000 operações!

Passos para Análise de Complexidade

Metodologia Sistemática

1. **Identificar operação básica**
2. **Contar frequência da operação**
3. **Expressar em função de n**
4. **Determinar ordem de crescimento**

Identificando a Operação Básica

Definição: Operação mais frequente e/ou custosa no algoritmo

Exemplos por tipo de algoritmo:

- **Busca**: Comparação de elementos
- **Ordenação**: Comparação ou troca
- **Multiplicação de matrizes**: Multiplicação
- **Parsing**: Comparação de caracteres
- **Grafos**: Visita de vértice/aresta

Foque na operação que domina o tempo total

Estrutura de Loops Aninhados

Padrão básico:

```
for i = 1 to n:
    for j = 1 to n:
        operacao_basica()  # tempo constante c
```

Interpretação:

- Para cada valor de i (1 até n)
- Executamos j de 1 até n
- Cada execução custa tempo c

Somatório duplo:

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} c$$

Resolvendo o loop interno primeiro:

$$\sum_{j=1}^{n} c = n \times c$$

Depois o loop externo:

$$T(n) = \sum_{i=1}^{n} (n \times c) = n \times (n \times c)$$

Resultado - Complexidade Quadrática

Simplificando:

$$T(n) = n \times (n \times c) \tag{15}$$
$$= n^2 \times c \tag{16}$$
$$= cn^2 = O(n^2) \tag{17}$$

Interpretação visual:

- Loop interno: n operações
- Loop externo: repete n vezes
- Total: $n \times n = n^2$ operações

Objetivo

Determinar como o **uso de memória** cresce com o tamanho da entrada

Tipos de espaço:

- **Espaço fixo**: Código + variáveis simples
- **Espaço variável**: Estruturas dinâmicas

Foco principal: Espaço auxiliar

Componentes do Espaço Variável

O que consideramos:

1. **Espaço para dados de entrada**
2. **Espaço para dados de saída**
3. **Espaço auxiliar** (temporário)

Função de espaço: $S(n) =$ memória usada em função de n

Unidades de medida:

- Palavras de memória
- Bytes
- Número de elementos

Metodologia - Análise Espacial

Processo similar à análise temporal:

1. Identificar estruturas de dados usadas
2. Contar células de memória necessárias
3. Expressar em função de n
4. Determinar ordem de crescimento

Notação: Usamos O, Ω, Θ como na análise temporal

Algoritmos Recursivos - Consideração Especial

Stack de chamadas recursivas

Cada chamada usa:

- Espaço para parâmetros
- Espaço para variáveis locais
- Endereço de retorno

Fórmula geral:

$$\text{Espaço total} = \text{Profundidade máxima} \times \text{Espaço por chamada}$$

Exemplos - Recursão e Espaço

Casos comuns:

- **Fibonacci recursivo**: O(n)
  - Profundidade: n
  - Espaço por chamada: O(1)
- **Busca binária recursiva**: O($\log n$)
  - Profundidade: $\log n$
  - Espaço por chamada: O(1)
- **Merge Sort**: O(n)
  - Profundidade: $\log n$, mas arrays auxiliares: O(n)

Exercício 1 - Operação Básica

Identifique a operação básica

Para cada algoritmo a seguir, determine qual é a operação que deve ser contada para análise de complexidade.

Exercício 1A - Busca em Array

```
for i = 1 to n:
    if array[i] == x:
        return i
```

Pergunta: Qual é a operação básica?

a) Incremento do loop (i++)

b) Comparação (array[i] == x)

c) Return

d) Acesso ao array (array[i])

Exercício 1B - Soma de Elementos

```
soma = 0
for i = 1 to n:
    soma = soma + array[i]
```

Pergunta: Qual é a operação básica?

a) Atribuição (soma = ...)

b) Adição (soma + array[i])

c) Acesso ao array (array[i])

d) Incremento do loop

Exercício 1C - Multiplicação de Matrizes

```
for i = 1 to n:
    for j = 1 to n:
        for k = 1 to n:
            C[i][j] += A[i][k] * B[k][j]
```

Pergunta: Qual é a operação básica?

a) Multiplicação (A[i][k] * B[k][j])

b) Adição (+=)

c) Acesso aos arrays

d) Incrementos dos loops

Respostas - Exercício 1

Respostas com justificativas:

A) b) Comparação (array[i] == x)

- É executada a cada iteração
- Determina o fluxo do algoritmo

B) b) Adição (soma + array[i])

- Operação aritmética principal
- Executada exatamente n vezes

C) a) Multiplicação (A[i][k] * B[k][j])

- Operação mais custosa computacionalmente
- Executada n^3 vezes

Exercício 2 - Análise de Complexidade

Determine a complexidade temporal

Para cada fragmento de código, encontre T(n) e a classificação Big O.

Exercício 2A - Loop Simples

```
for i = 1 to n:
    print(i)
```

Análise:

- Quantas iterações?
- Qual é $T(n)$?
- Qual é a complexidade $O(?)$?

Exercício 2B - Loop com Divisão

```
for i = 1 to n/3:
    operacao_constante()
```

Análise:

- Quantas iterações?
- Qual é T(n)?
- A constante 3 afeta a complexidade?

Exercício 2C - Loop com Incremento

```
for i = 1 to n by 2: // incremento 2
    operacao_constante()
```

Análise:

- Quantas iterações? (incremento de 2)
- Qual é T(n)?
- Qual é a complexidade final?

Respostas - Exercício 2

Análises completas:

A) $T(n) = n \times c = O(n)$ Linear

- Loop executa exatamente n vezes

B) $T(n) = \frac{n}{3} \times c = O(n)$ Linear

- Constante não altera a ordem de crescimento

C) $T(n) = \frac{n}{2} \times c = O(n)$ Linear

- Incremento de 2 reduz iterações, mas mantém linearidade

Princípio: Constantes multiplicativas são ignoradas em Big O

Exercício 3 - Análise Espacial

Determine a complexidade espacial

Analise o uso de memória dos algoritmos a seguir.

Exercício 3A - Cópia de Array

```
int resultado[n]; // array que vai salvar copia
for i = 1 to n: // n iteracoes
    resultado[i] = array[i] * 2; // copiando os valores do
        array multiplicados por 2
return resultado; // retornando o array de valores
    multiplicados por 2
```

Análise espacial:

- Quais estruturas de dados são criadas?
- Quantos elementos cada uma tem?
- Qual é S(n)?

Exercício 3B - Matriz Transposta

```
int transposta[n][n]; \\ definindo vari vel NxN para
    armazenar matriz transposta
for i = 1 to n: \\ n iteracoes
    for j = 1 to n: // n iteracoes
        transposta[j][i] = matriz[i][j]; // Operacao basica
            de transposicao vai acontecer N^2
return transposta; // retornando matriz transposta
```

Análise espacial:

- Qual é o tamanho da matriz transposta?
- Há outras estruturas auxiliares?
- Qual é S(n)?

Respostas - Exercício 3

Análises espaciais:

A) $S(n) = n \rightarrow$ O(n)

- Array `resultado`: n elementos
- Variáveis do loop: O(1)
- Total: n + O(1) = O(n)

B) $S(n) = n^2 \rightarrow$ O(n^2)

- Matriz `transposta`: n × n elementos
- Variáveis do loop: O(1)
- Total: n^2 + O(1) = O(n^2)

Regra: Conte apenas estruturas dinâmicas criadas pelo algoritmo

Resumo da Aula

O que aprendemos:

- **Complexidade Linear O(n)**: loops simples
- **Complexidade Logarítmica O(log n)**: divisão binária
- **Complexidade Quadrática O(n²)**: loops aninhados
- **Análise Espacial**: uso de memória

Notação Big O, Omega e Theta

Tópicos da próxima aula:

- Definições formais de O, Ω, Θ
- Demonstrações matemáticas rigorosas
- Propriedades e teoremas
- Análise de caso médio, melhor e pior
- Comparação entre diferentes algoritmos