Cálculo da Complexidade de Algoritmos Iterativos - Parte 2

Pedro Ximenes

Universidade Católica de Pernambuco (UNICAP)

Análise de Algoritmos – 2026.1

Roteiro da Aula

1. Revisão da Aula Anterior
2. Ordenação Quadrática
3. Melhor Caso, Pior Caso e Caso Médio
4. Novos Padrões de Análise
5. Técnicas Eficientes
6. Resumo da Aula
7. Exercícios
8. Gabarito dos Exercícios
9. Desafio – Mini-Projeto
10. Fechamento

Revisão essencial da Aula 4

Método dos 3 passos

1. Identificar as operações primitivas.
2. Contar quantas vezes cada operação acontece no pior caso.
3. Somar os custos e simplificar pela ordem de crescimento.

Padrões já conhecidos

Padrão	**Estrutura**	**Classe**
Sem loop	sequência simples	$O(1)$
1 loop	percorrer array uma vez	$O(n)$
Loop com divisão por 2	descartar metade a cada passo	$O(\log n)$
Dois loops aninhados	comparar todos com todos	$O(n^2)$
Três loops aninhados	visitar triplas	$O(n^3)$
Loops sequenciais	somar custos e pegar o maior termo	$O(\max)$

Selection Sort: intuição

Ideia do algoritmo

A cada rodada, procuramos o menor elemento da parte ainda não ordenada do vetor. Depois disso, colocamos esse menor elemento na primeira posição disponível.

Como o vetor evolui

- O vetor vai sendo dividido em duas partes.
- À esquerda: parte já organizada.
- À direita: parte ainda desorganizada.
- Em cada rodada, um novo elemento entra na parte organizada.

Selection Sort: pseudocódigo

Pseudocódigo comentado

```
1  algoritmo SelectionSort(V, n)
2      para i de 0 ate n-2 faca
3          // No inicio da rodada, assume que o menor esta em i
4          i_menor ← i
5          para j de i+1 ate n-1 faca
6              // Se aparecer valor menor, atualiza a posicao
                   do menor
7              se V[j] < V[i_menor] entao
8                  i_menor ← j
9              fim se
10         fim para
11         // Ao fim da rodada, coloca o menor valor na posicao
               i
12         troca(V[i], V[i_menor])
13     fim para
14 fim algoritmo
```

Selection Sort: exemplo de entrada

Entrada

$V = [7, 4, 5, 2]$

Rodada 1: $i = 0$

- Começamos assumindo que o menor está na posição 0.
- Então, por enquanto, o menor è 7.
- Comparamos com 4, depois com 5, depois com 2.
- O menor encontrado no final da rodada è 2.
- Trocamos 7 com 2.

$$[7, 4, 5, 2] \rightarrow [2, 4, 5, 7]$$

Selection Sort: continuando o exemplo

Rodada 2: $i = 1$

Agora olhamos apenas para a parte $[4, 5, 7]$. O menor já está na frente, então o vetor não muda.

$$[2, 4, 5, 7] \rightarrow [2, 4, 5, 7]$$

Rodada 3: $i = 2$

Agora olhamos apenas para a parte $[5, 7]$. Mais uma vez, o menor já está na frente.

$$[2, 4, 5, 7] \rightarrow [2, 4, 5, 7]$$

Observação

Mesmo quando o vetor quase não muda, o algoritmo ainda faz as comparações.

Selection Sort: contando as comparações

Quantas comparações acontecem?

- Na primeira rodada, comparamos com $n-1$ elementos.
- Na segunda rodada, comparamos com $n-2$ elementos.
- Na terceira rodada, comparamos com $n-3$ elementos.
- E assim por diante, até restar apenas 1 comparação.

$$(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}$$

Selection Sort: conclusão

Estrutura de custo

O custo dominante não vem das trocas. O custo dominante vem da busca repetida pelo menor elemento.

Resultado

$$\boxed{SelectionSort \in \Theta(n^2)}$$

Conclusão

O Selection Sort **sempre** faz essencialmente a mesma quantidade de comparações. Por isso, melhor caso, pior caso e caso médio ficam na mesma classe.

Bubble Sort: intuição

Ideia do algoritmo

Comparamos pares adjacentes. Quando um par está fora de ordem, fazemos uma troca. Ao final de cada passada, o maior elemento da parte não ordenada vai para o final.

Efeito de cada passada

Depois da primeira passada, o maior valor já fica no lugar certo. Depois da segunda passada, o segundo maior também fica no lugar certo.

Bubble Sort otimizado: pseudocódigo

Pseudocódigo comentado

```
1  algoritmo BubbleSortOtimizado(V, n)
2      para i de 0 ate n-2 faca
3          // A flag registra se houve troca nesta passada
4          trocou ← falso
5          para j de 0 ate n-2-i faca
6              // Se o par estiver fora de ordem, corrige
                   localmente
7              se V[j] > V[j+1] entao
8                  troca(V[j], V[j+1])
9                  trocou ← verdadeiro
10             fim se
11         fim para
12         // Se nada mudou, o vetor ja estava ordenado
13         se trocou = falso entao
14             retorne
15         fim se
16     fim para
17 fim algoritmo
```

Bubble Sort: exemplo de entrada

Entrada

$V = [5, 3, 4, 1]$

Passada 1

- Compara 5 e 3: troca.
- Compara 5 e 4: troca.
- Compara 5 e 1: troca.

$$[5, 3, 4, 1] \rightarrow [3, 4, 1, 5]$$

Leitura

Ao fim da primeira passada, o 5 já está em sua posição definitiva.

Bubble Sort: continuando o exemplo

Passada 2

- Compara 3 e 4: não troca.
- Compara 4 e 1: troca.

$$[3, 4, 1, 5] \rightarrow [3, 1, 4, 5]$$

Leitura

Ao fim da segunda passada, o 4 também vai para sua posição correta.

Bubble Sort: última passada relevante

Passada 3

- Compara 3 e 1: troca.

$$[3, 1, 4, 5] \rightarrow [1, 3, 4, 5]$$

Leitura

Depois disso, o vetor inteiro está ordenado.

Bubble Sort: pior caso

Quando acontece o pior caso?

Quando o vetor começa em ordem decrescente, porque quase toda comparação gera troca.

- Primeira passada: $n - 1$ comparações.
- Segunda passada: $n - 2$ comparações.
- Terceira passada: $n - 3$ comparações.
- E assim por diante.

Padrão

Surge novamente uma soma triangular.

Bubble Sort: conta do pior caso

$$(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}$$

$$\boxed{\text{Pior caso } \in \Theta(n^2)}$$

Bubble Sort: melhor caso

Quando acontece o melhor caso?

Quando o vetor já está ordenado.

O que muda com a versão otimizada?

- Fazemos uma primeira passada completa.
- Nenhuma troca acontece.
- A flag continua falsa.
- O algoritmo termina imediatamente.

$$\text{Melhor caso} \in \Theta(n)$$

Selection Sort x Bubble Sort

Algoritmo	**Melhor caso**	**Pior caso**
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort otimizado	$\Theta(n)$	$\Theta(n^2)$

Comparação importante

Dois algoritmos podem parecer semelhantes porque ambos usam dois loops, mas detalhes da execução mudam o melhor caso.

Por que falamos em casos diferentes?

Definições

- **Melhor caso**: entrada mais favorável.
- **Pior caso**: entrada mais custosa.
- **Caso médio**: comportamento esperado em entradas comuns.

Foco da análise

O foco principal continua sendo o pior caso, porque ele fornece uma garantia clara sobre o desempenho.

Exemplo simples: busca linear

Problema

Procurar o valor 8 em um vetor.

Entrada 1

$[8, 4, 3, 1]$

- O algoritmo olha a primeira posição.
- Já encontra o valor procurado.

$$\boxed{\text{Melhor caso } \in \Theta(1)}$$

Busca linear: pior caso e caso médio

Entrada 2

$[4, 3, 1, 8]$

- O algoritmo precisa olhar quase tudo para achar 8.

Entrada 3

$[4, 3, 1, 2]$

- O algoritmo olha todas as posições e não encontra 8.

$$\boxed{\text{Pior caso } \in \Theta(n)}$$

Caso médio

Em média, também esperamos um número linear de comparações.

Quando a entrada tem dois tamanhos

Situação

Nem sempre uma entrada è descrita apenas por n. Uma matriz, por exemplo, tem número de linhas e número de colunas.

Pergunta certa

Se o algoritmo percorre uma matriz, o custo depende de quantas linhas existem e de quantas colunas existem.

Exemplo simples: busca em matriz

Pseudocódigo comentado

```
1  algoritmo BuscaMatriz(M, n, m, x)
2      // Percorre as linhas da matriz
3      para i de 0 ate n-1 faca
4          // Percorre as colunas da linha atual
5          para j de 0 ate m-1 faca
6              // Testa se o valor procurado foi encontrado
7              se M[i][j] = x entao
8                  retorne verdadeiro
9              fim se
10         fim para
11     fim para
12     retorne falso
13 fim algoritmo
```

Busca em matriz: exemplo de entrada

Entrada

$$M = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix} \quad x = 9$$

- A matriz tem 2 linhas.
- Cada linha tem 3 colunas.
- Como 9 não está na matriz, visitamos todas as 6 posições.

$$2 \cdot 3 = 6$$

Busca em matriz: conclusão

Leitura da estrutura

- Loop externo: n linhas.
- Loop interno: m colunas para cada linha.

$$\boxed{f(n, m) \in O(n \cdot m)}$$

Observação

Se a matriz for quadrada, isto è, se $n = m$, recuperamos o padrão $O(n^2)$.

Exemplo simples: teste de primalidade

Pseudocódigo comentado

```
1  algoritmo EhPrimo(n)
2      // Elimina os casos menores que 2
3      se n < 2 entao
4          retorne falso
5      fim se
6      i ← 2
7      enquanto i * i ≤ n faca
8          // Se achou um divisor, n nao e primo
9          se n mod i = 0 entao
10             retorne falso
11         fim se
12         // Passa ao proximo candidato a divisor
13         i ← i + 1
14     fim enquanto
15     retorne verdadeiro
16 fim algoritmo
```

Análise – Verificação de Primalidade

Por que só verificar até $\sqrt{n}$?

Se n não é primo, então existe $n = a \times b$ com $a \leq b$, então:

$$a \leq \sqrt{n} \leq b$$

(Se ambos fossem maiores que $\sqrt{n}$, então $a \times b > \sqrt{n} \times \sqrt{n} = n$, impossível.) Logo, pelo menos um divisor satisfaz $\leq \sqrt{n}$. Se n tem um divisor maior que $\sqrt{n}$, então também tem um menor.

Análise de complexidade:

- O loop executa enquanto $i^2 \leq n$.
- Isso significa $i \leq \sqrt{n}$.
- Número de iterações: $\sqrt{n} - 1$.

$$\boxed{f(n) \in O(\sqrt{n})}$$

Eficiência

Para $n = 1.000.000$: apenas ≈ 1.000 iterações em vez de 1.000.000!

Teste de primalidade: exemplo de entrada

Entrada 1

$n = 29$

- Verificamos $i = 2, 3, 4, 5$.
- Não precisamos testar 6, 7, 8, ..., 28.
- Isso acontece porque $\sqrt{29} \approx 5{,}38$.

Entrada 2

$n = 30$

- Já em $i = 2$, descobrimos que 30 è divisível por 2.
- O algoritmo pode parar bem cedo.

Teste de primalidade: conclusão

Leitura da estrutura

O laço executa enquanto $i^2 \leq n$. Logo, o maior valor relevante de i è aproximadamente $\sqrt{n}$.

$$\boxed{f(n) \in O(\sqrt{n})}$$

Conclusão

Aqui o ponto importante não è decorar a prova algébrica completa. O ponto importante è perceber a relação entre a condição $i^2 \leq n$ e a raiz de n.

De onde aparece $O(\sqrt{n})$?

Sinal de alerta no pseudocódigo

Sempre que a condição de parada envolve algo como $i^2 \leq n$, vale suspeitar de $O(\sqrt{n})$.

Intuição

Se o laço para quando i^2 passa de n, então o valor de i não chega perto de n. Ele cresce apenas até a raiz de n.

Técnica dos Dois Ponteiros

O que é?

Uma técnica que utiliza **dois índices** (ponteiros) que se movem pelo array de forma coordenada, geralmente resultando em algoritmos $O(n)$ para problemas que, ingenuamente, seriam $O(n^2)$.

Variações comuns

- Dois ponteiros que se aproximam (início e fim)
- Dois ponteiros que avançam na mesma direção (lento e rápido)

Exemplo 1: palíndromo

Pseudocódigo comentado

```
1  algoritmo EhPalindromo(S, n)
2      esq ← 0
3      dir ← n - 1
4      enquanto esq < dir faca
5          // Se os extremos diferem, a palavra nao e
               palindromo
6          se S[esq] ≠ S[dir] entao
7              retorne falso
8          fim se
9          // Aproxima os ponteiros do centro
10         esq ← esq + 1
11         dir ← dir - 1
12     fim enquanto
13     retorne verdadeiro
14 fim algoritmo
```

Palíndromo: exemplo de entrada

Entrada

$S =$ radar

- Comparamos r com r.
- Depois comparamos a com a.
- O elemento do meio não precisa ser comparado com outro.

Leitura

Cada posição participa de no máximo uma comparação relevante.

$$\boxed{f(n) \in O(n)}$$

Exemplo 2: Two Sum em vetor ordenado

Problema

Dado um vetor ordenado, queremos saber se existem dois valores cuja soma è igual a k.

Pseudocódigo comentado

```
1  algoritmo TwoSum(V, n, k)
2      esq ← 0
3      dir ← n - 1
4      enquanto esq < dir faca
5          // Soma os dois extremos atuais
6          soma ← V[esq] + V[dir]
7          se soma = k entao
8              retorne verdadeiro
9          senao se soma < k entao
10             // Se a soma ficou pequena, aumenta o valor da esquerda
11             esq ← esq + 1
12         senao
13             // Se a soma ficou grande, diminui o valor da direita
14             dir ← dir - 1
15         fim se
16     fim enquanto
17     retorne falso
18 fim algoritmo
```

Two Sum: exemplo de entrada

Entrada

$V = [1, 3, 5, 7, 9]$ e $k = 8$

- Começamos com 1 e 9. Soma $= 10$.
- A soma ficou grande demais, então movemos o ponteiro da direita.
- Agora testamos 1 e 7. Soma $= 8$.
- Encontramos a resposta.

Leitura

Em cada passo, um dos ponteiros anda uma casa. Por isso, o total de movimentos è linear.

$$\boxed{f(n) \in O(n)}$$

Resumo operacional

Padrão	**Como reconhecer**	**Classe**
Selection Sort	soma triangular de comparações	$\Theta(n^2)$
Bubble Sort	soma triangular no pior caso	$\Theta(n^2)$
Melhor caso do Bubble	flag encerra cedo	$\Theta(n)$
Dois parâmetros	linhas $\times$ colunas	$O(n \cdot m)$
Raiz de n	condição com i^2	$O(\sqrt{n})$
Dois ponteiros	cada ponteiro anda poucas vezes	$O(n)$

Fechamento

Os padrões apresentados cobrem os casos mais frequentes desta unidade e servem como base para análises mais avançadas.

Exercícios: orientações

Estratégia de resolução

1. Identificar o padrão dominante de repetição.
2. Contar o custo de cada parte e montar a soma.
3. Simplificar pela ordem de crescimento e comparar com o gabarito.

Exercício 1

Determine a complexidade do algoritmo abaixo.

```
1  algoritmo Exercicio1(n)
2      // Comeca com i no valor n
3      i ← n
4      enquanto i ≥ 1 faca
5          // Executa n operacoes em cada rodada do while
6          para j de 1 ate n faca
7              // O(1)
8          fim para
9          // Reduz i pela metade
10         i ← i / 2
11     fim enquanto
12 fim algoritmo
```

Exercício 2

Determine a complexidade do algoritmo abaixo.

```
1  algoritmo Exercicio2(n)
2      para i de 1 ate n faca
3          // Reinicia j em 1 para cada valor de i
4          j ← 1
5          enquanto j < i faca
6              // Dobra j a cada passo
7              j ← j * 2
8          fim enquanto
9      fim para
10 fim algoritmo
```

Exercício 3

Determine a complexidade do algoritmo abaixo.

```
1  algoritmo Exercicio3(n, m)
2      para i de 0 ate n-1 faca
3          para j de 0 ate m-1 faca
4              // Para cada par (i,j), percorre n posicoes
5              para k de 0 ate n-1 faca
6                  // O(1)
7              fim para
8          fim para
9      fim para
10 fim algoritmo
```

Exercício 4

Determine a complexidade do algoritmo abaixo.

```
1  algoritmo Exercicio4(V, n)
2      para i de 0 ate n-1 faca
3          para j de 0 ate n-1-i faca
4              // Corrige pares adjacentes fora de ordem
5              se V[j] > V[j+1] entao
6                  troca(V[j], V[j+1])
7              fim se
8          fim para
9      fim para
10 fim algoritmo
```

Exercício 5

Qual è a complexidade no pior caso?

```
1  algoritmo Exercicio5(V, n)
2      para i de 0 ate n-1 faca
3          // Comeca no elemento atual e tenta desloca-lo para
               a esquerda
4          j ← i
5          enquanto j > 0 e V[j] < V[j-1] faca
6              // Troca o elemento atual com o anterior
7              troca(V[j], V[j-1])
8              j ← j - 1
9          fim enquanto
10     fim para
11 fim algoritmo
```

Exercício 6

Determine a complexidade do algoritmo abaixo.

```
1  algoritmo Exercicio6(n)
2      s ← 0
3      para i de 1 ate n faca
4          // Para cada i, repete i*i vezes
5          para j de 1 ate i*i faca
6              s ← s + 1
7          fim para
8      fim para
9      retorne s
10 fim algoritmo
```

Exercício 7

Determine a complexidade do algoritmo abaixo.

```
1 algoritmo Proposto1(n)
2     para i de n ate 1 (decrementando) faca
3         // O limite inferior depende de i
4         para j de i ate n faca
5             // O(1)
6         fim para
7     fim para
8 fim algoritmo
```

Exercício 8

Determine a complexidade do algoritmo abaixo.

```
1 algoritmo Proposto2(n)
2     i ← 1
3     enquanto i*i*i ≤ n faca
4         // Trabalho constante
5         i ← i + 1
6     fim enquanto
7 fim algoritmo
```

Exercício 9

Determine a complexidade do algoritmo abaixo.

```
1  algoritmo Proposto3(n)
2      para i de 1 ate n faca
3          // Reinicia j no valor n
4          j ← n
5          enquanto j > i faca
6              // Diminui j em blocos de tamanho i
7              j ← j - i
8          fim enquanto
9      fim para
10 fim algoritmo
```

Exercício 10

Determine a complexidade do algoritmo abaixo.

```
algoritmo Proposto4(n)
    para i de 1 ate n faca
        // Avanca no loop interno usando passo i
        para j de 1 ate n (passo i) faca
            // O(1)
        fim para
    fim para
fim algoritmo
```

Gabarito 1

- O while divide i por 2 a cada rodada: $\Theta(\log n)$.
- O loop interno executa n vezes em cada rodada.

$$\boxed{\Theta(n \log n)}$$

Gabarito 2

- Para cada i, o while dobra j.
- O custo interno è $\Theta(\log i)$.
- Somando para todos os valores de i, obtemos $\Theta(n \log n)$.

$$\boxed{\Theta(n \log n)}$$

Gabarito 3

- Primeiro loop: n vezes.
- Segundo loop: m vezes.
- Terceiro loop: n vezes.

$$\boxed{O(n^2 \cdot m)}$$

Gabarito 4

- O loop interno executa $(n-1)+(n-2)+\ldots+1$.
- Trata-se do padrão do Bubble Sort.

$$\boxed{\Theta(n^2)}$$

Gabarito 5

- Trata-se do padrão do Insertion Sort.
- No pior caso, o while anda i posições para cada i.
- Isso gera a soma $1 + 2 + \ldots + (n - 1)$.

$$\boxed{\text{Pior caso } \Theta(n^2)}$$

Gabarito 6

- Para cada i, o loop interno executa i^2 vezes.
- A soma total è $\sum_{i=1}^{n} i^2$.

$$\boxed{\Theta(n^3)}$$

Gabarito 7

- Para cada i, o loop interno executa aproximadamente $n - i + 1$ vezes.
- Isso produz outra soma triangular.

$$\boxed{\Theta(n^2)}$$

Gabarito 8

- O while para quando $i^3 > n$.
- Logo, i cresce até aproximadamente $\sqrt[3]{n}$.

$$\boxed{\Theta(n^{1/3})}$$

Gabarito 9

- Para cada i, o while anda em saltos de tamanho i.
- O custo aproximado por valor de i è n/i.
- A soma produz nH_n.

$$\boxed{\Theta(n \log n)}$$

Gabarito 10

- Para cada i, o loop interno avança em passo i.
- Assim, ele visita aproximadamente n/i posições.
- Somando para todos os valores de i, obtemos $\Theta(n \log n)$.

$$\boxed{\Theta(n \log n)}$$

Desafio: Mini-Projeto de Análise

Objetivo

Implementar e analisar **três soluções** para o problema de verificar se um array contém dois elementos cuja soma é igual a um valor k.

O que entregar

1. Código em qualquer linguagem (Python, Java, C, etc.)
2. Relatório com:
   - Análise de complexidade de cada solução (passo a passo)
   - Comparação teórica das três abordagens
   - Medições empíricas com diferentes tamanhos de entrada
   - Gráfico comparativo

Critérios para o desafio

- Implementar corretamente as 3 soluções.
- Explicar a complexidade de cada uma com clareza.
- Comparar teoria e experimento.
- Discutir a troca entre tempo e memória.

Próxima Aula

Aula 6 – Algoritmos Recursivos

- Relações de recorrência.
- Método da substituição.
- Método da àrvore de recursão.
- Teorema Mestre.

Dúvidas?

Referências

T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Algoritmos: Teoria e Prática*. 3a ed, Elsevier, 2012. Capítulos 2 e 3.

J. A. B. Monteiro. *Notas de Aula – Estruturas de Dados*. Disponível em: joaoarthurbm.github.io/eda.