45EST – Algoritmos e Estruturas de Dados

Complexidade de algoritmos

Prof. Marcelo de Souza

UDESC Ibirama
Bacharelado em Engenharia de Software

marcelo.desouza@udesc.br
Versão compilada em 17 de setembro de 2018

Leitura obrigatória:

- Capítulo 4 de Goodrich and Tamassia [2013] – Ferramentas de análise.

Leitura complementar:

- Capítulo 2 de Preiss [2001] – Análise de algoritmos.
- Capítulo 3 de Preiss [2001] – Notação assintótica.

Conceitos básicos

Algumas definições:

- **Algoritmo:** sequência de passos para realizar uma tarefa.
- **Estrutura de dados:** forma sistemática de organizar e acessar os dados.

Como definir se eles são bons?
Como comparar dois algoritmos ou duas estruturas?

Análise de algoritmos:

- Correção / corretude.
- Complexidade.
  - Tempo de execução.

- Consumo de memória.

Como analisar a **complexidade**?

- Executar o algoritmo e plotar os resultados.
  - Sensível às entradas escolhidas.
  - Comparação prejudicada. $\leftarrow$ software, hardware, entradas...
  - Necessário implementar e executar todas as opções.
- **Solução:** métodos analíticos.
  - Complexidade $\rightarrow$ função $f(n)$. $\leftarrow$ n é o tamanho da entrada.

Exemplo:

- Algoritmo A possui complexidade n.
  - Entrada de tamanho n – tempo máximo de processamento cn.
    - c: constante que modela a variação por hardware e software.
  - Melhor que um algoritmo B, com complexidade n^2.

Análise de algoritmos

Ideia geral:

- Contar o número de operações primitivas executadas.
- Cada operação primitiva terá um tempo de processamento constante.
- Quanto menor o número de operações, melhor o algoritmo.
- Operações primitivas:
  - Atribuição de valores.

- Operações aritméticas.
- Comparação de valores.
- Acesso a uma posição de um vetor.
- Recuperar a referência de um objeto.
- Chamada de método.
- Retorno de um método.

Exemplo:

Algoritmo `arrayMax(A, n)`:

```
1 // Entrada: um vetor A com n ≥ 1 elementos inteiros.
2 // Saída: o maior elemento de A.
3
4 currentMax ← A[0]
5 for i ← 1 to n − 1 do
6     if currentMax < A[i] then
7         currentMax ← A[i]
8
9 return currentMax
```

Contando as operações do algoritmo `arrayMax(A, n)`:

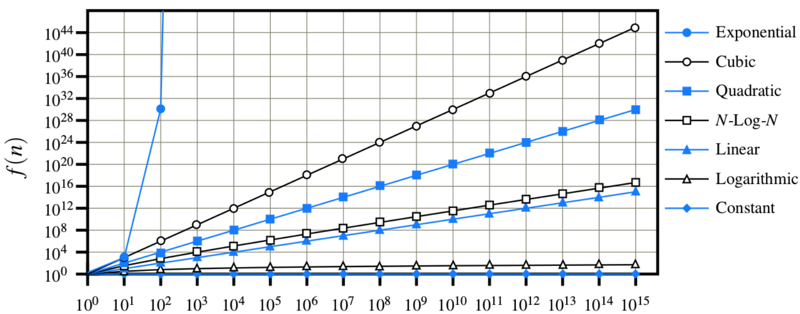
Linha	Operações
4	2
5	1 inicialização + n comparações + $2(n-1)$ incrementos = $3n-1$
6	$2(n-1) = 2n-2$
7	de 0 [nunca entra] a $2(n-1) = 2n-2$ [entra todas as vezes]
9	1

- Complexidade no **melhor caso**: $5n$.
- Complexidade no **pior caso**: $7n - 2$.
  - Para qualquer entrada de tamanho n, o algoritmo executará no máximo $7n - 2$ operações.
  - É simples de conduzir/estimar.
  - Sabe-se que o algoritmo analisado nunca será pior que a estimativa.

Principais tipos de função de complexidade:

- Constante $\rightarrow 1$
- Logarítmica $\rightarrow \log n$
- Linear $\rightarrow n$
- n-log-n $\rightarrow n \log n$
- Quadrática $\rightarrow n^2$
- Cúbica $\rightarrow n^3$
- Polinomial $\rightarrow n^k$
- Exponencial $\rightarrow a^n$

Taxas de crescimento das funções de complexidade:



Exercícios:

- Calcule o número máximo de passos (pior caso) para o algoritmo abaixo.

```
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        if(grade[i][j] != 0) {
            // Operação com complexidade 1
        }
    }
}
```

Resposta:

- Laço externo executa $3n + 2$ operações.
  - Inicialização (1), comparações $(n+1)$ e incremento $(2n)$.
- Laço interno possui a mesma complexidade, mas é executado n vezes.
  - Logo, $n(3n+2) = 3n^2 + 2n$.
- Condicional é executado n^2 vezes. Logo, $2n^2$.
- Operação é executada n^2 vezes, no pior caso.
- Resultado: $6n^2 + 5n + 2$.

Análise assintótica

Problemas da análise completa:

- Muito detalhado.
- Oneroso. $\leftarrow$ ver exemplo do algoritmo `arrayMax`.
- Pouca precisão. $\leftarrow$ não considera baixo nível, hardware...

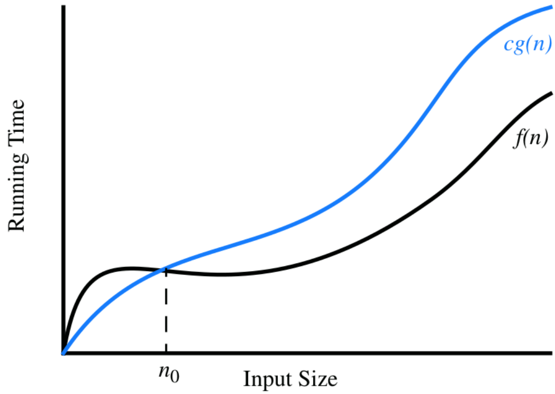
O que importa:

- A **taxa de crescimento** do tempo de execução em função de n.
- Para modelar isso, usamos a notação O.

Notação O

"Sejam $f(n)$ e $g(n)$ funções mapeando números inteiros (tamanho da entrada) em reais (tempo de execução). Dizemos que $f(n)$ é $O(g(n))$ se existe uma constante real $c > 0$ e uma constante inteira $n_0 \geq 1$ tais que $f(n) \leq cg(n)$ para todo inteiro $n \geq n_0$."

Exemplo de relação definida pela notação O:



Proposição. A função $7n - 2$ é $O(n)$.

Demonstração. Ao escolher $c = 7$ e $n_0 = 1$, temos que $7n - 2 \leq cn$, para todo inteiro $n \geq n_0$. Logo, $7n - 2$ é $O(n)$. ■

Logo, a complexidade do algoritmo `arrayMax` é $O(n)$, isto é, complexidade linear.

Observações:

- A notação O determina que uma função $f(n)$ é "menor ou igual a" outra função $g(n)$, descontando-se um fator constante, a medida que n cresce para infinito.
- Um algoritmo A com complexidade $O(n^2)$ nunca terá um tempo de execução superior a n^2, para uma determinada entrada n.

Regras para definir a complexidade O:

- Função polinomial: sempre considerar o maior grau.
  - $5n^4 + 3n^3 + 2n^2 + 4n + 1$ é $O(n^4)$.
  - $n^3 + 600n$ é $O(n^3)$.
- Constantes e multiplicadores são eliminados.
  - $2^{n+2} + 4$ é $O(2^n)$.
  - $4n^3$ é $O(n^3)$.
- Função mista: sempre considerar o termo de maior complexidade.
  - $5n^2 + 3n \log n + 2n + 5$ é $O(n^2)$.
  - $3 \log n + 2$ é $O(\log n)$.
  - $2n + 100 \log n$ é $O(n)$.
- Sempre considerar a menor complexidade possível para a notação O.
  - É verdade que $4n^2 + 10$ é $O(n^4)$, mas é melhor dizer que é $O(n^2)$.
- Sempre considerar a representação mais simples.
  - $4n^2 + 2 \log n$ é $O(n^2)$, o que é melhor que $O(n^2 + \log n)$.

Outras notações:

- Notação Ω: define que a função é maior ou igual a $\Omega(g(n))$.
- Notação Θ: define que a função cresce na mesma taxa que $\Theta(g(n))$.

Comparação

Comparação 1: crescimento do tempo de execução em função do tamanho da entrada.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Comparação 2: tamanho máximo do problema que diferentes algoritmos podem resolver em tempos limite distintos.

Running	Maximum Problem Size (n)		
Time (μs)	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Comparação 3: aumento no tamanho máximo a ser resolvido usando um computador 256 vezes mais rápido (m é o tamanho máximo da tabela anterior).

Running Time	New Maximum Problem Size
$400n$	$256m$
$2n^2$	$16m$
2^n	$m+8$

Exemplos

Exemplo 1

Algoritmo `sumNumbers(n1, n2)`:

```
1 // Soma dois números inteiros.
2 public static int sumNumbers(int n1, int n2) {
3     int result = n1 + n2;
4     return result;
5 }
```

Análise:

- Linha 3: 2 operações.
- Linha 4: 1 operação.
- Complexidade constante: $O(1)$.
- Algoritmo de tempo constante.

Exemplo 2

Algoritmo `repeat1(char c, int n)`:

```
// Compõe uma String com o caracter c repetido n vezes.
public static String repeat1(char c, int n) {
    String answer = "";
    for (int j = 0; j < n; j++)
        answer += c;
    return answer;
}
```

Análise:

- O comando `answer += c` implica em criar uma nova String, copiar cada caracter da String antiga para ela, e acrescentar o caracter c.

- A cada iteração, a linha 5 executa k operações, sendo k o tamanho da String `answer`.
  - Operações: $\sum_{j=0}^{n-1} j = \dfrac{n(n+1)}{2}$. Logo, sua complexidade é $O(n^2)$.
- Linhas 3, 4 e 6 apresentam operações em tempo constante.
- Complexidade quadrática: $O(n^2)$.
- Algoritmo de tempo quadrático.

Exemplo 3

Algoritmo `repeat2(char c, int n)`:

```
1 // Compõe uma String com o caracter c repetido n vezes.
2 public static String repeat2(char c, int n) {
3    StringBuilder sb = new StringBuilder();
4    for (int j = 0; j < n; j++)
5       sb.append(c);
6    return sb.toString();;
7 }
```

Análise:

- O comando `sb.append(c)` é executado em tempo constante.
- Linhas 3, 4 e 6 apresentam operações em tempo constante.
- Complexidade linear: $O(n)$.
- Algoritmo de tempo linear.
- OBS: o algoritmo é o mesmo, o que muda é a estrutura de dados.

Exemplo 4

Algoritmo `disjoint1(int[] vA, int[] vB, int[] vC)`:

```
1  // Retorna true se não existe nenhum elemento comum nos três grupos.
2  // Cada vetor possui elementos distintos dentro de si.
3  public static boolean disjoint1(int[ ] vA, int[ ] vB, int[ ] vC) {
4     for (int a : vA)
5        for (int b : vB)
6           for (int c : vC)
7              if ((a == b) && (b == c))
8                 return false;
9     return true;
10 }
```

Análise:

- A operação constante da linha 7 é repetida $n \times n \times n = n^3$ vezes.
- Complexidade cúbica: $O(n^3)$.
- Algoritmo de tempo cúbico.

Exemplo 5

Algoritmo `disjoint2(int[] vA, int[] vB, int[] vC)`:

```
// Retorna true se não existe nenhum elemento comum nos três grupos.
// Cada vetor possui elementos distintos dentro de si.
public static boolean disjoint2(int[ ] vA, int[ ] vB, int[ ] vC) {
   for (int a : vA)
      for (int b : vB)
         if (a == b)
            for (int c : vC)
               if (a == c)
                  return false;
   return true;
}
```

Análise:

- Os laços das linhas 4 e 5 sempre são executados – $O(n^2)$.
- No máximo n pares (a, b) são iguais, portanto o laço da linha 7 é executado no máximo n vezes.
- Complexidade $n^2 + n = O(n^2)$.
- Algoritmo de tempo quadrático.

Exemplo 6

Algoritmo `unique1(int[] data)`:

```java
// Retorna true se não existe elemento duplicado no vetor.
public static boolean unique1(int[ ] data) {
   int n = data.length;
   for (int j = 0; j < n - 1; j++)
      for (int k = j + 1; k < n; k++)
         if (data[j] == data[k])
            return false;
   return true;
}
```

Análise:

- O laço interno é executado $(n-1) + (n-2) + \cdots + 2 + 1$ vezes.
- Complexidade quadrática $O(n^2)$.
- Algoritmo de tempo quadrático.

Exemplo 7

Algoritmo `unique2(int[] data)`:

```
// Retorna true se não existe elemento duplicado no vetor.
// O vetor é ordenado para verificar apenas os pares de elementos.
public static boolean unique2(int[ ] data) {
   int n = data.length;
   Arrays.sort(data);                 // Operação O(n log n)
   for (int j = 0; j < n - 1; j++)
      if (data[j] == data[j+1])
         return false;
   return true;
}
```

Análise:

- O vetor é percorrido apenas uma vez.
- Complexidade linear $O(n)$.
- Algoritmo de tempo linear.

Exemplo 8

Algoritmo `prefixAverage1(double[] x)`:

```
// Retorna um vetor onde cada posição j armazena a média dos
// elementos x[0]...x[j].
public static double[] prefixAverage1(double[] x) {
   int n = x.length;
   double[ ] a = new double[n];
   for (int j=0; j < n; j++) {
      double total = 0;
      for (int i=0; i <= j; i++)
         total += x[i];
      a[j] = total / (j+1);
   }
```

```
12      return a;
13  }
```

Análise:

- Linhas 4 e 12 são executadas em tempo constante.
- Linha 5 executa em tempo $O(n)$.
- Os laços (linhas 6 e 8) totalizam $O(n^2)$.
- Complexidade quadrática: $O(n^2)$.
- Algoritmo de tempo quadrático.

Exemplo 9

Algoritmo `prefixAverage2(double[] x)`:

```
// Retorna um vetor onde cada posição j armazena a média dos
// elementos x[0]...x[j].
public static double[] prefixAverage2(double[] x) {
    int n = x.length;
    double[] a = new double[n];
    double total = 0;
    for (int j=0; j < n; j++) {
        total += x[j];
        a[j] = total / (j+1);
    }
    return a;
}
```

Análise:

- O laço percorre o vetor uma única vez.
- Complexidade linear: $O(n)$.
- Algoritmo de tempo linear.

Atividades

1. Leia a respeito das sete funções usadas em Goodrich and Tamassia [2013] e as proposições e provas a respeito das mesmas.
2. Leia a respeito das técnicas de prova de correção de algoritmos apresentadas em Goodrich and Tamassia [2013].
3. Analise o tempo de execução do algoritmo `BinarySum` (Trecho de Código 3.34 de Goodrich and Tamassia [2013]) usando valores arbitrários para o parâmetro n.
4. Faça os seguintes exercícios de Goodrich et al. [2014].

R-4.1: Desenhe o gráfico das funções $8n$, $4n \log n$, $2n^2$, n^3 e 2^n usando uma escala logarítmica para os eixos x e y, isto é, se o valor da função $f(n)$ é y, desenhe esse ponto com a coordenada x em $\log x$ e a coordenada y em $\log y$.

R-4.2: O número de operações executadas pelos algoritmos `A` e `B` é $8n \log n$ e $2n^2$, respectivamente. Determine n_0 tal que `A` seja melhor que `B` para $n \geq n_0$.

R-4.3: O número de operações executadas pelos algoritmos `A` e `B` é $40n^2$ e $2n^3$, respectivamente. Determine n_0 de maneira que `A` seja melhor que `B` para $n \geq n_0$.

R-4.8: Ordene as funções a seguir por sua taxa assintótica de crescimento.

- $4n \log n + 2n$
- 2^{10}
- $2^{\log n}$
- $3n + 100 \log n$

- $4n$
- 2^n
- $n^2 + 10n$
- n^3
- $n \log n$

R-4.9: Forneça uma caracterização O em termos de n do tempo de execução do algoritmo abaixo.

```java
/** Returns the sum of the integers in given array. */
public static int example1(int[] arr) {
   int n = arr.length, total = 0;
   for (int j=0; j < n; j++)
      total += arr[j];
   return total;
}
```

R-4.10: Forneça uma caracterização O em termos de n do tempo de execução do algoritmo abaixo.

```java
/** Returns the sum of the integers with even index in given array. */
public static int example2(int[] arr) {
   int n = arr.length, total = 0;
   for (int j=0; j < n; j += 2)
      total += arr[j];
   return total;
}
```

R-4.11: Forneça uma caracterização O em termos de n do tempo de execução do algoritmo abaixo.

```java
/** Returns the sum of the prefix sums of given array. */
public static int example3(int[ ] arr) {
   int n = arr.length, total = 0;
```

```java
    for (int j=0; j < n; j++)
        for (int k=0; k <= j; k++)
            total += arr[j];
    return total;
}
```

R-4.12: Forneça uma caracterização O em termos de n do tempo de execução do algoritmo abaixo.

```java
/** Returns the sum of the prefix sums of given array. */
public static int example4(int[ ] arr) {
    int n = arr.length, prefix = 0, total = 0;
    for (int j=0; j < n; j++) {
        prefix += arr[j];
        total += prefix;
    }
    return total;
}
```

R-4.13: Forneça uma caracterização O em termos de n do tempo de execução do algoritmo abaixo.

```java
/** Returns the number of times second array stores sum of prefix sums from first. */
public static int example5(int[] first, int[] second) {
    int n = first.length, count = 0;
    for (int i=0; i < n; i++) {
        int total = 0;
        for (int j=0; j < n; j++)
            for (int k=0; k <= j; k++)
                total += first[k];
        if (second[i] == total) count++;
    }
    return count;
}
```

R-4.28: Para cada função $f(n)$ e tempo t da tabela a seguir, determine o maior tamanho de n para um problema P que pode ser resolvido em tempo t se o algoritmo para resolver P consome $f(n)$ microssegundos (uma das entradas já foi feita).

	1 segundo	1 hora	1 mês	1 século
$\log n$	$\approx 10^{300000}$			
n				
$n \log n$				
n^2				
2^n				

R-4.29: O algoritmo `A` executa uma computação em tempo $O(\log n)$ para cada entrada de um arranjo de n elementos. Qual o pior caso em relação ao tempo de execução de `A`?

R-4.30: Dado um arranjo `X` de n elementos, o algoritmo `B` escolhe $\log n$ elementos de `X`, aleatoriamente, e executa um cálculo em tempo $O(n)$ para cada um. Qual o pior caso em relação ao tempo de execução de `B`?

R-4.31: Dado um arranjo `X` de n elementos inteiros, o algoritmo `C` executa uma computação em tempo $O(n)$ para cada número par de `X` e uma computação em tempo $O(\log n)$ para cada elemento ímpar de `X`. Qual o melhor caso e o pior caso em relação ao tempo de execução de `C`?

R-4.32: Dado um arranjo `X` de n elementos, o algoritmo `D` chama o algoritmo `E` para cada elemento `X[i]`. O algoritmo `E` executa em tempo $O(i)$ quando é chamado sobre um elemento `X[i]`. Qual o pior caso em relação ao tempo de execução do algoritmo `D`?

R-4.34: Existe uma cidade bem conhecida (que ficará anônima aqui) cujos habitantes têm a reputação de gostarem de uma refeição somente se essa refeição for a melhor que já experimentaram na vida. Caso contrário, eles a odeiam. Assumindo que a qualidade das refeições está distribuída de maneira uniforme ao longo da vida da pessoa, qual o número esperado de habitantes dessa cidade que estão felizes com suas refeições?

P-4.60: Implemente `prefixAverage1` e `prefixAverage2`, e execute uma análise experimental dos seus tempos de execução. Visualize seus tempos de execução como uma função do tamanho da entrada usando um gráfico *di-log*.

P-4.61: Execute uma análise experimental cuidadosa que compare os tempos relativos de execução dos métodos apresentados nos exercícios R-4.16 a R-4.20.

P-4.62: Execute uma análise experimental para testar a hipótese de que o método da biblioteca Java, `java.util.Arrays.sort` executa em um tempo médio $O(n \log n)$.

P-4.63: Execute uma análise experimental para determinar o maior valor de n para cada um dos dois algoritmos para resolver o problema do elemento único, de maneira que o algoritmo execute em um minuto ou menos.

Referências

Goodrich, M. T. and Tamassia, R. (2013). *Estruturas de Dados & Algoritmos em Java*. Bookman Editora, 5th edition.

Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. (2014). *Data structures and algorithms in Java*. John Wiley & Sons, 6th edition.

Preiss, B. R. (2001). *Estruturas de dados e algoritmos: padrões de projetos orientados a objetos com Java*. Campus.