Algoritmos e Estruturas de Dados I

Noções Básicas de Complexidade de Algoritmos

Mirtha Lina Fernández Venero
Sala 529-2, Bloco A
mirtha.lina@ufabc.edu.br
http://professor.ufabc.edu.br/~mirtha.lina/aedi.html

6 de março de 2019

Agenda

Aulas Anteriores: Vetores, Listas, Pilhas, Filas?

- Dado um problema, como escolher a estrutura de dados apropriada?

Aulas Anteriores: Vetores, Listas, Pilhas, Filas?

- Dado um problema, como escolher a estrutura de dados apropriada? Depende das operações a serem realizadas e o custo delas

Aulas Anteriores: Vetores, Listas, Pilhas, Filas?

- Dado um problema, como escolher a estrutura de dados apropriada? Depende das operações a serem realizadas e o custo delas
- Custo? Dos recursos que serão usados pelo algoritmo que implementa a operação
  - número de linhas de código, dificuldade de implementação (?): depende da linguagem e o programador
  - tempo de execução e quantidade de memória interna/externa: depende do computador

Aulas Anteriores: Vetores, Listas, Pilhas, Filas?

- Dado um problema, como escolher a estrutura de dados apropriada? Depende das operações a serem realizadas e o custo delas
- Custo? Dos recursos que serão usados pelo algoritmo que implementa a operação
  - número de linhas de código, dificuldade de implementação (?): depende da linguagem e o programador
  - tempo de execução e quantidade de memória interna/externa: depende do computador

Como medir o custo ou complexidade de tempo e espaço dum algoritmo independentemente do hardware, da linguagem de programação, compilador, SO, estilo de codificação, etc?

Objetivos da Análise da Complexidade de Algoritmos

- ajudar a determinar qual algoritmo é mais eficiente para resolver um problema
- medir como o tempo ou espaço aumenta com relação ao tamanho da entrada

Objetivos da Análise da Complexidade de Algoritmos

- ajudar a determinar qual algoritmo é mais eficiente para resolver um problema
- medir como o tempo ou espaço aumenta com relação ao tamanho da entrada

Tamanho da entrada: número de elementos de dados que são relevantes na entrada do algoritmo. Varia dependendo do problema

- Número de elementos dum conjunto/sequência
- Dimensões duma matriz
- quantidade de bits na representação dum número
- número de vértices e/ou arestas dum grafo

Análise da Complexidade de Algoritmos*

- **Análise de caso pior**: considera as entradas para a qual o algoritmo tem o maior custo
- **Análise de caso melhor**: considera as entradas para as quais o algoritmo tem o menor custo
- **Análise de caso médio**: calcula a média do custo sobre todas entradas, assumindo uma distribuição

* Será estudado em profundidade na disciplina Análise de Algoritmos

Análise da Complexidade de Algoritmos*

- **Análise de caso pior**: considera as entradas para a qual o algoritmo tem o maior custo
- **Análise de caso melhor**: considera as entradas para as quais o algoritmo tem o menor custo
- **Análise de caso médio**: calcula a média do custo sobre todas entradas, assumindo uma distribuição

Importante

1. **O custo exato do algoritmo é irrelevante**. O importante é obter uma <u>boa</u> aproximação ou limite (*tigh bound*)
2. **Entradas pequenas são irrelevantes**. O importante é o comportamento do algoritmo quando o tamanho da entrada é grande (*asymptotic complexity*)

* Será estudado em profundidade na disciplina Análise de Algoritmos

Agenda

Análise Assintótica de Algoritmos

- Assume um modelo abstrato de computador com um conjunto **básico** de operações e seus custos
- O custo de tempo é uma função $T(n)$ onde n representa o tamano da entrada.

Análise Assintótica de Algoritmos

- Assume um modelo abstrato de computador com um conjunto **básico** de operações e seus custos
- O custo de tempo é uma função $T(n)$ onde n representa o tamano da entrada. **Exemplo**: Insertion-Sort

INSERTION-SORT(A)	*cost*
for $j = 2$ **to** $A.length$	c_1
$key = A[j]$	c_2
// Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0
$i = j - 1$	c_4
while $i > 0$ and $A[i] > key$	c_5
$A[i+1] = A[i]$	c_6
$i = i - 1$	c_7
$A[i+1] = key$	c_8

Ver Cormen cap 2.

Análise Assintótica de Algoritmos

- Assume um modelo abstrato de computador com um conjunto **básico** de operações e seus custos
- O custo de tempo é uma função $T(n)$ onde n representa o tamano da entrada. **Exemplo**: Insertion-Sort, $n = A.length$

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

Ver Cormen cap 2.

Análise Assintótica de Algoritmos

- Assume um modelo abstrato de computador com um conjunto **básico** de operações e seus custos
- O custo de tempo é uma função $T(n)$ onde n representa o tamano da entrada. **Exemplo**: Insertion-Sort, $n = A.length$

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$\quad i = j - 1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\quad\quad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$\quad\quad i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + (c_2 + c_4)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Análise Assintótica de Algoritmos

$$T(n) = c_1 n + (c_2 + c_4)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- **Caso melhor**: A já ordenado $\Rightarrow \forall j = 2 \ldots n, t_j = 1$

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
$\quad i = j - 1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\quad\quad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$\quad\quad i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Análise Assintótica de Algoritmos

$$T(n) = c_1 n + (c_2 + c_4)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- **Caso pior**: A em ordem decrescente $\Rightarrow \forall j = 2 \ldots n, t_j = j$

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j - 1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$$

Notações assintóticas ou de Bachmann-Landau (O)

Permitem descrever o comportamento assintótico duma função quando o argumento tende a infinito.

- **Notação O**: expressa um limite superior para o comportamento assintótico de uma função, de forma aproximada

$$O(g(n)) = \{\ f(n) \mid \exists c > 0, n_0, \forall n > n_0, f(n) \leq c * g(n)\ \}$$

Informalmente, $f(n) \in O(g(n))$ (se escreve $f(x) = O(g(x))$), significa que $f(n)$ não cresce mais rapidamente que $g(n)$, para valores de n suficientemente grandes (como se $f(x) \leq g(x)$)

Exemplo: $5n = O(n^2)$, $10n^2 + 5n = O(n^2)$, $n^3 \neq O(10n^2)$

Notações assintóticas ou de Bachmann-Landau (Ω)

Permitem descrever o comportamento assintótico duma função quando o argumento tende a infinito.

- **Notação Ω**: expressa um limite inferior para o comportamento assintótico de uma função, de forma aproximada

$$\Omega(g(n)) = \{\ f(n) \mid \exists c > 0, n_0, \forall n > n_0, f(n) \geq c * g(n)\ \}$$

Informalmente, $f(n) \in \Omega(g(n))$ (se escreve $f(x) = \Omega(g(x))$), significa que $f(n)$ cresce mais rapidamente que $g(n)$, para valores de n suficientemente grandes (como se $f(x) \geq g(x)$)

Exemplo: $n^2 = \Omega(300n + 1000)$, $10n^2 + 5n = \Omega(n^2)$

Notações assintóticas ou de Bachmann-Landau (Θ)

Permitem descrever o comportamento assintótico duma função quando o argumento tende a infinito.

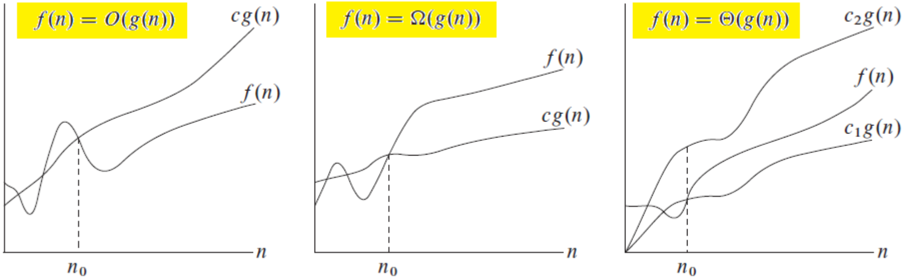
- **Notação Θ**: expressa um limite firme ou restrito para o comportamento assintótico de uma função, de forma aproximada

$$\Theta(g(n)) = \{\ f(n) \mid \exists c_1 > 0, c_2 > 0, n_0, \forall n > n_0, \\ c_1 * g(n) \leq f(n) \leq c_2 * g(n)\ \}$$

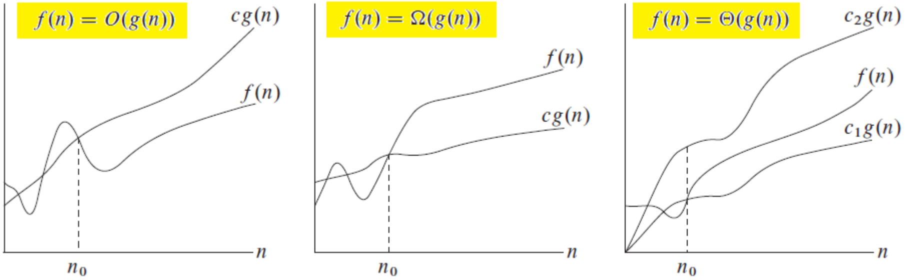
Informalmente, $f(n) \in \Theta(g(n))$ (se escreve $f(x) = \Theta(g(x))$), significa que $f(n)$ cresce tão rapidamente quanto $g(n)$ dentro dum fator, para valores de n suficientemente grandes.

Exemplo: $10n^2 + 5n = \Theta(n^2)$, $n \neq \Theta(n^2)$, $n^3 \neq \Theta(10n^2)$

Algumas Propriedades das Notações O, Ω, Θ



Algumas Propriedades das Notações O, Ω, Θ



1. Todas reflexivas e transitivas; Θ é simétrica;
2. $f(x) = O(g(x))$ sse $g(x) = \Omega(f(x))$
3. $f(x) = \Theta(g(x))$ sse $f(x) = O(g(x))$ e $f(x) = \Omega(g(x))$
4. $O(kf(x)) = O(f(x)), \forall k \neq 0$
5. $O(f(x)) + O(g(x)) = O(max(f(x), g(x)))$

Propriedades das Notações O, Ω, Θ

Permitem desprezar constantes e termos de menor grau. Pex:

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + (c_2 + c_4)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Aqui, pelo caso pior $T(n) = O(n^2)$. Porém, $T(n) \neq \Theta(n^2)$ pois no caso melhor $T(n) \neq \Omega(n^2)$, e sim $T(n) = \Omega(n)$.

Agenda

Pequena guia para a análise de $T(n)$ no caso pior

- Operações predefinidas sobre tipos básicos: $O(1)$
- Sequência de instruções: $max(T(I_1), \ldots, T(I_n))$
- Condicional: $max(T(cond + then), T(cond + else))$
- Laços: $O(\sum_{it=1}^{iterNum} T(cond + body))$

Pequena guia para a análise de $T(n)$ no caso pior

- Operações predefinidas sobre tipos básicos: $O(1)$
- Sequência de instruções: $max(T(I_1), \dots, T(I_n))$
- Condicional: $max(T(cond + then), T(cond + else))$
- Laços: $O(\sum_{it=1}^{iterNum} T(cond + body))$

Exemplo: Qual o custo $T(n)$ do seguinte trecho de programa?

```
if( a+b % c ) {        // test
  i = 0;                 // assigment operator
  while (i<n)            // test, iterations?
    d[i++] = a+b;          // body
} else {
  for(i=0; i<n; i++)       // test, iterations?
    for(j=0; j<n; j++)       // test, iterations?
      f[i][j] = c;             // body
}                        // T(if)?
```

Pequena guia para a análise de $T(n)$ no caso pior

- Operações predefinidas sobre tipos básicos: $O(1)$
- Sequência de instruções: $max(T(I_1), \ldots, T(I_n))$
- Condicional: $max(T(cond + then), T(cond + else))$
- Laços: $O(\sum_{it=1}^{iterNum} T(cond + body))$

Exemplo: Qual o custo $T(n)$ do seguinte trecho de programa?

```
for(i=0; i<n; i++)
 for(j=0; j<n; j++)
   for(k=0; k<=j; k++)
     f[i][j][k] = c;
```

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Pequena guia para a análise de $T(n)$ no caso pior

- Operações predefinidas sobre tipos básicos: $O(1)$
- Sequência de instruções: $max(T(I_1), \ldots, T(I_n))$
- Condicional: $max(T(cond + then), T(cond + else))$
- Laços: $O(\sum_{it=1}^{iterNum} T(cond + body))$

Exemplo: Qual o custo $T(n)$ do seguinte trecho de programa?

```
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    for(k=0; k<=j; k++)
      f[i][j][k] = c;
```

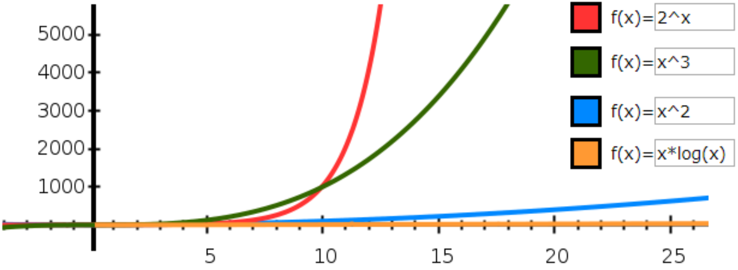
```
for(i=1; i<=n; i*=2)
  for(j=0; j<i; j++)
    count++;
```

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}, \quad c \neq 1$$

Classes de Complexidades

Permite definir uma hierarquia na complexidade dos algoritmos



$$O(1) \subset O(\log n) \subset O(n) \subset O(n^2) \subset O(2^n) \subset O(n!) \subset O(n^n)$$

- $\forall a, b, a > 0, O((\log n)^b) \subset O(n^a)$
- $\forall a, b, a > 1, O(n^b) \subset O(a^n)$

Agenda

Análise de Algoritmos recursivos

Recursividade é um recurso de programação de grande importância

```
double powerR(float base, int n)
{
  if (n == 0)
    return 1;
  else
    return base * powerR(base, n-1);
}
```

```
double powerI(float base, int n)
{
  double pow = 1;
  for(int i=0; i<n; i++)
    pow *= base;
  return pow;
}
```

Vantagens

- Fornece uma forma natural, simples e elegante de resolver um problema
- Facilidade para analisar as propriedades de um algoritmo

Análise de Algoritmos recursivos

Recursividade é um recurso de programação de grande importância

```
double powerR(float base, int n)
{
  if (n == 0)
    return 1;
  else
    return base * powerR(base, n-1);
}
```

```
double powerI(float base, int n)
{
  double pow = 1;
  for(int i=0; i<n; i++)
    pow *= base;
  return pow;
}
```

Vantagens

- Fornece uma forma natural, simples e elegante de resolver um problema
- Facilidade para analisar as propriedades de um algoritmo

Como analizar a complexidade dum algoritmo recursivo?!

Algoritmos recursivos e Recorrências

As propriedades de um programa recursivo podem ser analisadas usando uma função recursiva, chamada **relação de recorrência**

```
double powerR(float base, int n)
{
  if (n == 0)
    return 1;
  else
    return base * powerR(base, n-1);
}
```

Na matemática, uma **relação de recorrência** é definida para um conjunto de valores iniciais e em termos de ela mesma.

$$f(0) = c_0, \quad f(1) = c_1, \quad \ldots, \quad f(k) = c_k$$

$$f(n) = g(f(n-1), f(n-2), \ldots, f(n-t)), \quad n > k$$

Algoritmos recursivos e Recorrências

As propriedades de um programa recursivo podem ser analisadas usando uma função recursiva, chamada **relação de recorrência**

```
double powerR(float base, int n)
{
  if (n == 0)
    return 1;
  else
    return base * powerR(base, n-1);
}
```

- Número de chamadas recursivas:

$$C(0) = 0, \quad C(n) = C(n-1) + 1$$

- Tempo de execução:

$$T(0) = c_1, \quad T(n) = T(n-1) + c_2$$

Relações de recorrências e seus Tipos

$$f(0) = c_0, \quad f(1) = c_1, \quad \ldots, \quad f(k) = c_k$$
$$f(n) = g(f(n-1), f(n-2), \ldots, f(n-t)), \quad n > k$$

- **Linear**: se não há produtos ou potências de termos recursivos, e.g. $C(n) - \sqrt{n}C(n-1) = n$
- **Não Linear**: e.g. $C(n) = C(n-1) * C(n-2) + C(n-3)$
- **Homogênea**: se não há um termo não recursivo
- de **segundo/terceiro/quarto ordem** (em geral de ordem ou grau fixo), dependendo do menor termo recursivo, e.g. $D(n) - 3 * D(n-2) + D(n-4) = 1$ é de ordem 4
- **com coeficientes constantes ou não**
- **Divisão e conquista**: $T(n) = \sum_{i=1}^{k} a_i * T(\frac{n}{b_i}) + f(n)$

Exemplos de Recorrências Simples

- Linear, homogênea, de ordem 1 e coeficiente constante

$$A(0) = 1, \quad A(n) = 2 * A(n-1)$$

- Linear, homogênea, de ordem 1 e coeficiente variável

$$B(0) = 1, \quad B(n) = n * B(n-1)$$

- Linear, não homogênea, de ordem 1

$$C(0) = 0, \quad C(n) = C(n-1) + 1$$

- Linear, homogênea, de ordem 2

$$F(0) = 1, \;\; F(1) = 1, \;\; F(n) = F(n-1) + F(n-2)$$

Ver Apêndice 1

Algoritmos e recorrências de divisão e conquista

```
double powerDC(float base, unsigned int n)
{
  if (n == 0)
    return 1;
  else
    return powerDC(base * base, n/2) * ( n%2 ) ? base : 1;
}
```

As recorrências mais simples de divisão e conquista têm a forma

$$T(n) = aT(n/b) + f(n)$$

- $a > 0$ é o número de sub-problemas de tamanho n/b
- $b > 1$ e $f(n)$ é o custo de dividir e combinar as sub-soluções

Como obter a solução duma recorrência, i.e. uma função não recursiva equivalente?

```
double powerR(float base, int n)
{
  if (n == 0)
    return 1;
  else
    return base * powerR(base, n-1);
}
```

$$T(0) = c_1, \quad T(n) = T(n-1) + c_2, \quad n > 0$$

Como obter a solução duma recorrência, i.e. uma função não recursiva equivalente?

```
double powerR(float base, int n)
{
  if (n == 0)
    return 1;
  else
    return base * powerR(base, n-1);
}
```

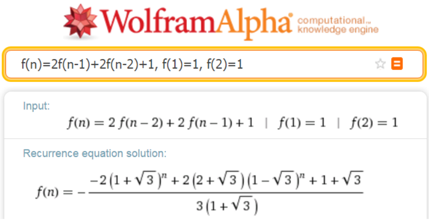
$$T(0) = c_1, \quad T(n) = T(n-1) + c_2, \quad n > 0$$

$$T(n) = T(n-1) + c_2 = T(n-2) + c_2 + c_2 = T(n-3) + 3 * c_2 = \ldots$$

$$= T(n-k) + k * c_2 = \ldots = T(0) + c_2 * n = c_1 + c_2 * n$$

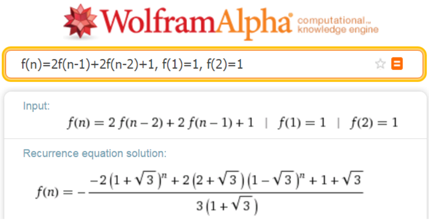
Como obter a solução duma recorrência, i.e. uma função não recursiva equivalente?

- Não existe procedimento geral para resolver recorrências
- Recorrências lineares de ordem fixo com coeficientes constantes sempre podem ser resolvidas



Como obter a solução duma recorrência, i.e. uma função não recursiva equivalente?

- Não existe procedimento geral para resolver recorrências
- Recorrências lineares de ordem fixo com coeficientes constantes sempre podem ser resolvidas



- A **análise de algoritmos** não precisa duma solução exata; **basta uma aproximação assintótica**
- Para algumas recorrências é usado o **Teorema Mestre**

Agenda

Teorema Mestre

Receita para resolver assintóticamente algumas recorrências

Variantes Simples do Teorema Mestre

Receita para resolver assintóticamente algumas recorrências

- $T(n) = aT(n-b) + n^c$, $a > 0, b > 0, c \geq 0$

$$T(n) = \begin{cases} O(n^{c+1}) & \text{if } a = 1 \\ O(n^c a^{n/b}) & \text{if } a > 1 \end{cases}$$

Teorema Mestre

Receita para resolver assintóticamente algumas recorrências

- $T(n) = aT(n-b) + n^c,\ a > 0, b > 0, c \geq 0$

$$T(n) = \begin{cases} O(n^{c+1}) & \text{if } a = 1 \\ O(n^c a^{n/b}) & \text{if } a > 1 \end{cases}$$

Exemplo: Qual o custo $T(n)$ da seguinte função?

```
double powerR(float base, int n)
{
  if (n == 0)
    return 1;
  else
    return base * powerR(base, n-1);
}
```

Teorema Mestre

Receita para resolver assintóticamente algumas recorrências

- $T(n) = aT(n-b) + n^c$, $a > 0, b > 0, c \geq 0$

$$T(n) = \begin{cases} O(n^{c+1}) & \text{if } a = 1 \\ O(n^c a^{n/b}) & \text{if } a > 1 \end{cases}$$

- $T(n) = aT(\frac{n}{b}) + n^c$, $a > 0, b > 0, c \geq 0$

$$T(n) = \begin{cases} \Theta(n^{\log_b^a}) & if \quad \log_b^a > c \quad (Caso\ 1) \\ \Theta(n^c \log_b^n) & if \quad \log_b^a = c \quad (Caso\ 2) \\ \Theta(n^c) & if \quad \log_b^a < c \quad (Caso\ 3) \end{cases}$$

Qual função é mais eficiente?

```
double powerR(float base, int n)
{
  if (n == 0)
    return 1;
  else
    return base * powerR(base, n-1);
}
```

```
double powerDC(float base, unsigned int n)
{
  if (n == 0)
    return 1;
  else
    return powerDC(base * base, n/2) * ( n%2 ) ? base : 1;
}
```

https://repl.it/@mirthalina/powerItVsRec

Qual função é mais eficiente?

```
void imprimir_lista(linked_node *atual) {
    while (atual != NULL) {
        printf("%d ", atual->data);
        atual = atual->next;
    }
    printf("\n");
}

void imprimir_lista_rec(linked_node *atual) {
    if (atual == NULL) {
        printf(" \n");
    } else {
        printf("%d ", atual->data);
        imprimir_lista_rec(atual->next);
    }
}
```

https://repl.it/@mirthalina/ImprimirListaItVsRec

Exemplo do Teorema Mestre - $T(n) = aT(n-b) + n^c$

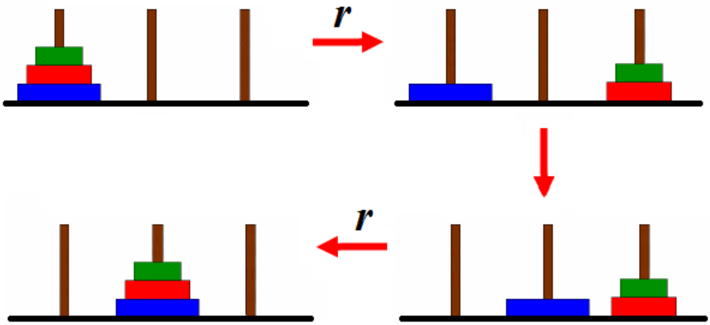
$$T(n) = \begin{cases} O(n^{c+1}) & \text{if } a = 1 \\ O(n^c a^{n/b}) & \text{if } a > 1 \end{cases}$$

Exemplo: Torres de Hanoi, solução recursiva

Exemplo do Teorema Mestre - $T(n) = aT(n-b) + n^c$

$$T(n) = \begin{cases} O(n^{c+1}) & \text{if } a = 1 \\ O(n^c a^{n/b}) & \text{if } a > 1 \end{cases}$$

Exemplo: Torres de Hanoi, solução recursiva



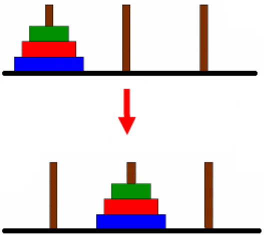
Exemplo do Teorema Mestre - $T(n) = aT(n-b) + n^c$

$$T(n) = \begin{cases} O(n^{c+1}) & \text{if } a = 1 \\ O(n^c a^{n/b}) & \text{if } a > 1 \end{cases}$$

Exemplo: Torres de Hanoi, solução recursiva

```
// dir == -1-> left; dir == 1-> right
// (move in a circular way)

void HanoiTower(int disksNum, int dir){
  if (disksNum == 0)
    return;
  HanoiTower(disksNum-1, -dir);
  moveDisk(disksNum, dir);
  HanoiTower(disksNum-1, -dir);
}
```



$$T(n) = 2T(n-1) + 1$$

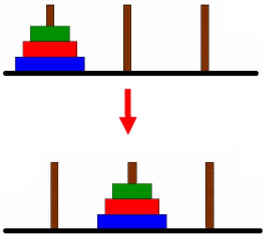
Exemplo do Teorema Mestre - $T(n) = aT(n-b) + n^c$

$$T(n) = \begin{cases} O(n^{c+1}) & \text{if } a = 1 \\ O(n^c a^{n/b}) & \text{if } a > 1 \end{cases}$$

Exemplo: Torres de Hanoi, solução recursiva

```
// dir == -1-> left; dir == 1-> right
// (move in a circular way)

void HanoiTower(int disksNum, int dir){
  if (disksNum == 0)
    return;
  HanoiTower(disksNum-1, -dir);
  moveDisk(disksNum, dir);
  HanoiTower(disksNum-1, -dir);
}
```



$$T(n) = 2T(n-1) + 1 = O(2^n)$$

Exemplos Caso 1 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

1. $\log_b^a > c \Rightarrow T(n) = \Theta(n^{\log_b^a})$

   Exemplo: $T(n) = 4T(n/2) + n$

Exemplos Caso 1 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

1. $\log_b^a > c \Rightarrow T(n) = \Theta(n^{\log_b^a})$

Exemplo: $T(n) = 4T(n/2) + n$

$a = 4, b = 2, \log_b^a = 2 > c = 1 \Rightarrow T(n) = \Theta(n^2)$

Exemplos Caso 1 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

1. $\log_b^a > c \Rightarrow T(n) = \Theta(n^{\log_b^a})$

Exemplo: $T(n) = 4T(n/2) + n$

$a = 4, b = 2, \log_b^a = 2 > c = 1 \Rightarrow T(n) = \Theta(n^2)$

Exemplo: $T(n) = 2T(n/2) + \sqrt{n}$

Exemplos Caso 1 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

1. $\log_b^a > c \Rightarrow T(n) = \Theta(n^{\log_b^a})$

Exemplo: $T(n) = 4T(n/2) + n$

$a = 4, b = 2, \log_b^a = 2 > c = 1 \Rightarrow T(n) = \Theta(n^2)$

Exemplo: $T(n) = 2T(n/2) + \sqrt{n}$

$a = 2, b = 2, \log_b^a = 1 > c = 1/2 \Rightarrow T(n) = \Theta(n)$

Exemplos Caso 2 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

2. $\log_b^a = c \Rightarrow T(n) = \Theta(n^c * \log_b^n)$

   Exemplo: $T(n) = 4T(n/2) + n^2$

Exemplos Caso 2 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

2. $\log_b^a = c \Rightarrow T(n) = \Theta(n^c * \log_b^n)$

Exemplo: $T(n) = 4T(n/2) + n^2$

$a = 4, b = 2, \log_b^a = 2 = c \Rightarrow T(n) = \Theta(n^2 * \log n)$

Exemplos Caso 2 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

2. $\log_b^a = c \Rightarrow T(n) = \Theta(n^c * \log_b^n)$

Exemplo: $T(n) = 4T(n/2) + n^2$

$a = 4, b = 2, \log_b^a = 2 = c \Rightarrow T(n) = \Theta(n^2 * \log n)$

Exemplo: $T(n) = 2T(n/2) + n$

Exemplos Caso 2 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

2. $\log_b^a = c \Rightarrow T(n) = \Theta(n^c * \log_b^n)$

Exemplo: $T(n) = 4T(n/2) + n^2$

$$a = 4, b = 2, \log_b^a = 2 = c \Rightarrow T(n) = \Theta(n^2 * \log n)$$

Exemplo: $T(n) = 2T(n/2) + n$

$$a = 2, b = 2, \log_b^a = 1 = c \Rightarrow T(n) = \Theta(n \log n)$$

Exemplos Caso 3 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

3. $\log_b^a < c \Rightarrow T(n) = \Theta(n^c)$

   Exemplo: $T(n) = 4T(n/2) + n^3$

Exemplos Caso 3 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

3. $\log_b^a < c \Rightarrow T(n) = \Theta(n^c)$

Exemplo: $T(n) = 4T(n/2) + n^3$

$a = 4, b = 2, \log_b^a = 2 < c = 3 \Rightarrow T(n) = \Theta(n^3)$

Exemplos Caso 3 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

3. $\log_b^a < c \Rightarrow T(n) = \Theta(n^c)$

Exemplo: $T(n) = 4T(n/2) + n^3$

$a = 4, b = 2, \log_b^a = 2 < c = 3 \Rightarrow T(n) = \Theta(n^3)$

Exemplo: $T(n) = 3T(n/4) + n\sqrt{n}$

Exemplos Caso 3 do Teorema Mestre - $T(n) = aT(\frac{n}{b}) + n^c$

3. $\log_b^a < c \Rightarrow T(n) = \Theta(n^c)$

Exemplo: $T(n) = 4T(n/2) + n^3$

$a = 4, b = 2, \log_b^a = 2 < c = 3 \Rightarrow T(n) = \Theta(n^3)$

Exemplo: $T(n) = 3T(n/4) + n\sqrt{n}$

$a = 3, b = 4, \log_4^3 = 0.793 < c = 1.5 \Rightarrow T(n) = \Theta(n\sqrt{n})$

Agenda

Considerações sobre o Teorema Mestre

- Muito útil para analisar algoritmos de divisão e conquista
- O comportamento assintótico de $T(n)$ não muda se $T(n/b)$ é substituído por $T(\lfloor n/b \rfloor)$ ou $T(\lceil n/b \rceil)$.
- Tem variantes mais gerais mas nem sempre pode ser aplicado

Considerações sobre o Teorema Mestre

- Muito útil para analisar algoritmos de divisão e conquista
- O comportamento assintótico de $T(n)$ não muda se $T(n/b)$ é substituído por $T(\lfloor n/b \rfloor)$ ou $T(\lceil n/b \rceil)$.
- Tem variantes mais gerais mas nem sempre pode ser aplicado

Complexidade das Estruturas de Dados Lineares

Data Structure	Time								Space
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

http://bigocheatsheet.com/

Classes de Complexidades

Complexidade	Nome	Exemplo
$O(1)$	Constante	Expressões e atribuições interiras e reais
$O(\log \log n)$	Logaritmica	Busca por interpolação
$O(\log n)$	Logaritmica	Busca binária
$O(n)$	Linear	Busca sequencial
$O(n \log n) = O(\log n!)$	Quase Linear	Métodos de ordenação eficientes
$O(n^c)$	Polinonial	Métodos de ordenação simples.
$O(c^n)$ $c>1$	Exponencial	Todas as combinações de elementos
$O(n!)$	Fatorial	Todas as permutações de elementos

Classes de Complexidades

n	log n	n	n log n	n^2	2^n	n!
10	0.003ns	0.01ns	0.033ns	0.1ns	1ns	3.65ms
20	0.004ns	0.02ns	0.086ns	0.4ns	1ms	77years
30	0.005ns	0.03ns	0.147ns	0.9ns	1sec	8.4×10^{15}yrs
40	0.005ns	0.04ns	0.213ns	1.6ns	18.3min	--
50	0.006ns	0.05ns	0.282ns	2.5ns	13days	--
100	0.07	0.1ns	0.644ns	0.10ns	4×10^{13}yrs	--
1,000	0.010ns	1.00ns	9.966ns	1ms	--	--
10,000	0.013ns	10ns	130ns	100ms	--	--
100,000	0.017ns	0.10ms	1.67ms	10sec	--	--
1'000,000	0.020ns	1ms	19.93ms	16.7min	--	--
10'000,000	0.023ns	0.01sec	0.23ms	1.16days	--	--
100'000,000	0.027ns	0.10sec	2.66sec	115.7days	--	--
1,000'000,000	0.030ns	1sec	29.90sec	31.7 years	--	--

http://cooervo.github.io/Algorithms-DataStructures-BigONotation/index.html

Agenda

Referências Bibliográficas

- **Introduction to Algorithms**, 3rd Edition. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, 2009, caps 2,3,4
- An Introduction to the Analysis of Algorithms, 2nd Edition. Robert Sedgewick and Philippe Flajolet, 2013
- Algorithm Design: Foundation, Analysis, and Internet Examples. Michael T. Goodrich and Roberto Tamassia, 2002
- Projeto de Algoritmos, 2da Edição, Nivio Ziviani, 2007
- Wikipedia: Big O notation, Master theorem