Relações de Recorrência

Disciplina: Algoritmos e Estruturas de Dados II

Álvaro Luiz Fazenda

alvaro.fazenda@unifesp.br

Bibliografia básica

- Projeto de algoritmos – com implementações em PASCAL e C
  - Nívio Ziviani – 2ª Edição – Thomson – 2004
    - Cap. 2
- Algoritmos – Teoria e Prática
  - Thomas H. Cormem et al – Elsevier – 2002
    - Cap. 4

Recursividade (Revisão)

- Um procedimento que chama a si mesmo, direta ou indiretamente, é dito ser recursivo
- Recursividade permite descrever algoritmos, geralmente, de forma mais clara e concisa, especialmente problemas recursivos por natureza ou que utiliza estruturas recursivas
- Exemplos: Busca em profundidade em árvore binária, algoritmo fatorial, Ordenação por intercalação (MergeSort), etc

Implementação de Recursividade

- Usa-se uma pilha para armazenar os dados usados em cada chamada de um procedimento recursivo que ainda não terminou
- Todos os dados não globais vão para a pilha, registrando o estado corrente da computação
- Quando uma ativação anterior prossegue, os dados da pilha são recuperados

Exemplo: Percorrimento INORDER

```
struct Data_Tree {
  int Value;
  Data_Tree *Left;
  Data_Tree *Right;
  };
...
void show_inorder(Data_Tree *node){
   if (node) {
         show_inorder(p->Left);
         cout << p->value << endl;
         show_inorder(p->Right);
         }
}
```

Terminação em Procedimentos Recursivos

- É fundamental que a chamada recursiva a um procedimento **P** esteja sujeita a uma condição **B**, a qual se torna não-satisfeita em algum momento da computação
- Esquema para procedimentos recursivos:
  - composição C de comandos Si e P
  - $P \equiv if\ B\ then\ C[Si\ ,\ P]$
  - Para demonstrar que uma repetição termina, define-se uma função $f(x)$, sendo x o conjunto de variáveis do programa, tal que:
    - $f(x) \leq 0$ implica na condição de terminação;
    - $f(x)$ é decrementada a cada iteração.

Terminação em Proced. Recursivos (II)

- Uma forma simples de garantir terminação é associar um parâmetro n para P (no caso por valor) e chamar P recursivamente com $n - 1$
  - $P \equiv if\ n > 0\ then\ P[Si\ ,\ P\ (n - 1)]$
  - É necessário mostrar que o nível mais profundo de recursão é finito, e também possa ser mantido pequeno, pois cada ativação recursiva usa uma parcela de memória para acomodar as variáveis
    - Gera problema de estouro de Pilha (*Stack Overflow*)

Quando Não Usar Recursividade

- Nem todo problema de natureza recursiva deve ser resolvido com um algoritmo recursivo
- Estes podem ser caracterizados pelo esquema:
  - $P \equiv \textit{if } B \textit{ then } (S, P)$
- Tais programas são facilmente transformáveis em uma versão não recursiva
  - $P \equiv (x := x0 \; ; \textit{ while } B \textit{ do } S)$

Quando Não Usar Recursividade (II)

- Cálculo dos números de Fibonacci
  - $f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$ para $n \geq 2$
  - O procedimento recursivo obtido diretamente da equação é o seguinte:

```
integer FibRec(int n) {
      if (n<2) return(n); // Cond. Parada
      else return(FibRec(n-1) + FibRec(n-2));
}
```

  - O programa é extremamente ineficiente porque recalcula o mesmo valor várias vezes
  - Existe solução óbvia não recursiva

Quando Não Usar Recursividade (III)

- **Número de chamadas recursivas = número de Fibonacci !**
- Complexidade de tempo e espaço: $\mathbf{f(n) = O(\Phi^n)}$
- $\mathbf{\Phi} = (1 + \sqrt{5})/2 = 1{,}61803...$
  - *Golden ratio*
  - Exponencial!!!

Quando Não Usar Recursividade (IV)

- Versão não recursiva do Cálculo de Fibonacci

```
int FibIter(int n) {
    int i = 1, k, F = 0;
    for (k = 1; k <= n; k++) {
        F += i;
        i = F - i; }
}
```

  - Complexidade: ***O(n)***
    - **Conclusão: não usar recursividade cegamente!**

Exercícios propostos:

- a) Implementar versões recursivas e não recursivas dos algoritmos de Fibonacci e cálculo de fatorial
  - Medir tempos de processamento e limites (robustez) dos códigos implementados aumentando-se *n*
    - Deve-se plotar em gráfico os tempos de execução, relatar os problemas ocorridos e suas instâncias, especificar a máquina (CPU) utilizada
- b) Implemente uma função recursiva para computar o valor de 2^n
- c) Oque faz a função abaixo?:

```
int f(int a, int b) {
        if (a<b) return(a);
        else return(f(a-b, b));
}
```

Analise da eficiência de algorítmos recursivos

- Encontrar relação de recorrência
  - Analisando o algoritmo
- Resolver relação de recorrência
  - Vários métodos possíveis
- Encontrar complexidade (notação assintótica) com base nos métodos tradicionais

Relação de Recorrência

- Exemplo: Calcular a função fatorial $F(n) = n!$ para um número inteiro não negativo qualquer n
- $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n$ para $n \geq 1$
- $0! = 1$ por definição
- Algoritmo:

```
int fatrec(int n) {
   if (n==0) return(1);
   else return(fatrec(n-1)*n); }
```

Relação de Recorrência
Ex: Fatorial (II)

- O **número de multiplicações** ***M(n)*** necessários para calculá–la deve satisfazer a igualdade:
  - $M(n) = \quad M(n-1) \quad + \quad 1$ para $n > 0$
  - $\quad\quad [calc.\ F(n\text{-}1)] \quad [mult.\ F(n\text{-}1)*n]$
  - Deve-se notar que a equação define $M(n)$ implicitamente como uma função de seu valor em outro ponto, isto é n–1

Relação de Recorrência
Ex: Fatorial (III)

- **Resolvendo a relação de recorrência:**
  - Encontrar uma fórmula explícita para a seqüência *M(n)* somente em termos de *n*
  - Existem muitas sequências que satisfazem esta relação de recorrência
  - Recorrência e a condição inicial para o **número de multiplicações** do algoritmo *M(n)* (para *n=0*, não há multiplicação):
    - $\boldsymbol{M(n) = M(n-1) + 1}$ **para** $\boldsymbol{n > 0}$
    - $\boldsymbol{M(0) = 0}$

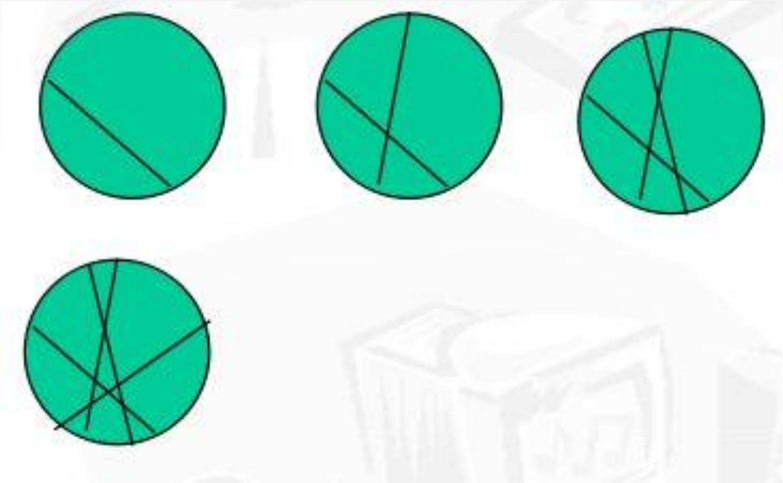
Encontre as relações de recorrência (I):

- Faça um algoritmo recursivo para encontrar o maior valor de uma sequência de números. Para isso, encontre primeiro a equação de recorrência
- Encontre a relação de recorrência $T(n)$ para um valor genérico n considerando os seguintes valores da tabela abaixo:

n	1	2	3	4	5	6
$T(n)$	1	9	20	34	51	71

Encontre as relações de recorrência (II):

- Considere os seguintes cortes de uma determinada pizza:
  - 1 corte, 2 fatias;
  - 2 cortes, 4 fatias;
  - 3 cortes, 7 fatias;
  - 4 cortes, 11 fatias.



  - Quantos pedaços obteremos com n cortes na pizza?
    - Encontre uma relação de recorrência que resolva o problema acima

Resolvendo a relação de recorrência do Fatorial

- Método das **substituições retrógradas (ou expansão telescópica)** para resolver relações de recorrência:
  - $M(n)=M(n-1)+1$ subst. $M(n-1)=M(n-2)+1$
  - $=[M(n-2)+1]+1=M(n-2)+2$
  - $=[M(n-3)+1]+2=M(n-3)+3$
  - Vê-se uma fórmula geral emergindo para o padrão: $\boldsymbol{M(n) = M(n-i)+i}$
  - Substituindo $i = n$ na fórmula padrão:
  - $\boldsymbol{M(n) = M(n-1)+1= \ldots = M(n-i)+i = \ldots = M(n)=M(n-n)+n = n}$
  - $\boldsymbol{M(n) = O(n)}$

Resolva as relações de recorrências

- $T(n) = 1, \quad \text{se } n = 0$
- $T(n) = 2\,T(n-1) + 1, \quad \text{se } n > 0$

- $T(n) = 1 \quad \text{se } n = 1$
- $T(n) = T(n/2) + 1 \quad \text{se } n > 1$

- $T(n) = 1 \quad \text{se } n = 1$
- $T(n) = 2T(n/2) + n \quad \text{se } n > 1$

Exemplo de solução de recorrências

$$\begin{cases} T(n) = 1 & \text{se } n = 1 \\ T(n) = T(n/2) + 1 & \text{se } n > 1 \end{cases}$$

$T(N/2) + 1$

$\boldsymbol{T(N/4) + 1} + 1$

$\boldsymbol{T(N/8) + 1} + 1 + 1$

$T(N/2^3) + 3 \Rightarrow \boldsymbol{T(N/2^i) + i}$

$\boldsymbol{N=2^i\ ,\ i=\log_2 N} \Rightarrow T(2^i/2^i) + \log_2 N \Rightarrow \boldsymbol{1 + \log_2 N}$

$\boldsymbol{O(\log_2 N)}$

Outros métodos para resolver recorrências

- Método da Substituição
  - 1) Pressupor a forma de solução
  - 2) Usar a indução matemática para encontrar as constantes e mostrar que a solução funciona
  - É um método eficiente, porém, só pode ser usando quando é fácil pressupor a forma da resposta
    - As vezes é difícil apresentar uma boa suposição

Método da Substituição
Exemplo

- $T(n) = 2T(n/2) + n$
  - Supor: $T(1) = O(1)$
  - "Chutar": $O(n^3)$ (deve-se provar Θ e Ω separadamente)
    - Implica supor: $T(k) \leq ck^3$, para $k < n$
  - Provar que: $T(n) \leq cn^3$
    - $T(n) = 2T(n/2) + n$
    - $\quad = 2c(n/2)^3 + n$
    - $\quad = c/4(n^3) + n \quad \leq cn^3$
    - Desde que: $2c(n/2)^3 + n >= 0$ e $n = 1$ e $c = 2$
  - **Este limitante muito é muito alto??**

Método da Substituição
Exemplo (II)

- Trocando o Limitante por $O(n^2)$
  - $T(n) = O(n^2)$
  - Provar que: $T(n) \leq cn^2$
    - $T(n) = 2T(n/2) + n$
    - $= 2c(n/2)^2 + n$
    - $= c/2(n^2) + n \leq cn^2$
    - Desde que: $2c(n/2)^2 + n >= 0$ e $n = 1$ e $c >= 4$
  - **Válido, porem pode ser ainda alto!!**

Método da Substituição
Exemplo (III)

- Trocando o Limitante por $O(n)$
  - $T(n) = O(n)$
  - Provar que: $T(n) \le cn$
    - $T(n) = 2T(n/2) + n$
    - $= 2c(n/2) + n$
    - $= cn + n \le cn$ ***** ERRADO!!! *****
    - Não existe valor de $c>1$ que torne a expressão verdadeira

Método da Substituição
Exemplo (IV)

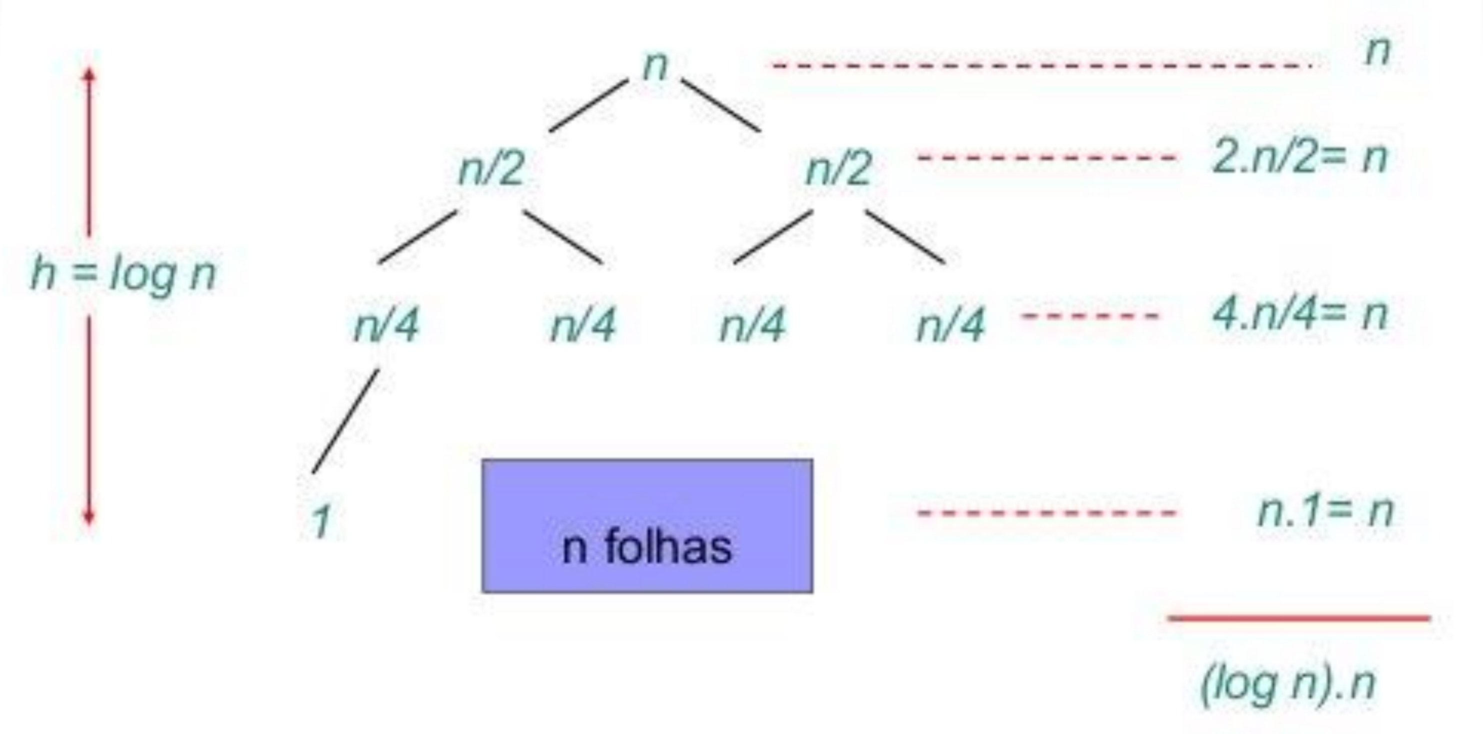
- Trocando o Limitante por $O(n\log_2 n)$ [entre n^2 e n]
  - $T(n) = O(n\log_2 n)$
  - Provar que: $T(n) \leq cn\log_2 n$
    - $T(n) = 2T(n/2) + n$
    - $= 2c(n/2 \log_2(n/2)) + n$
    - $= cn \log_2(n/2)) + n = cn \log_2 n - cn \ \log_2 2 + n$
    - $= cn \log_2 n - cn + n \ \leq \ cn \log_2 n$
    - Para $c>1$

Método de Árvore de Recursão

- Talvez o método mais intuitivo
- Consiste em desenhar uma árvore cujos nós representam os tamanhos dos correspondentes problemas
  - Cada nível *i* contém todos os subproblemas de profundidade *i*
  - Dois aspectos importantes:
    - A altura da árvore.
    - O número de passos executados de cada nível.
  - A solução da recorrência é a soma de todos os passos de todos os níveis

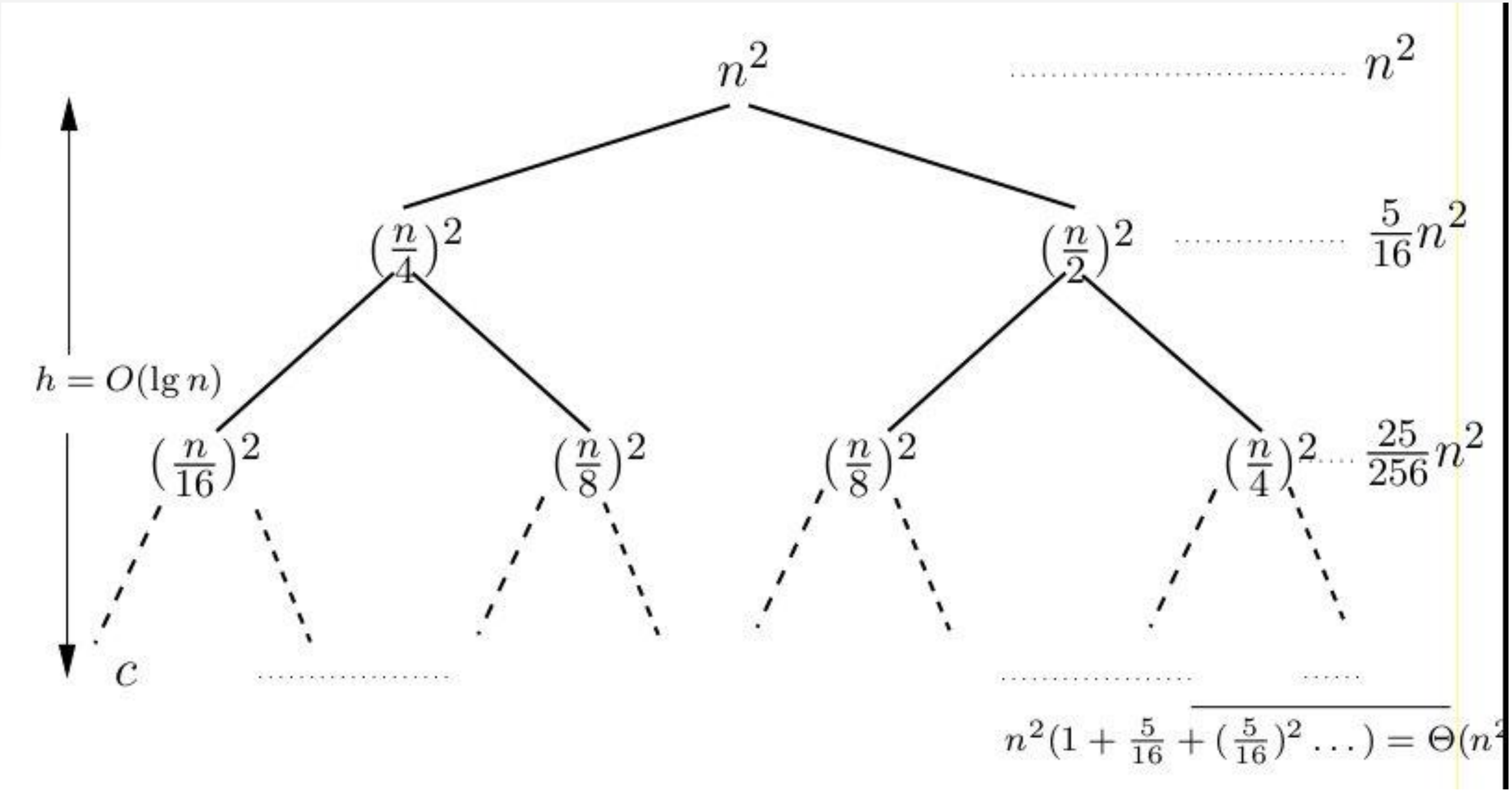
Árvore de Recursão Exemplo (I)

- Resolver:
  - $T(1) = 1$
  - $T(n) = 2T(n/2) + n$



Árvore de Recursão
Exemplo (II)

- $T(1) = 1$
- $T(n) = T(n/4) + T(n/2) + n^2$



Método Mestre

- Fornece um processo de “livro de receitas” para resolver recorrências da forma:
  - $T(n) = aT(n/b) + f(n)$
  - Onde $a \geq 1$, $b > 1$ são constantes e $f(n)$ é uma função assintoticamente positiva
  - A recorrência acima, descreve o tempo de execução de um algoritmo que divide um problema de tamanho n em a subproblemas de tamanho n/b
      - Interpreta-se n/b com o significado de $\lceil n/b \rceil$ ou $\lfloor n/b \rfloor$
    - O custo de dividir o problema e combinar os resultados é descrito por $f(n)$

Método Mestre (II)

- $T(n)$ da forma: $aT(n/b) + f(n)$, $a,b > 1$
  - Casos:
  - 1. Se $f(n) = O(n^{\log_b a - \varepsilon})$, para algum $\varepsilon > 0$, tem-se que: $T(n) = \Theta(n^{\log_b a})$
  - 2. Se $f(n) = O(n^{\log_b a})$, tem-se que: $T(n) = \Theta(n^{\log_b a} \log_2 n)$
  - 3. Se $f(n) = O(n^{\log_b a + \varepsilon})$, para algum $\varepsilon > 0$ e se $af(n/b) \leq cf(n)$ para algum $c > 0$ e n suficientemente grande, tem-se que: $T(n) = \Theta(f(n))$

Método Mestre (III)

- Nos três casos estamos comparando a função *f(n)* com a expressão: $n^{\log_b a}$
  - A solução para a recorrência é dada pela maior das duas funções:
    - No caso 1 a expresão $n^{\log_b a}$ é maior, então a solução: $T(n) = \Theta(n^{\log_b a})$
    - No caso 3 a função *f(n)* é maior, então a solução: $T(n) = \Theta(f(n))$
    - No caso 2 a função *f(n)* e a expressão são de mesma dimensão sendo introduzido um fator $log_2 n$, sendo a solução: $T(n) = \Theta(n^{\log_b a} \log_2 n) = \Theta(f(n) log_2 n)$

Método Mestre (IV)

- **Atenção! O teorema não cobre todos os casos possíveis**
- Apenas aqueles em que *f(n)* é menor do que $n^{\log_b a}$ por um fator polinomial e aqueles em que *f(n)* é maior do que $n^{\log_b a}$ por um fator polinomial
- Lacunas entre os casos 1 e 2 e entre os casos 2 e 3 o método não poderá ser usado

Método Mestre (V)
Exemplos

- MergeSort:
  - $T(n) = 2T(n/2) + n$
  - $a = b = 2$
  - $f(n) = n$
  - $log_b a = 1$ => Cai no caso 2
  - Logo, $T(n) = \Theta(nlog_2 n)$

Método Mestre (VI)
Exemplos

- $T(n) = 9T(n/3) + n$
  - $a = 9, b = 3$
  - $f(n) = n$
  - $log_b a = 2$ => Se $\varepsilon = 1$, Cai no caso 1
  - Logo, $T(n) = \Theta(n^2)$

Método Mestre (VII)
Exemplos

- $T(n) = 4T(n/2) + n$
  - $a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2$, $f(n) = n$
  - **Caso 1:** $f(n) = O(n^{2-\varepsilon})$ para $\varepsilon = 1$
  - $T(n) = \Theta(n^2)$
- $T(n) = 4T(n/2) + n^2$
  - $a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2$, $f(n) = n^2$
  - **Caso 2:** $f(n) = \Theta\ (f(n)\log_2 n)$
  - $T(n) = \Theta(n^2 \log_2 n)$

Método Mestre (VIII)
Exemplos

- Exemplos onde o Teorema Master não se aplica:
  - $T(n) = T(n - 1) + n$
  - $T(n) = T(n - a) + T(a) + n, \ (a \geq 1)$
  - $T(n) = T(\alpha n) + T((1 - \alpha)n) + n, \ (0 < \alpha < 1)$
  - $T(n) = T(n - 1) + log_2 n$