# Análise de Algoritmos Recursivos
# Relações de Recorrência, Árvores e o Teorema Mestre

Prof. Pedro Ximenes

Universidade Católica de Pernambuco (UNICAP)

Análise de Algoritmos – 2026.2

# Objetivos de Aprendizagem

Nesta aula, avançamos da análise iterativa para os **algoritmos recursivos**. Ao final desta aula, você será capaz de:

- **Entender** por que funções auto-referenciadas exigem relações de recorrência para medir seu tempo de execução.
- **Modelar** a recorrência $T(n) = aT(n/b) + f(n)$ a partir do pseudocódigo.
- **Construir e interpretar Árvores de Recursão** passo a passo (Fatorial, Merge Sort e Busca Binária).
- **Dominar o Teorema Mestre**: compreender a intuição dos 3 casos e identificar classes assintóticas com rapidez.
- **Reconhecer** quando o Teorema Mestre não pode ser aplicado.

# Roteiro da Aula

## 1. Do Iterativo ao Recursivo

Limitações da contagem de laços e o dilema da auto-referência.

## 2. Relações de Recorrência

Conceito, caso base e modelagem formal de $T(n)$.

## 3. Árvore de Recursão

O método em 5 passos passo a passo:
• Fatorial • Merge Sort • Busca Binária

## 4. O Teorema Mestre

Batalha Raiz vs Folhas ($n^{\log_b a}$), os 3 casos e quando não aplicar.

## 5. Quizzes em Sala

5 questões objetivas de fixação com gabarito comentado.

## 6. Síntese e Conclusão

Quadro comparativo dos métodos e regra de ouro.

# Recapitulando: Como Analisávamos Algoritmos Iterativos?

Nas aulas anteriores, analisamos algoritmos iterativos em **3 passos**:

1. **Identificar operações primitivas**:
   - Avaliação de expressões booleanas;
   - Operações matemáticas (+, -, *, /);
   - Retorno de métodos;
   - Atribuições e acessos a posições de vetores.
2. **Identificar a quantidade de vezes** que cada primitiva executa.
3. **Somar** essas execuções para obter uma função $f(n)$.

## Ordem de Crescimento Assintótico

Eliminamos constantes e termos de menor magnitude:

$$f(n) = 70n + 32n + 231 \implies f(n) \in \Theta(n)$$

# O Que Acontece com Algoritmos Recursivos?

Para algoritmos iterativos, sabemos quantas vezes o laço executa olhando diretamente para o laço.
Mas vejamos uma função clássica recursiva: o cálculo do **Fatorial**:

```
1 funcao fatorial(n)
2     se n == 0 ou n == 1 entao
3         retorne 1
4     senao
5         retorne n * fatorial(n - 1)
6     fim_se
7 fim_funcao
```

Identificando as primitivas:

- `se n == 0 ou n == 1`: expressão booleana $\rightarrow$ custo $\Theta(1)$.
- `retorne 1`: retorno $\rightarrow$ custo $\Theta(1)$.
- `n * ...`: produto aritmético $\rightarrow$ custo $\Theta(1)$.
- `fatorial(n - 1)`: **????**

# O Impasse da Auto-Referência

## O Problema Central

Qual o custo de executar `fatorial(n - 1)` e quantas vezes essa chamada será feita?

- Não conseguimos responder de forma direta porque a função está definida **em termos dela mesma**!
- O custo de resolver o problema para tamanho $n$ depende do custo para resolver o problema de tamanho $n - 1$.
- Funções dessa natureza são chamadas de **Relações de Recorrência**.

## O Nosso Desafio

Precisamos de métodos adequados para **modelar** e **resolver** essas relações de recorrência.

# O que é uma Relação de Recorrência?

## Definição

Uma **relação de recorrência** é uma equação ou inequação que descreve uma função $T(n)$ em termos dela mesma, considerando entradas estritamente menores.

Toda recorrência em análise de algoritmos possui duas partes:

1. **Caso Base**: ponto de parada da recursão (resposta direta sem novas chamadas recursivas). Custo: $\Theta(1)$.
2. **Passo Recursivo**: trabalho de dividir o problema + custo das chamadas recursivas + trabalho de combinar a solução.

# Extraindo a Recorrência: Exemplo do Fatorial

Olhando para o algoritmo do Fatorial:

- Se $n \leq 1$: apenas um teste e um retorno $\implies$ $T(1) = 1$.
- Se $n > 1$: faz **uma** chamada recursiva com entrada $(n-1)$, somada a um número fixo de operações primitivas locais.

## Equação de Recorrência do Fatorial

$$T(n) = T(n-1) + \Theta(1)$$

Simplificando as constantes para a análise assintótica:

$$T(n) = T(n-1) + 1, \quad \text{com } T(1) = 1$$

**Em português:** *"O tempo para calcular o fatorial de n é igual ao tempo de calcular o fatorial de $n-1$, mais 1 unidade de trabalho local."*

# O Método da Árvore de Recursão

A ideia da **Árvore de Recursão** é simular a execução do algoritmo através de uma estrutura em árvore:

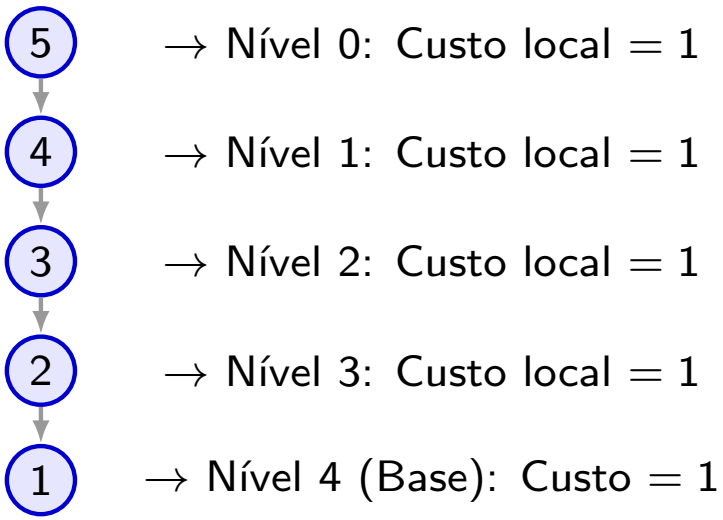
- Cada **nó** representa o custo local de processamento daquele subproblema;
- Cada **aresta** representa uma chamada recursiva filha.

## O Protocolo em 5 Passos

1. **Estabelecer** a relação de recorrência.
2. **Expandir** a árvore de execução a partir da raiz.
3. **Determinar a altura máxima** $h$ da árvore (atingir o caso base).
4. **Somar o custo de cada nível** (soma horizontal).
5. **Somar o custo total** de todos os níveis (soma vertical).

# Exemplo 1: Fatorial Concreto ($n = 5$)

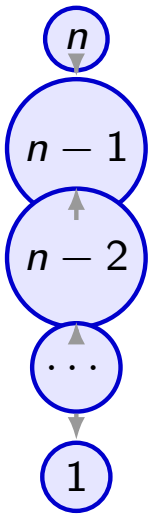
Visualizando a árvore para a entrada $n = 5$:




Custo total $= 1 + 1 + 1 + 1 + 1 = 5$ operações primitivas.

# Exemplo 1: Fatorial com Entrada Arbitrária $n$

Expandindo para uma entrada genérica de tamanho $n$:




$\rightarrow$ Nível 0: Custo $= 1$

$\rightarrow$ Nível 1: Custo $= 1$

$\rightarrow$ Nível 2: Custo $= 1$

$\rightarrow$ Nível $i$: Custo $= 1$

$\rightarrow$ Nível $h$: Custo $= 1$

Como cada chamada gera apenas 1 subproblema, a árvore é uma cadeia linear!

# Exemplo 1: Fatorial – Altura da Árvore

## Passo 3: Determinar a Altura Máxima $h$

A cada nível descido, a entrada reduz em 1 unidade:

- Nível 0: tamanho $n$
- Nível 1: tamanho $n - 1$
- Nível 2: tamanho $n - 2$
- Nível $h$: tamanho $n - h$

O caso base ocorre quando a entrada chega a 1:

$$n - h = 1 \implies \mathbf{h = n - 1}$$

# Exemplo 1: Fatorial – Custo Total e Complexidade

## Passos 4 e 5: Custo por Nível e Custo Total

- O número total de níveis é $h + 1 = (n - 1) + 1 = n$ níveis.
- Cada nível gasta exatamente 1 operação local.

Somando o trabalho de todos os níveis:

$$T(n) = \sum_{i=0}^{n-1} 1 = n \cdot 1 = n$$

## Conclusão Assintótica

O algoritmo do Fatorial possui complexidade linear:

$$\mathbf{T(n) \in \Theta(n)}$$

# Exemplo 2: Merge Sort (Divisão e Conquista)

Vejamos agora um algoritmo onde o problema é **dividido pela metade**:

```
1  procedimento mergeSort(A, ini, fim)
2      se ini < fim entao
3          meio ← (ini + fim) / 2
4          mergeSort(A, ini, meio)              // metade esquerda
5          mergeSort(A, meio + 1, fim)          // metade direita
6          intercala(A, ini, meio, fim)         // combinacao linear
               : Θ(n)
7      fim_se
8  fim_procedimento
```

- Faz **duas** chamadas recursivas, cada uma com metade do vetor $(n/2)$.
- Executa a rotina `intercala`, que junta as duas metades ordenadas em tempo linear: custo $\Theta(n)$.

# A Recorrência do Merge Sort

Somando as partes que compõem o algoritmo:

$$T(n) = \underbrace{T(n/2)}_{\text{metade esquerda}} + \underbrace{T(n/2)}_{\text{metade direita}} + \underbrace{n}_{\text{rotina intercala}}$$

## Forma Simplificada da Recorrência

$$T(n) = 2 \cdot T(n/2) + n, \quad \text{com } T(1) = 1$$

**Significado:** *"Para ordenar n elementos, gastamos o dobro do tempo de ordenar n/2 elementos, mais n passos para intercalar as respostas."*

# Anatomia Geral de Divisão e Conquista

Muitas relações de recorrência seguem o formato padrão:

**Equação Geral**

$$T(n) = a \cdot T(n/b) + f(n)$$

**O que significa cada termo em português?**

- $\mathbf{a \geq 1}$: quantidade de subproblemas (chamadas recursivas).
- $\mathbf{b > 1}$: fator pelo qual a entrada é dividida em cada chamada.
- $\mathbf{f(n)}$: custo de divisão e combinação fora da recursão.

**No Merge Sort:**

$a = 2$ (duas chamadas), $b = 2$ (metade da entrada), $f(n) = n$ (intercala).

# Quiz Rápido: Extraindo Recorrências de Código

**Qual é a relação de recorrência do algoritmo abaixo?**

```
1  funcao processa(A, n)
2      se n ≤ 1 entao retorne A[1] fim_se
3
4      processa(A[1 ... n/2], n/2)
5      processa(A[n/2 + 1 ... n], n/2)
6
7      para i de 1 ate n faca
8          para j de i + 1 ate n faca
9              // operacao elementar O(1)
10         fim_para
11     fim_para
12 fim_funcao
```

# Quiz Rápido: Resolução Comentada

## Análise Passo a Passo

1. **Chamadas recursivas:** O algoritmo chama `processa` 2 vezes, cada uma com $n/2$ elementos $\implies 2 \cdot T(n/2)$.
2. **Trabalho fora da recursão:** Os laços aninhados executam:

$$\sum_{i=1}^{n}(n-i) \approx \frac{n^2}{2} \implies \Theta(n^2)$$

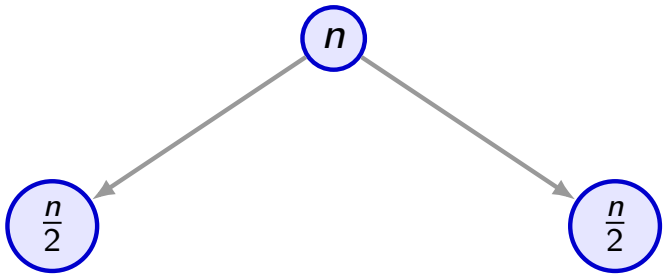
## Relação de Recorrência Final

$$\mathbf{T(n) = 2 \cdot T(n/2) + n^2}$$

com $a = 2$, $b = 2$ e $f(n) = n^2$.

# Merge Sort: Árvore Passo 1 (Níveis 0 e 1)

Vamos expandir a árvore de $T(n) = 2T(n/2) + n$:

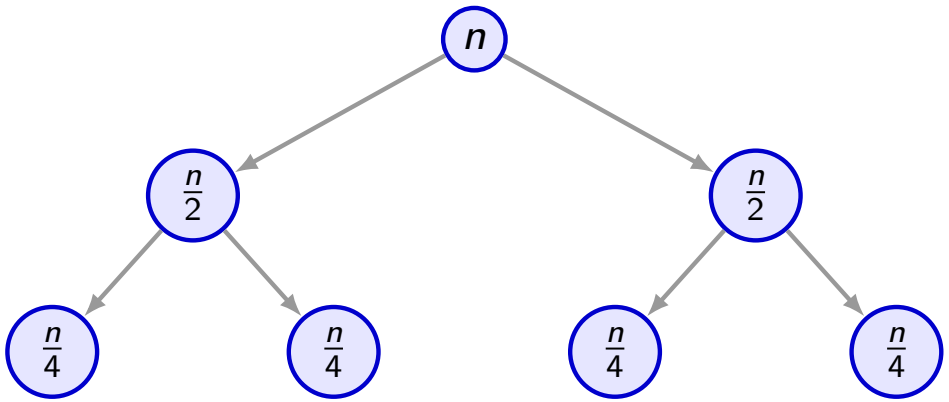



$\rightarrow$ Nível 0: 1 nó de custo $n \implies \mathbf{n}$

$\rightarrow$ Nível 1: $2 \times \frac{n}{2} \implies \mathbf{n}$

- Nível 0 (raiz): faz trabalho local $n$.
- Nível 1: 2 nós filhos, cada um fazendo trabalho local $n/2$. Total do nível: $2 \cdot (n/2) = n$.

# Merge Sort: Árvore Passo 2 (Nível 2 e Nível $i$)




$\rightarrow$ Nível 0: $1 \cdot n = \mathbf{n}$

$\rightarrow$ Nível 1: $2 \cdot \frac{n}{2} = \mathbf{n}$

$\rightarrow$ Nível 2: $4 \cdot \frac{n}{4} = \mathbf{n}$

## Padrão Geral em um Nível $i$

Existem $2^i$ nós no nível $i$, cada um com entrada de tamanho $\frac{n}{2^i}$.
**Custo do nível $i$:** $2^i \cdot \left(\frac{n}{2^i}\right) = \mathbf{n}$.
*Todos os níveis da árvore gastam exatamente a mesma quantidade de trabalho: $n$!*

# Merge Sort: Árvore Passo 3 (Cálculo da Altura)

## Passo 3: Determinar a Altura $h$

A cada nível descido, a entrada é dividida por 2:

- Nível 0: tamanho $\frac{n}{2^0} = n$
- Nível 1: tamanho $\frac{n}{2^1} = \frac{n}{2}$
- Nível $i$: tamanho $\frac{n}{2^i}$
- Nível $h$ (folhas): atinge o caso base (tamanho 1).

Igualando o tamanho do nó folha a 1:

$$\frac{n}{2^h} = 1 \iff 2^h = n \implies h \cdot \log_2 2 = \log_2 n \implies \mathbf{h = \log_2 n}$$

# Merge Sort: Árvore Passo 4 (Soma Total e Conclusão)

Agora calculamos a soma de todos os níveis da árvore:

- Cada nível tem custo total: $n$.
- A altura da árvore é: $h = \log_2 n$.
- Quantidade de níveis: $h + 1 \approx \log_2 n$ níveis.

## Custo Total $T(n)$

$$T(n) = \sum_{i=0}^{\log_2 n} n = \underbrace{n + n + n + \cdots + n}_{\log_2 n \text{ vezes}} = n \cdot \log_2 n$$

## Conclusão Assintótica

$$\mathbf{T(n) \in \Theta(n \log n)}$$

# Exemplo 3: Busca Binária

O algoritmo de Busca Binária localiza um elemento em um vetor ordenado descartando metade do espaço de busca a cada iteração:

```
1  funcao buscaBinaria(A, x, ini, fim)
2      se ini ≤ fim entao
3          meio ← (ini + fim) / 2
4          se A[meio] == x entao
5              retorne meio
6          senao se x < A[meio] entao
7              retorne buscaBinaria(A, x, ini, meio - 1)  //
                   busca esq.
8          senao
9              retorne buscaBinaria(A, x, meio + 1, fim)  //
                   busca dir.
10         fim_se
11     senao
12         retorne -1 // nao encontrado
13     fim_se
14 fim_funcao
```

# A Recorrência da Busca Binária

## Pegadinha Muito Comum!

Embora haja duas chamadas recursivas no código (`se ... senao`), elas são mutualmente exclusivas!
**Apenas UMA chamada recursiva é de fato executada por passo!**

Portanto, para o pior caso:

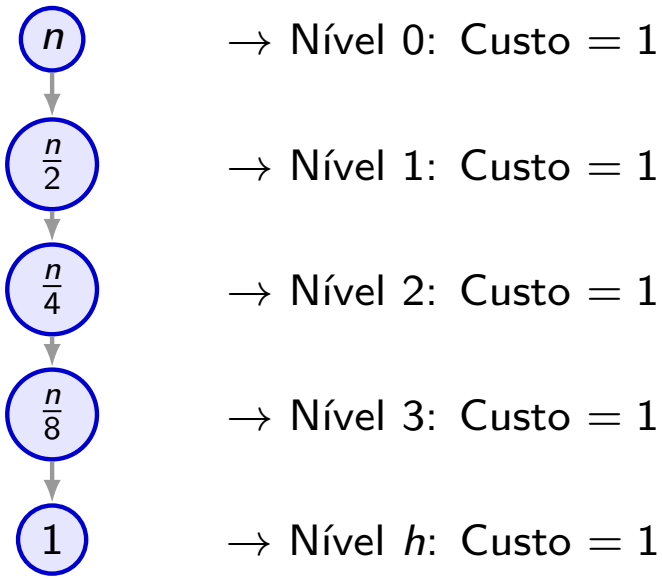
- Apenas 1 subproblema com metade do tamanho: $T(n/2)$.
- Custo local (cálculo de `meio` e comparações): $\Theta(1)$.

## Equação de Recorrência

$$T(n) = T(n/2) + 1, \quad \text{com } T(1) = 1$$

# Busca Binária: Árvore de Recursão

Como há apenas 1 chamada recursiva por nível, a árvore **não ramifica**:



# Busca Binária: Altura da Árvore

## Passo 3: Determinar a Altura Máxima $h$

A cada chamada recursiva, o tamanho da entrada é dividido por 2:

- Nível 0: tamanho $n = \frac{n}{2^0}$
- Nível 1: tamanho $\frac{n}{2} = \frac{n}{2^1}$
- Nível 2: tamanho $\frac{n}{4} = \frac{n}{2^2}$
- Nível $h$: tamanho $\frac{n}{2^h}$

A recursão encerra quando a entrada atinge o tamanho 1:

$$\frac{n}{2^h} = 1 \implies 2^h = n \implies \mathbf{h = \log_2 n}$$

# Busca Binária: Custo Total e Complexidade

## Passos 4 e 5: Soma Total de Todos os Níveis

- A árvore possui $h \approx \log_2 n$ níveis.
- Cada nível gasta custo constante 1 (cálculo de meio e comparações).

Somando os custos:

$$T(n) = \sum_{i=0}^{\log_2 n} 1 = 1 \cdot \log_2 n = \log_2 n$$

## Conclusão Assintótica

$$\mathbf{T(n) \in \Theta(\log n)}$$

*Muito mais rápida que a Busca Linear recursiva, que custa $\Theta(n)$!*

# Por que Precisamos do Teorema Mestre?

A árvore de recursão é uma excelente ferramenta visual, mas desenhar árvores para toda e qualquer função é **trabalhoso e demorado**.

## A Ideia do Teorema Mestre

O **Teorema Mestre** (Master Theorem) é um atalho analítico para resolver diretamente relações de recorrência de divisão e conquista:

$$T(n) = a \cdot T(n/b) + f(n)$$

com $a \geq 1$, $b > 1$ e $f(n)$ não-negativa.

Em vez de desenhar a árvore inteira, basta **comparar duas funções**:

1. $f(n)$: o trabalho realizado na raiz (divisão + combinação).
2. $n^{\log_b a}$: o trabalho acumulado nas folhas da árvore de recursão.

# A Batalha Fundamental: Raiz vs Folhas

## O Expoente Crítico: Quantidade de Folhas da Árvore

Se a árvore ramifica em $a$ filhos a cada passo e atinge altura $\log_b n$:

$$\text{Número de Folhas} = a^{\log_b n} = \mathbf{n^{\log_b a}}$$

O Teorema Mestre é simplesmente uma **batalha assintótica** entre:

- $\mathbf{n^{\log_b a}}$: trabalho na base da recursão (as folhas).
- $\mathbf{f(n)}$: trabalho na raiz (divisão e combinação).

Quem crescer mais rápido dita a complexidade de todo o algoritmo!

# Os 3 Casos do Teorema Mestre (Visão Intuitiva)

Basta comparar a ordem de crescimento de $f(n)$ com $n^{\log_b a}$:

| **Caso** | **Comparação** | **Solução** $T(n)$ |
|---|---|---|
| **Caso 1** | $f(n) < n^{\log_b a}$<br>*(As folhas dominam!)* | $\mathbf{\Theta(n^{\log_b a})}$ |
| **Caso 2** | $f(n) = \Theta(n^{\log_b a})$<br>*(Empate! Custo uniforme)* | $\mathbf{\Theta(n^{\log_b a} \log n)}$<br>ou $\mathbf{\Theta(f(n) \log n)}$ |
| **Caso 3** | $f(n) > n^{\log_b a}$<br>*(A raiz domina!)* | $\mathbf{\Theta(f(n))}$ |

*No Caso 3, para funções polinomiais usuais, exige-se a condição de regularidade* $a \cdot f(n/b) \leq c \cdot f(n)$ *com* $c < 1$.

# Exemplo TM 1: $T(n) = 8T(n/2) + 1000n^2$ – Passo 1

Vamos resolver passo a passo:

## Passo 1: Identificar os Parâmetros

Comparando com $T(n) = aT(n/b) + f(n)$:

- $a = 8$ (número de chamadas recursivas)
- $b = 2$ (fator de redução da entrada)
- $f(n) = 1000n^2$ (trabalho local de divisão/combinação)

## Passo 2: Calcular o Expoente Crítico das Folhas

$$\log_b a = \log_2 8 = 3 \implies n^{\log_b a} = \mathbf{n^3}$$

# Exemplo TM 1: $T(n) = 8T(n/2) + 1000n^2$ – Passo 2

## Passo 3: Comparar $f(n)$ com $n^{\log_b a}$

- $f(n) = 1000n^2 \in \Theta(n^2)$
- $n^{\log_b a} = n^3$
- As constantes são ignoradas na comparação: comparamos $n^2$ com $n^3$.
- Como $n^2$ cresce estritamente mais devagar que $n^3$:

$$\mathbf{f(n) < n^{\log_b a}}$$

## Passo 4: Conclusão (Caso 1)

As folhas dominam o custo total da computação:

$$\mathbf{T(n) \in \Theta(n^3)}$$

# Exemplo TM 2: $T(n) = 2T(n/2) + 10n$ – Passo 1

Vamos resolver passo a passo:

## Passo 1: Identificar os Parâmetros

Comparando com $T(n) = aT(n/b) + f(n)$:

- $a = 2$
- $b = 2$
- $f(n) = 10n$

## Passo 2: Calcular o Expoente Crítico das Folhas

$$\log_b a = \log_2 2 = 1 \implies n^{\log_b a} = n^1 = \mathbf{n}$$

# Exemplo TM 2: $T(n) = 2T(n/2) + 10n$ – Passo 2

## Passo 3: Comparar $f(n)$ com $n^{\log_b a}$

- $f(n) = 10n \in \Theta(n)$
- $n^{\log_b a} = n \in \Theta(n)$
- Comparando as ordens de grandeza: $n$ e $n$ têm **a mesma taxa de crescimento**!

$$\mathbf{f(n) = \Theta(n^{\log_b a})}$$

## Passo 4: Conclusão (Caso 2 – Empate)

Multiplica-se pelo fator logarítmico referente à altura da árvore:

$$\mathbf{T(n) \in \Theta(n \log n)}$$

*(O mesmo resultado obtido no Merge Sort!)*

# Exemplo TM 3: $T(n) = 2T(n/2) + n^2$ – Passo 1

Vamos resolver passo a passo:

## Passo 1: Identificar os Parâmetros

Comparando com $T(n) = aT(n/b) + f(n)$:

- $a = 2$
- $b = 2$
- $f(n) = n^2$

## Passo 2: Calcular o Expoente Crítico das Folhas

$$\log_b a = \log_2 2 = 1 \implies n^{\log_b a} = n^1 = \mathbf{n}$$

# Exemplo TM 3: $T(n) = 2T(n/2) + n^2$ – Passo 2

## Passo 3: Comparar $f(n)$ com $n^{\log_b a}$

- $f(n) = n^2$
- $n^{\log_b a} = n$
- Claramente, $n^2$ cresce estritamente mais rápido do que $n$:

$$\mathbf{f(n) > n^{\log_b a}}$$

## Passo 4: Conclusão (Caso 3 – A Raiz Domina)

O trabalho de divisão na raiz domina toda a computação da árvore:

$$\mathbf{T(n) \in \Theta(n^2)}$$

# Quando NÃO Aplicar o Teorema Mestre?

O Teorema Mestre é poderoso, mas possui limites bem definidos:

- **Decréscimo constante (não divisão fracionária):**
  Ex: $T(n) = T(n-1) + 1$ (Fatorial) ou $T(n) = 2T(n-1) + 1$ (Hanói).
  *Não há fator $b > 1$. Solução: use a Árvore de Recursão!*
- **Número de subproblemas $a$ ou fator $b$ variáveis:**
  Ex: $T(n) = 2^n \cdot T(n/2) + n$ (aqui $a = 2^n$ não é constante).
- **Subproblemas com tamanhos desiguais:**
  Ex: $T(n) = T(n/3) + T(2n/3) + n$ (Quicksort assimétrico).
  *Solução: use a Árvore de Recursão para somar os níveis!*
- **Brechas assintóticas não polinomiais:**
  Ex: $T(n) = 2T(n/2) + n \log n$ (o fator $\log n$ não é polinomial $n^\epsilon$).

# Quiz 1: Altura da Árvore de Execução

**Pergunta:** Qual é a altura da árvore de execução da **Busca Binária** e da **Busca Linear Recursiva**, respectivamente?

A) $\Theta(n)$ e $\Theta(n)$
B) $\Theta(\log n)$ e $\Theta(n)$
C) $\Theta(n \log n)$ e $\Theta(n^2)$
D) $\Theta(\log n)$ e $\Theta(2^n)$

# Quiz 1: Resolução Comentada

## Resposta Correta: B ($\Theta(\log n)$ e $\Theta(n)$)

- **Busca Binária:** A entrada é dividida por 2 a cada passo:

$$\frac{n}{2^h} = 1 \implies 2^h = n \implies \mathbf{h = \Theta(\log n)}$$

- **Busca Linear:** A entrada é subtraída de 1 a cada passo:

$$n - h = 1 \implies \mathbf{h = \Theta(n)}$$

# Quiz 2: Aplicação Rápida do Teorema Mestre

**Pergunta:** A qual classe $\Theta$ pertence a relação de recorrência:

$$T(n) = 8 \cdot T(n/2) + 1001 \cdot n^2$$

A) $\Theta(n^2)$
B) $\Theta(n \log n)$
C) $\Theta(n^3)$
D) $\Theta(2^n)$

# Quiz 2: Resolução Comentada

## Resposta Correta: C ($\Theta(n^3)$)

- $a = 8, b = 2 \implies n^{\log_b a} = n^{\log_2 8} = n^3$.
- $f(n) = 1001n^2 \in \Theta(n^2)$.
- Como $n^2$ cresce mais devagar que $n^3$, estamos no **Caso 1**:

$$\mathbf{T(n) \in \Theta(n^3)}$$

# Quiz 3: Aplicação Rápida do Teorema Mestre

**Pergunta:** A qual classe $\Theta$ pertence a relação de recorrência:

$$T(n) = 2 \cdot T(n/2) + 15 \cdot n$$

A) $\Theta(n)$
B) $\Theta(n^2)$
C) $\Theta(n \log n)$
D) $\Theta(\log n)$

# Quiz 3: Resolução Comentada

## Resposta Correta: C ($\Theta(n \log n)$)

- $a = 2, b = 2 \implies n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
- $f(n) = 15n \in \Theta(n)$.
- Mesma ordem de grandeza assintótica $\implies$ **Caso 2 (Empate)**:

$$\mathbf{T(n) \in \Theta(n \log n)}$$

# Quiz 4: Significado da Recorrência

**Pergunta:** O que significa, em português, a relação de recorrência $T(n) = 3 \cdot T(n/2) + n^2$?

A) Há 2 chamadas recursivas, cada uma dividindo a entrada por 3.

B) Cada uma das 3 chamadas recursivas diminui a entrada em 2 unidades.

C) Há 3 chamadas recursivas; cada uma divide a entrada pela metade; o custo de divisão/combinação é quadrático ($\Theta(n^2)$).

D) Trata-se da relação de recorrência do Merge Sort.

# Quiz 4: Resolução Comentada

## Resposta Correta: C

- $a = 3 \implies$ exatamente 3 subproblemas recursivos.
- $b = 2 \implies$ a entrada de cada chamada é reduzida à metade.
- $f(n) = n^2 \implies$ o trabalho fora das chamadas tem custo quadrático.

# Quiz 5: Identificando Inadequações

**Pergunta:** Qual das seguintes relações de recorrência **NÃO** se adequa ao Teorema Mestre clássico?

A) $T(n) = 7 \cdot T(n/13) + 1$

B) $T(n) = 2 \cdot T(n/2) + n^2$

C) $T(n) = 2^n \cdot T(n/2) + n^n$

D) $T(n) = T(n/2) + 1$

E) $T(n) = 2 \cdot T(n/2) + 10n$

# Quiz 5: Resolução Comentada

**Resposta Correta: C**

Na alternativa C, o número de chamadas é $a = 2^n$, que varia exponencialmente com $n$. O Teorema Mestre exige que $a$ seja uma constante fixa ($a \geq 1$).

# Quadro Comparativo dos Métodos

| Método | Quando Usar | Vantagens / Limitações |
|---|---|---|
| **Árvore de Recursão** | Recorrências gerais, divisão assimétrica, intuição visual. | Altamente didática; permite somar os níveis; exige desenhar a árvore. |
| **Teorema Mestre** | Divisão uniforme $aT(n/b) + f(n)$ com $a, b$ constantes. | Extremamente rápido e direto; restrito a divisões proporcionais. |

## Regra de Ouro do Estudante

1. Sempre teste se a recorrência se encaixa no **Teorema Mestre**.
2. Se não se aplicar (ex: decréscimo $n-1$, divisões desiguais), use a **Árvore de Recursão**!

# Próximos Passos

- **Fixação:** Pratique os 3 casos do Teorema Mestre na lista de exercícios da disciplina.
- **Próxima Aula:** Aplicação em algoritmos fundamentais de ordenação (Merge Sort, Quicksort e Heapsort) e limites inferiores ($\Omega(n \log n)$).

# Dúvidas?

*"Dividir para conquistar: a essência dos algoritmos eficientes."*