Eficiência de Algoritmos Recursivos

Disciplina: Estrutura de Dados I

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação
Universidade Estadual do Norte Fluminense

Análise de Algoritmos Recursivos

- **Algoritmo Recursivo:**
- Algoritmo que na sua descrição contém uma ou mais chamadas a si mesmo.

- **Exemplo 1:** Calcular o fatorial de um número n.

  - 3! = 3x2x1= 6
  - 5! = 5x4x3x2x1=120
  - 10! = 10x9x....x2x1 = 3.628.800
  - 20! = 20x19x....x2x1= 2.432.902.008.176.640.000

Onde armazenar?

  - 1000.000.000 10^9 bilhão
  - 1000.000.000.000 10^{12} trilhão
  - 1000.000.000.000.000 10^{15} quatrilhão
  - 1000.000.000.000.000.000 10^{18} quintilhão

Análise de Algoritmos Recursivos
O Calculo do Fatorial

- **Exemplo 1:** Calcular o fatorial de um número n.

- O calculo do fatorial de um número n pode ser definido mediante uma relação de recorrência.

- **Relação de Recorrência (ou relação recursiva):**
- É uma função definida mediante uma expressão que contem a mesma função .

$$n! = \begin{cases} 1, & \text{se n=0} \\ n(n-1)!, & \text{se n>0} \end{cases}$$

Base da recursão

Relação de recorrência

Análise de Algoritmos Recursivos
O Calculo do Fatorial

- **Exemplo 1:** Calcular o fatorial de um número n.

```
FATORIAL (n: inteiro): inteiro;                    { retorna o fatorial do número n}
inicio
    se n=0 então
        FATORIAL ← 1                                { caso base }
    senão
        FATORIAL ←  n * FATORIAL (n-1)              { caso da relação recorrente }
    fim-se
fim
```

Se n=0, $T(n) = 1$

Se n>0, $T(n) = 1+T(n-1)$

Análise de Algoritmos Recursivos
O Calculo do Fatorial

- **Método do Desdobramento:**
- Consiste em aplicar repetidamente a função de recorrência no termo de menor ordem (fazer substituições sucessivas) até atingir:
  - o caso base
  - ou conseguir identificar a forma geral

$$T(n) = \begin{cases} 1, & \text{se } n=0 \quad \text{(caso base)} \\ 1 + T(n-1), & \text{se } n>0 \quad \text{(relação de recorrência)} \end{cases}$$

- Para n>0:

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

- Substituindo na função de recorrência T(n-1), temos:

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

Análise de Algoritmos Recursivos
O Calculo do Fatorial

- Para n>0:

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

- Substituindo na função de recorrência T(n-1), $T(n) = 1 + T(n\text{-}1)$ temos:

$$T(n) = 1 + (1 + T(n\text{-}2))$$

- Substituindo na função de recorrência T(n-2), $T(n) = 1 + (1 + T(n\text{-}2))$ temos:

$$T(n) = 1 + (1 + (1 + T(n\text{-}3)) \qquad \boxed{T(n) = 3 + T(n\text{-}3)}$$

- Generalizando para k<n, temos:

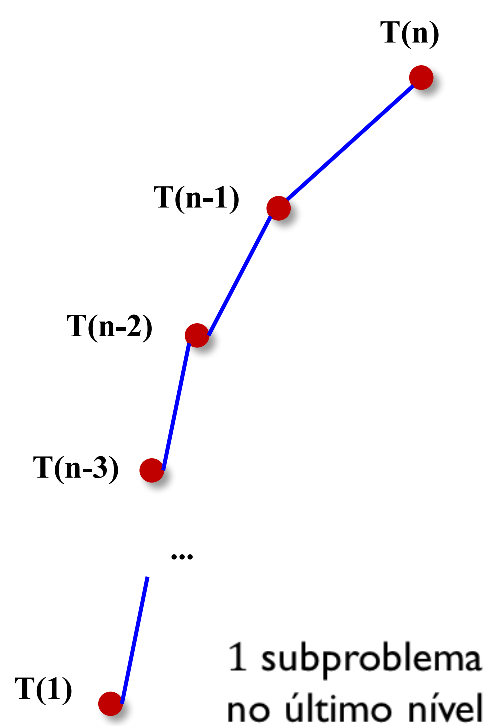
$$\boxed{T(n) = k + T(n\text{-}k)}$$

- Para k=n:

$$\boxed{T(n) = n + T(0)} \qquad \boxed{T(n) = n + 1} \qquad \boxed{T(n) = O(n)}$$

Análise de Algoritmos Recursivos
O Calculo do Fatorial

- A árvore recursiva gerada pelo problema de cálculo do fatorial tem a seguinte caraterística.
- Observe que o tipo de decomposição do problema. Um número pequeno de subproblemas é gerado.



Análise de Algoritmos Recursivos
O Calculo do Fatorial

- Observe que o calculo da função fatorial tem um comportamento linear $O(n)$.
- O número de operações cresce proporcionalmente com o tamanho do problema n.
- A única questão que limita o calculo do fatorial é valor do resultado, que cresce exponencialmente, e que ao produzir números muito grandes, exige formas de armazenamento especiais ou estendidas.

10	10	3.628.800	10^6 Milhão
15	15	1.307.674.368.000	10^{12} Trilhão
20	20	24.329.020.200.817.664.000	10^{18} Quintilhão

Algoritmo MergeSort

Princípio de Dividir e Conquistar

- O algoritmo MERGE-SORT(A, p, r) aplica o princípio de **divisão e conquista** para realizar a ordenação do arranjo A.
- O princípio de Dividir e Conquistar é típico de **algoritmos recursivos**, e compreende três etapas básicas**:**

- **Dividir.-** O problema é dividido em um determinado número de subproblemas.
- **Conquistar.-** Cada subproblema é resolvido de maneira recursiva. Ou seja, dividindo ele novamente em subproblemas e assim sucessivamente até que o tamanho do problema resulte na solução de maneira direta ou trivial.
- **Combinar.-** Utilizar a solução dos subproblemas menores para produzir a solução para um problema maior.

O Procedimento de Intercalação

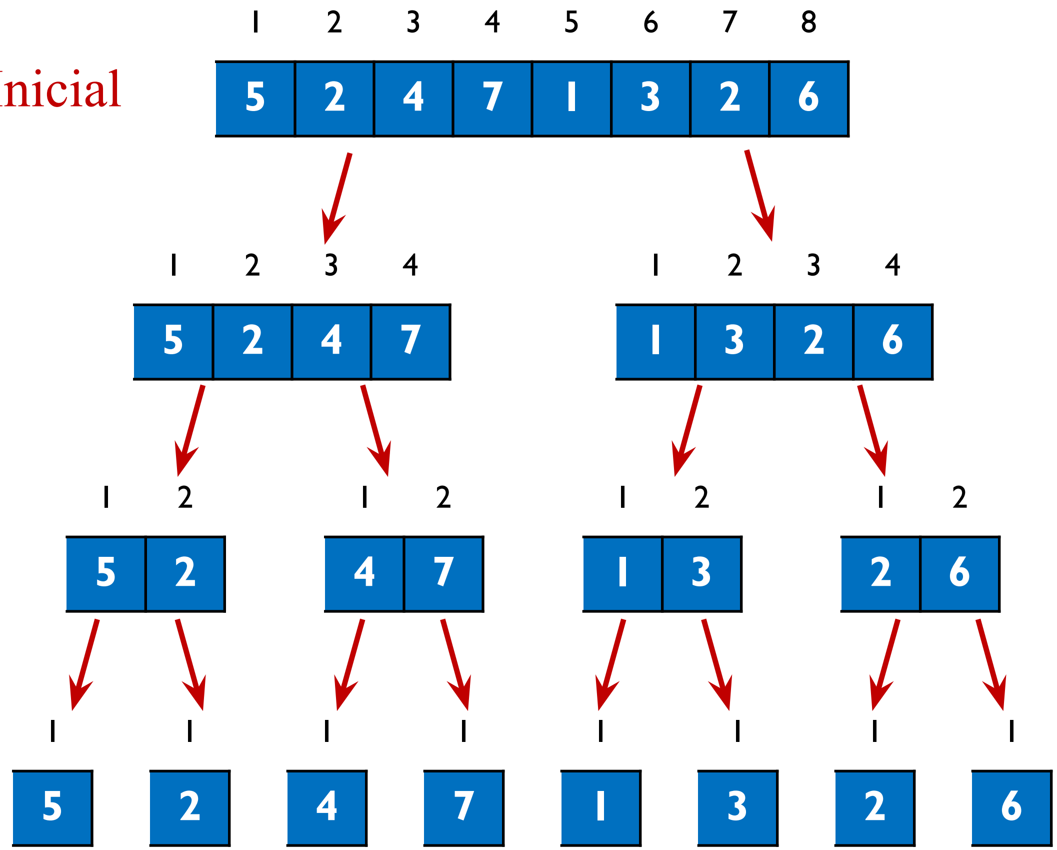
Ordenação de um Subarranjo

- No procedimento MERGE (A, p, q, r), **intercalação,** é o procedimento chave do algoritmo Merges*ort.*
- Este é o procedimento realiza a ordenação de um subarranjo do vetor A, delimitado pelos índices p e r, (extremos esquerdo e direito).
- Alem disso, o algoritmo utiliza o índice intermediário q, que serve para dividir o subarranjo em duas partes.
- O procedimento pressupõe que os subarranjos A[p..q] e A[q+1..r] já se encontram em sequência ordenada.

O Principio de Dividir e Conquistar

Dividindo o Problema

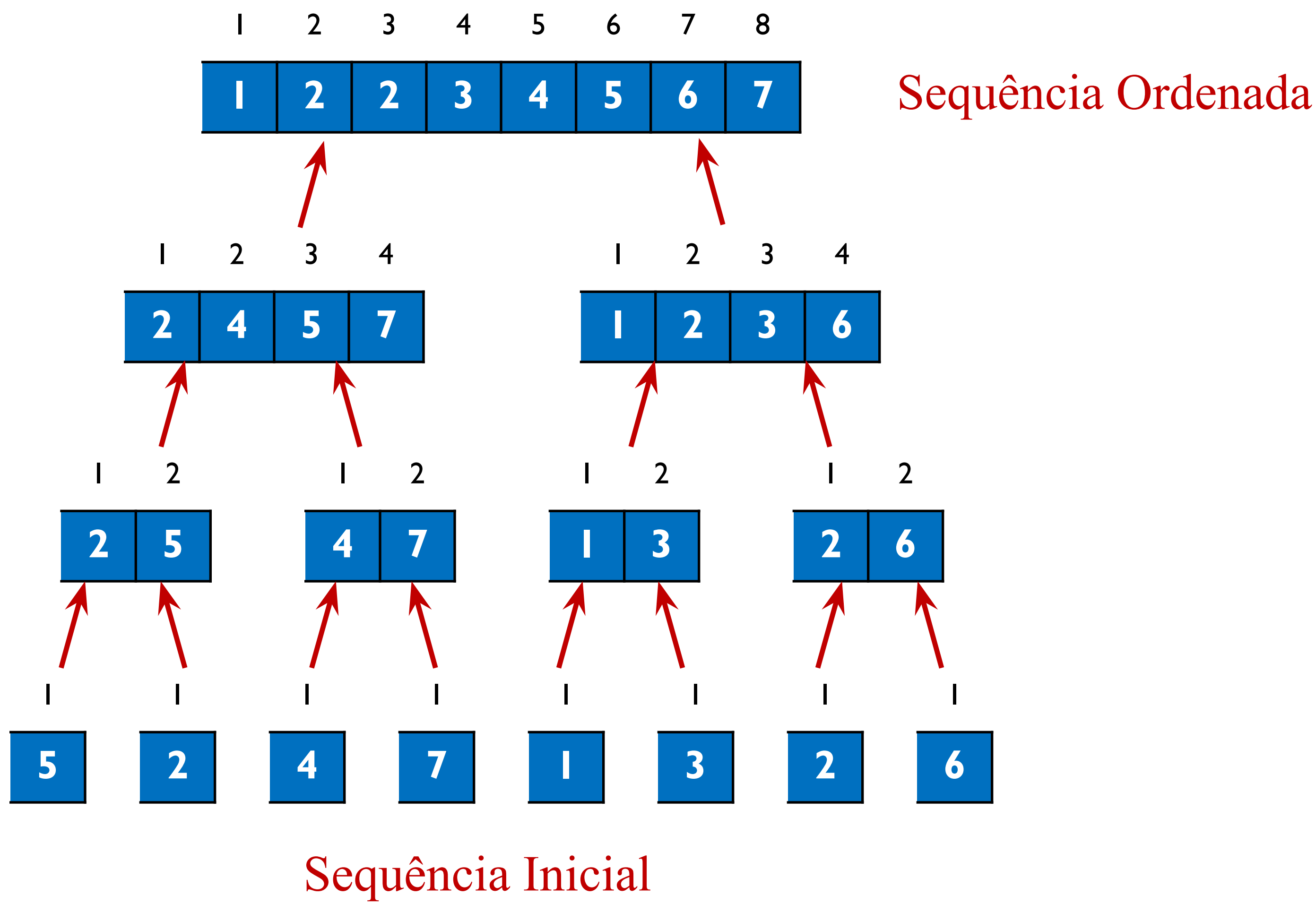
Sequência Inicial



Subproblemas Triviais

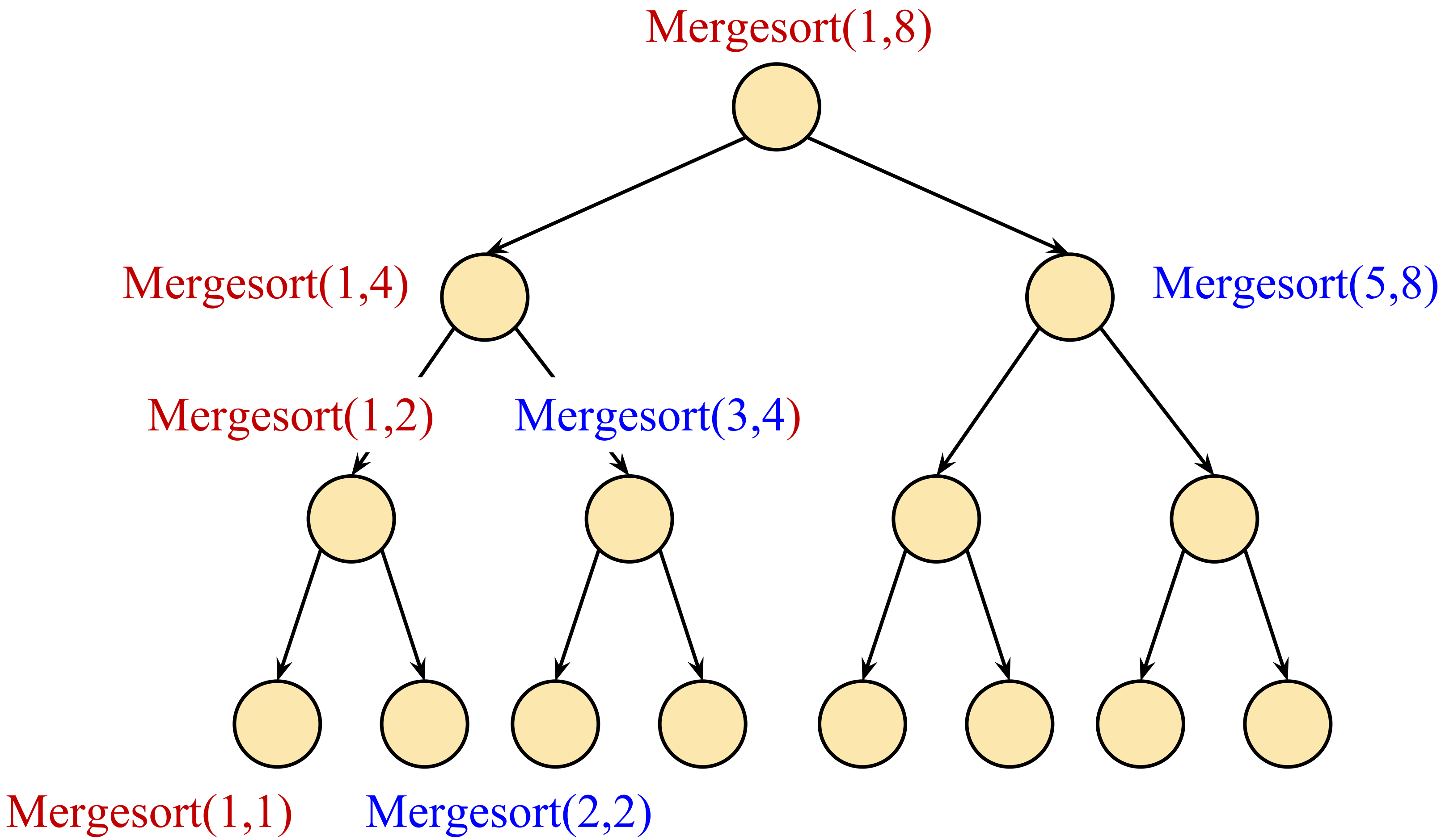
O Principio de Dividir e Conquistar

Combinando a solução dos subproblemas



Algoritmo Mergesort

Sequência de Chamadas Recursivas

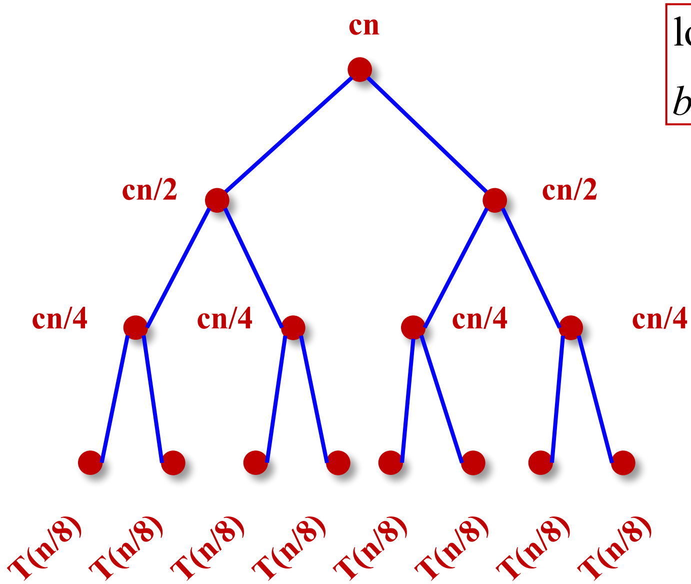


Análise de Algoritmos Recursivos

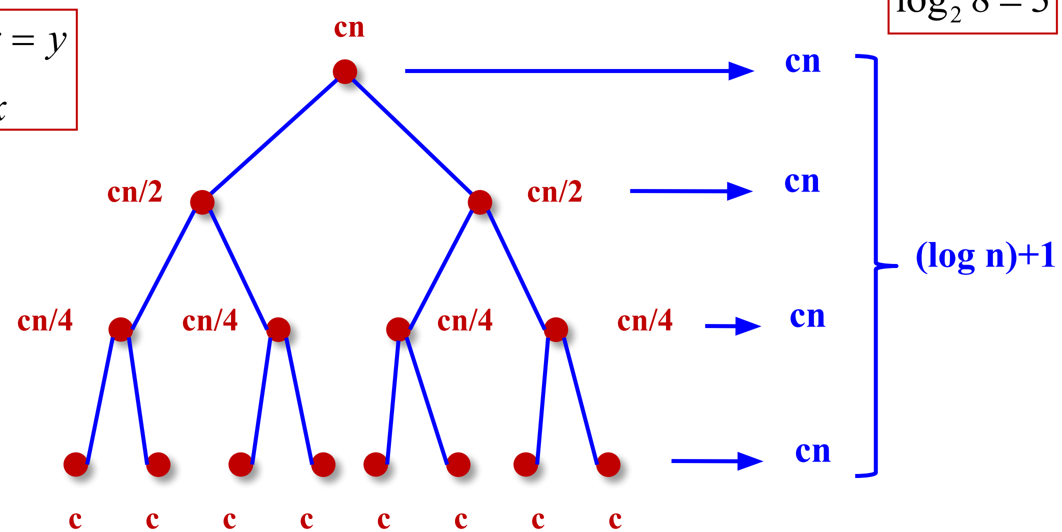
Algoritmo MergeSort

- O custo do algoritmo Mergesort, esta diretamente associado ao processo de intercalação e ao número de intercalações.
- Fazendo as contas, a complexidade do algoritmo resulta em O(n*log n).

$\log_2 8 = 3$



$$\log_b x = y$$
$$b^y = x$$



(log n+1) (cn)

n = 8 $\Rightarrow$ n/8 =1 $\Rightarrow$ T(1) = c

(log n+1) (cn) = cn logn + cn = O(n log n)

Análise de Algoritmos Recursivos

Algoritmo MergeSort

- **Método do Desdobramento:**
- Consiste em aplicar repetidamente a função de recorrência no termo de menor ordem (fazer substituições sucessivas) até atingir:
  - o caso base
  - ou conseguir identificar a forma geral

Tempo constante

$$T(n) = \begin{cases} c, & \text{se } n=1 \quad \text{(Intercalação)} \\ 2.T(n/2) + cn, & \text{se } n>1 \quad \text{(Divisão + Intercalação)} \end{cases}$$

- Para n>0:

$$T(n) = 2.T(n/2) + cn$$

- Substituindo na função de recorrência T(n/2), temos:

$$T(n) = 2.(2.T(n/4)+c.n/2) + cn$$

Análise de Algoritmos Recursivos
Algoritmo MergeSort

- Para n>0:

$$T(n) = 2.T(n/2) + cn$$

- Substituindo na função de recorrência T(n/2), temos:

$$T(n) = 2.T(n/2) + cn$$

$$T(n) = 2.(2.T(n/4)+c.n/2) + cn$$

$$T(n) = 2^2.T(n/4) + 2cn$$

- Substituindo na função de recorrência T(n/4), temos:

$$T(n) = 2^2.T(n/4) + 2cn$$

$$T(n) = 2^2.(2.T(n/8)+c.n/4) + 2cn$$

$$\boxed{T(n) = 2^3.T(n/8) + 3cn}$$

- Generalizando para k<n, temos:

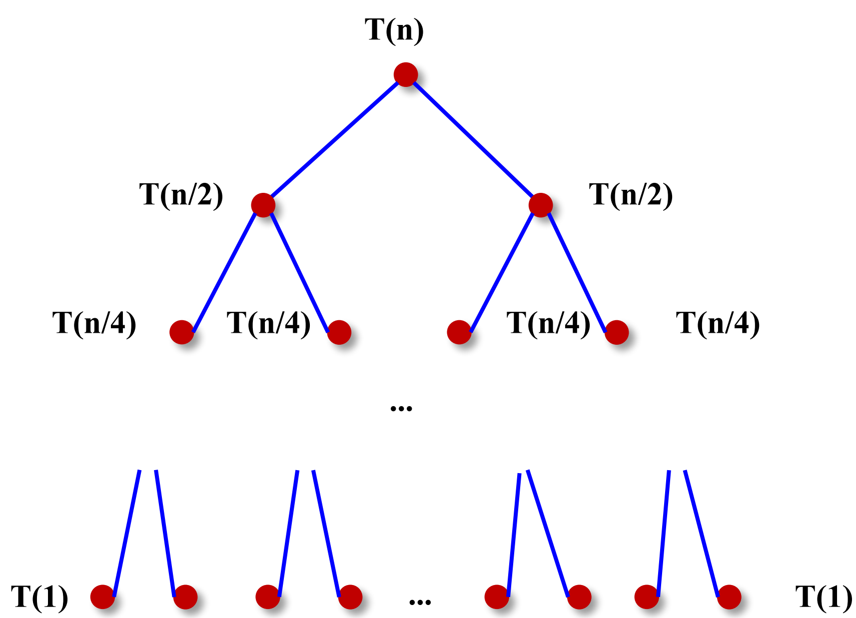
$$\boxed{T(n) = 2^k.T(n/2^k) + kcn}$$

- Considerando k, tal que $n=2^k$, temos:

$$\boxed{T(n) = n.T(1) + (\log n).cn} \quad \boxed{T(n) = c.n + c.n\,(\log n)} \quad \boxed{T(n) = O(n \log n)}$$

Análise de Algoritmos Recursivos
Algoritmo MergeSort

- A árvore recursiva gerada pelo problema da ordenação Mergesort tem a seguinte caraterística.
- Observe que a decomposição do problema (simplificação) divide o problema original em dois subproblemas com tamanho igual a metade do problema original.
- O número de subproblemas gerados não é muito grande.



n subproblemas no último nível

Análise de Algoritmos Recursivos

Algoritmo MergeSort

- Observe que o calculo da função de ordenação mergesort tem um comportamento $O(n\ log\ n)$.
- O número de operações cresce proporcionalmente a $n\ log\ n$, onde n é tamanho do problema, conforme indicado na tabela.
- Problemas de tamanho relativamente grande podem ser resolvidos em tempo relativamente curto.

$$\boxed{\log_2 10 = \frac{\log_e 10}{\log_e 2} = \frac{\ln 10}{\ln 2}}$$

10	33,21
100	664,38
1.000	9.965,78
10.000	132.877,12
100.000	1.660.964,04
1.000.000	19.931.568,56

10^1 Decena

$\boxed{1Ghz = 10^9 \text{ operações/seg}}$

10^2 Centena

10^3 Milhar

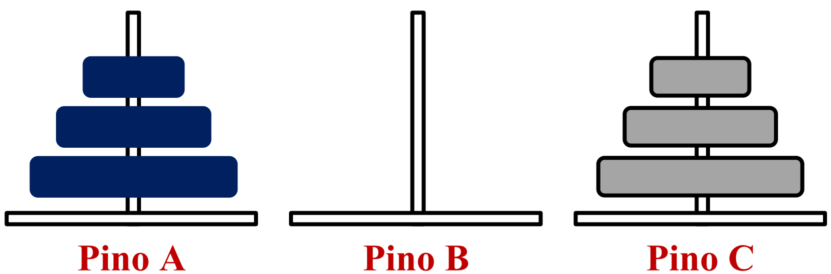
$10^2 \times 10^3$ Centena de milhar $\boxed{10^5/10^9 = 10^{-4} = 0{,}0001}$

10^6 Milhão $\boxed{10^6/10^9 = 10^{-3} \approx \text{milésimos seg}}$

$10^1 \times 10^6$ Dezena de milhão $\boxed{10^{-2} \approx \text{centésimos seg}}$

Análise de Algoritmos Recursivos
O Problema das Torres de Hanói

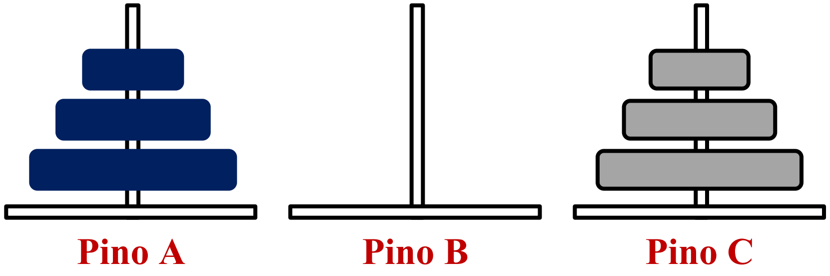
- “No inicio dos tempos, Deus criou as Torres de Brahma com 64 discos de ouro puro, dispostos em agulhas de diamante. Quando a hora chegou, ordenou a um grupo de monges de um Monastério no Tibet que dessem inicio à movimentação dos discos segundo regras determinadas e alertou que concluído o trabalho as torres desmoronariam dando inicio ao final dos tempos”.



Análise de Algoritmos Recursivos

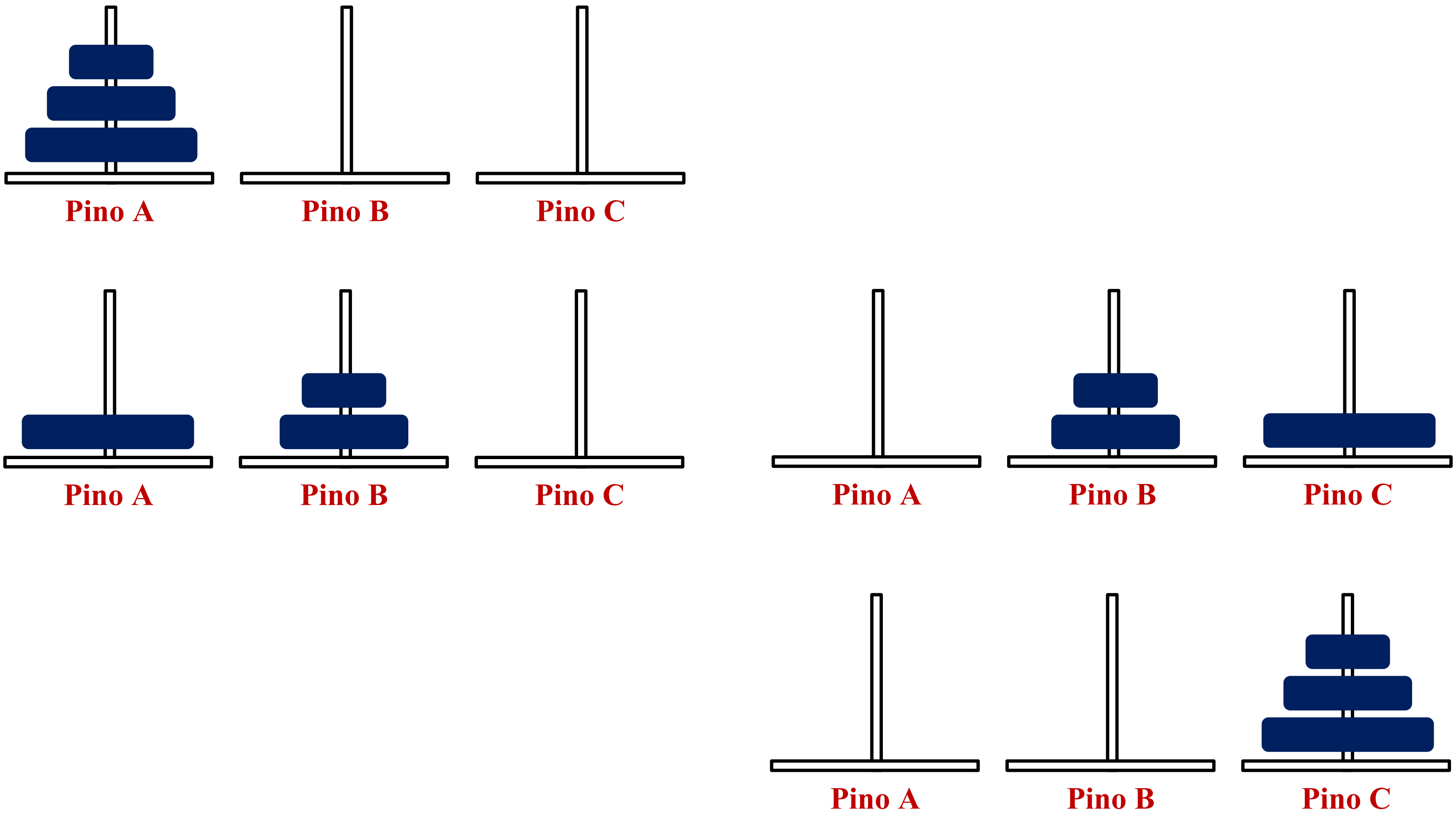
O Problema das Torres de Hanói

- **Exemplo 2:**
- **(Versão Simplificada com 3 discos).** Deslocar os 3 discos do pino A para o pino C usando o pino B como intermediário sempre que seja necessário, atendendo as seguintes restrições:
  - Somente um disco pode ser movimentado de cada vez;
  - Um disco maior não pode ser colocado sobre um disco menor;
  - Realizar o menor número de movimentos.

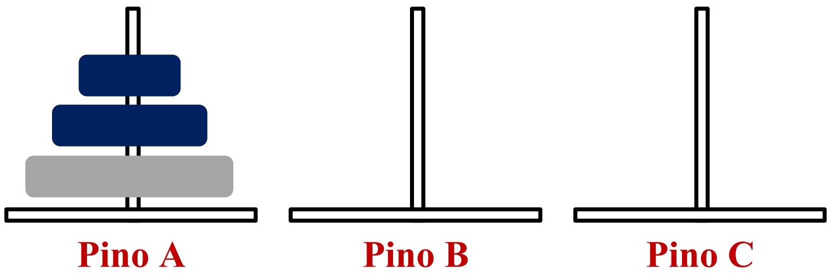


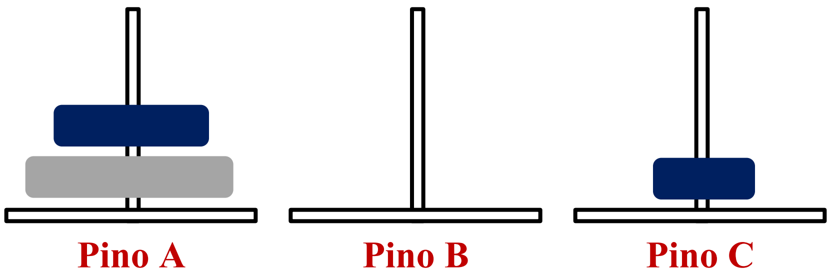
Análise de Algoritmos Recursivos

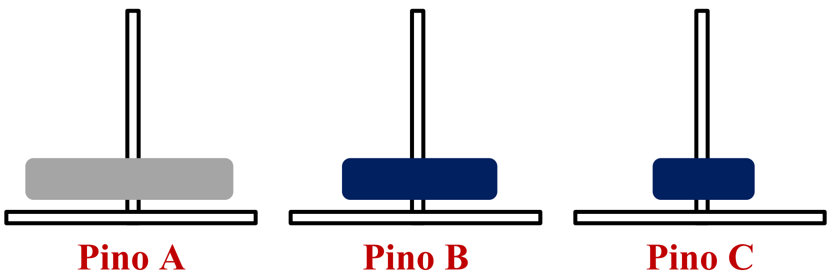
O Problema das Torres de Hanói

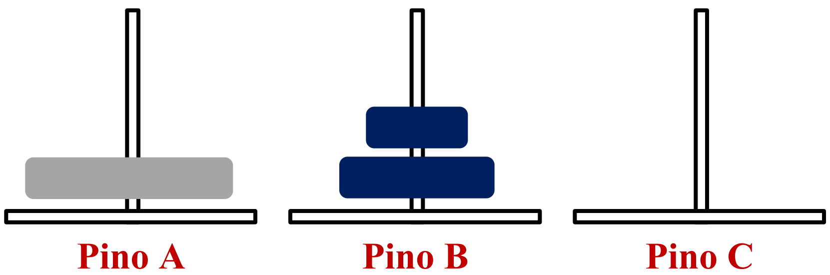


Análise de Algoritmos Recursivos
O Problema das Torres de Hanói



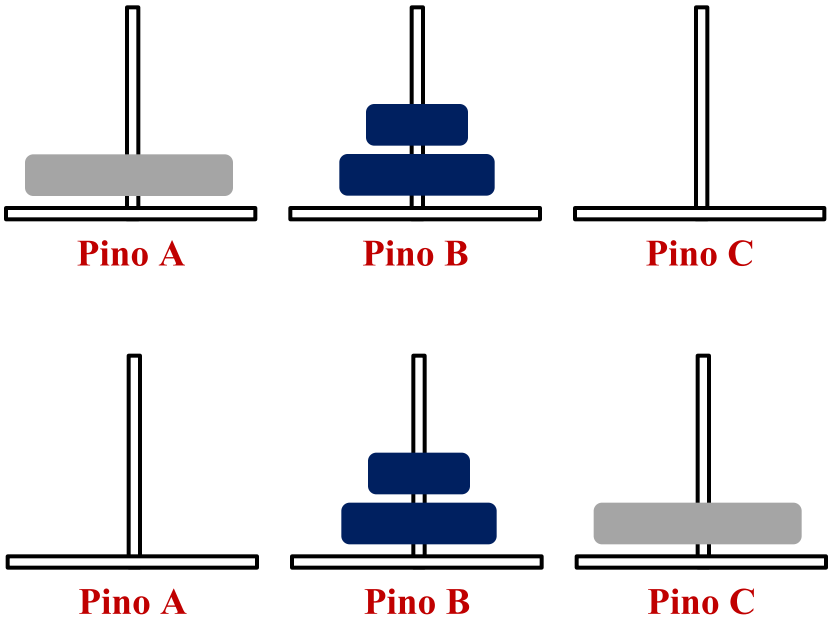






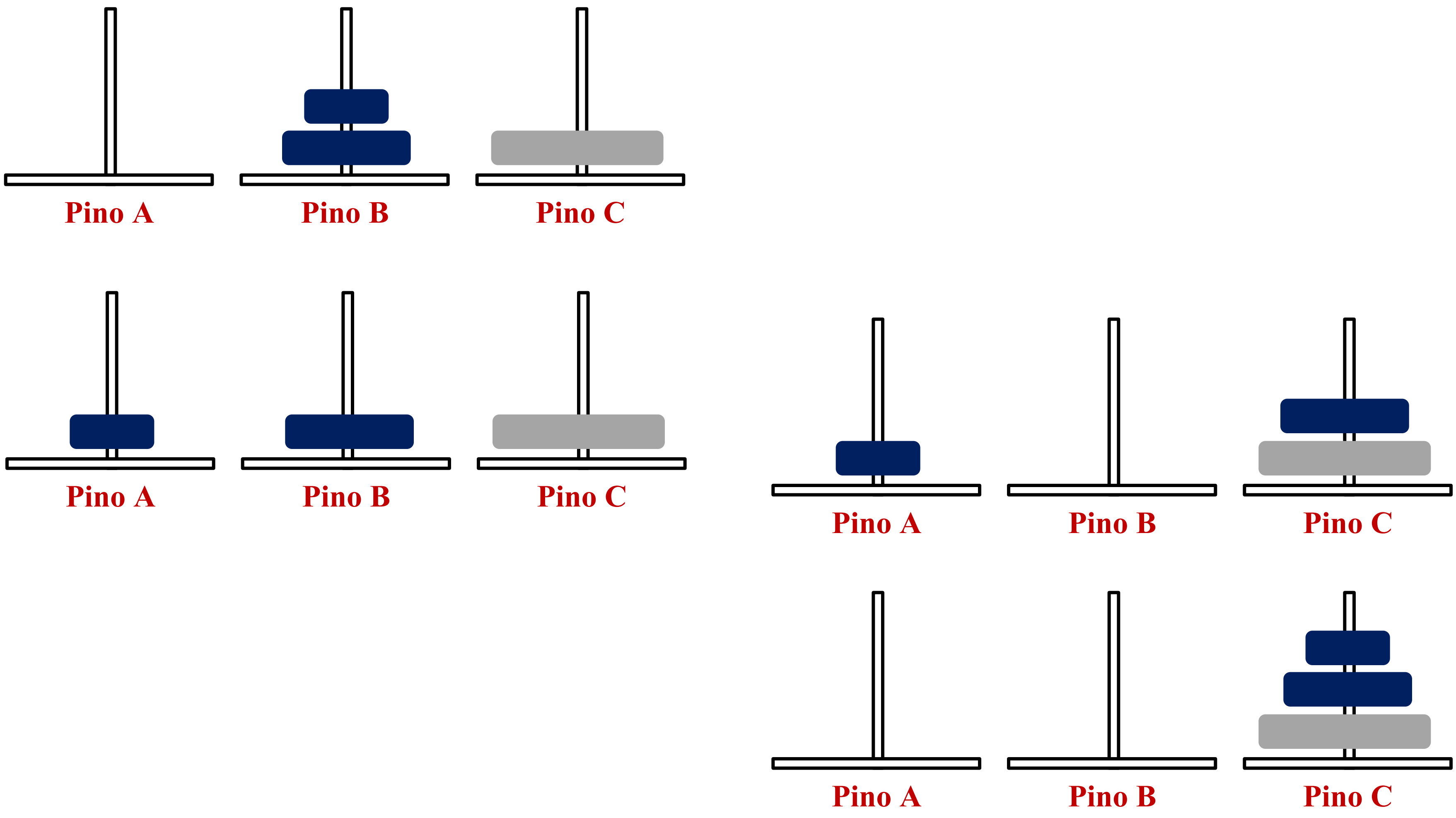
Análise de Algoritmos Recursivos

O Problema das Torres de Hanói



Análise de Algoritmos Recursivos

O Problema das Torres de Hanói



Análise de Algoritmos Recursivos
O Problema das Torres de Hanói

- **Exemplo 2:** Deslocar os n discos do pino A para o pino C usando o pino B como intermediário.

```
HANOI (n, A, C ,B);                          { move n discos de A para C usando B}
inicio
    se n=1 então
        MOVER (n, A, C)                      { caso base, move disco n de A para C }
    senão
        HANOI (n-1, A, B ,C)                 { move n-1 discos de A para B usando C }
        MOVER (n, A, C)                      { move disco n de A para C }
        HANOI (n-1, B, C ,A)                 { move n-1 discos de B para C usando A }
    fim-se
fim
```

$$T(n) = \begin{cases} 1, & \text{se } n=1 \quad \text{(caso base)} \\ 2.T(n-1)+1, & \text{se } n>1 \quad \text{(relação de recorrência)} \end{cases}$$

Análise de Algoritmos Recursivos

O Problema das Torres de Hanói

- Aplicando o Método do Desdobramento:

$$T(n) = \begin{cases} 1, & \text{se } n=1 \quad \text{(caso base)} \\ 2.T(n\text{-}1)+1, & \text{se } n>1 \quad \text{(relação de recorrência)} \end{cases}$$

- Para n>1: $T(n) = 2.T(n\text{-}1) + 1$

- Substituindo na função de recorrência T(n-1), temos: $T(n) = 2.T(n\text{-}1) + 1$

$$T(n) = 2.(2.T(n\text{-}2)+1) + 1 \qquad T(n) = 2^2\,T(n\text{-}2) + 2 + 1$$

- Substituindo na função de recorrência T(n-2), temos: $T(n) = 2^2\,T(n\text{-}2) + 2 + 1$

$$T(n) = 2^2(2.T(n\text{-}3)+1) + 2 + 1 \qquad \boxed{T(n) = 2^3.T(n\text{-}3) + 2^2 + 2 + 1}$$

- Generalizando para k<n

$$\boxed{T(n) = 2^k.T(n\text{-}k) + 2^{k-1} + 2^{k-2} + \ldots + 2 + 1}$$

Análise de Algoritmos Recursivos

O Problema das Torres de Hanói

$$\boxed{T(n) = 2^{k}.T(n\text{-}k) + 2^{k-1} + 2^{k-2} + \ldots + 2 + 1}$$

- O processo de substituição termina quando o termo T(n-k) cai no **caso base (condição de parada)**, T(n-k)=1;
- Isso acontece quando (n-k) =1, onde k = (n-1);
- Substituindo ambas relações na expressão anterior temos:

$$T(n) = 2^{n-1}.T(1) + 2^{n-1-1} + 2^{n-1-2} + \ldots + 2 + 1$$

$$\boxed{T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2 + 1}$$

- Usando a expressão da **progressão geométrica**:

$$2^{n} + 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 = 2^{n+1} - 1$$

- Temos:

$$\boxed{T(n) = 2^{n} - 1 = O(2^{n})}$$

Análise de Algoritmos Recursivos
O Problema das Torres de Hanói (Revisado)

- Aplicando o Método do Desdobramento:

Tempo constante

$$T(n) = \begin{cases} c, & \text{se } n=1 \text{ (caso base)} \\ 2.T(n\text{-}1)+c, & \text{se } n>1 \text{ (relação de recorrência)} \end{cases}$$

- Para n>1: $T(n) = 2.T(n\text{-}1) + c$
- Substituindo na função de recorrência T(n-1), temos: $T(n) = 2.T(n\text{-}1) + c$

$$T(n) = 2.(2.T(n\text{-}2)+c) + c \qquad T(n) = 2^2 T(n\text{-}2) + 2c + 1c$$

- Substituindo na função de recorrência T(n-2), temos: $T(n) = 2^2 T(n\text{-}2) + 2c + 1c$

$$T(n) = 2^2 (2.T(n\text{-}3)+c) + 2c + 1c \qquad \boxed{T(n) = 2^3.T(n\text{-}3) + \left(2^2 + 2 + 1\right)c}$$

- Generalizando para k<n

$$\boxed{T(n) = 2^k.T(n\text{-}k) + \left(2^{k-1} + 2^{k-2} + \ldots + 2 + 1\right)c}$$

Análise de Algoritmos Recursivos
O Problema das Torres de Hanói (Revisado)

$$\boxed{T(n) = 2^k.T(n\text{-}k) + \left(2^{k-1} + 2^{k-2} + \ldots + 2 + 1\right)c}$$

- O processo de substituição termina quando o termo T(n-k) cai no **caso base (condição de parada)**, T(n-k)=c;
- Isso acontece quando (n-k) =1, onde k = (n-1);
- Substituindo ambas relações na expressão anterior temos:

$$T(n) = 2^{n-1}.T(1) + \left(2^{n-1-1} + 2^{n-1-2} + \ldots + 2 + 1\right)c$$

$$\boxed{T(n) = \left(2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2 + 1\right)c}$$

- Usando a expressão da progressão geométrica:

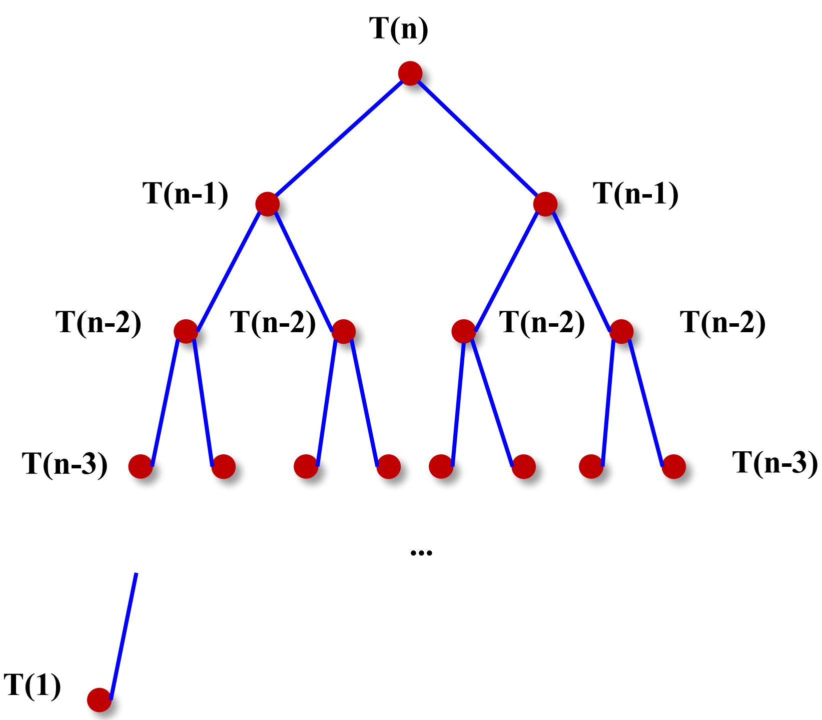
$$2^n + 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 = 2^{n+1} - 1$$

- Temos:

$$\boxed{T(n) = \left(2^n - 1\right)c = O\left(2^n\right)}$$

Análise de Algoritmos Recursivos
O Problema das Torres de Hanói

- A árvore recursiva gerada pelo problema das Torres de Hanoi tem a seguinte caraterística.
- Observe que o grau de decomposição do problema (simplificação) não é muito grande. Uma grande quantidade de subproblemas é gerada.



Análise de Algoritmos Recursivos

O Problema das Torres de Hanói

- Qual é o tamanho máximo de problema que poderemos resolver no caso das torres de Hanói?.
- Isso depende, muito mais do tamanho do problema n, que da velocidade de processamento do seu computador.
- Como a função 2^n é uma função que cresce de maneira exponencial, o número de operações ou movimentos realizados será muito grande, a partir de um certo tamanho n.
- A a partir de um certo tamanho n o problema se torna intratável.

10	1.024
20	1.048.576
30	1.073.741.824
40	1.099.511.627.776
...	...
64	18.446.744.073.709.551.616

$1Ghz = 10^9$ operações/seg

10^3 = Milhar

10^6 = Milhão

10^9 = Bilhão — ≈ 1seg

10^{12} = Trilhão — $10^{12}/10^9 = 10^3$ segs. $\approx 16{,}6$ minutos

10^{18} = Quintilhão — $10^{18}/10^9 = 10^9$ segs. $\approx 31{,}7$ anos

Referências

- Thomas **Cormen**, Charles **Leiserson,** et al.. Algoritmos. Teoria e Prática. 2ª Edição. 2002. Capítulo 4.