Recorrências

Recursão
Relações de recorrência
Divisão e Conquista
Método de substituição
Método de Árvore de Recursão
Teorema Mestre

Recursividade

- Um procedimento que chama a si mesmo, direta ou indiretamente, é dito ser **recursivo**
- Permite descrever algoritmos de forma mais clara e concisa, especialmente para problemas de natureza recursiva

Recursividade

- Implementação
  - Dados locais usados em cada chamada de um procedimento recursivo são armazenados em uma pilha
  - Dados são recuperados quando uma dada chamada retorna de uma outra chamada

Recursividade

- Terminação
  - A chamada recursiva deve estar sujeita a uma condição que se torna falsa em algum momento da execução.
    - Falta de terminação gera estouro de pilha (stack overflow)
  - Ex:
    - Chamar um procedimento para *n*-1 enquanto *n*>0.

Recorrência

- Equação ou desigualdade que descreve a função em termos do seu próprio valor em entradas menores
- O tempo de execução de algoritmos recursivos pode ser descrito por uma recorrência

$T(n)$: tempo para problema de tamanho n

a subproblemas, cada um de tamanho $\frac{n}{b}$

- $aT(n/b)$

$D(n)$: tempo para dividir o problema

$C(n)$: tempo para combinar as soluções dos subproblemas

$$T(n) = \begin{cases} \Theta(1) & \text{Se } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{caso contrário} \end{cases}$$

Exemplo

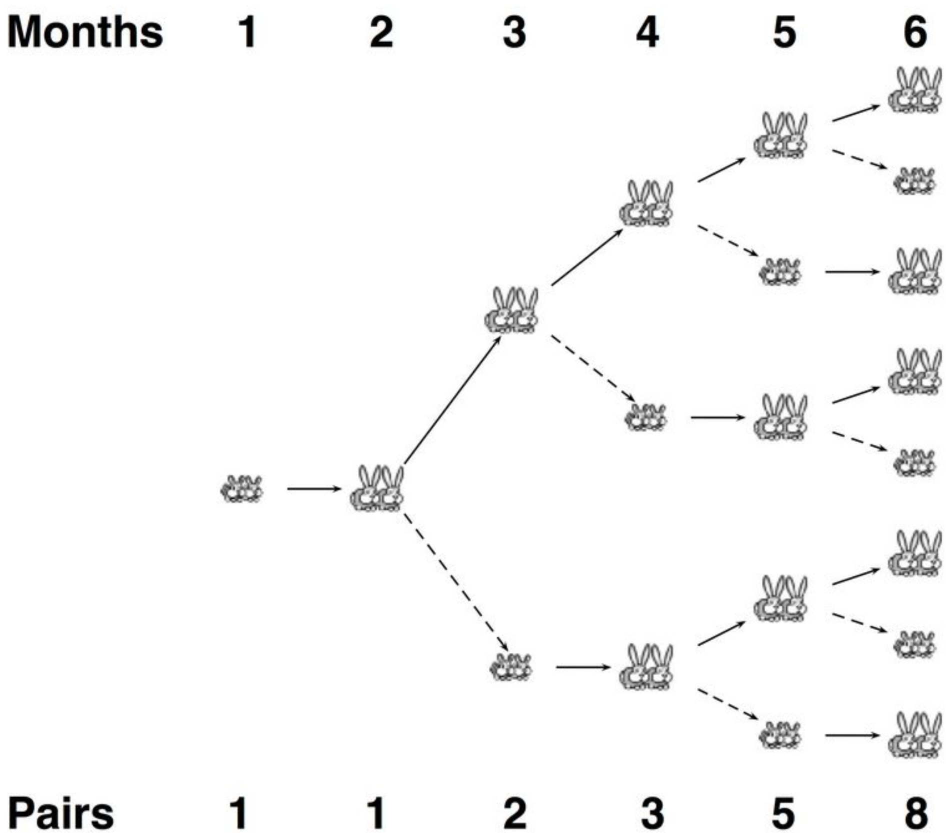
- Fatorial

```
Fatorial(n){
   If n≤1 then
      return 1;
   else
      return (n*Fatorial(n-1));
}
```

$$T(n) = \begin{cases} \Theta(1) \text{ se } n \leq 1 \\ T(n-1) + \Theta(1) \text{ se } n > 1 \end{cases}$$

Sequência de Fibonacci

- Reprodução de uma população de coelhos



Exemplo

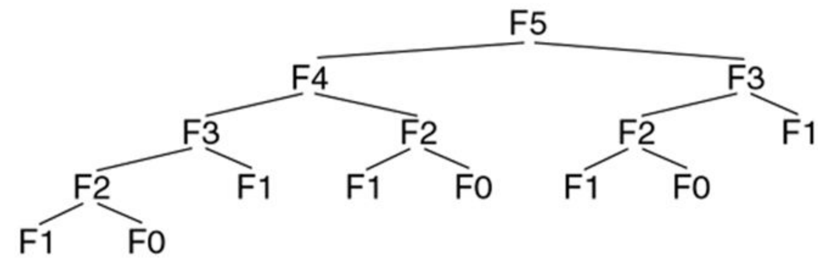
- Fibonacci

```
Fibonacci(n){
    If n≤1 then
        return n;
    else
        return (Fibonacci(n-1)+Fibonacci(n-2));
}
```

$$T(n) = \begin{cases} \Theta(1) \text{ se } n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) \text{ se } n > 1 \end{cases}$$

Exemplo

- Fibonacci




```
Fibonacci(n){
    If n≤1 then
        return n;
    else
        return (Fibonacci(n-1)+Fibonacci(n-2));
}
```

$$T(n) = \begin{cases} \Theta(1) \text{ se } n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) \text{ se } n > 1 \end{cases}$$

Número de chamadas recursivas = Número de Fibonacci

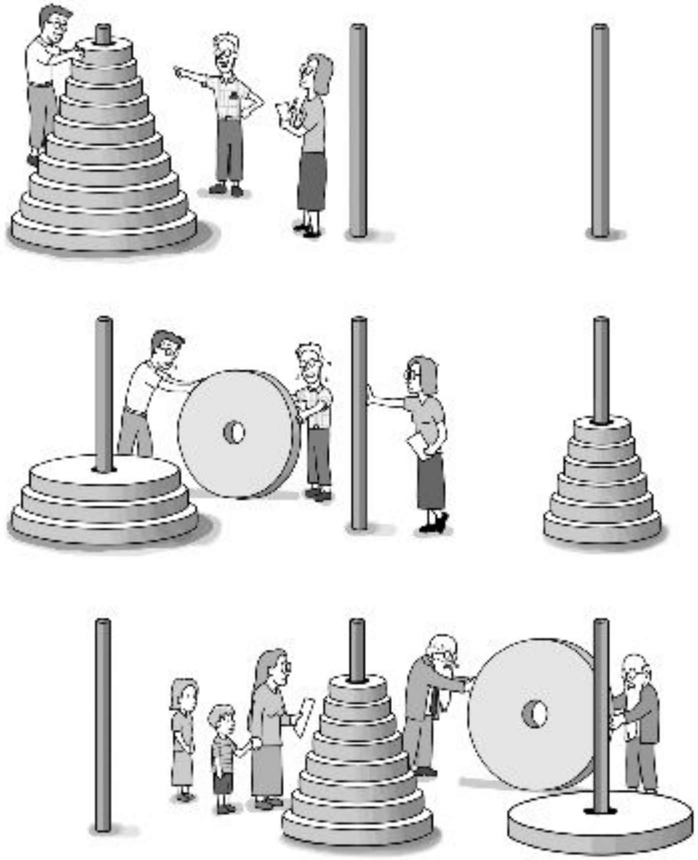
Complexidade de tempo e espaço: $f(n) = O(\Phi^n)$

$$\Phi = \left(\frac{1+\sqrt{5}}{2}\right) = 1{,}61803\ldots$$

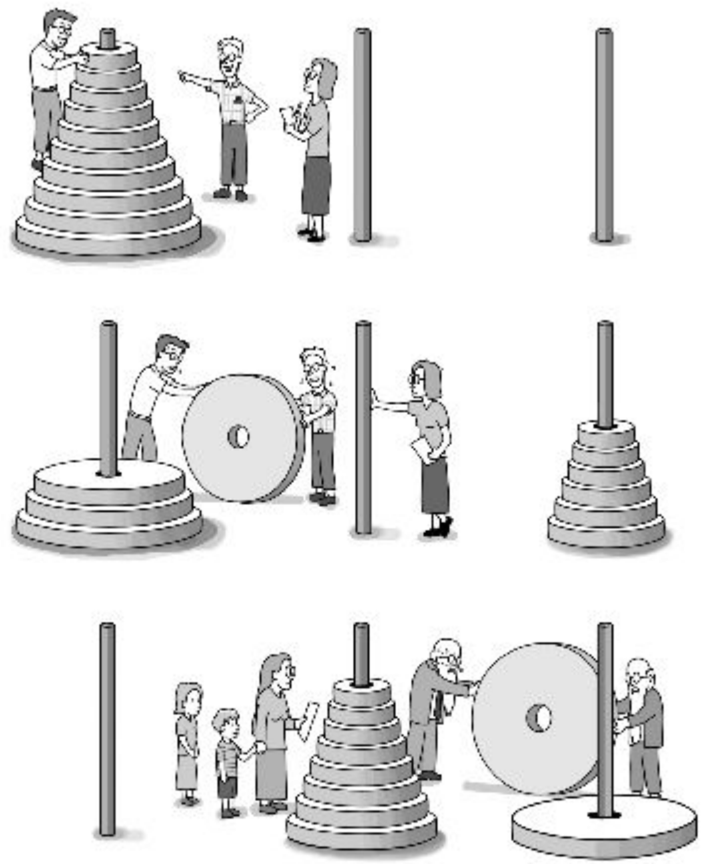
Golden ratio

Complexidade Exponencial!

Torre de Hanói



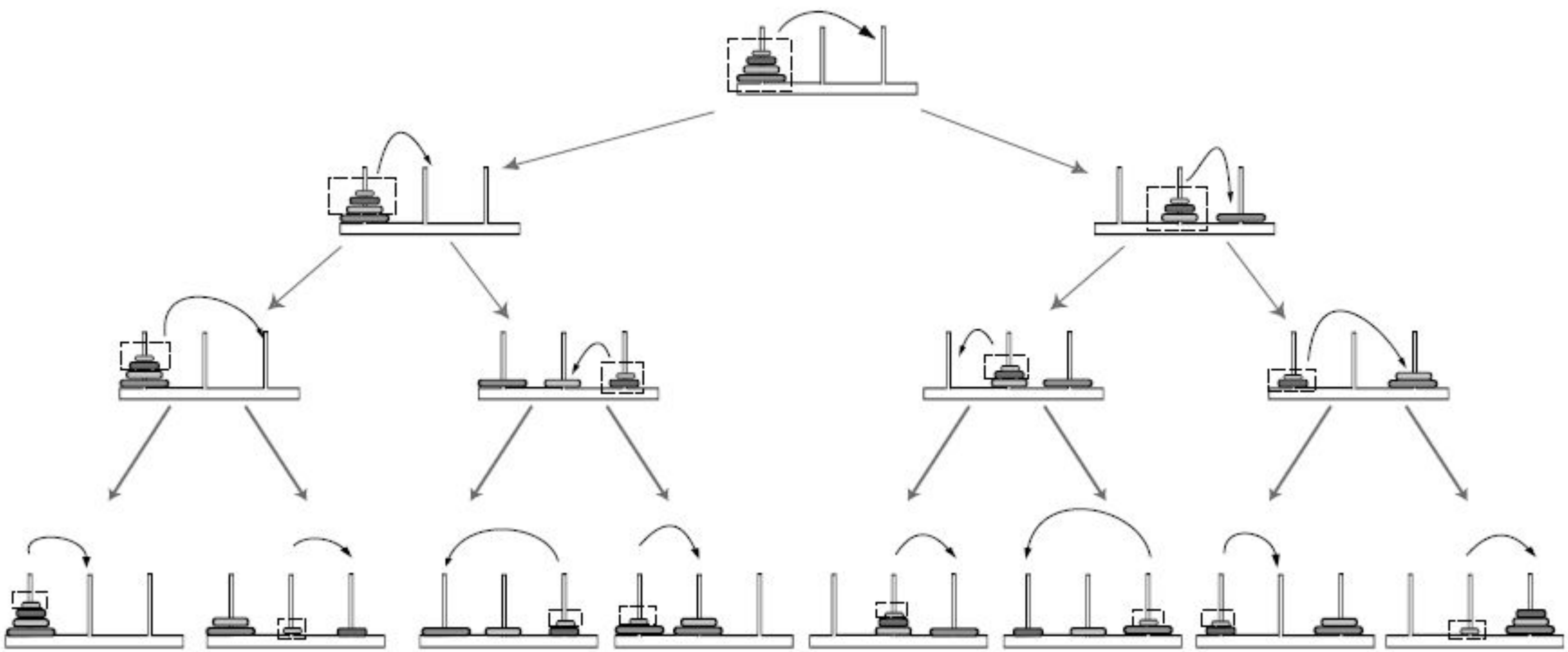
Torre de Hanói



- Mover n-1 discos do pino inicial para o auxiliar
- Mover disco da base do pino inicial para o final
- Mover n-1 discos do pino auxiliar para o final

Torre de Hanói

```
int hanoi(int n, int inicio, int aux, int fim){
    if(n == 1)
        printf("Move disco de %d para %d\n",inicio,fim);
    hanoi(n-1,inicio,fim,aux);
    printf("Move disco de %d para %d\n",inicio,fim);
    hanoi(n-1,aux,inicio,fim);
}
```

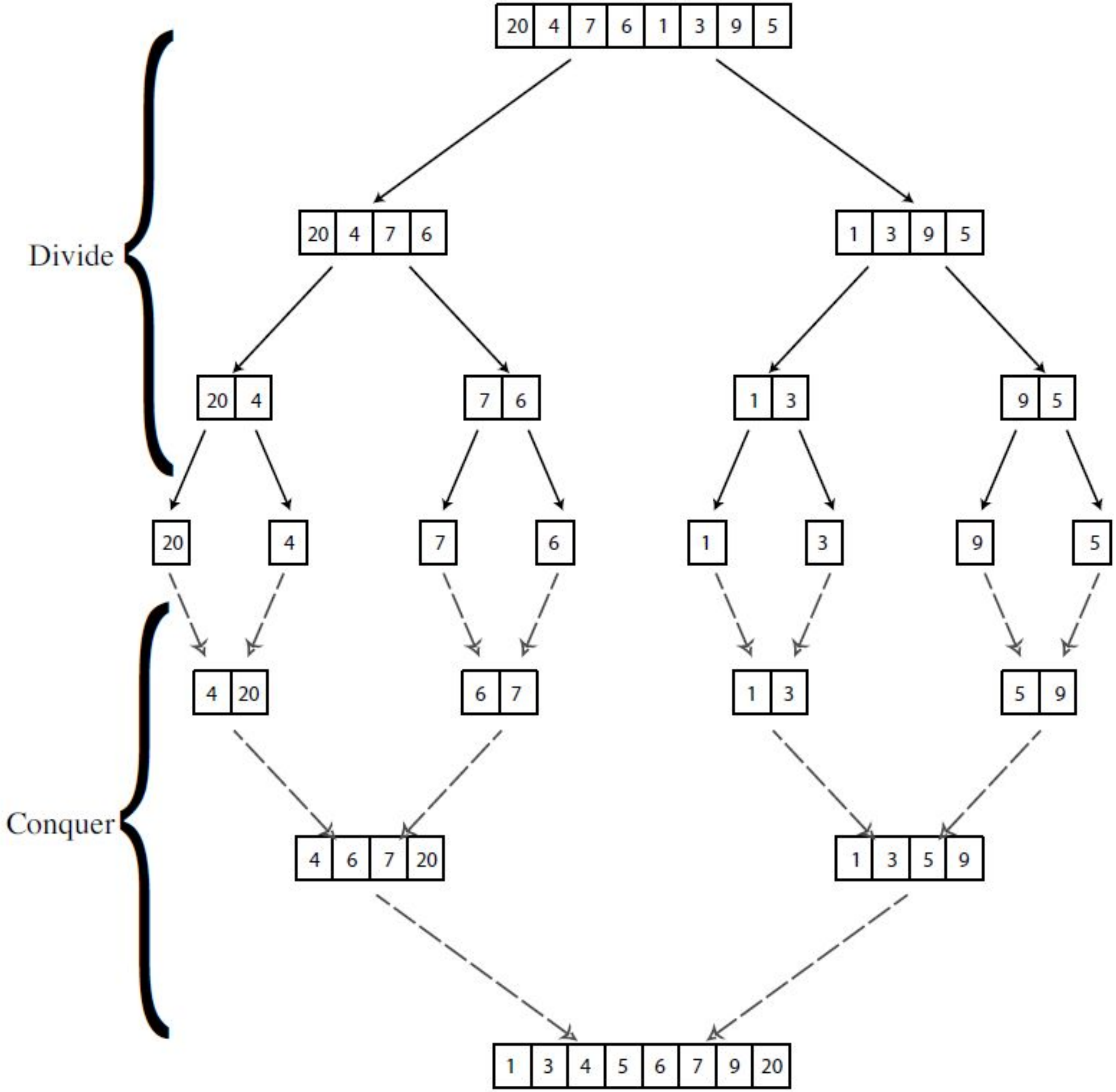


Abordagem Divisão e Conquista

- **Dividir**
  - Desmembra o problema em vários subproblemas menores que são semelhantes ao problema original
- **Conquistar**
  - Resolve os subproblemas recursivamente
- **Combinar**
  - Combina as soluções dos subproblemas para resolver o problema original

Exemplo

- Merge Sort
  - **Dividir**
    - Divide a sequência de n elementos em 2 subsequências de cerca de $n/2$ elementos cada
  - **Conquistar**
    - Realiza chamadas recursivas para cada uma das subsequências até chegar ao caso base
  - **Combinar**
    - Faz a intercalação das 2 sequências ordenadas


Divide
Conquer
20 4 7 6 1 3 9 5
20 4 7 6
1 3 9 5
20 4
7 6
1 3
9 5
20
4
7
6
1
3
9
5
4 20
6 7
1 3
5 9
4 6 7 20
1 3 5 9
1 3 4 5 6 7 9 20

Merge

MERGE(A, p, q, r)

```
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Ilustração do procedimento Merge (1)

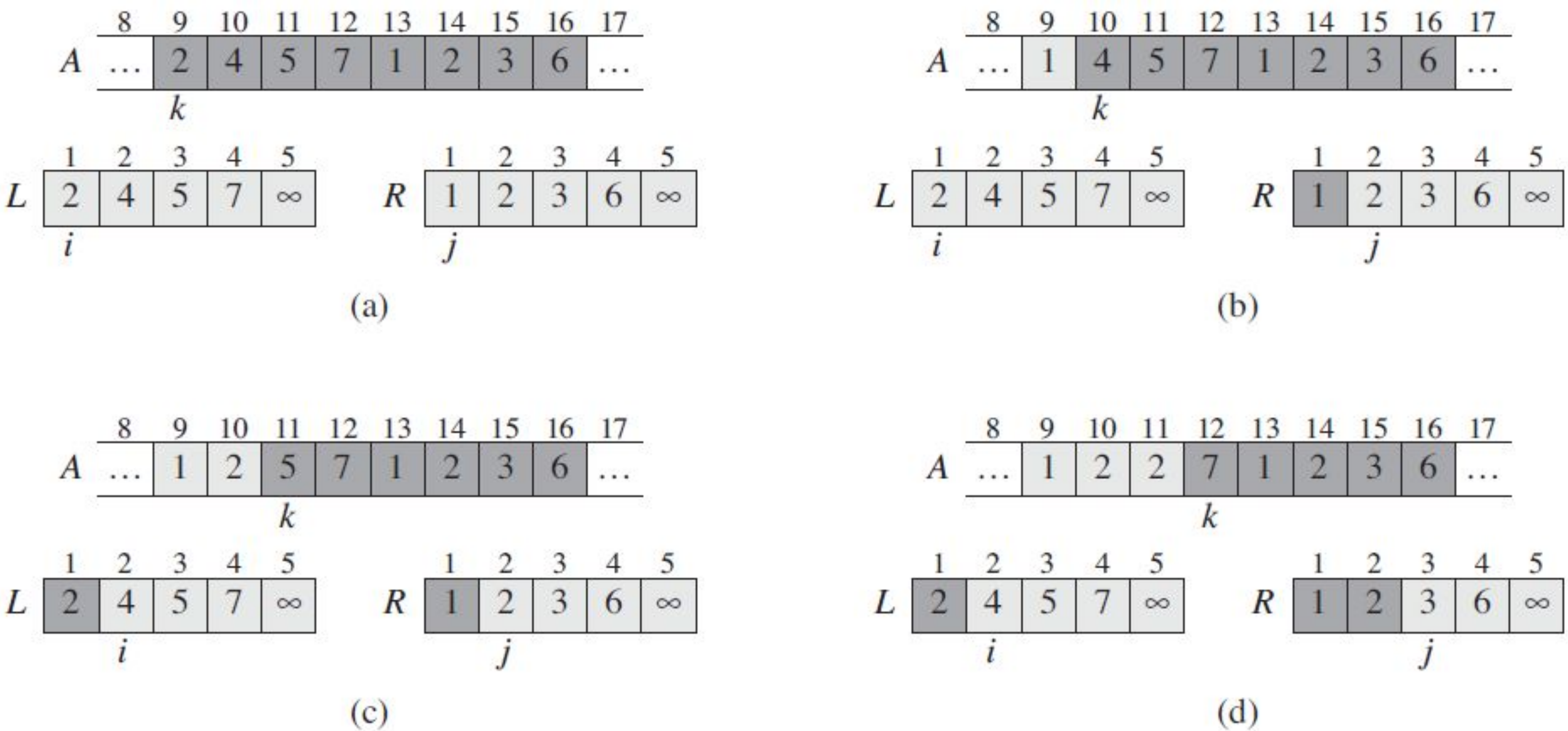
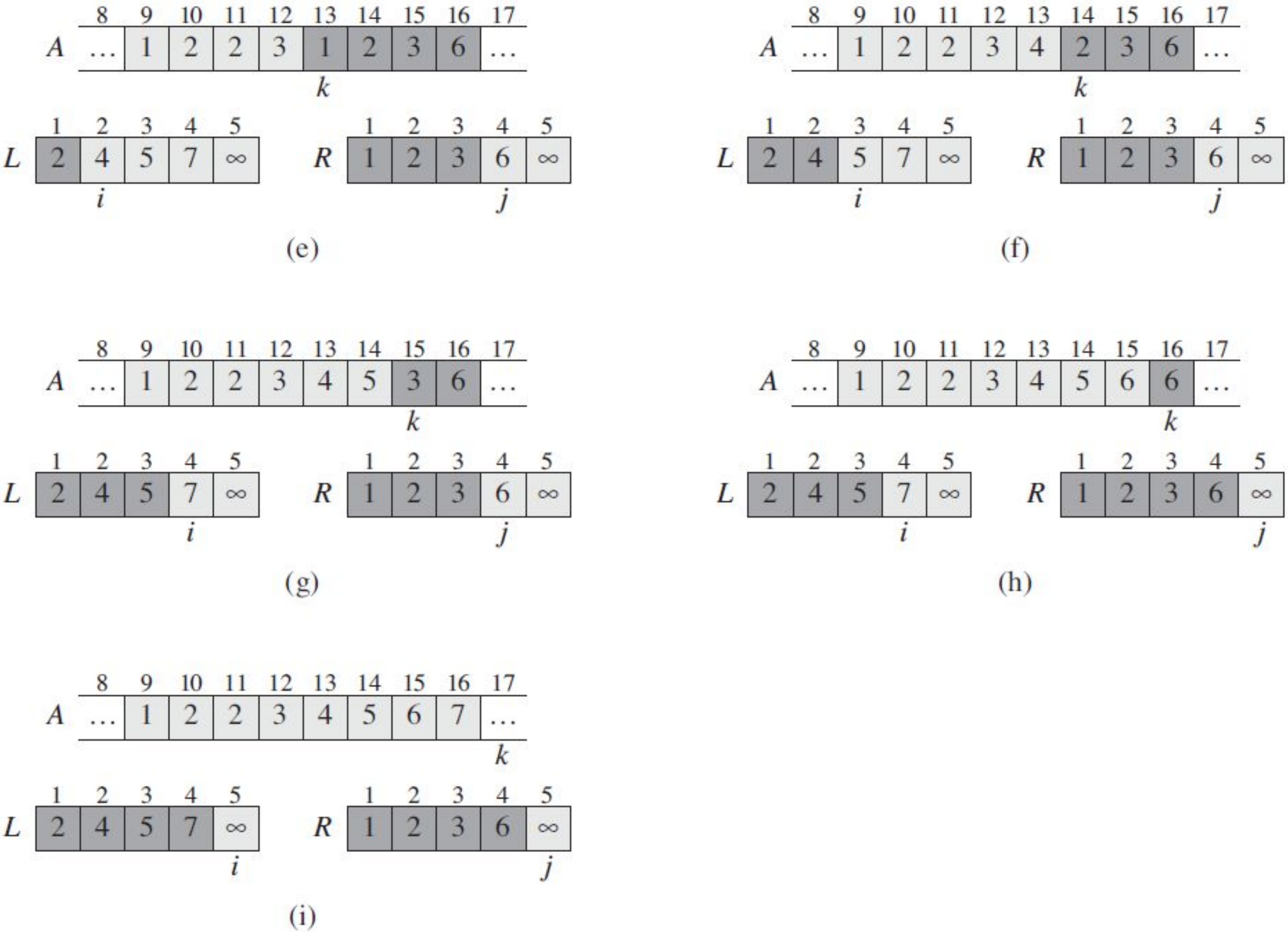


Ilustração do procedimento Merge (2)



Análise do procedimento Merge

```
MERGE(A, p, q, r)
1  n1 = q - p + 1
2  n2 = r - q
3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4  for i = 1 to n1
5          L[i] = A[p + i - 1]
6  for j = 1 to n2
7          R[j] = A[q + j]
8  L[n1 + 1] = ∞
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13         if L[i] ≤ R[j]
14                 A[k] = L[i]
15                 i = i + 1
16         else A[k] = R[j]
17                 j = j + 1
```

- Linhas 1..3, 8..11
  - Tempo constante
- Loops linhas 4..7
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Loop linhas 12..17
  - n iterações, cada uma de tempo constante
- Logo, o procedimento Merge é executado em $\Theta(n)$.

Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Análise do Merge Sort

- **Dividir**
  - Apenas calcula o meio do subarranjo. Então $D(n) = \Theta(1)$
- **Conquistar**
  - Resolve recursivamente para cada um dos dois subproblemas de tamanhos $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$ até chegar ao caso base
- **Combinar**

$$T(n) = \begin{cases} \Theta(1) & \text{Se } n = 1, \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n) & \text{se } n > 1 \end{cases}$$

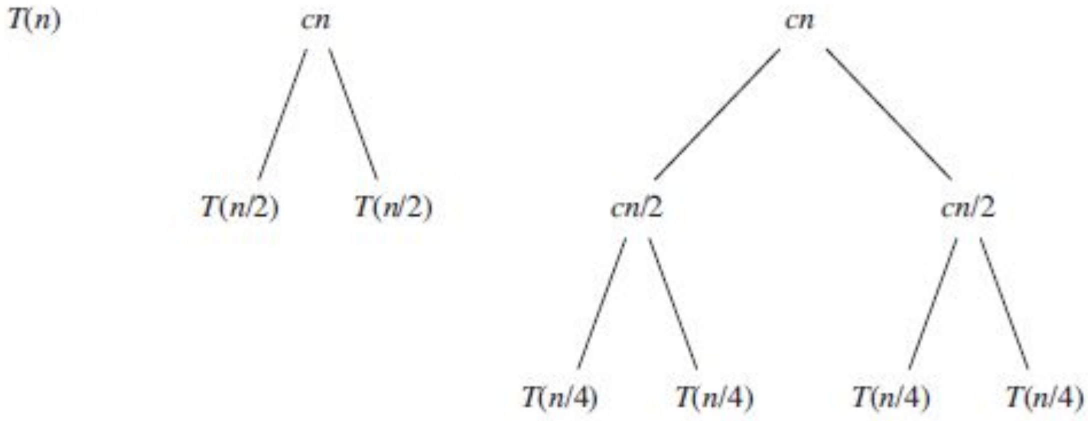
Análise do Merge Sort

- Vamos assumir n como sendo uma potência de 2.
  - Cada etapa de divisão cria duas subsequências de tamanho $n/2$.

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

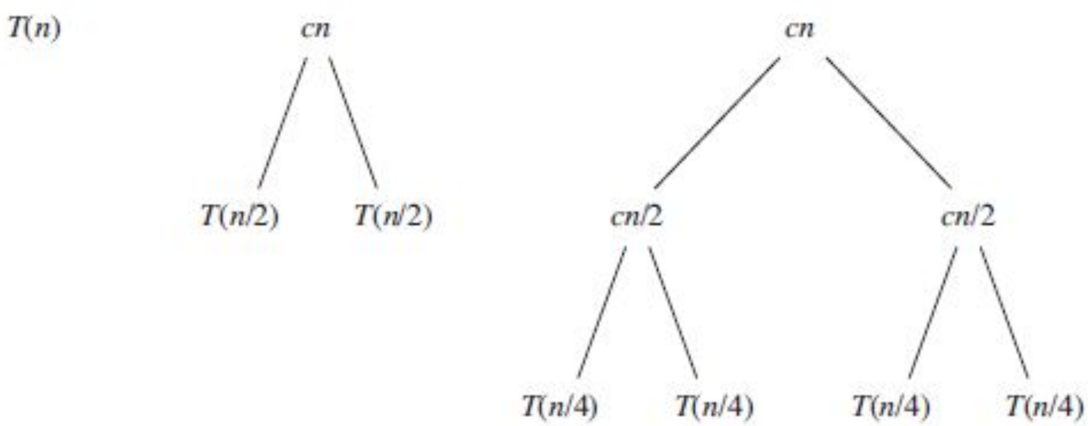
- Condição limite
  - O caso base pode ser resolvido em tempo constante na maioria dos casos. Portanto, em geral, $T(n) = \Theta(1)$, para n suficientemente pequeno.
  - Por conveniência, podemos, em geral, omitir a condição limite das recorrências. No caso do Merge sort, podemos descrever a recorrência simplesmente como

$$T(n) = 2T(n/2) + \Theta(n)$$

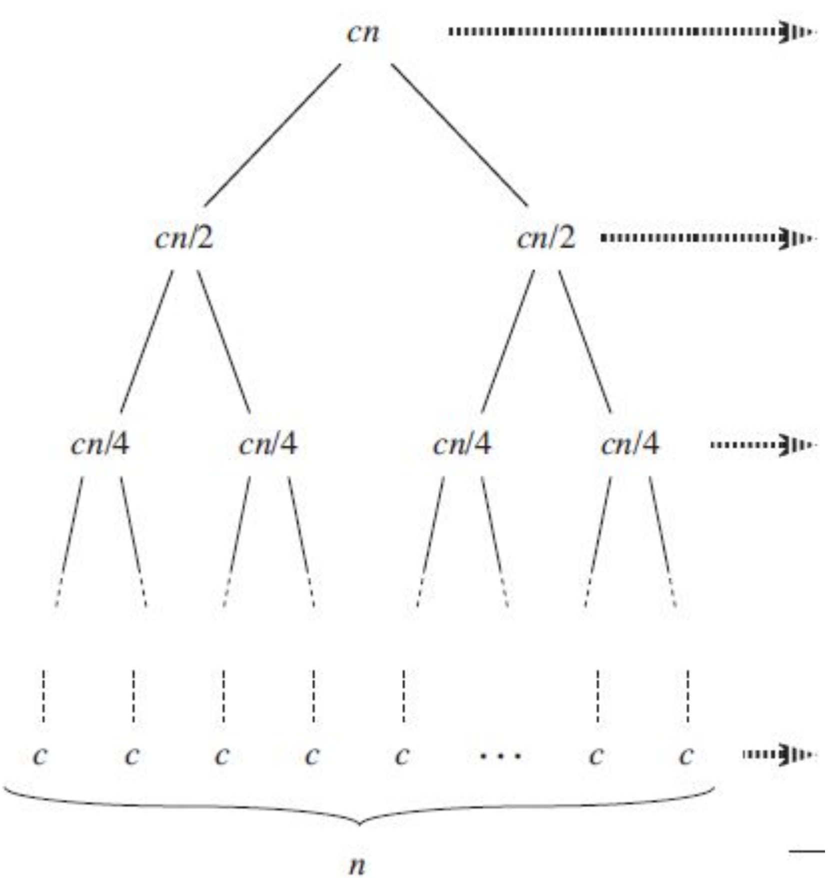


(a) (b) (c)

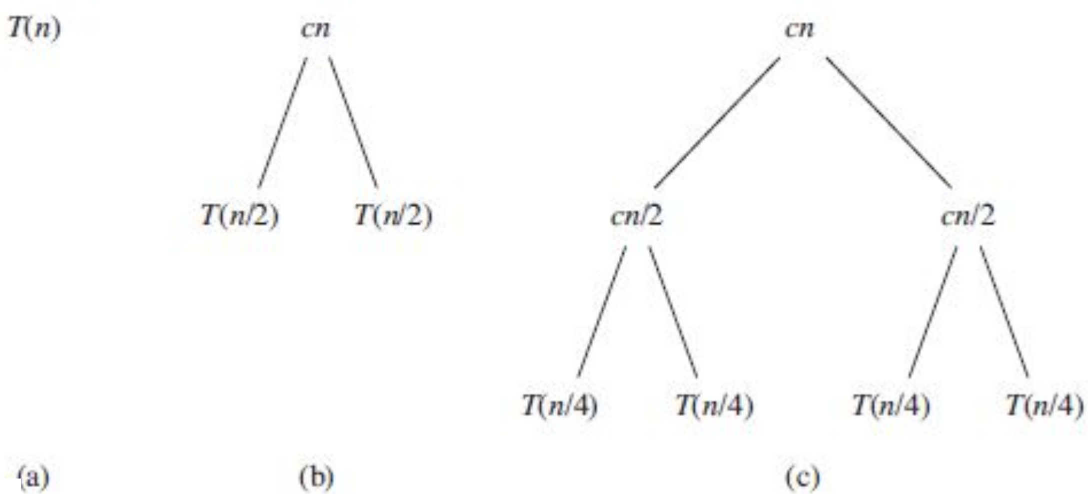
$$T(n) = \begin{cases} \Theta(1) & \text{Se } n = 1, \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

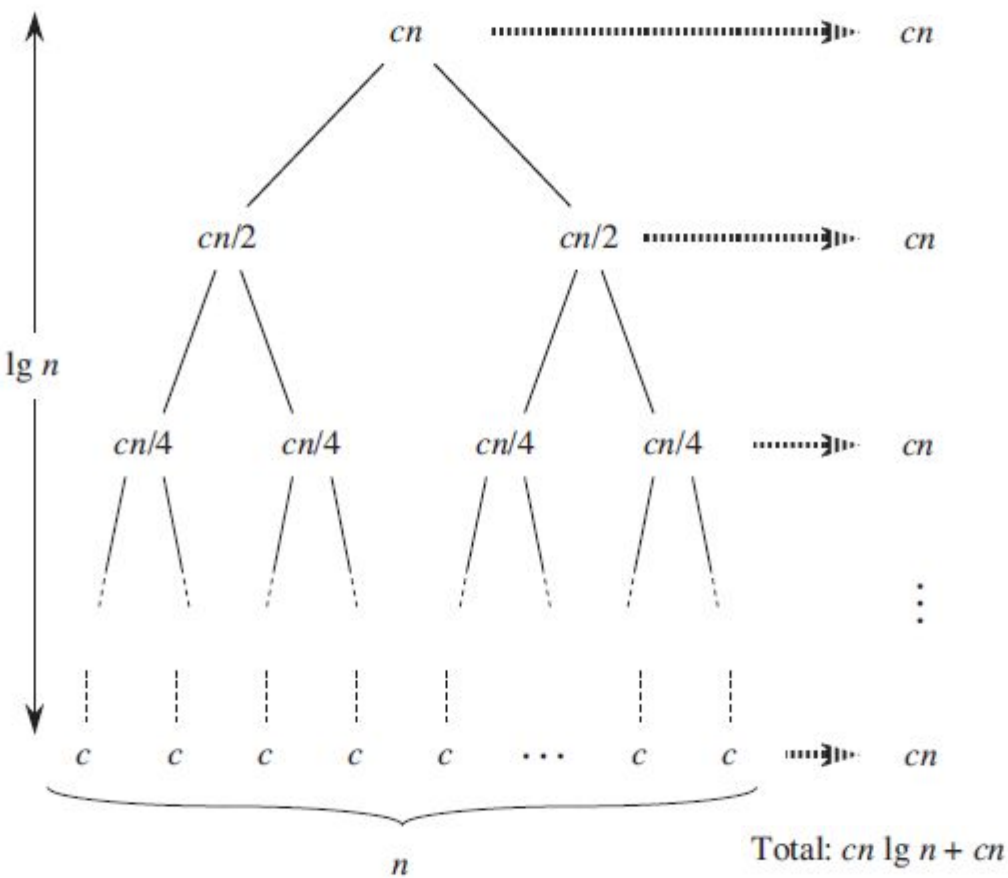


(a) (b) (c)



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

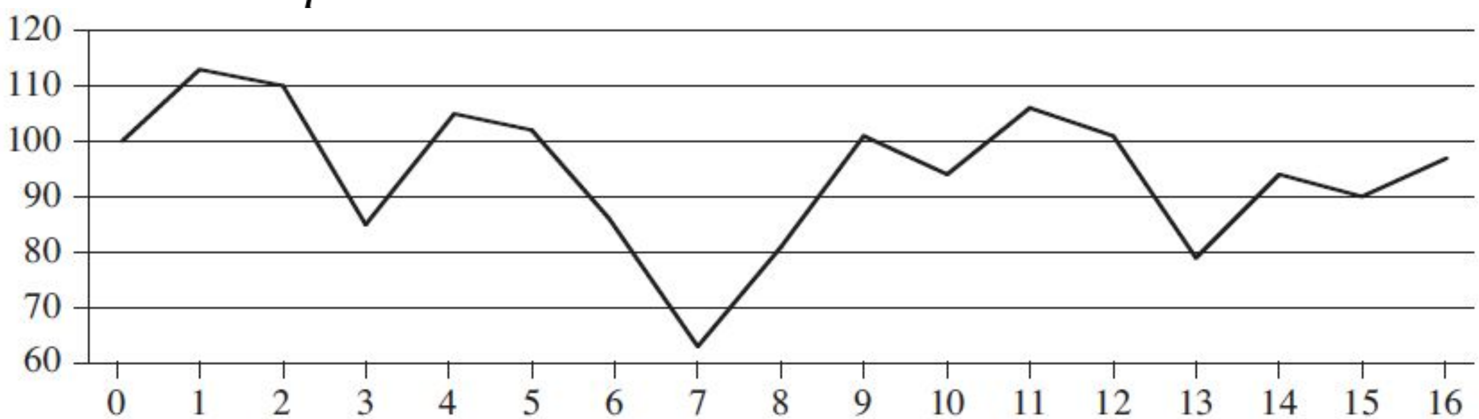




$$T(n) = \begin{cases} \Theta(1) & \text{Se } n = 1, \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

Maximizando o lucro

- Encontrar o dia de compra e de venda de uma ação de forma a maximizar o lucro
- Entrada: Cotações diárias de uma dada ação
- Saída: d_{compra}, d_{venda}



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Força bruta

- Solução
  - Tentar todas os possíveis pares de possíveis datas de compra e venda
  - Para um período de n dias
    - $\binom{n}{2}$ pares de dias, ou seja, $\Theta(n^2)$
  - No melhor dos casos, podemos checar cada par de dias em tempo constante
    - Portanto, o tempo desta solução ficaria em $\Omega(n^2)$

- Conseguimos encontrar solução em $o(n^2)$?

Problema do subarranjo máximo

- Ao invés de considerar as cotações diárias, utilizaremos as variações diárias das cotações da ação.

- Problema: Encontrar um subarranjo contíguo cujos valores possuem a soma máxima entre todos os possíveis subarranjos.
- Entrada: Arranjo de n números inteiros
- Saída: Índices de início e fim do subarranjo máximo

Solução

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

- Força bruta?
  - Solução por força bruta continua pelo menos quadrática, pois a quantidade de subarranjos é $\Theta(n^2)$.
- Divisão e conquista?
  - Como dividir o problema em subproblemas menores e resolvê-los recursivamente?

Divisão e conquista

- Para um arranjo $A[low..high]$, podemos encontrar um ponto intermediário mid que divida o arranjo em 2
- Neste caso, a solução $A[i..j]$ para este subarranjo poderá estar em um dos seguintes subarranjos

- 1) Inteiramente em $A[low..mid]$, onde $low \leq i \leq j \leq mid$
- 2) Inteiramente em $A[mid+1..high]$, onde $mid < i \leq j \leq high$
- 3) Cruzando o ponto mid, onde $low \leq i \leq mid < j \leq high$

Divisão e conquista

- Os casos 1 e 2, por serem instâncias menores do problema, podem ser resolvidos recursivamente
- O caso 3 precisa ser resolvido separadamente
  - Encontrar um subarranjo máximo que cruze o ponto central
  - Problema do mesmo tamanho do problema inicial, em quanto tempo podemos resolver isso?

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

- Número de iterações

$$(mid - low + 1) + (high - mid) = high - low + 1$$

$$= n$$

- Portanto, o procedimento tem tempo $\Theta(n)$

Solução

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])            // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Análise

- Caso base, n=1
  - $T(n) = \Theta(1)$
- Divisão do problema em 2 subproblemas de tamanhos aproximadamente n/2
- Tempo gasto no procedimento FIND-MAX-CROSSING-SUBARRAY é $\Theta(n)$
- Combinar (linhas 7-11) é executado em tempo constante, assim como as primeiras linhas
- Portanto, $T(n) = \Theta(1) + 2T(n/2) + \Theta(n)$

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

Método da Substituição

- Passos:
  1)Pressupor a forma da solução
  2)Usar indução matemática para encontrar constantes e mostrar que a solução funciona

- É um método eficiente, mas só pode ser usado quando é fácil pressupor a forma da solução

Exemplo

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

1) Chute: $O(n \log n)$

2) Provar que $T(n) \leq cn \log n$ para uma escolha apropriada de uma constante $c > 0$

  - Assumir que o limite vale para todo positivo $m < n$
    - $m = \lfloor n/2 \rfloor$

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$$

Exemplo

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

1) Chute: $O(n \log n)$

2) Provar que $T(n) \leq cn \log n$ para uma escolha apropriada de uma constante $c > 0$

- Assumir que o limite vale para todo positivo $m < n$
  - $m = \lfloor n/2 \rfloor$

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$$

$T(n) = 2T(\lfloor n/2 \rfloor) + n$ (por definição)

$T(n) \leq 2c \left\lfloor \frac{n}{2} \right\rfloor \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$ (por hipótese)

$$\leq cn \log\left(\frac{n}{2}\right) + n$$

$$= cn \log n - cn \log 2 + n$$

$$= cn \log n - cn + n$$

$$= cn \log n - (cn - n)$$

$\leq cn \log n$ para $c \geq 1$

Exemplo

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

- Caso base:
  - $T(1) \leq c\ 1\ \log 1 = 0$
  - Se houver condição limite $T(1) = 1$, basta encontrar n_0 tal que a relação é válida para $n \geq n_0$ (notação assintótica).
    - Seja $n_0 = 2$, substituir $T(1)$ por $T(2)$ e $T(3)$ como caso base.
    - $T(2) = 2T(1) + 2 = 4$
    - $T(3) = 2T(1) + 3 = 5$

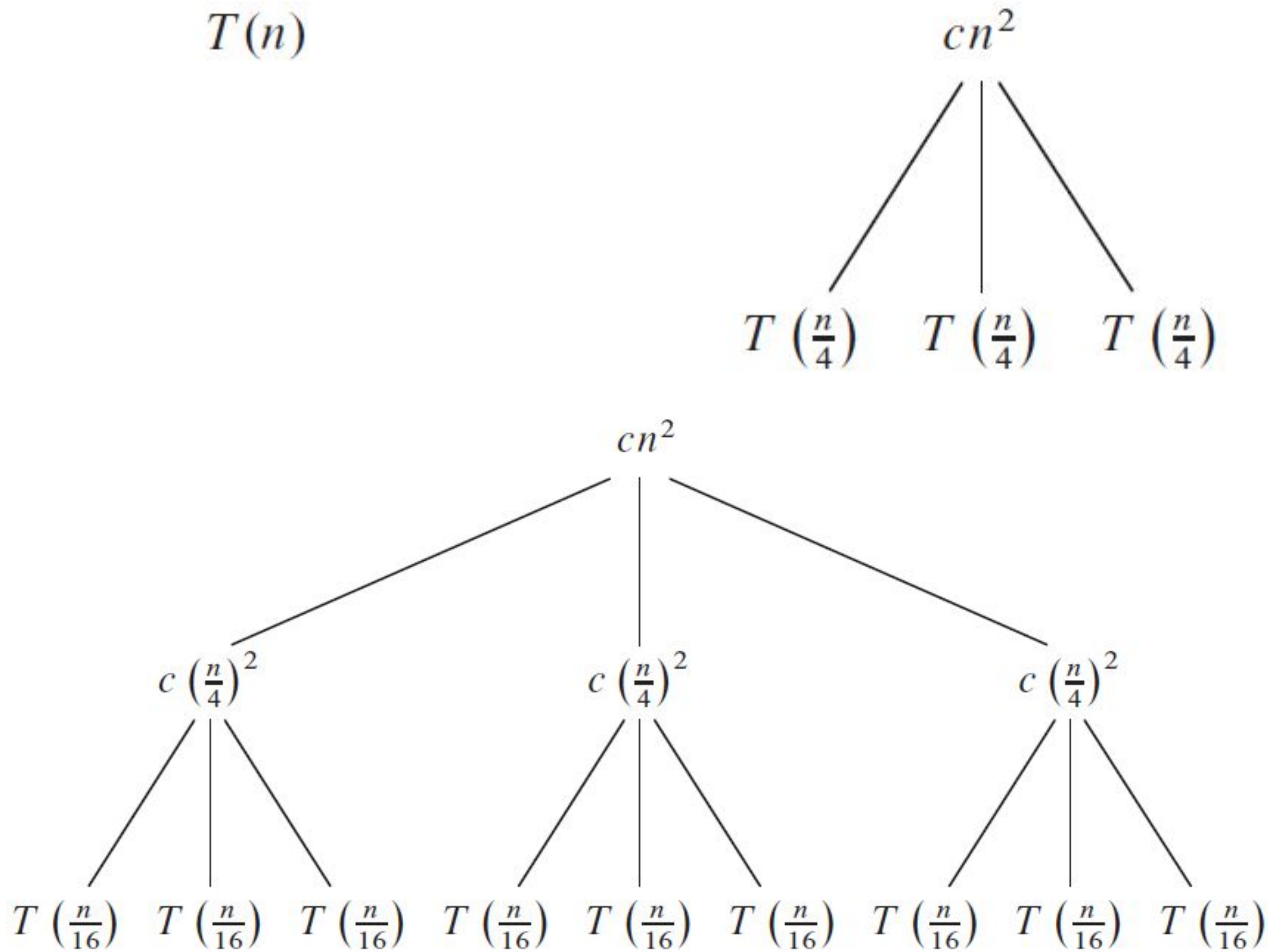
    - $T(2) \leq c\ 2\ \log 2$ e $T(3) \leq c\ 3\ \log 3$
      - Para $c \geq 2$, os casos bases $T(2)$ e $T(3)$ são válidos

Método de Árvore de Recursão

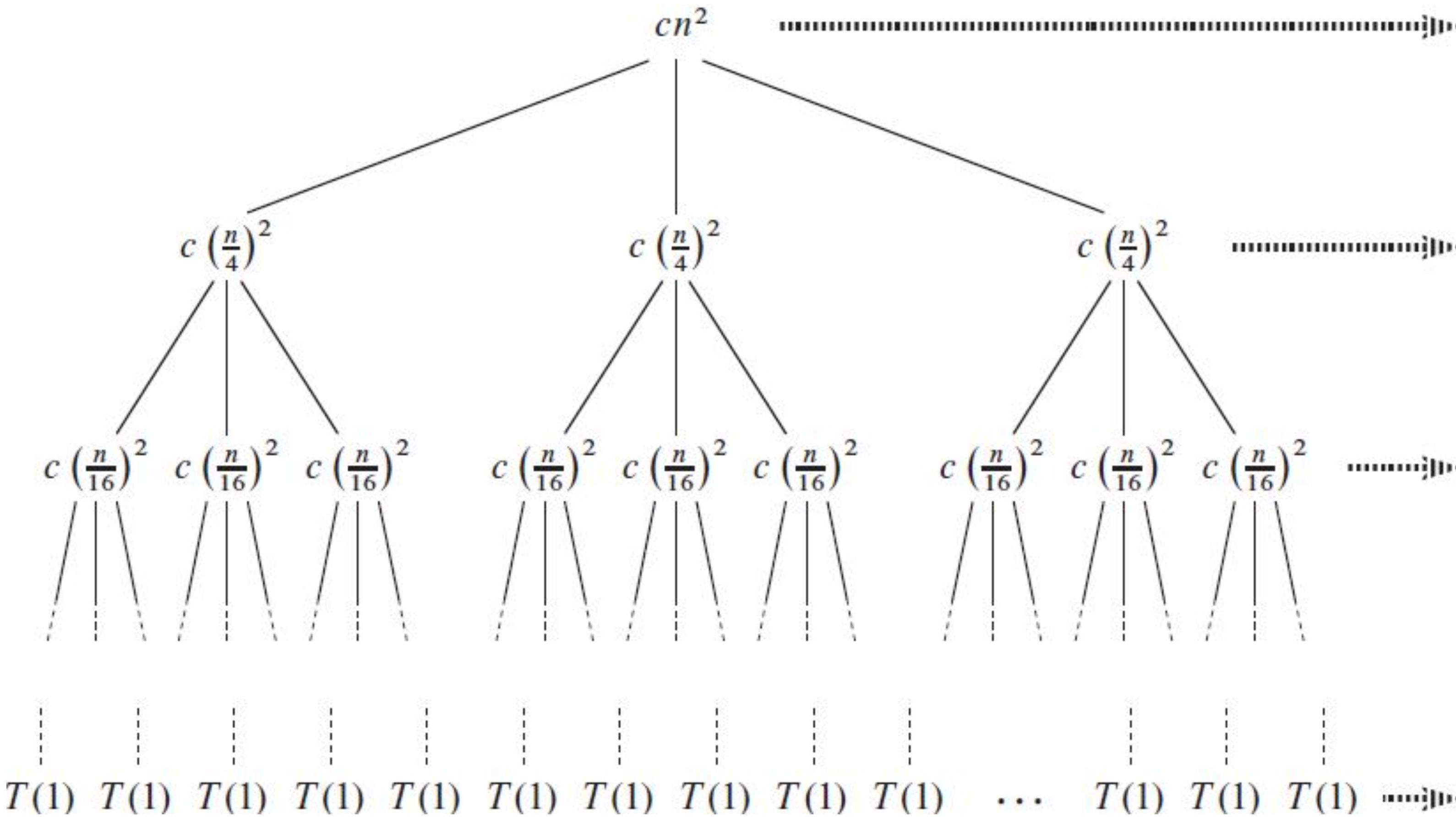
- Método mais intuitivo
  - No método da substituição, podemos ter problemas para obter uma boa pressuposição para a solução
  - Pode ser utilizado para encontrar um bom chute para o método da substituição
- Árvore de Recursão
  - Cada nó representa o custo de um subproblema
  - Cada nível *i* contém todos os subproblemas de profundidade *i*
  - Aspectos importantes
    - Altura da árvore
    - Número de passos executados em cada nível
  - Solução da recorrência
    - Soma de todos os passos de todos os níveis

Exemplo

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

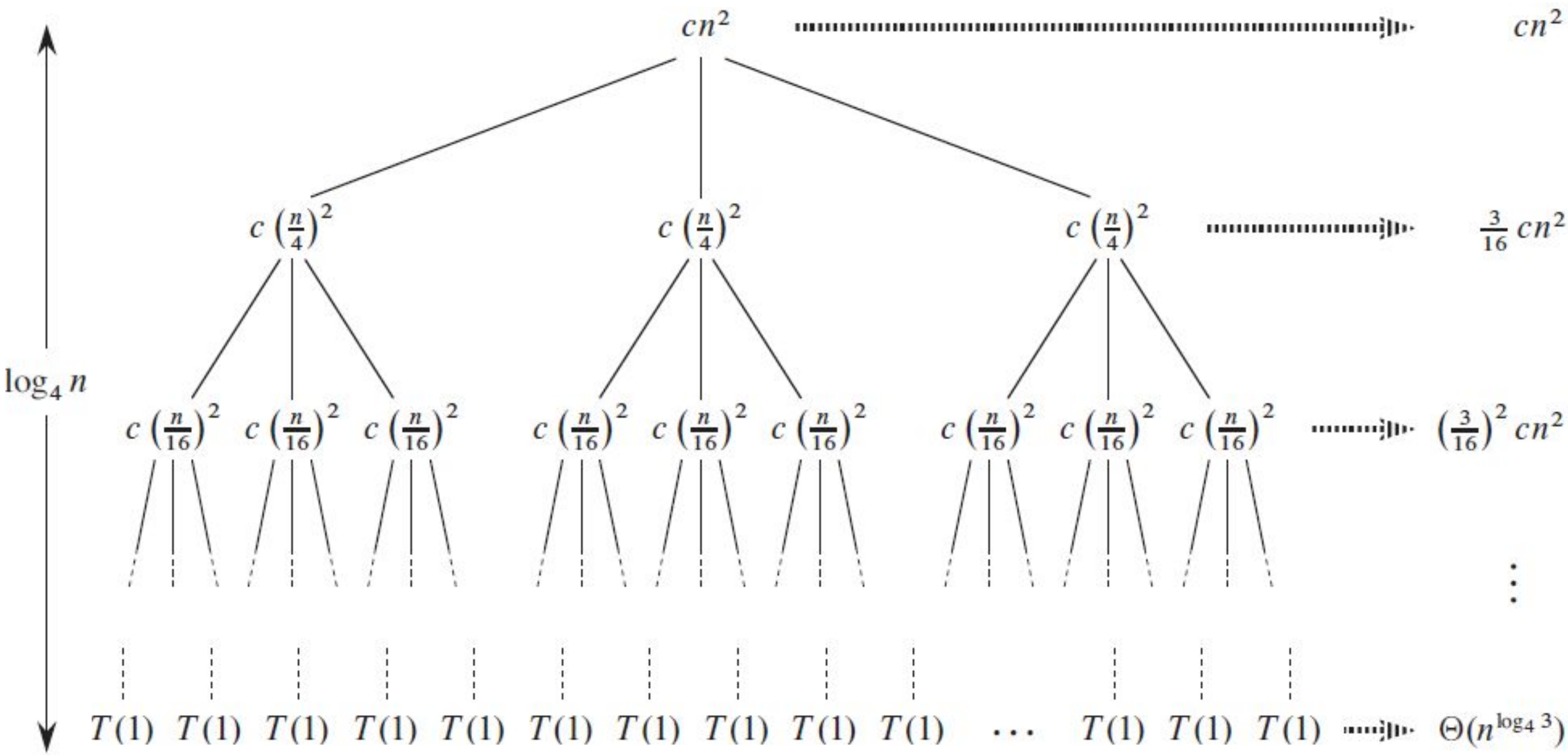


$$T(n) = 3T(n/4) + cn^2$$



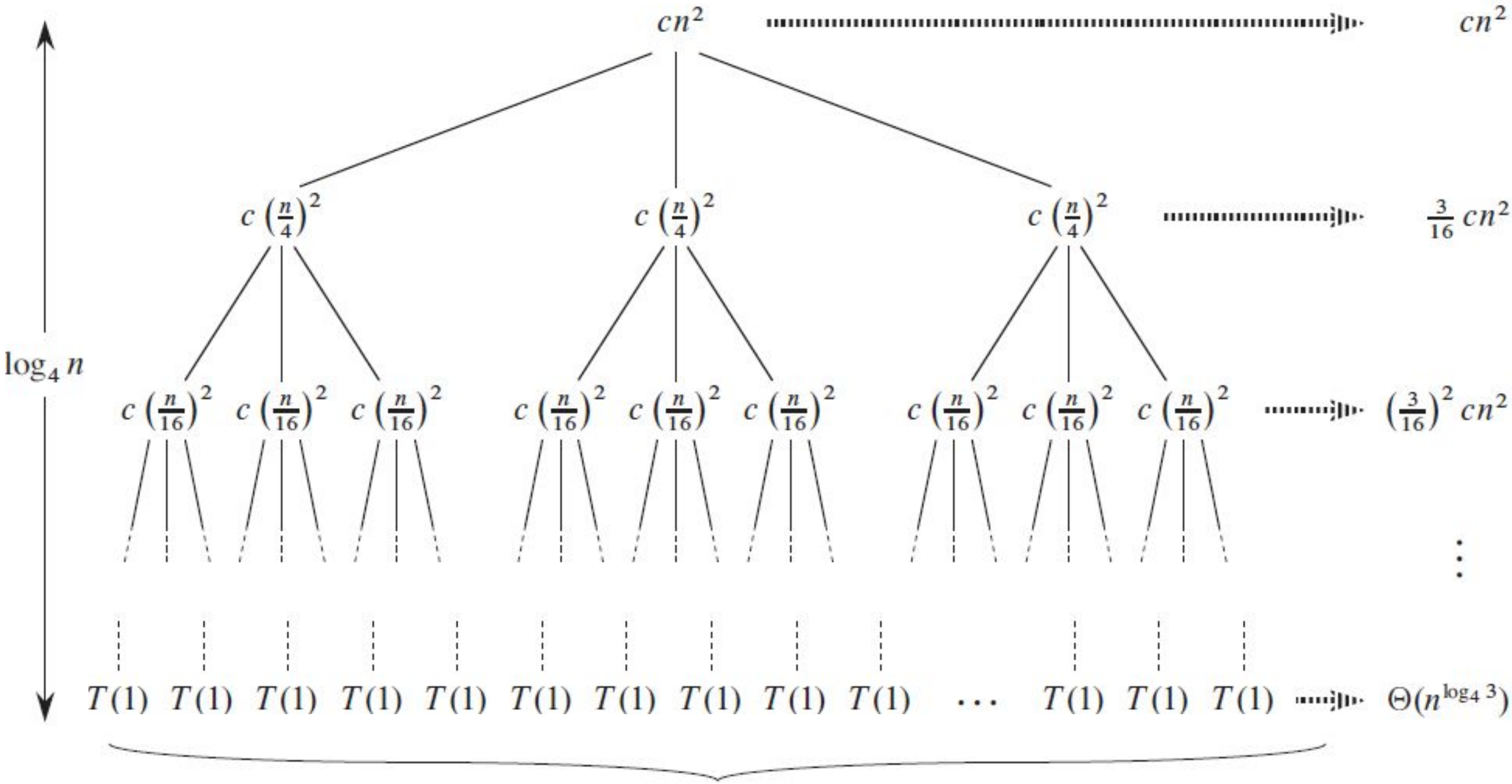
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$
$T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $\cdots$ $T(1)$ $T(1)$ $T(1)$

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



cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$T(1)$
$\log_4 n$
cn^2
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$

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



$n^{\log_4 3} = 3^{\log_4 n}$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})
\end{aligned}
$$

Série geométrica

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$$

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})
\end{aligned}
$$

Série geométrica

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$$

$$
\begin{aligned}
T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) .
\end{aligned}
$$

Teorema Mestre

- Fornece um processo de “livro de receitas” para resolver recorrências da forma:
  - $T(n) = aT(n/b) + f(n)$
  - Onde $a \geq 1$ e $b > 1$ 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, cada um de tamanho n/b
  - Resolve cada subproblema recursivamente
  - Combina as soluções dos subproblemas em uma solução do problema original

Teorema Mestre

- Sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida sobre os inteiros não negativos como:
  - $T(n) = aT(n/b) + f(n)$
  - Onde n/b pode ser $\lceil n/b \rceil$ ou $\lfloor n/b \rfloor$.

Teorema Mestre

- Sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida sobre os inteiros não negativos como:
  - $T(n) = aT(n/b) + f(n)$
  - Onde n/b pode ser $\lceil n/b \rceil$ ou $\lfloor n/b \rfloor$.
- Então, $T(n)$ pode ser limitada assintoticamente como a seguir:

  1) Se $f(n) = O(n^{\log_b a - \varepsilon})$ para algum $\varepsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$.

  2) Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$.

  3) Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ para algum $\varepsilon > 0$, e se $af\left(\frac{n}{b}\right) \leq cf(n)$ para algum $c < 1$ e para todo n suficientemente grande, então $T(n) = \Theta(f(n))$.

Teorema Mestre

- Sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida sobre os inteiros não negativos como:
  - $T(n) = aT(n/b) + f(n)$
  - Onde n/b pode ser $\lceil n/b \rceil$ ou $\lfloor n/b \rfloor$.
- Então, $T(n)$ pode ser limitada assintoticamente como a seguir:

  1) Se $f(n) = O(n^{\log_b a - \varepsilon})$ para algum $\varepsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$.

  2) Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$.

  3) Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ para algum $\varepsilon > 0$, e se $af\left(\frac{n}{b}\right) \leq cf(n)$ para algum $c < 1$ e para todo n suficientemente grande, então $T(n) = \Theta(f(n))$.

- Em cada um dos casos, comparamos $f(n)$ com a função $n^{\log_b a}$. Intuitivamente, a solução da recorrência é determinada pela maior das funções.

  $n^{\log_b a}$ “>” $f(n) \Rightarrow T(n) = \Theta(n^{\log_b a})$

  $n^{\log_b a}$ “=“ $f(n) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$

  $n^{\log_b a}$ “<“ $f(n) \Rightarrow T(n) = \Theta(f(n))$

Teorema Mestre

- Quando o teorema mestre **não** pode ser utilizado
  - Lacuna entre os casos 1 e 2
    - $f(n)$ é menor que $n^{\log_b a}$, mas não polinomialmente menor
  - Lacuna entre os casos 2 e 3
    - $f(n)$ é maior que $n^{\log_b a}$, mas não polinomialmente maior
    - Condição de regularidade no caso 3 não for válida

Exemplo 1

- $T(n) = 9T(n/3) + n$

$$a = 9$$
$$b = 3$$
$$f(n) = n$$

- Logo, $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$.
- Como $f(n) = O(n^{\log_3 9 - \varepsilon})$, onde $\varepsilon = 1$, podemos aplicar o caso 1 do teorema mestre e concluir que:

$$T(n) = \Theta\left(n^{\log_3 9}\right) = \Theta(n^2).$$

Exemplo 2

- $T(n) = T(2n/3) + 1$

$$a = 1$$
$$b = 3/2$$
$$f(n) = 1$$

- Logo, $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.
- Como $f(n) = \Theta\left(n^{\log_{3/2} 1}\right) = 1$, então aplica-se o caso 2 e portanto:
  - $T(n) = \Theta(\log n)$.

Exemplo 3

- $T(n) = 3T(n/4) + n \log n$

$$a = 3$$
$$b = 4$$
$$f(n) = n \log n$$

- Verificamos que $n^{\log_b a} = n^{\log_4 3} = n^{0,793}$.
- Como $f(n) = \Omega\left(n^{\log_4 3+\varepsilon}\right)$, onde ε ≈ 0,2, então aplica-se o caso 3 se $af\left(\frac{n}{b}\right) \leq cf(n)$

$$af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right)\log(n/4) \leq \left(\frac{3}{4}\right)n \log n = cf(n)$$

para $c = 3/4$ e n suficientemente grande. Logo,

$$T(n) = \Theta(\,n\log n).$$

Exemplo 4

- $T(n) = 2T(n/2) + n \log n$

$$a = 2$$
$$b = 2$$
$$f(n) = n \log n$$

- Verificamos que $n^{\log_b a} = n$.
- Como $f(n) = n \log n$ é assintoticamente maior do que $n^{\log_b a} = n$, o caso 3 parece se aplicar, porém

  $\frac{f(n)}{n^{\log_b a}} = \frac{n \log n}{n} = \log n.$

  Logo, $f(n)$ não é *polinomialmente* maior do que n, pois $\log n$ é assintoticamente menor do que n^{ε} para qualquer constante positiva ε. Consequentemente, a recorrência recai na lacuna entre os casos 2 e 3.

Exercício

1) Mostre pelo método de substituição que

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

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

$$T(n) = T(n/2) + 1 = O(\log n)$$

Exercício

2) Encontre um bom limite superior assintótico para a recorrência utilizando o método de árvore de recursão :

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

Exercícios

3) Resolva as recorrências

a) $T(n) = 2T(n/2) + n$

b) $T(n) = 4T(n/2) + n$

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

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

Exercício

4) Seja o seguinte algoritmo que calcula o valor de um inteiro positivo x elevado a um inteiro positivo n.

a) Determine a recorrência que descreve a quantidade de multiplicações efetuadas.

b) Encontre limites assintóticos justos inferior e superior para o algoritmo.

```
int potencia(int x, int n){
   double y;
   if(n==0) return 1;
   y = potencia(x,n-1)*x;
   return y;
}
```

Exercício

5) Seja A[1..n] um vetor com n números distintos. Se i < j e A[i] > A[j], então o par (i,j) é chamado de uma inversão de A. Projete um algoritmo por divisão e conquista que determina o número de inversões em qualquer permutação de n elementos em tempo $\Theta(n \log n)$ no pior caso. Qual recorrência descreve o número de comparações no pior caso desse algoritmo? Mostre que essa recorrência possui complexidade $\Theta(n \log n)$.

Referências

- CLRS, Introduction to Algorithms, 3rd ed.
  - 2.3, 4, 4.1, 4.3, 4.4, 4.5