PANC: Projeto e Análise de Algoritmos

Aula 04: Paradigma de Divisão e Conquista e Ordenação utilizando MergeSort (Lógica e Complexidade)

Breno Lisi Romano

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Sumário

- Revisão de Conteúdo
- Divisão e Conquista
- Ordenação por Intercalação (*Merge Sort*)
  - Proposta de Divisão e Conquista Adotada
  - Lógica
  - Exemplo Prático
  - Pseudocódigo
  - Implementação
- Análise da Complexidade do *Merge Sort*

Recapitulando...

- **T(n)** é a função de complexidade que representa a **medida de custo da execução** de um **algoritmo** para uma instância de **tamanho n**:
  - A função de **complexidade de tempo T(n)** mede o tempo necessário para executar um algoritmo **(número de instruções) → Quantidade de Operações Executadas**

- **Ordenação por Inserção (*Insertion Sort*):**
  - Caracterizada pelo **princípio** no qual se **divide** o **array** em **dois segmentos**: um já **ordenado** e o outro **não ordenado**
    - O **progresso** se desenvolve em **n-1 interações**
    - Em cada interação: um **elemento do segmento não ordenado** é **transferido** para o **primeiro segmento**, e **inserido** na **posição correta** em relação aos demais elementos já existentes
  - **Análise da Complexidade – T(n):**
    - **Melhor Caso:** T(n) é Linear – O(n)
    - **Pior Caso:** T(n) é Quadrático – $O(n^2)$

Divisão e Conquista (1)

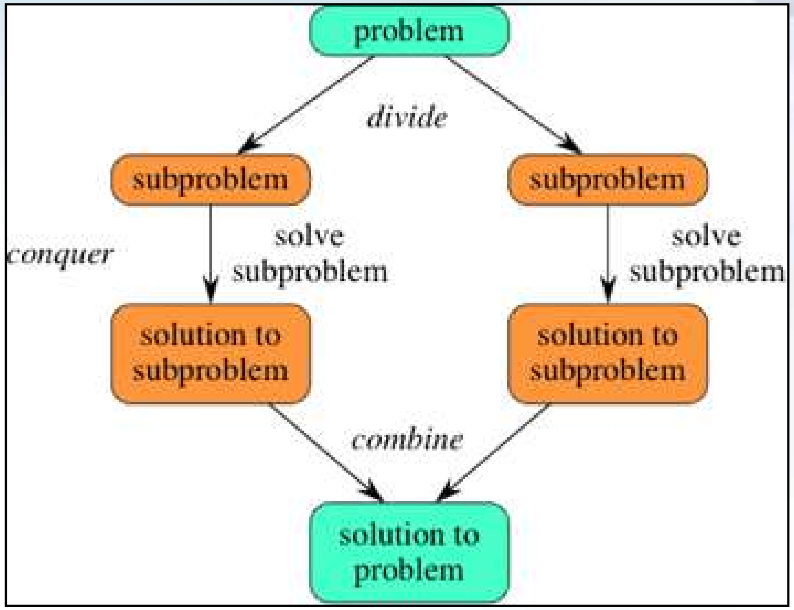
- *"Divide-and-Conquer is perhaps the most commonly used algorithm design technique in computer science.*
- *Faced with a big problem P, divide it into smaller subproblems, solve these sub-problems, and combine their solutions into a solution for P.*
- *But how do you solve the smaller problems?*
- *Simply divide each of the small problems into smaller problems, and keep doing this until the problems become so small that it is trivial to solve them.*
- *Sound like recursion? Not surprisingly, a recursive procedure is usually the easiest way of implementing divide-and-conquer"*
- **- Ian Parberry, *Problems on Algorithms***

Divisão e Conquista: Definição (2)

- Divisão e Conquista (ou Dividir e Conquistar, ou ainda, D&C) é um **paradigma de solução de problemas** no qual tentamos **simplificar a solução do problema** original **dividindo-o em subproblemas** menores e **resolvendo-os** (ou "conquistando-os") separadamente
- O processo:
  - **Dividir** o problema original em **subproblemas** – normalmente com a metade (ou algo próximo disto) do tamanho do problema original, porém com a mesma estrutura
  - **Conquistar**, ou determinar a solução dos subproblemas, comumente, de maneira recursiva – que agora se tornam mais "fáceis"
  - Se necessário, **combinar** as soluções dos subproblemas para produzir a **solução completa** para o problema original

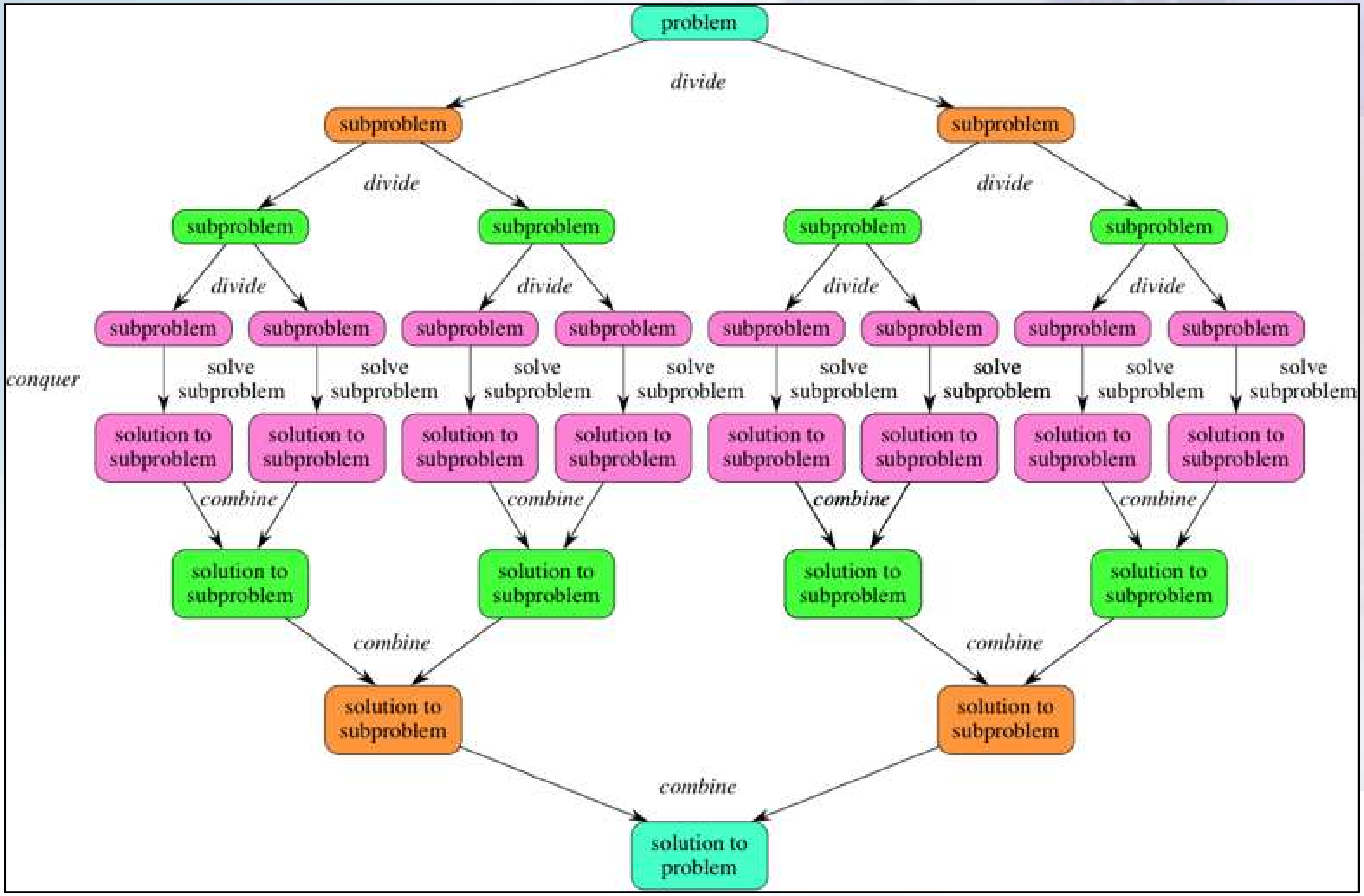
Divisão e Conquista: Definição (3)

- **Passos** de um algoritmo de Divisão e Conquista: **Dividir**, **Conquistar**, **Combinar**
- **Ilustração**: Visualizar um passo, assumindo que cada passo de dividir cria dois subproblemas
  - Alguns algoritmos de dividir-e-conquistar criam mais de dois



Divisão e Conquista: Definição (4)

- **Ilustração**: Se expandirmos mais duas etapas recursivas, ele terá esta aparência



Como se cria pelo menos dois subproblemas, um algoritmo de Divisão e Conquista faz múltiplas chamadas recursivas

Divisão e Conquista: Observação (5)

- Conforme descrito anteriormente, pode-se **resolver subproblemas de estrutura igual** de maneira recursiva
- Eventualmente, é necessário **resolver alguns subproblemas diferentes do problema original** em termos de estrutura
- Consideram-se estes **subproblemas** como **parte do processo de combinar** as soluções

Divisão e Conquista: Utilização (6)

- Existem **quatro condições** que indicam se o paradigma Divisão e Conquista pode ser aplicado com sucesso:
  - Deve ser possível **dividir o problema** em **subproblemas**
  - A **combinação** de **resultados** deve ser **eficiente**
  - Os **subproblemas** devem possuir **tamanhos parecidos** dentro de um mesmo nível
  - A **solução** dos **subproblemas** são **operações repetidas** ou correlacionadas

Divisão e Conquista: Solução Genérica (7)

D&CGenerico(x)
Entrada: (sub)problema x
se x *é o caso base* **então**
    retorna *resolve*(x);
senão
    Divida x em n subproblemas $x_0, x_1, \ldots, x_{n-1}$;
    para $i \leftarrow 0$ ***até*** $n - 1$ **faça**
        $y_i \leftarrow$ **D&CGenerico**(x_i);
    fim
    Combine $y_0, y_1, \ldots, y_{n-1}$ em y;
    retorna y;
fim

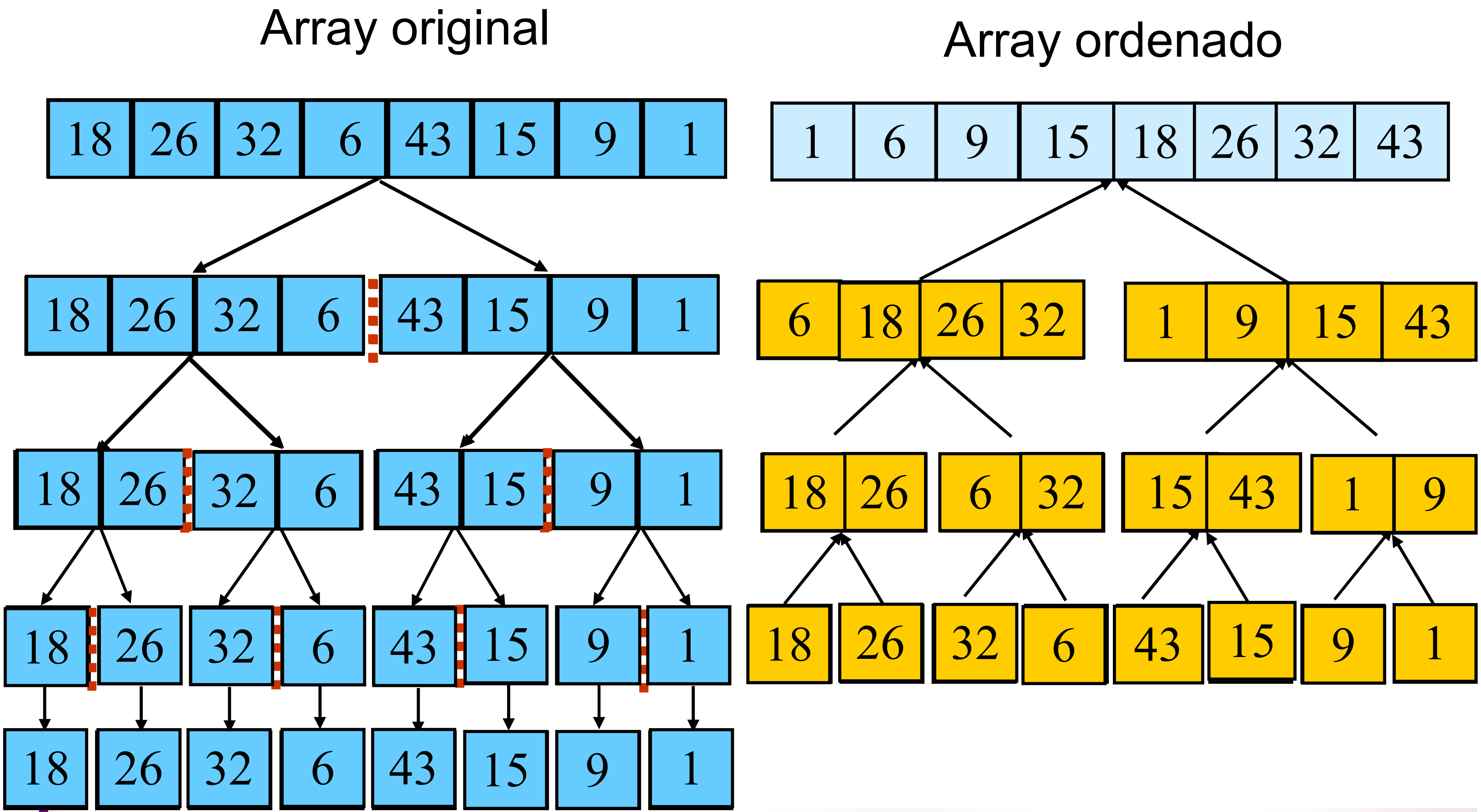
Aplicação do paradigma de Divisão e Consquisa.....

ORDENAÇÃO POR INTERCALAÇÃO (*MERGE SORT*)

Merge Sort: Divisão e Conquista

- *Mergesort* é um **algoritmo** para resolver o problema de **ordenação de arrays** e um exemplo clássico do **uso do paradigma de Divisão e Conquista** (*to merge* = intercalar)
  - Duas abordagens: **Top-Down (Recursiva)** e Bottom-Up (Iterativa)
- **Descrição** do ***Mergesort*** em alto nível (**Top-Down**):
  - **Divisão**: Divide a sequência de n elementos que deve ser ordenada em duas subsequências de n/2 elementos cada uma
    - **Condição de parada da Recursão**: quando for ordenar apenas um elemento, este caso será a sub-solução elementar
  - **Conquista**: Ordena as duas subsequências recursivamente, utilizando a ordenação por intercalação (Algoritmo Mergesort)
  - **Combinação**: Intercala as duas subsequências ordenadas para produzir a resposta ordenada (Algoritmo Intercala)

Merge Sort: Ilustração para um Array (n=8)



Merge Sort: Algoritmo

- ***Mergesort*:** O objetivo é rearranjar um array A[p...r], com p ≤ *r, em* ordem crescente

$\text{MERGESORT}(A, p, r)$

Divisão

se $p < r$

então $q \leftarrow \lfloor (p + r)/2 \rfloor$

Conquista

$\text{MERGESORT}(A, p, q)$

$\text{MERGESORT}(A, q + 1, r)$

Combinação

$\text{INTERCALA}(A, p, q, r)$

	p				q				r
A	66	33	55	44	99	11	77	22	88

Merge Sort: Ilustração do Algoritmo (1)

- ***Mergesort:*** O objetivo é rearranjar um array A[p...r], com p ≤ *r, em* ordem crescente

```
MERGESORT(A, p, r)
  se p < r
     então q ← ⌊(p + r)/2⌋
           MERGESORT(A, p, q)
           MERGESORT(A, q + 1, r)
           INTERCALA(A, p, q, r)
```

	p				q				r
A	33	44	55	66	99	11	77	22	88

Merge Sort: Ilustração do Algoritmo (2)

- ***Mergesort**:* O objetivo é rearranjar um array A[p...r], com p ≤ *r, em* ordem crescente

```
MERGESORT(A, p, r)
    se p < r
        então q ← ⌊(p + r)/2⌋
              MERGESORT(A, p, q)
              MERGESORT(A, q + 1, r)
              INTERCALA(A, p, q, r)
```

	p				q				r
A	33	44	55	66	99	11	22	77	88

Merge Sort: Ilustração do Algoritmo (3)

- ***Mergesort**:* O objetivo é rearranjar um array A[p...r], com p ≤ *r, em* ordem crescente

```
MERGESORT(A, p, r)
  se p < r
      então q ← ⌊(p + r)/2⌋
            MERGESORT(A, p, q)
            MERGESORT(A, q + 1, r)
            INTERCALA(A, p, q, r)
```

	p				q				r
A	11	22	33	44	55	66	77	88	99

Merge Sort: Algoritmo Intercala()

- O que significa intercalar dois (sub)arrays ordenados?
- **Problema:** Dados A[p...q] e A[q+1...r] crescentes, rearranjar A[p...r] de modo que ele fique em ordem crescente

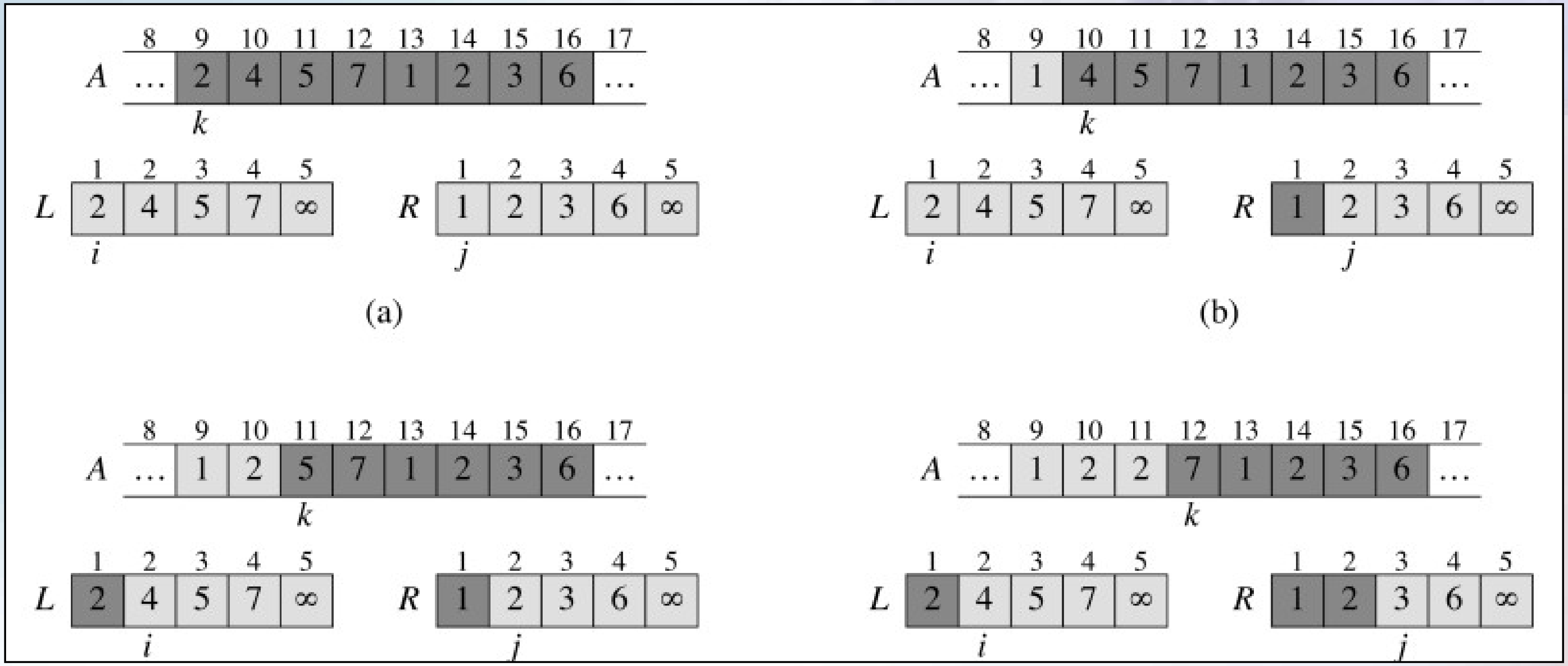
Entrada:

	p				*q*				*r*
A	22	33	55	77	99	11	44	66	88

Saída:

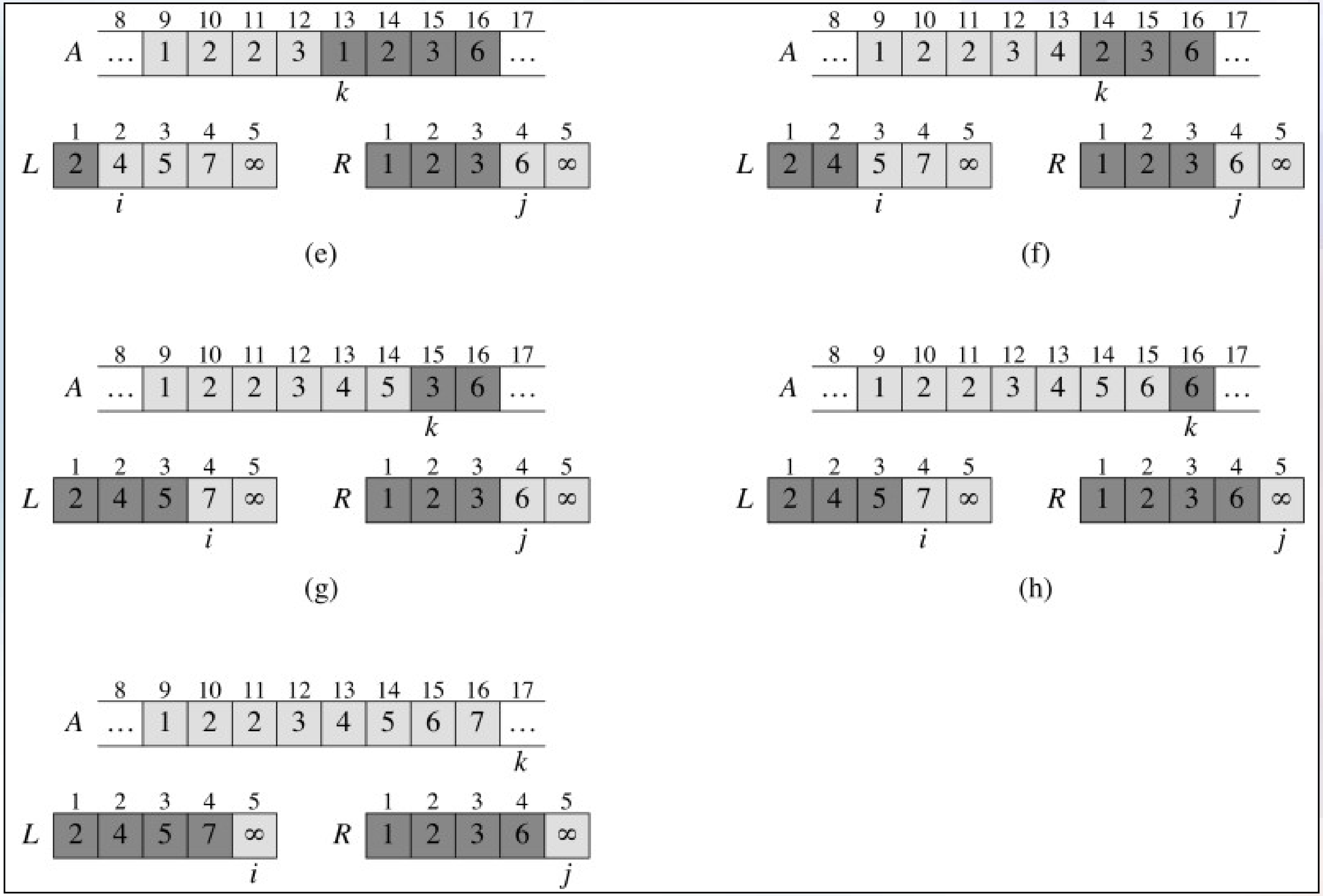
	p				*q*				*r*
A	11	22	33	44	55	66	77	88	99

Merge Sort: Algoritmo Intercala() – Com Sentinela (1)



Ref.: Algoritmos – Cormen (2012)

Merge Sort: Algoritmo Intercala() – Com Sentinela (2)



Ref.: Algoritmos – Cormen (2012)

Merge Sort: Algoritmo Intercala() – Com Sentinela (3)

```
INTERCALA(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
sejam L[1..n1 + 1] e R[1..n2 + 1] novos vetores
para i ← 1 até n1 faça
    L[i] ← A[p + i − 1]
para j ← 1 até n2 faça
    R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
para k ← p até r faça
    se L[i] ≤ R[j] então
        A[k] ← L[i]
        i ← i + 1
    senão
        A[k] = R[j]
        j ← j + 1
```

Merge Sort: Algoritmo Intercala() – Sem Sentinela (1)

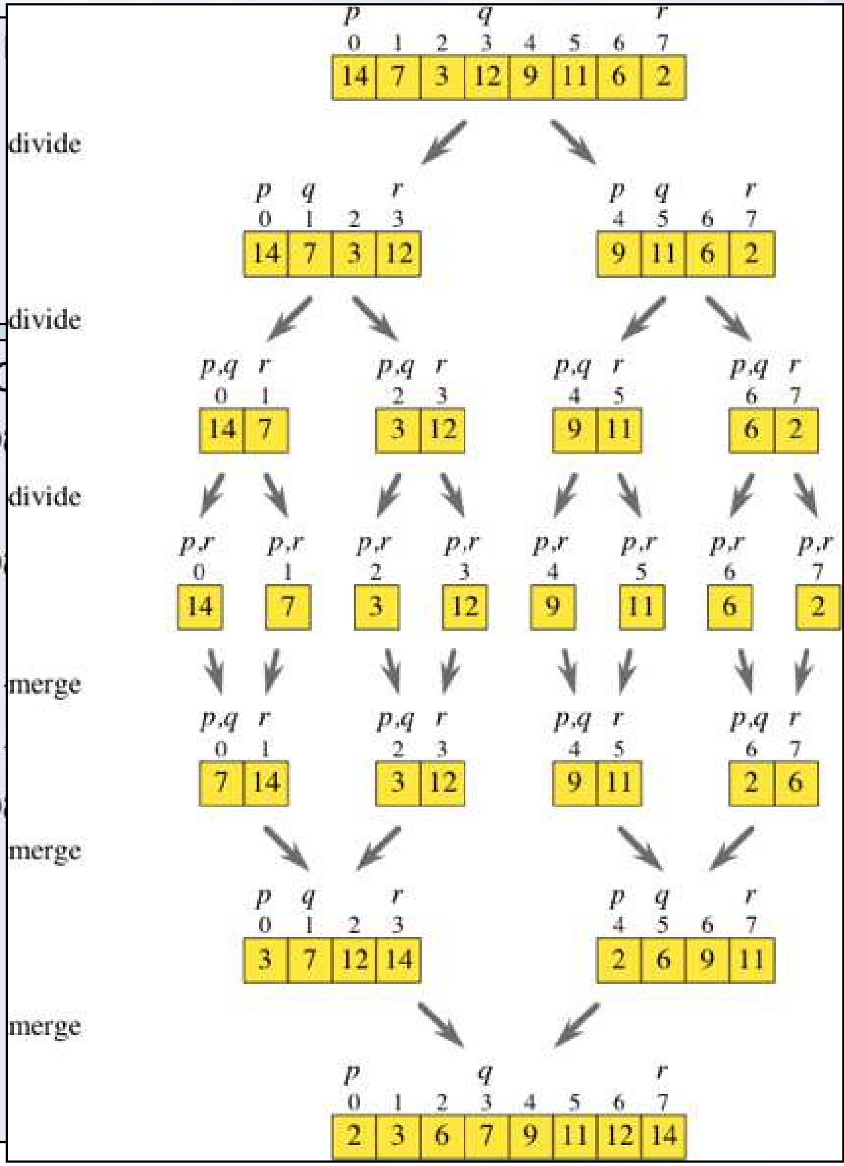
A	11	22	33	44	55	66	77	88	99
					j	*i*			
B	22	33	55	77	99	88	66	44	11

Merge Sort: Algoritmo Intercala() – Sem Sentinela (2)

```
INTERCALA(A, p, q, r)
    para i ← p até q faça
        B[i] ← A[i]
    para j ← q + 1 até r faça
        B[r + q + 1 − j] ← A[j]
    i ← p
    j ← r
    para k ← p até r faça
        se B[i] ≤ B[j]
            então A[k] ← B[i]
                  i ← i + 1
            senão A[k] ← B[j]
                  j ← j − 1
```

Merge Sort: Ilustração do Algoritmo Exemplo Completo

- **A[] = [14, 7, 3, 12, 9, 11, 6, 2]**



Merge Sort: Análise da Complexidade Algoritmo Intercala

- **Pior Caso – T(n)***:*

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

INTERCALA(A, p, q, r)

para $i \leftarrow p$ **até** q **faça**
    $B[i] \leftarrow A[i]$
para $j \leftarrow q+1$ **até** r **faça**
    $B[r+q+1-j] \leftarrow A[j]$
$i \leftarrow p$
$j \leftarrow r$
para $k \leftarrow p$ **até** r **faça**
    se $B[i] \leq B[j]$
        então $A[k] \leftarrow B[i]$
                $i \leftarrow i+1$
        senão $A[k] \leftarrow B[j]$
                $j \leftarrow j-1$

- **Tamanho da Entrada:** n = r – p + 1
- **Complexidade T(n):** O(n) → Linear

Merge Sort: Análise da Complexidade Algoritmo Mergesort (1)

- **Pior Caso – T(n):** Assumir que o Tamanho do Array é Par

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

Linha	Consumo de Tempo	
1	1	
2	1	
3	T(n/2)	$T(\lceil n/2 \rceil)$
4	T(n/2)	$T(\lfloor n/2 \rfloor)$
5	n: Complexidade do Intercala	
T(n)	**T(n/2) + T(n/2) + O(n) + 2**	

Se não fosse par!

Merge Sort: Análise da Complexidade Algoritmo Mergesort (2)

- **Fórmula de Recorrência** (ou seja, uma fórmula definida em termos de si mesma)**:**
  - **T(1) = O(1)** se n=1
  - **T(n) = T(n/2)+ T(n/2)+ O(n)** se n>1

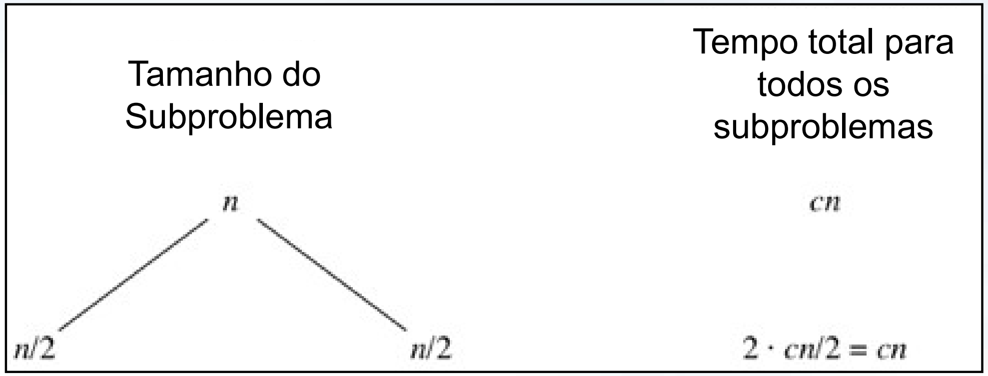
- Em geral, ao utilizar-se do **paradigma de Divisão e Conquista**, adota-se **recursividade** → **Complexidade** T(n) é uma **Fórmula de Recorrência**
  - Precisamos aprender a **resolver recorrência** → Encontrar uma **"Fórmula Fechada"** para T(n)
  - Veremos em outras aulas como resolver recorrência
  - Mas queremos calcular a Complexidade T(n) do *MergeSort*

Merge Sort: Análise da Complexidade Algoritmo Mergesort (3)

- **Fórmula de Recorrência:**
  - **T(1) = c** se n=1
  - **T(n) = 2.T(n/2) + c.n** se n>1

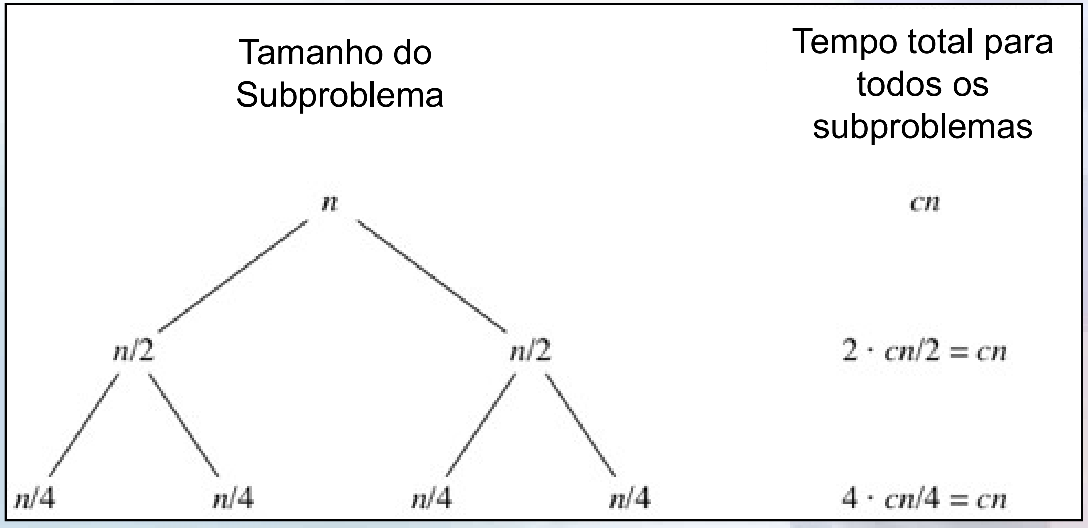
  onde c é um a constante para o tempo exigido em resolver problemas de tamanho 1, bem como o tempo por elemento do (sub)array para as etapas de dividir e combinar

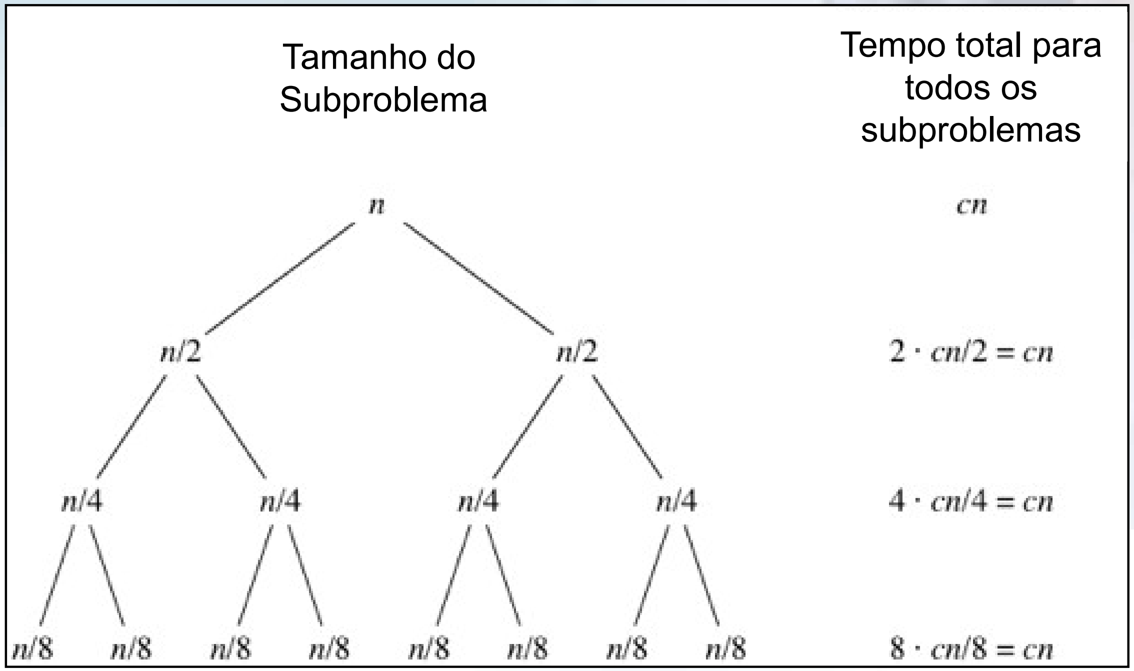
- Vamos resolver com uma "Árvore de Recorrência":



Merge Sort: Análise da Complexidade Algoritmo Mergesort (4)

- **Fórmula de Recorrência:**
  - **T(1) = c**
  - **T(n) = 2.T(n/2) + c.n**

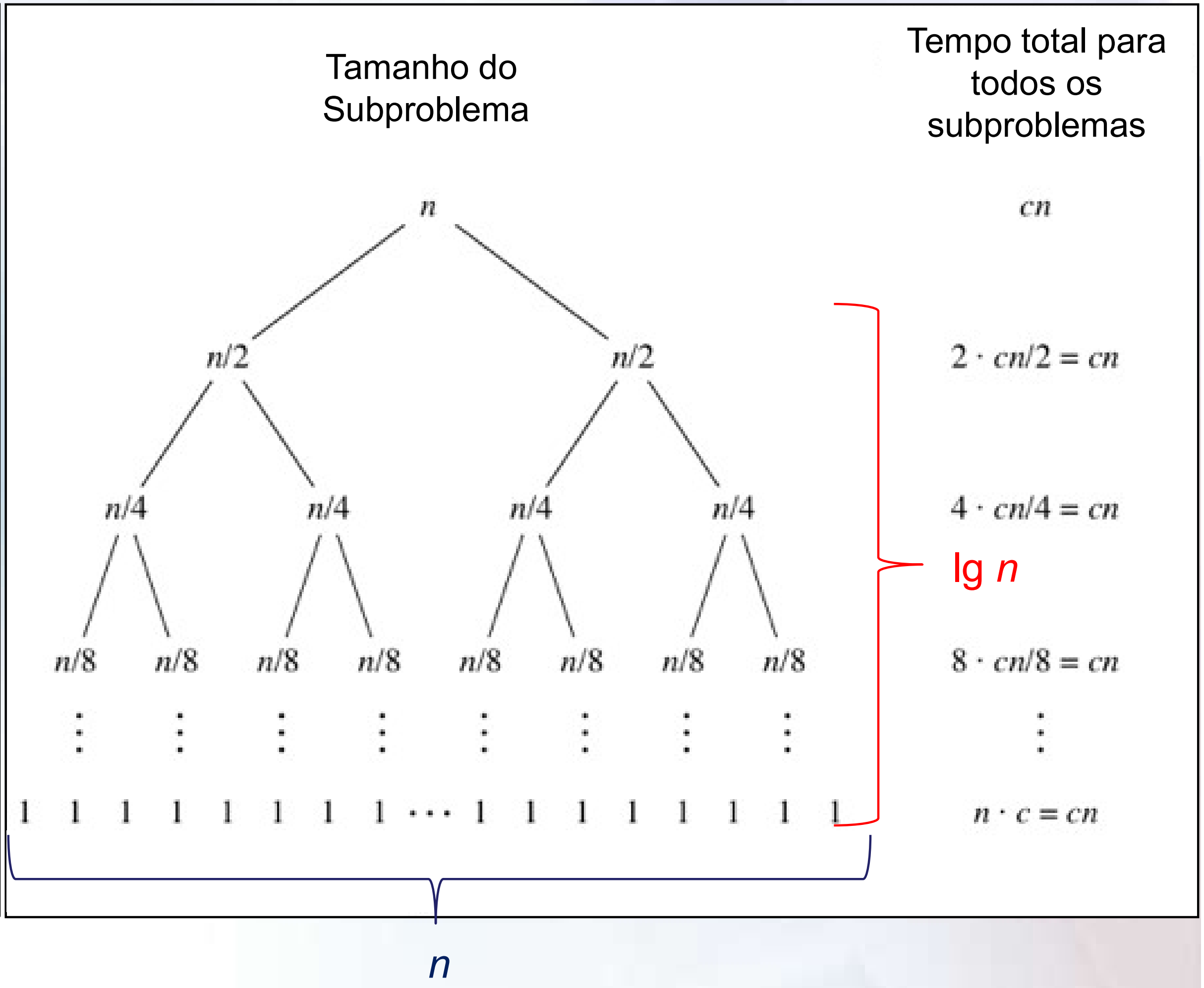




Merge Sort: Análise da Complexidade Algoritmo Mergesort (5)

- **Quantos nós tem na árvore com a altura i?**
  - n elementos
- **E qual a relação com a altura da árvore?**
  - $2^i = n$
  - $\lg 2^i = \lg n$
  - $i = \lg n$ ($\lg n = \log_2 n$)
- **Quantos níveis tem a árvore?**
  - lg n + 1 (Obs.: +1 → h = 0)

Altura (h) / nº de nós	
0	$2^0 = 1$
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
i	$2^i = n$



- **Complexidade T(n):**
  - c.n . (lg n + 1) → T(n) = c.n lg n + c.n → **T(n) = O(n.lg n)**

Merge Sort: Conclusão

- ***MergeSort*** necessita de espaço O(n) para **armazenar o array temporário**
- **Complexidade** T(n) do *MergeSort*:
  - **Pior Caso** - T(n): O (n.lg n)
  - **Melhor Caso** - T(n): O (n.lg n)
  - **Caso Médio** - T(n): O (n.lg n)
- Pode-se dizer que o **UP** de tempo para a **ordenação** é **O(n.log n)**
- Como o **UP** e o **LB** da **ordenação** são de **mesma ordem**:
  - A **ordenação** é um problema **computacionalmente resolvido**
  - Qualquer **algoritmo** que ordena n números em tempo **O(n.log n) é ótimo**

PANC: Projeto e Análise de Algoritmos

Aula 04: Paradigma de Divisão e Conquista e Ordenação utilizando MergeSort (Lógica e Complexidade)

Breno Lisi Romano

Dúvidas???

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista