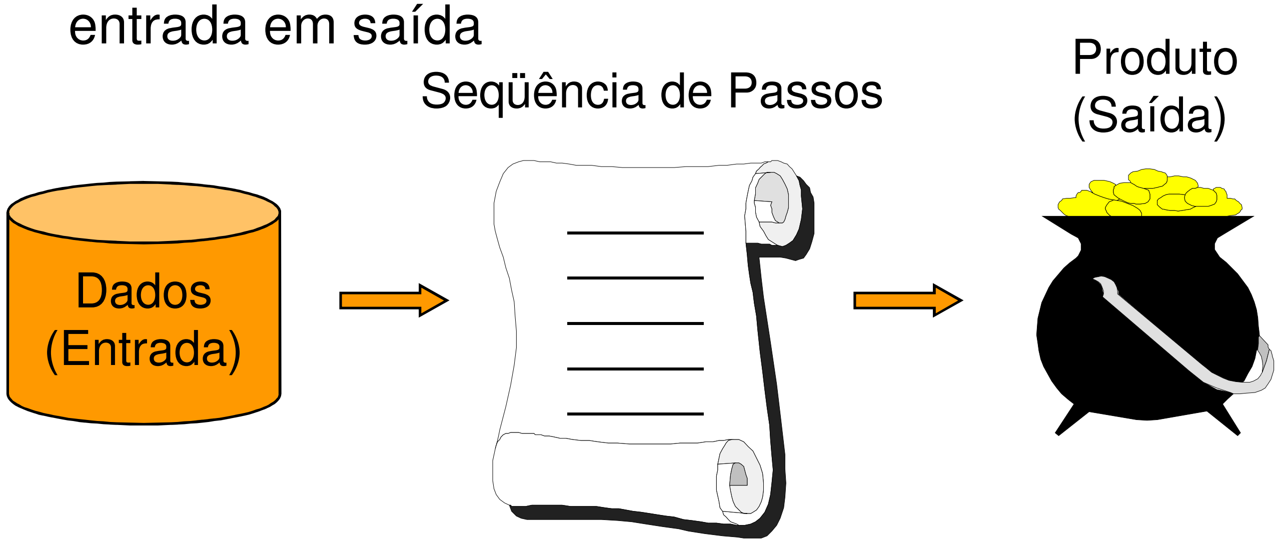
Algoritmos e Estrutura de Dados

Aula 3 – Conceitos Básicos de Algoritmos

Prof. Tiago A. E. Ferreira

Definição de Algoritmo

- Informalmente...
  - Um **Algoritmo** é qualquer procedimento computacional bem definido que toma algum valor (ou conjunto de valores) como **entrada** e produz algum valor (ou conjunto de valores) como **saída**.
    - Seqüência de passos computacionais que transforma entrada em saída



Exemplo: Problema de Ordenação

- Entrada:
  - Uma seqüência de *n* números:
    - $< a_1, a_2, ..., a_n >$
- Saída:
  - Uma permutação dos números de entrada:
    - $< a'_1, a'_2, ..., a'_n >$, *tal que* $a'_1 \leq a'_2 \leq ... \leq a'_n$ *(ordenação crescente)*
- Algoritmo:
  - Seqüência de comandos que leva uma instância de entrada em uma correta saída.

Notações

- Instância de um problema:
  - É a entrada, que satisfaz a quaisquer restrições impostas pelo problema, necessária para se calcular uma solução do problema
- Algoritmo correto:
  - É quando, para qualquer instância do problema, o algoritmo pára com a saída correta.
  - Resolve o problema computacional

Formas de Descrição de Algoritmos

- Pseudo-código
- Linguagem de programação
- Fluxograma
- Linguagem natural (ambigüidade!)

Desenvolvimento

- Identificação de etapas
- Detalhamento de cada etapa
- Seqüência de operações básicas sobre os dados considerados

Estrutura Dados

- É o meio para armazenar e organizar dados com o objetivo de facilitar o acesso e as modificações dos mesmos.
- Há vários tipos de estrutura de dados
  - Cada uma tem seus pontos forte e fracos

Estrutura de Dados

- Tipos de Dados
  - int, char, float, etc.

- Tipos Abstratos de Dados (TAD)
  - Filas, Pilhas, Listas, etc.

- Estruturas de dados: Método particular de se implementar um TAD

Custos

- Infelizmente os computadores têm recursos limitados!
  - Recurso "**poder de processamento**" (TEMPO)
  - Recurso "**armazenagem de dados**" (MOMÓRIA)

- Dois algoritmos distintos que realizam a mesma tarefa podem diferenciar brutalmente em relação aos custos em tempo e memória!

Exemplo

- Seja dois métodos de ordenação:
  - Ordenação por inserção:
    - Custo em tempo: c_1n^2 para ordenar n números
  - Ordenação por intercalação:
    - Custo em tempo: c_2nlog_2n para ordenar n números
- Suponha dois computadores:
  - Computador A:
    - Executa 1.000.000.000 de instruções por segundo
  - Computador B:
    - Executa 10.000.000 de instruções por segundo

Exemplo (Cont.)

- O melhor programador do mundo implementa a ordenação por inserção em código de máquina no computador A
- Um programador mediano implementa a ordenação por intercalação em linguagem de alto-nível no computador B
- Tempo em cada computador (ordenar um milhão de números)
  - Computador A ($c_1 = 2$)

$$\frac{2 \cdot \left(10^6\right)^2 \; instruções}{10^9 \; instruções / segundo} = 2.000 \; segundos$$

  - Computador B ($c_2 = 50$)

$$\frac{50 \cdot 10^6 \log_2 10^6 \; instruções}{10^7 \; instruções / segundo} \approx 100 \; segundos$$

Exemplo (Cont.)

- Desta forma:
  - Mesmo utilizando um compilador fraco, o computador B funciona 20 vezes mais rápido que o computador A!
  - Este exemplo mostra que a escolha do algoritmo pode ser bem mais crítica do que a escolha do Hardware e da linguagem e/ou experiência do programador!
- Portando:
  - Tanto os algoritmos quanto o Hardware constituem uma tecnologia!
  - O desempenho total do sistema depende da escolha correta de ambos!

Problema de Ordenamento

- Vamos analisar o problema de ordenamento:
  - Entrada:
    - Uma seqüência de *n* números:
      - $< a_1, a_2, ..., a_n >$
  - Saída:
    - Uma permutação dos números de entrada:
      - $< a'_1, a'_2, ..., a'_n >$, *tal que* $a'_1 \leq a'_2 \leq ... \leq a'_n$ *(ordenação crescente)*

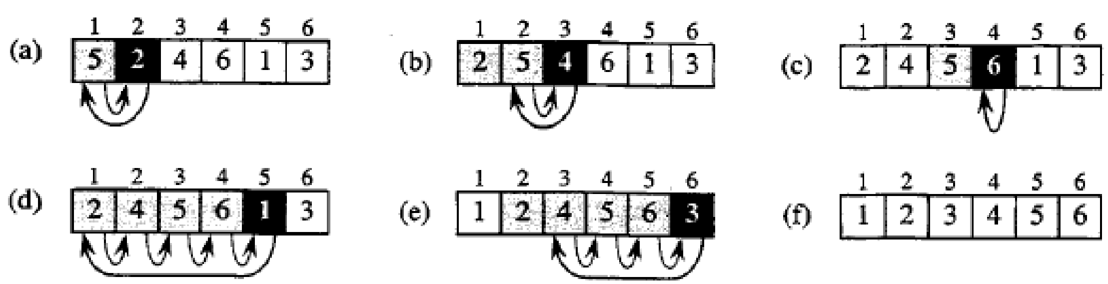
- *Obs.: os números que deseja-se ordenar serão chamados de* ***chaves***

Ordenamento por Inserção

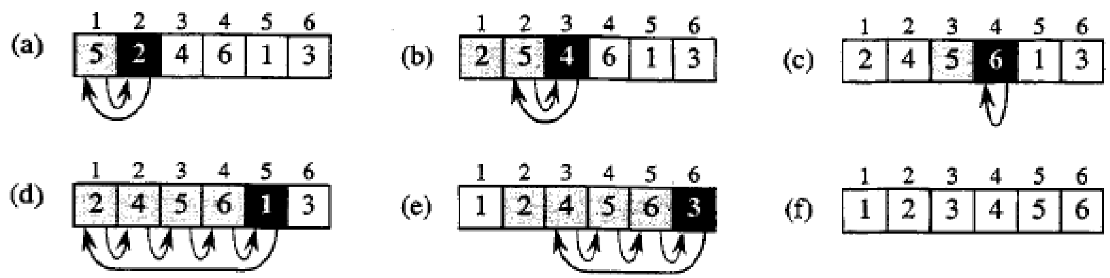
- Há várias formas de definirmos um algoritmo de ordenamento. Vamos começar pelo **Ordenamento do Inserção**
  - O ordenamento por inserção segue uma idéia bastante intuitiva:
    - Jogo de cartas: arrumamos as carta em uma certa seqüência a medida que as pegamos

Procedimento: Ordenamento por Inserção

- Procedimento: *insertion-sort*
- Entrada:
  - Arranjo de números *A[1...n]*
- Saída:
  - Arranjo de números *A[1...n]* ordenados
    - *Os números de entrada são ordenados no local*
- Exemplo: Seja *A=<5 2 4 6 1 3>*



Pseudo-Código




```
INSERTION-SORT(A)
for j ← 2 to comprimento[A]
    do chave ← A[j]
       ▷ Inserir A[j] na seqüência ordenada A[1..j – 1].
       i ← j – 1
       while i > 0 e A[i] > chave
           do A[i + 1] ← A[i]
              i ← i – 1
       A[i + 1] ← chave
```

Loop invariante

Loop **for**

Loop **While**

O símbolo ▷ indica um comentário

Loop Invariante

- Dada uma condição qualquer que desejamos observar em um algoritmo, podemos um ***loop* invariante.**
- Um Loop Invariante tem as Propriedades:
  - **Inicialização**: ele é verdadeiro antes da primeira iteração
  - **Manutenção**: Se for verdadeiro antes de uma iteração do loop, continuará verdadeiro antes da próxima iteração
  - **Término**: Quando o loop termina, o invariante fornece uma propriedade útil que ajuda a mostrar que o algoritmo é correto

Para o Ordenamento por Inserção

- **Inicialização:**
  - *j = 2,* o sub-arranjo *A[1..j-1]* consiste apenas no único elemento *A[1],* que é elemento de *A* e está ordendado
    - *Isto mostra que o loop invariante é válido antes da primeira iteração*
- **Manutenção:** (cada iteração mantém o loop invariante)
  - A medida que o **for** corre, desloca-se uma casa à direita em *A* (*A[j-1], A[j-2], A[j-3],...*) a procura da posição ideal para *A[j], mantendo o* sub-arranjo *A[1..j-1] ordenado*
- **Término:** (o que ocorre ao fim do loop)
  - *j = n+1,* sendo gerado o sub-arranjo *A[1..n]* que contem todos os elementos da seqüência de entrada e está ordenado
  - **Logo o algoritmo é correto!**

Análise de Algoritmos

- Analisar Algoritmo é...
  - Prever recursos necessários
    - Tempo de processamento
    - Memória necessária
    - Largura de banca para comunicações
    - Etc...
- Com a finalidade de...
  - Descartar algoritmos inviáveis
  - Escolher algoritmo correto mais barato computacionalmente

Analisando o Ordenamento por Inserção

- O custo do algoritmo de ordenação dependerá:
  - Tamanho da entrada
    - Número de itens na entrada (***n***) para o problema
  - Grau do pré-ordenamento da entrada
- Tempo de execução de um algoritmo
  - Em uma determinada entrada, é o número de operações primitivas ou “etapas” executadas
    - Pode-se definir uma etapa por um passo no algoritmo que seja o mais independente possível da máquina
    - Considera-se que cada linha do pseudo-código leve um tempo constante para sua execução
      - *i-ésima* linha, tempo c_i

Custos por linha e total

INSERTION-SORT(A)	*custo*	*vezes*
for $j \leftarrow 2$ **to** *comprimento*$[A]$	c_1	n
do *chave* $\leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Inserir $A[j]$ na seqüência ordenada $A[1..j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ e $A[i] >$ *chave*	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow$ *chave*	c_8	$n-1$

Nº de vezes que o teste do *While* é executado

Custo Total:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) - c_8(n-1).$$

Melhor caso

- No melhor caso, a entrada já se encontra ordenada!
  - Custo:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

    - Neste caso o teste do ***While*** é executado apenas uma vez para cada passo do ***for***.
    - Este custo pode ser escrito como ***an+b***, onde *a* e *b* são constantes, i.e., uma função linear em *n*

Pior Caso

- No pio caso, a entrada se encontra ordenada de forma decrescente! (ordem inversa de que se deseja ordenar)
  - Custo:
    - Para o pior caso, o teste do ***While* é repetido *j* vezes para cada passo do *for*.**
    - **Então:**

$$\sum_{j=2}^{n} j = \frac{n(n-1)}{2} - 1 \qquad\qquad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Pior Caso

- Portanto o custo total é dado por:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n-1)}{2} - 1\right)$$

$$+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n$$

$$- (c_2 + c_4 + c_5 + c_8).$$

- Esta é uma função do tipo: ***an*^{2}*+bn+c***
  - Um função quadrática de *n*

Análise do Pior Caso e do Caso Médio

- A análise mais utilizada é a do **pior caso**
  - A análise do pior caso é um limite superior
    - Conhecê-lo é a garantia que o algoritmo não irá demorar mais tempo do que o calculado!
  - Em vários problemas, esta é a situação mais comum
  - Em muitos problemas o **caso médio** é quase tão ruim que o **pior caso**.
    - Contudo, em alguns caso, como em algoritmos probabilísticos (ou estocásticos) , o caso médio é o de maior interesse.

Projeto de Algoritmos

- Abordagem de dividir e conquistar
  - **Dividir** o problema em um determinado número de subproblemas
  - **Conquistar** os subproblemas, reescrevendo-os recursivamente
  - **Combinar** as soluções dadas aos subproblemas

Exemplo de Dividir e Conquistar

- Algoritmo de **ordenação por intercalação**
  - **Dividir:** divide a seqüência de **n** elementos a serem ordenados em duas subseqüências de **n/2**
  - **Conquistar:** Classifica as duas subseqüências recursivamente
  - **Combinar:** faz a intercalação das duas subseqüências ordenadas

Algoritmo de Ordenação por Intercalação

- A operação chave está no passo de combinação, onde são intercaladas (merge) duas subseqüências já ordenadas
  - Será utilizado o procedimento MERGE(A,p,q,r)
    - Onde A é um arranjo, p,q e r são índices de enumeração dos elementos do arranjo, tais que $p \leq q < r$
  - É pressuposto que os sub-arranjos A[p..q] e A[q+1..r] estejam em seqüência ordenada

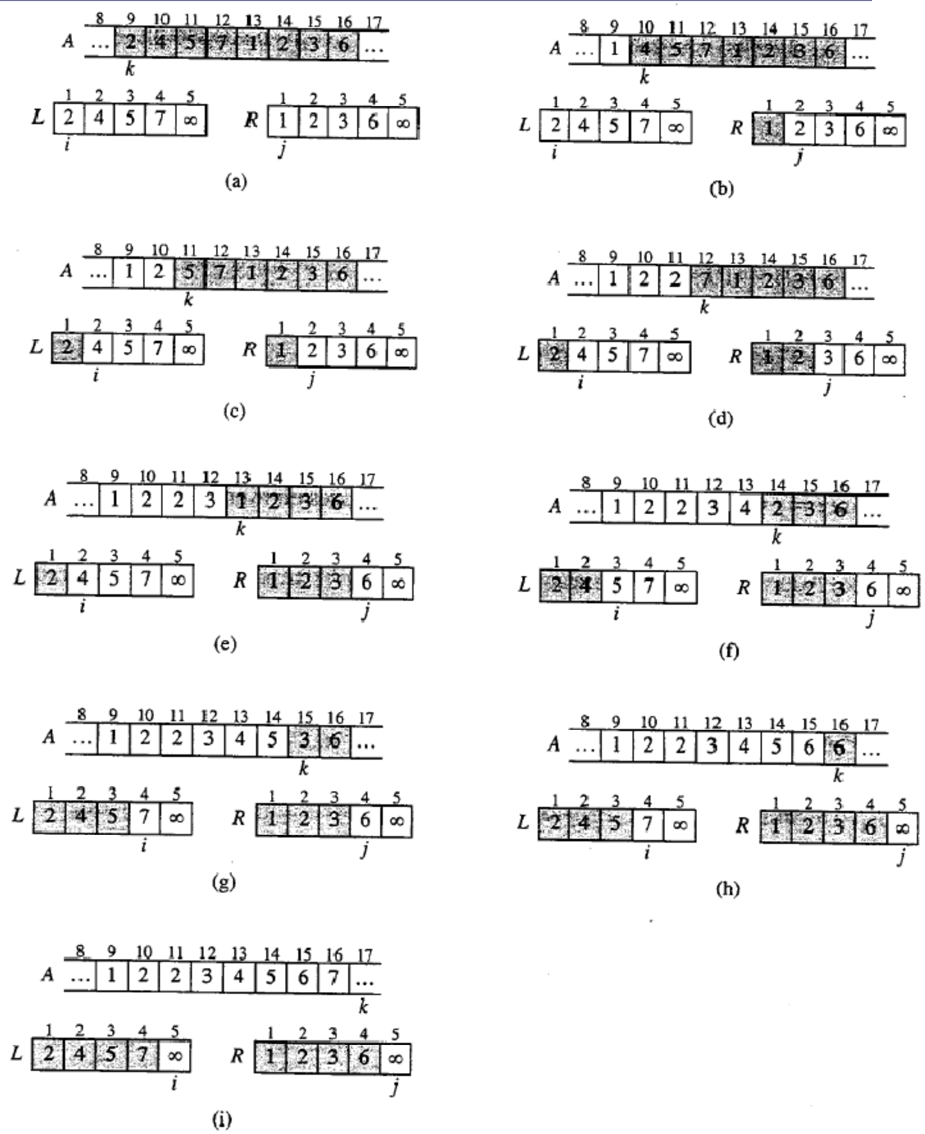
Combinação

- Suponha a existência de duas pilha de cartas com as numerações para cima
  - Será gerada uma pilha de saída, onde será depositada a carta de menor valor dentre as que estão expostas nas duas pilhas iniciais. Esta pilha de saída é formada com as cartas viradas para baixo.
  - Ao término, será gerada uma única pilha ordenada com todas as cartas das duas pilhas iniciais
  - Sendo n o número total de cartas (duas pilhas iniciais), o custo em tempo será $\Theta(n)$

Algoritmo MERGE(A,p,q,r)

MERGE(A, P, q, r)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
criar arranjos $L[1..n_1 + 1]$ e $R[1..n_2 + 1]$
for $i \leftarrow 1$ **to** n_1
    do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ **to** n_2
    do $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ **to** r
    do if $L[i] \le R[j]$
        then $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$

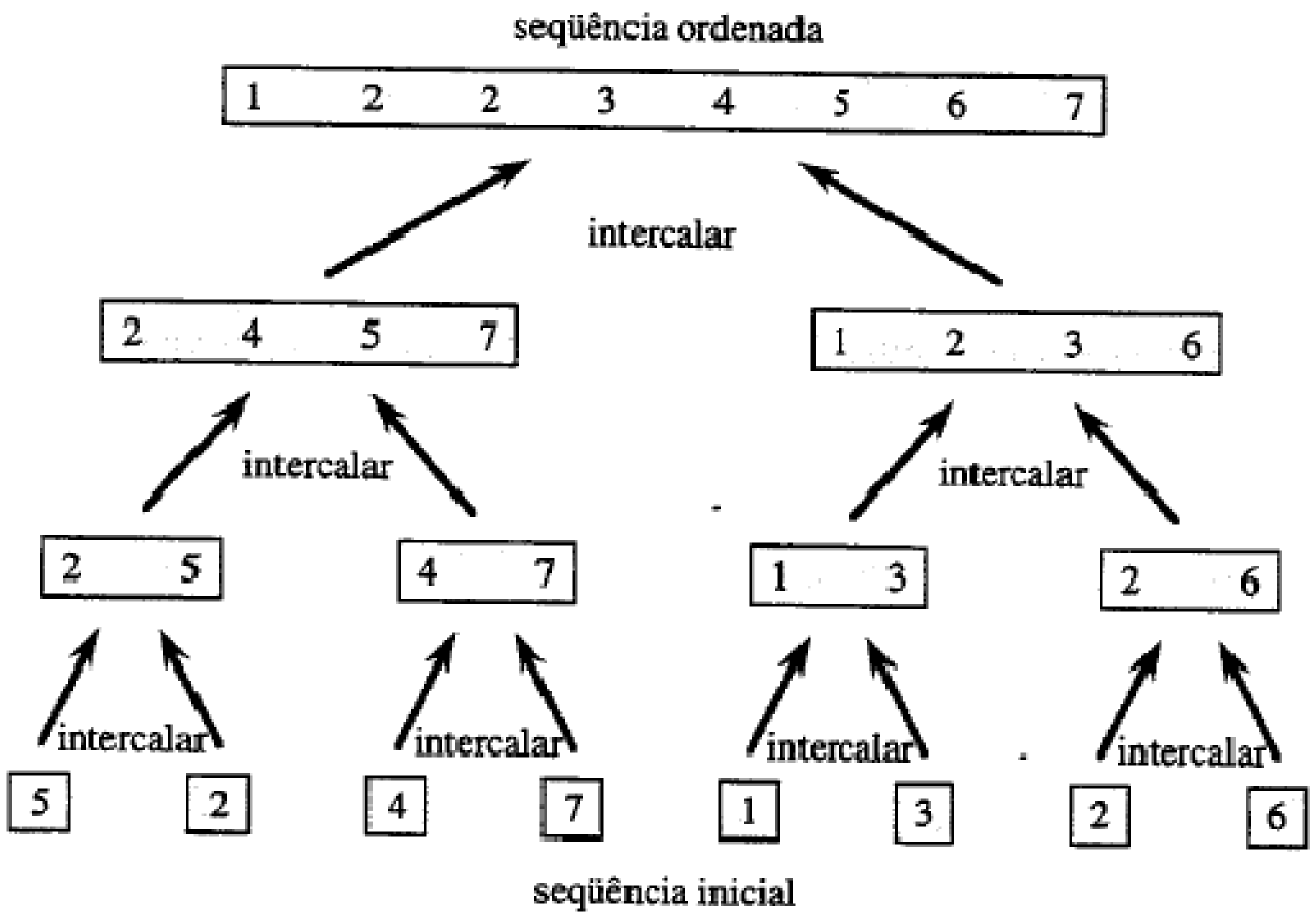


Algoritmo de ordenação por intercalação

- O Algoritmo MERGE-SORT(A,p,r) ordena os elementos do sub-arranjo A[p..r]
  - Se p≥r, então o arranjo tem 1 elemento
  - Caso contrario, é calculado o índice q que particiona o arranjo A[p..r] em dois sub-arranjos: A[p..q], com $\lceil n/2 \rceil$ elementos, e A[q+1,r] com $\lfloor n/2 \rfloor$ elementos

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

MERGER-SORT(A,1,Comprimento[A])



Algoritmos Recursivos

- Um algoritmo que tem uma chamada a si próprio, seu tempo de execução freqüentemente pode ser descrito por uma **equação de recorrência** ou **recorrência**
  - Para n pequeno, $n \leq c$ (c é uma cte qq), a solução direta demorará um tempo constante, $\Theta(1)$
  - Caso contrário, o problema será dividido em *a* subproblemas de comprimento 1/b do comprimento total
    - E, seja D(n) o tempo para dividir o problema, e C(n) o tempo para combinar as soluções

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

Análise da Ordenação por Intercalação

- Sem perda de generalidade, vamos supor que o comprimento do arranjo é uma potência de 2
  - A ordenação por intercalação sobre um único elemento demora um tempo constante
  - Quando n>1, deve-se desmembrar o tempo de execução:
    - Dividir: Calcula o ponto médio do subarranjo, demorando um tempo constante, $D(n)=\Theta(1)$
    - Conquistar: são resolvidos dois problemas recursivamente, cada um com tamanho de n/2, custo de $2T(n/2)$
    - Combinar: algoritmo MERGE, $C(n)=\Theta(n)$

Análise da Ordenação por Intercalação

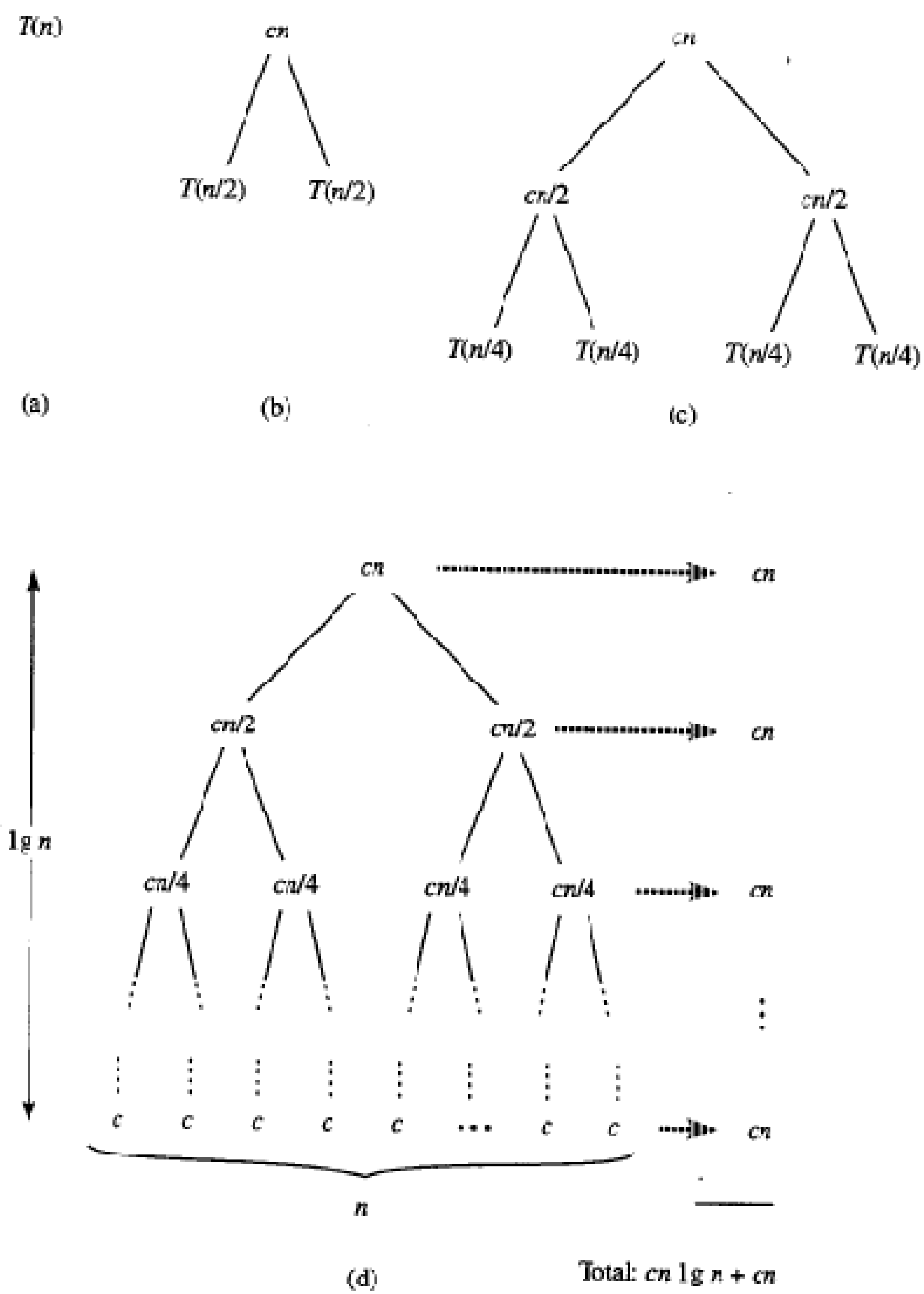
- Custo total:

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

  - Se chamarmos *c* uma cte que represente o tempo exigido para resolver problemas de tamanho 1:

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

- É possível perceber que T(n)=O(n lg n), lg é o log na base 2?



T(n)
cn
T(n/2)
T(n/2)
cn
cn/2
cn/2
T(n/4)
T(n/4)
T(n/4)
T(n/4)
(a)
(b)
(c)
cn
cn
cn/2
cn/2
cn
cn/4
cn/4
cn/4
cn/4
cn
lg n
c
c
c
c
c
...
c
c
cn
n
(d)
Total: cn lg n + cn