Ordenação

Projeto por indução simples de algoritmos de ordenação
Projeto por indução forte de algoritmos de ordenação
Heaps
Análise de algoritmos de ordenação

Ordenação

- Problema
  - Sejam n números $a_1, a_2, \dots, a_n$, ordenar os números em ordem crescente. Em outras palavras, encontrar uma permutação da entrada tal que $a'_1 \leq a'_2 \leq \ \dots \leq a'_n$.

Projeto por indução simples

- Hipótese de Indução Simples
  - Sabemos ordenar um conjunto de $n - 1 \geq 1$ inteiros.

- Caso base: $n = 1$. Um conjunto de um único elemento está ordenado.
- Passo da Indução (Primeira Alternativa): Seja S um conjunto de $n \geq 2$ inteiros e x um elemento qualquer de S. Por hipótese de indução, sabemos ordenar o conjunto $S - x$, basta então inserir x na posição correta para obtermos S ordenado.
- Esta indução dá origem ao algoritmo incremental Insertion Sort.

Insertion sort

```
OrdenacaoInsercaoRec(A, n)
if n ≥ 2
    OrdenacaoInsercaoRec(A, n − 1)
    v = A[n]
    j = n − 1
    while (j > 0) and (A[j] > v)
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = v
```

Análise do Insertion Sort

- Número de comparações e de trocas é dado pela recorrência

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n, & n > 1 \end{cases}$$

- Logo, a complexidade é de $\Theta(n^2)$ no pior caso.

Versão iterativa

- Trocar a recursão por um novo laço de repetição

```
OrdenacaoInsercao(A)
for i=2 to n
  v = A[i]
  j = i-1
  while (j > 0) and (A[j] > v)
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = v
```

Projeto por indução simples

- Hipótese de Indução Simples
  - Sabemos ordenar um conjunto de $n - 1 \geq 1$ inteiros.

- Caso base: $n = 1$. Um conjunto de um único elemento está ordenado.
- Passo da Indução (Segunda Alternativa): Seja S um conjunto de $n \geq 2$ inteiros e x o **menor** elemento de S. Então x deve ocupar a primeira posição da sequência ordenada de S. Por hipótese de indução, sabemos ordenar o conjunto $S - x$, basta então inserir x na posição correta para obtermos S ordenado.

Selection sort

```
OrdenacaoSelecaoRec(A, i, n)
if i < n
  min = i
  for j=i+1 to n
    if A[j] < A[min]
      min = j
  troca(A[min],A[i])
  OrdenacaoSelecaoRec(A,i+1,n)
```

Análise do Selection Sort

- Número de comparações é dado pela recorrência

$$T(n) = \begin{cases} 1, & n = 1 \\ T(n-1) + n, & n > 1 \end{cases}$$

- O número de trocas é dado pela recorrência

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + 1, & n > 1 \end{cases}$$

- Logo, a complexidade do tempo de execução do Selection Sort é de $\Theta(n^2)$ no pior caso.
  - O número de trocas é $\Theta(n)$.

Versão iterativa

```
OrdenacaoSelecao(A)
for i=1 to n-1
  min = i
  for j=i+1 to n
    if A[j] < A[min]
      min = j
  troca(A[min],A[i])
```

Projeto por indução simples

- Hipótese de Indução Simples
  - Sabemos ordenar um conjunto de $n - 1 \geq 1$ inteiros.
- Caso base: $n = 1$. Um conjunto de um único elemento está ordenado.
- Passo da Indução (Terceira Alternativa): Seja S um conjunto de $n \geq 2$ inteiros e x o **maior** elemento de S. Então x deve ocupar a última posição da sequência ordenada de S. Para cada elemento do conjunto S, da esquerda para a direita, trocamos os elementos adjacentes com ordem invertida. Por hipótese de indução, sabemos ordenar o conjunto $S - x$, basta então inserir x na posição correta para obtermos S ordenado.
- Parecido com o Selection sort, mas se implementarmos o posicionamento do maior elemento através de trocas com elementos adjacentes, obtemos o Bubble sort.

```
OrdenacaoBolha(A)
for i=n downto 1
  for j=2 to i
    if A[j-1] > A[j]
      troca(A[j-1],A[j])
```

Análise do Bubble Sort

- Número de comparações e trocas no pior caso é dado pela recorrência

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n - 1, & n > 1 \end{cases}$$

- Logo, a complexidade é de $\Theta(n^2)$ no pior caso.
  - O Bubble Sort realiza mais trocas do que o Selection Sort.

Heapsort

- Mesmo princípio da ordenação por seleção
- Algoritmo
  1) Selecione o maior item do vetor de n posições

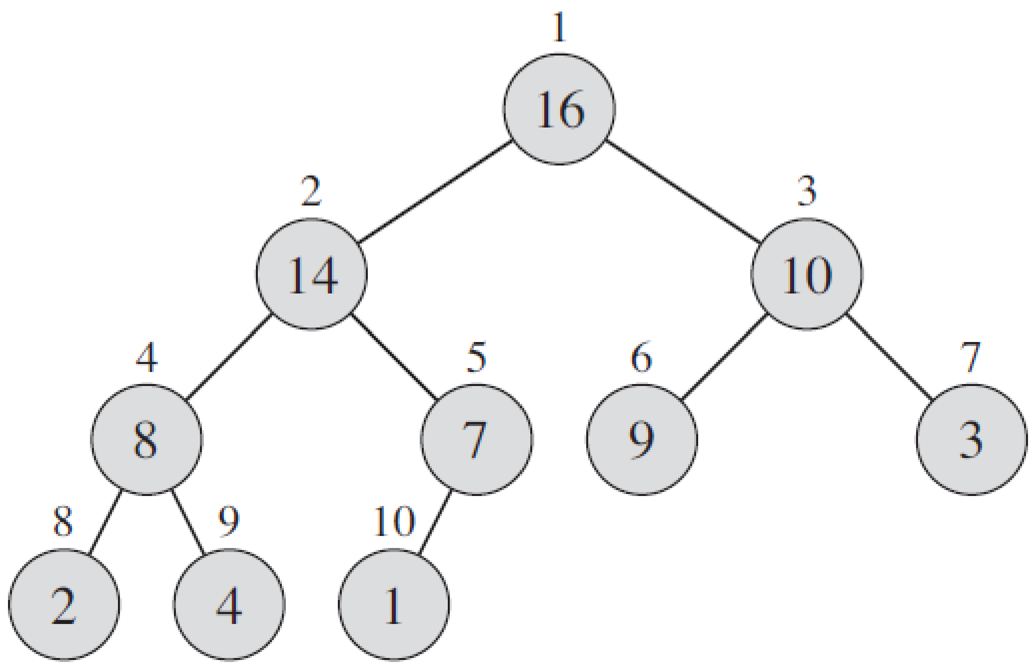
  2) Troque-o com o item da posição n do vetor

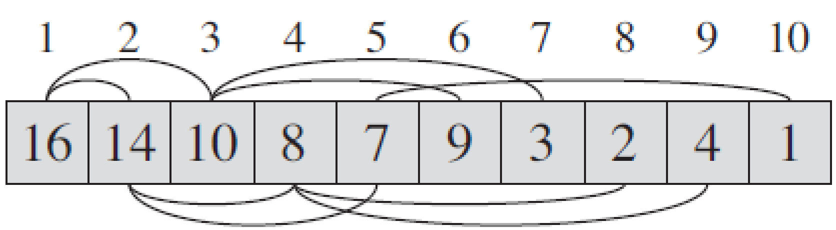
  3) Repita estas operações com os $n - 1$ itens restantes, depois com os $n - 2$ itens, e assim sucessivamente
- Utilizar uma heap (fila de prioridade) para otimizar a busca do passo 1

Heap

- Heap
  - Árvore binária quase completa que satisfaz as seguintes propriedades:
    - É completa até o penúltimo nível
    - No último nível, as folhas estão mais a esquerda possível
    - O valor de um nó é maior do que os valores dos nós filhos na sua sub-árvore

Exemplo





Raíz é $A[1]$

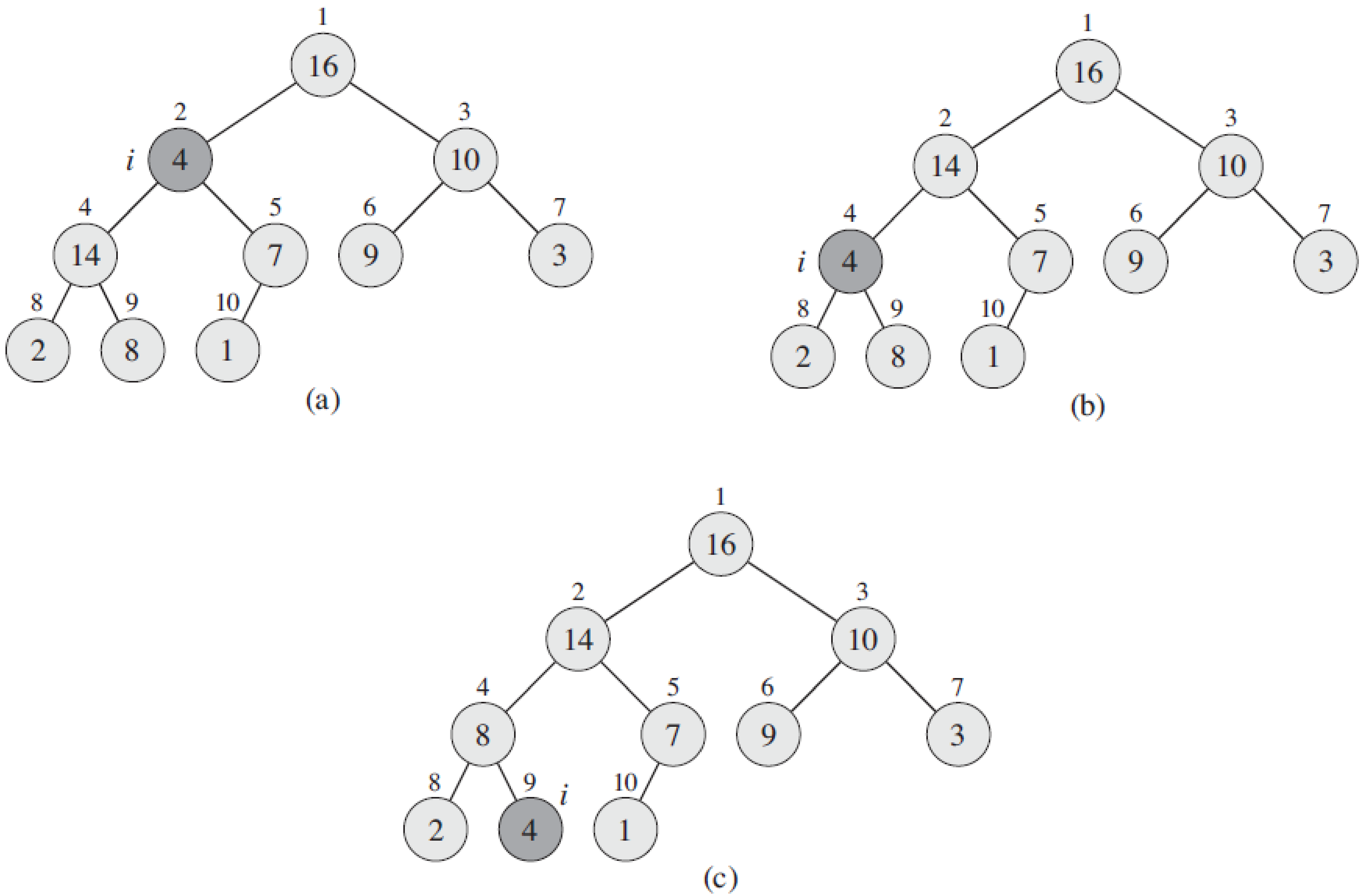
Para cada nó A[i], temos:

- Filho a esquerda: $A[2 * i]$
- Filho a direita: $A[2 * i + 1]$
- Pai: $A[\lfloor i/2 \rfloor]$
- $A[Parent(i)] \geq A[i]$ (Heap máximo)

- Procedimento para manter a propriedade Max-Heap

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

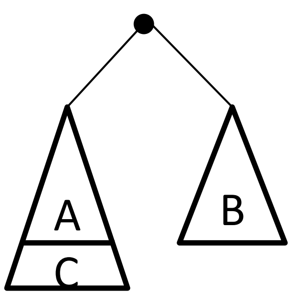
Exemplo



1
16
2
i
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
(a)
1
16
2
14
3
10
4
i
4
5
7
6
9
7
3
8
2
9
8
10
1
(b)
1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
i
4
10
1
(c)

Análise do MAX-HEAPFY()

- O algoritmo é executado uma única vez em cada nível da heap com operações de tempo constante. Então a complexidade é proporcional a altura da árvore.
- Logo, no pior caso é $\Theta(h) = \Theta(\log n)$ e no melhor caso é $\Theta(1)$.
- Lema: O tamanho máximo do subproblema (subárvore) para MAX-HEAPFY é $2n/3$.
- Demonstração: Este caso ocorre quando a árvore possui metade das folhas no último nível.



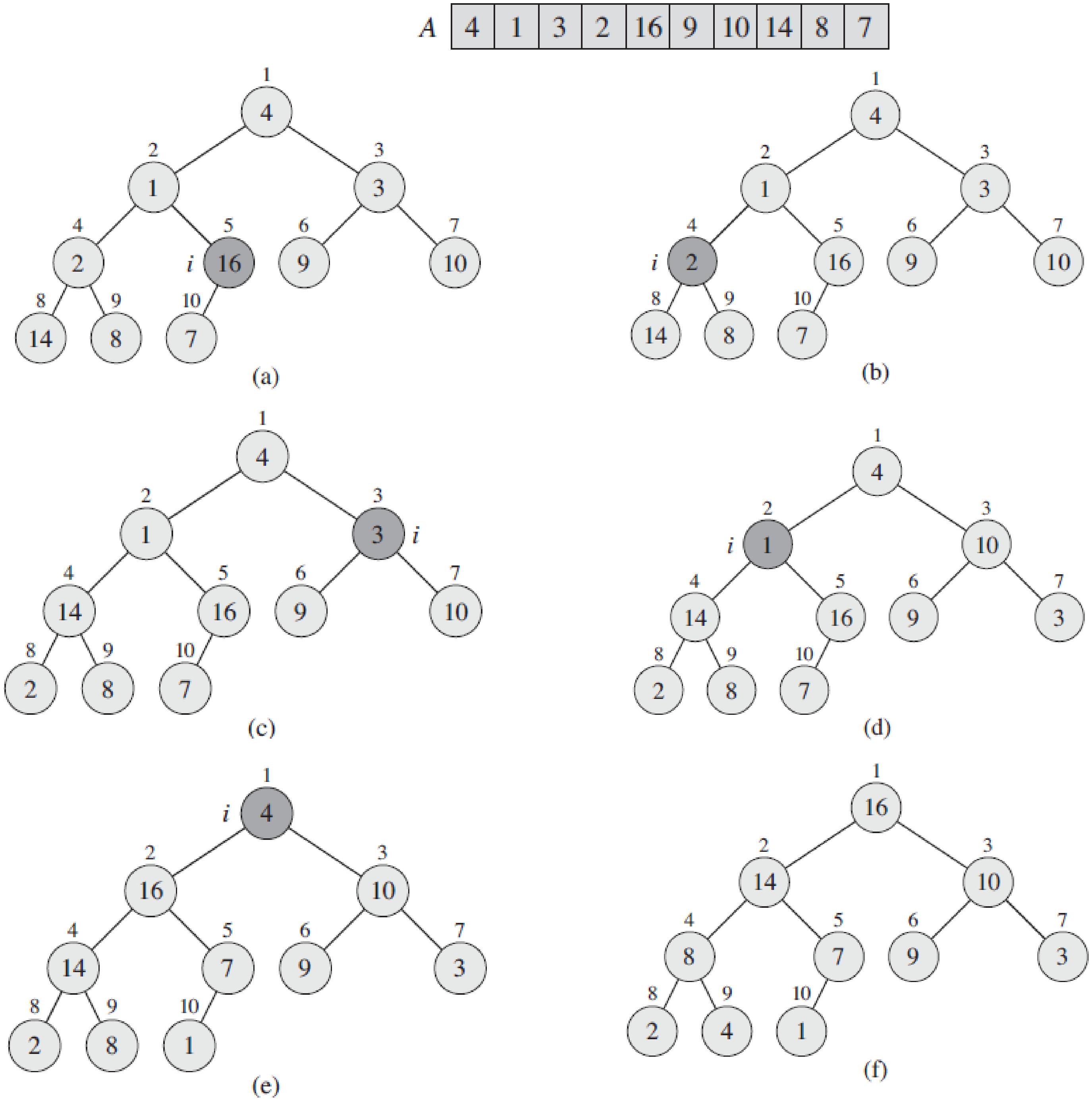
A e B são subárvores completas e C é o último nível da árvore heap.
Considerando que uma árvore binária completa de altura h tem $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ nós até o penúltimo nível e 2^h no último, então se em C existem x nós, então A e B possuem $x - 1$ nós cada. O total de nós da árvore é:

$$x + (x-1) + (x-1) + 1 = n$$
$$x + 2(x-1) + 1 = n$$
$$x = \frac{(n+1)}{3}$$

Portanto, o limite superior para o tempo do MAX-HEAPFY no *pior caso* é dado por
$T(n) \leq T(2n/3) + \Theta(1)$
$T(n) = \Theta(\log n)$ (caso 2 do teorema mestre).

- Procedimento para converter um arranjo em max-heap

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```



A 4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
(f)

Análise do BUILD-MAX-HEAP

- Propriedades de max-heap
  - Altura de um heap de n elementos: $\lfloor \log n \rfloor$
  - No máximo $\lceil n/2^{h+1} \rceil$ nós de altura h (pode ser demonstrado por indução em h)

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2\,. \end{aligned}$$

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$.

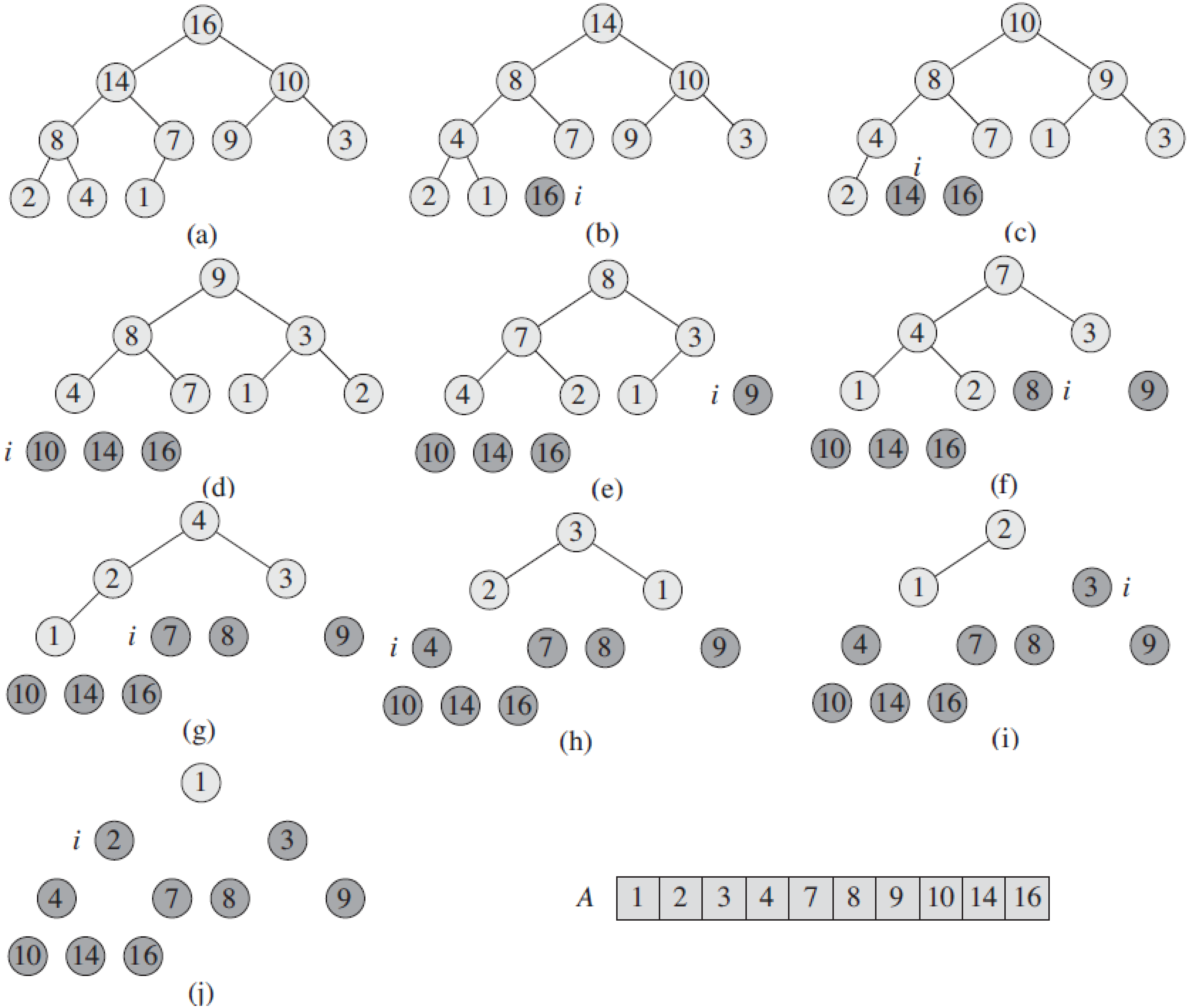
$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n)\,. \end{aligned}$$

Heapsort

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size - 1
        MAX-HEAPIFY(A, 1)
```

- BUILD-MAX-HEAP tem complexidade $\Theta(n)$
- MAX-HEAPFY
  - É executado $n-1$ vezes
  - MAX-HEAPFY é executado em $\Theta(\log n)$ no pior caso
    - Executa no máximo uma vez a cada nível
    - A cada nível, executa um número constante de operações
- Então, o Heapsort, no pior caso, tem complexidade de

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


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
A 1 2 3 4 7 8 9 10 14 16

Heaps

- Aplicação de heap
  - Fila de prioridade eficiente
- Max-heap
  - A raiz contém o maior elemento de sua subárvore
- Min-heap
  - A raiz contém o menor elemento de sua subárvore

Max-heap

- Máximo elemento em um max-heap
  - Primeiro elemento do arranjo (A[1])
- Remover e retornar o elemento máximo
  - Trocar o elemento na raiz pelo último elemento (folha) e chamar Max-Heapfy(A,1) para corrigir o heap

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
MAX-HEAPIFY(A, 1)
return max
```

Max-heap

- Aumentar o valor de uma chave na posição i
  - Pode violar as regras de um max-heap
  - Percorrer um caminho de i até a raiz trocando os elementos fora de ordem

HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
    error “new key is smaller than current key”
$A[i] = key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
    exchange $A[i]$ with $A[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$

Max-heap

- Inserir uma nova chave
  - Aumentar o tamanho do heap e inserir uma chave de tamanho mínimo ($-\infty$) nessa posição
  - Chamar função para aumentar o valor desta nova posição para o valor da chave a ser inserida

MAX-HEAP-INSERT(A, key)

$A.heap\text{-}size = A.heap\text{-}size + 1$

$A[A.heap\text{-}size] = -\infty$

HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

Projeto por indução forte

- Hipótese de Indução Forte
  – Sabemos ordenar um conjunto de $1 \leq k < n$ inteiros.

- Caso base: $n = 1$. Um conjunto de um único elemento está ordenado.
- Passo da Indução (Primeira Alternativa): Seja S um conjunto de $n \geq 2$ inteiros. Podemos particionar S em 2 conjuntos, S_1 e S_2, de tamanhos $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$. Como $n \geq 2$, cada partição possui menos de n elementos. Por hipótese de indução, sabemos ordenar os conjuntos S_1 e S_2. Para obter S ordenado basta então intercalar as partições ordenadas S_1 e S_2.

Merge sort

MERGE-SORT(A, p, r)

if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT$(A, q + 1, r)$
    MERGE(A, p, q, r)

Análise do Merge sort

- Número de comparações e de trocas é dado pela recorrência

$$T(n) = \begin{cases} 0 & \text{se } n = 1, \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & \text{se } n > 1 \end{cases}$$

- Logo, a complexidade é de $\Theta(n \log n)$ no pior caso.

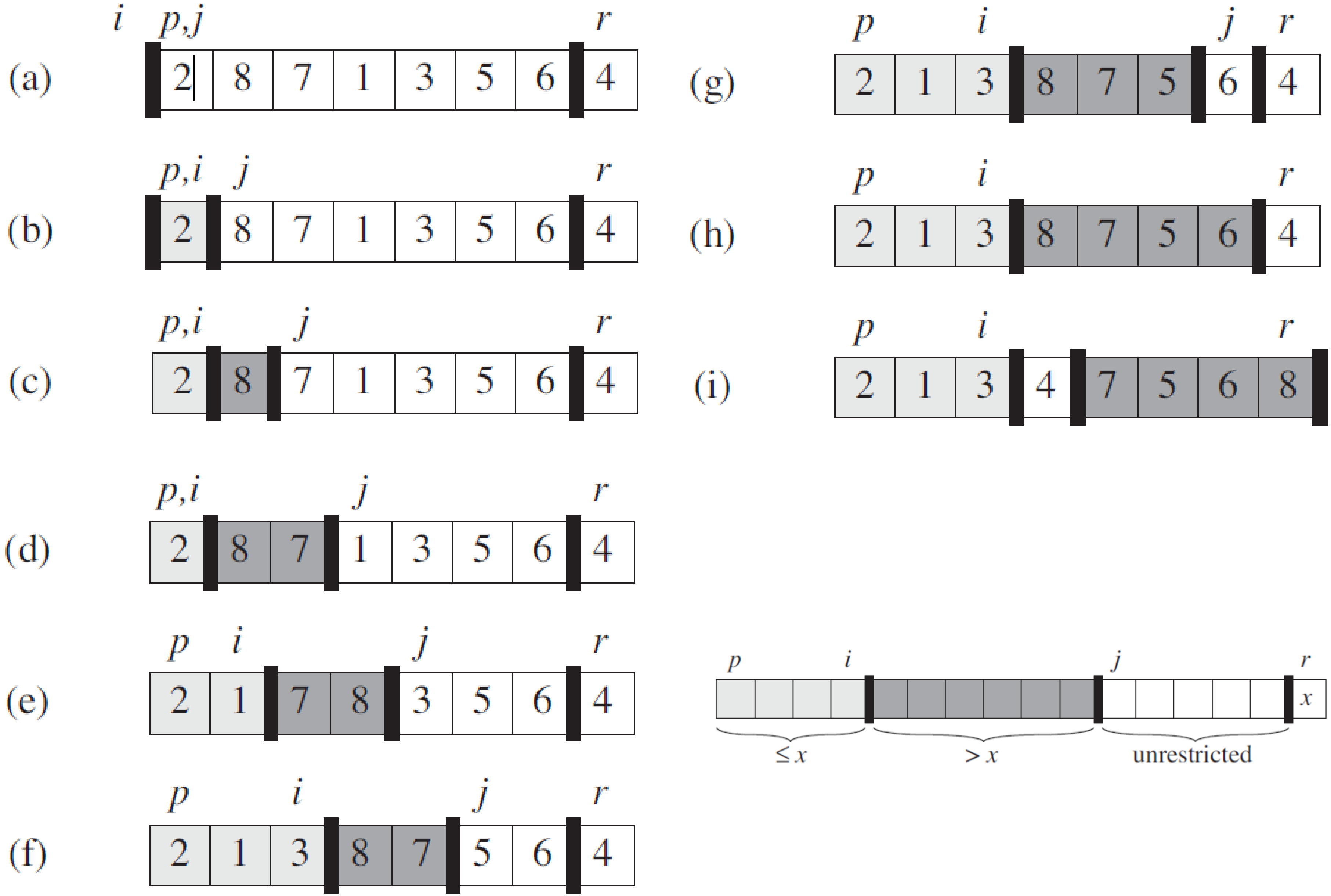
Projeto por indução forte

- Hipótese de Indução Forte
  - Sabemos ordenar um conjunto de $0 \leq k < n$ inteiros.

- Caso base: $n \leq 1$. Um conjunto de um único elemento ou vazio já está ordenado.
- Passo da Indução (Segunda Alternativa): Seja S um conjunto de $n \geq 2$ inteiros e x um elemento qualquer de S. Sejam S_1 e S_2, os subconjuntos de $S - x$ dos elementos menores ou iguais a x e maiores que x, respectivamente. Como $n \geq 2$, cada partição possui menos de n elementos. Por hipótese de indução, sabemos ordenar os conjuntos S_1 e S_2. Para obter S ordenado basta então concatenar S_1 ordenado, x e S_2 ordenado.

Quicksort

- Utiliza o processo de dividir e conquistar para ordenar um subarranjo $A[p..r]$
  - Dividir: particionar uma sequência $A[p..r]$ nas subsequências $A[p..q-1]$ e $A[q+1..r]$ tais que cada elemento de $A[p..q-1]$ seja menor que os elementos de $A[q+1..r]$.
  - Conquistar: ordenar recursivamente os dois subarranjos $A[p..q-1]$ e $A[q+1..r]$ por quicksort.
  - Combinar: Nada a ser feito: o arranjo $A[p..r]$ já está ordenado.


(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8
p i j r x
≤ x
> x
unrestricted

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Análise do Pior Caso

- O desempenho do Quicksort depende se o particionamento é balanceado ou não.
- O pior caso ocorre quando o particionamento é desbalanceado.

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

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

Análise do Melhor Caso

- O melhor caso ocorre quando o particionamento produz subproblemas de tamanho até $n/2$.

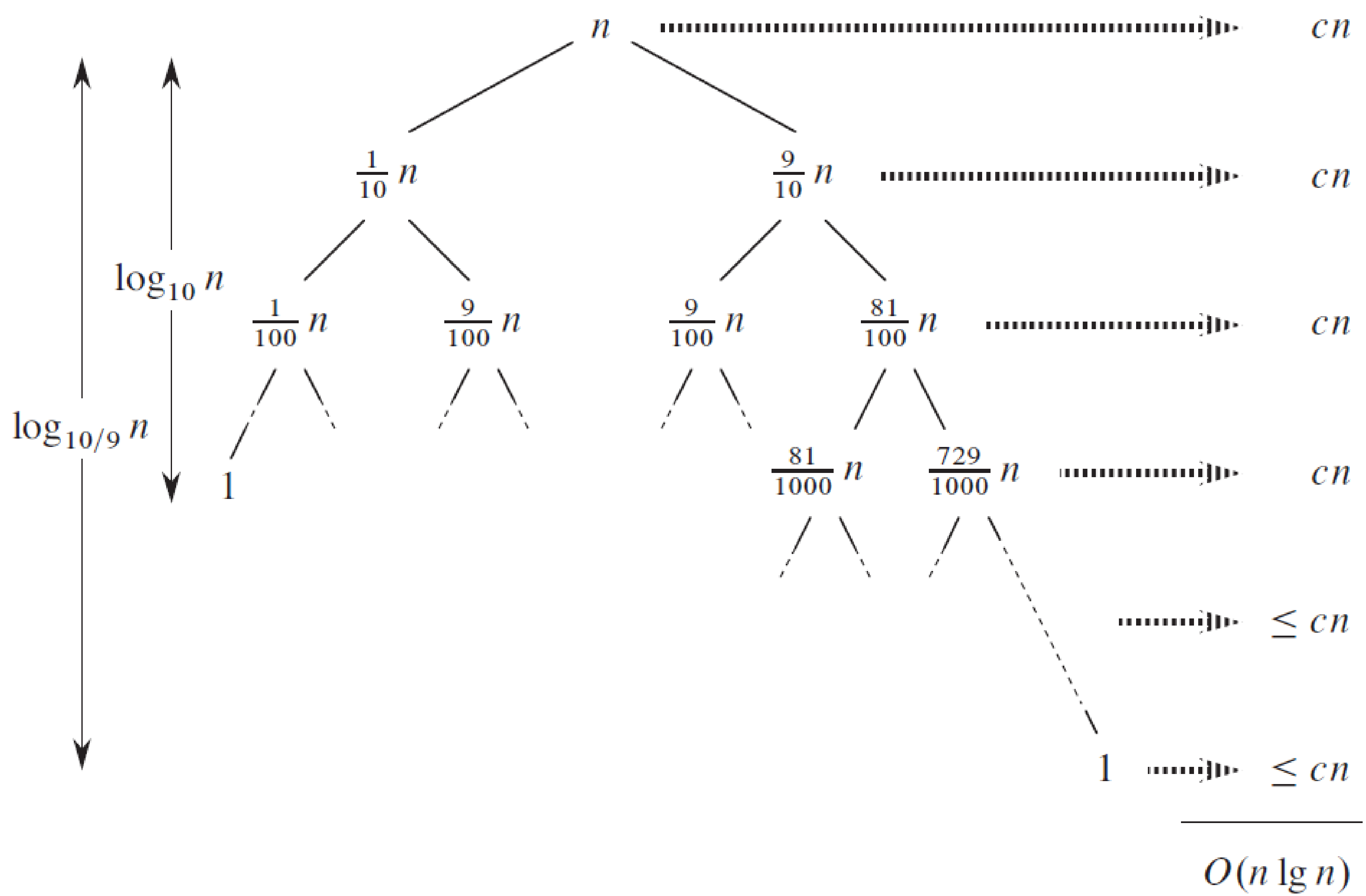
$T(n) = 2T(n/2) + \Theta(n)$

$T(n) = \Theta(n \log n)$ (caso 2 do teorema mestre)

Intuição para o caso médio

- Suposição: balanciomento na proporção 9:1.

$T(n) = T(9n/10) + T(n/10) + \Theta(n)$



Exercício

1) A entrada consiste de k vetores ordenados de números reais. Projete um algoritmo para intercalar todos esses vetores e obter um vetor ordenado. Seu algoritmo deve ter complexidade $O(n \lg k)$ onde n é o numero total de elementos em todos os vetores. Note que os vetores não precisam ter o mesmo tamanho.

Exercício

2) Projete um algoritmo por divisão e conquista que encontra a mediana de um vetor de inteiros. Utilize a ideia do algoritmo do quicksort. Qual a complexidade desta solução no caso médio?

Referências

- CLRS, Introduction to Algorithsm, 3rd ed.
  - Cap. 6, 7.1, 7.2