

CT-234



Estruturas de Dados,
Análise de Algoritmos e
Complexidade Estrutural

Carlos Alberto Alonso Sanches

CT-234



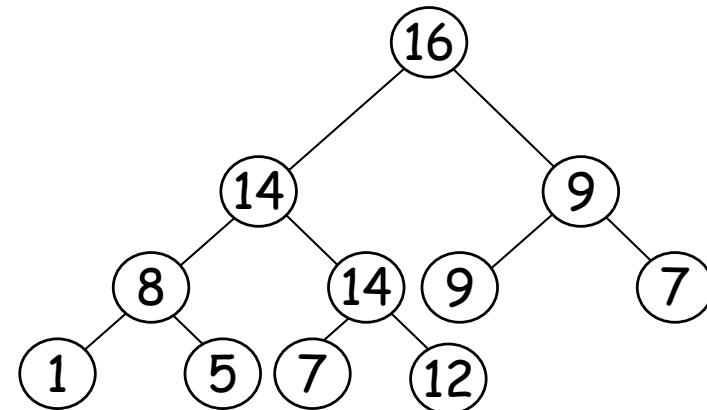
6) Ordenação

HeapSort, QuickSort, Rede Bitônica

A estrutura heap

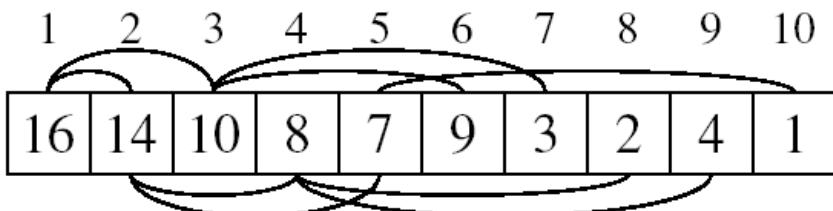
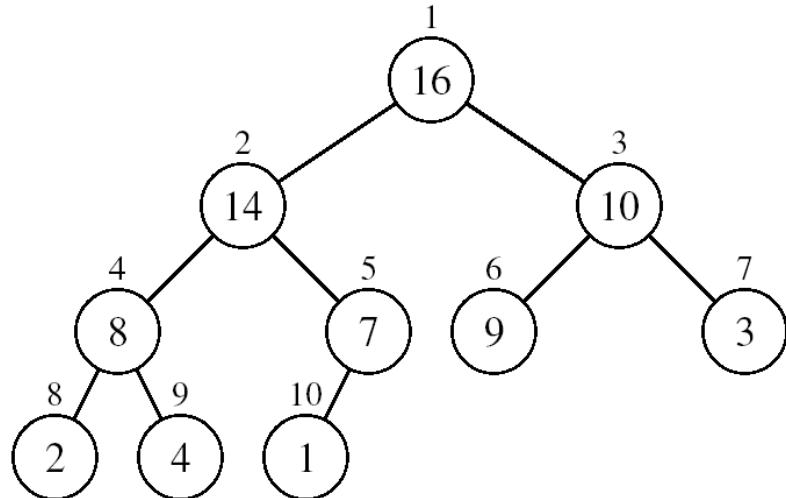
- Heap é uma árvore binária com duas propriedades:
 - 1) Balanceamento: é uma árvore completa, com a eventual exceção do último nível, onde as folhas estão sempre nas posições mais à esquerda.
 - 2) Estrutural: o valor armazenado em cada nó não é menor que os de seus filhos.

Exemplo:



- Observação: há também o caso análogo, em que o valor de cada nó não é maior que os de seus filhos.

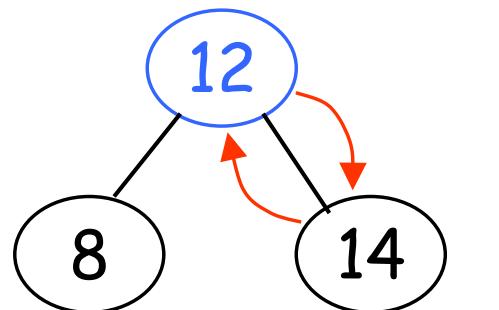
Representação de heap com vetor



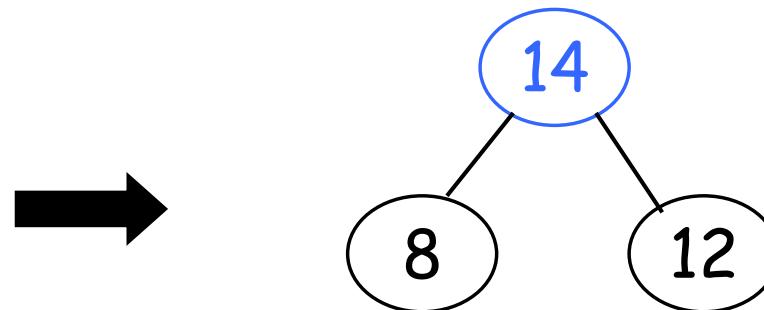
- Armazenamento de um *heap* com n elementos em um vetor v :
 - A raiz está em $v[1]$
 - O filho esquerdo de $v[i]$ é $v[2i]$
 - O filho direito de $v[i]$ é $v[2i+1]$
- O pai de $v[i]$ será $v[\lfloor i/2 \rfloor]$.
- Os elementos do subvetor $v[\lfloor n/2 \rfloor + 1] .. n]$ são as folhas.
- É fácil constatar que a altura do *heap* é $\Theta(\log n)$.

Algoritmo Sift

- Dado um *heap*, suponhamos que ocorra uma alteração no valor presente na sua raiz.
- Caso ela perca a propriedade estrutural, poderá recuperá-la trocando de valor com o seu filho maior.
- Isso pode ser feito através do algoritmo *Sift*:



Nó azul não tem a
propriedade estrutural



Nó azul tem a
propriedade estrutural

- Como o filho trocado também pode perder a propriedade estrutural, será preciso chamar *Sift* para ele.

Algoritmo Sift

Reorganiza “para baixo” o *heap* alterado na posição i :

```
Sift(i, n) {
    esq = 2i;
    dir = 2i+1;
    maior = i;
    if (esq <= n && v[esq] > v[i])
        maior = esq;
    if (dir <= n && v[dir] > v[maior])
        maior = dir;
    if (maior != i) {
        aux = v[i];
        v[i] = v[maior];
        v[maior] = aux;
        Sift(maior, n);
    }
}
```

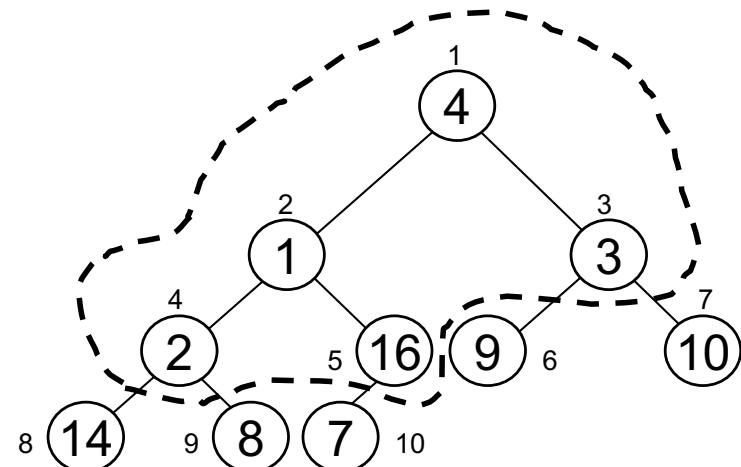
Tempo: $O(\log n)$

Exercício: reescrever *Sift* em formato não recursivo.

Transformação de um vetor em um *heap*

- O algoritmo *Build* transforma o vetor $v[1..n]$, já inicializado, em um *heap* de tamanho n .
- Como as posições entre $\lfloor n/2 \rfloor + 1$ e n são as folhas do *heap*, basta aplicar *Sift* entre as posições $\lfloor n/2 \rfloor$ e 1, nessa ordem.
- Exemplo:

v	4	1	3	2	16	9	10	14	8	7
	1	2	3	4	5	6	7	8	9	10

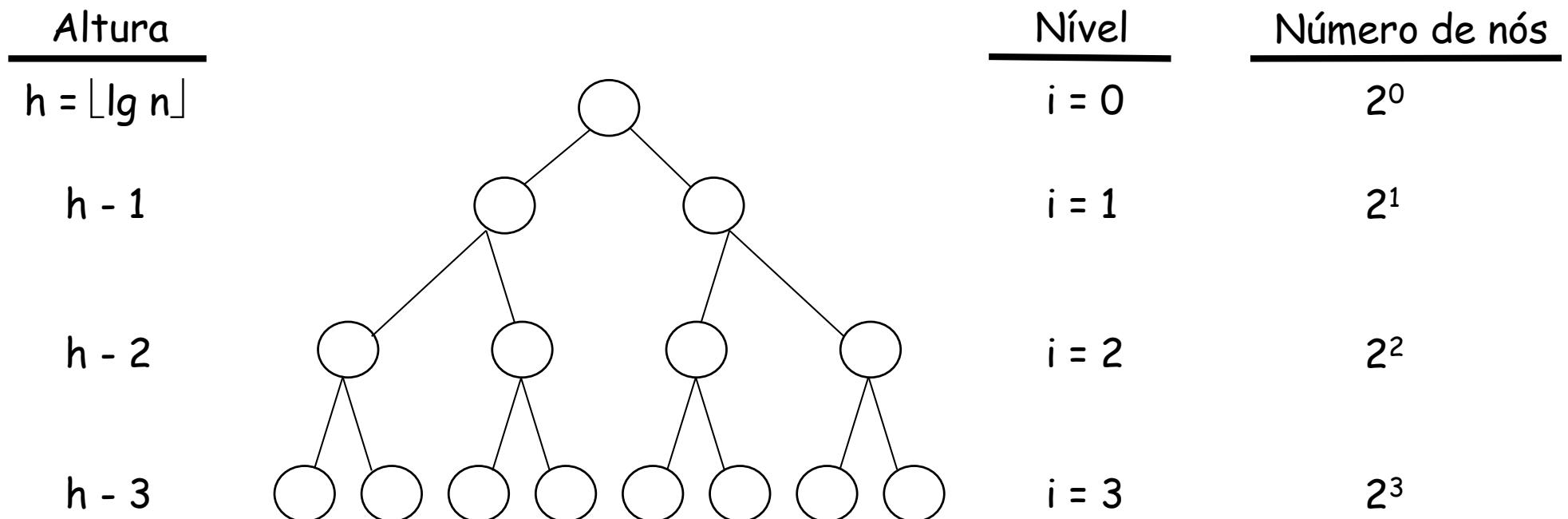


```
Build(v) {  
    for (i =  $\lfloor n/2 \rfloor$ ; i > 0; i--)  
        Sift(i, n);  
}
```

Complexidade de
tempo: $O(n \log n)$?

Complexidade de tempo de Build

- O tempo gasto por $Sift(i, n)$ é proporcional à altura do nó i .
- O pior caso ocorre com a árvore completa:



$$T(n) = 2^0h + 2^1(h-1) + \dots + 2^h(h-h)$$

$$T(n) = \sum_{i=0}^h 2^i(h-i)$$

Complexidade de tempo de Build

Tempo de pior caso, correspondente a uma árvore completa com n nós e altura h :

Multiplicando o numerador e o denominador por 2^h :

Troca de variáveis ($k = h - i$):

Sabemos que $h = \lg n$ e que essa somatória é menor que a correspondente somatória até ∞ :

Sabemos também que essa somatória é menor que 2:

$$T(n) = \sum_{i=0}^h 2^i (h-i)$$

$$= \sum_{i=0}^h \frac{h-i}{2^{h-i}} 2^h$$

$$= 2^h \sum_{k=0}^h \frac{k}{2^k}$$

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

$$= \Theta(n)$$

Filas de prioridade

- Fila de prioridade é um tipo abstrato de dados com as seguintes operações:
 - *Max* (ou *Min*): retorna o elemento com prioridade máxima (ou mínima) presente na fila
 - *ExtractMax* (ou *ExtractMin*): extrai e retorna o elemento de prioridade máxima (ou mínima) presente na fila
 - *Modify(k, x)* : atribui prioridade x ao elemento que esteja na posição k da fila
 - *Insert(x)* : insere na fila um elemento com prioridade x
- *Heap* é uma boa implementação para filas de prioridade.
- Supomos que o *heap* utilize um vetor v , e que a variável *size* armazene o seu tamanho corrente.

Operação Max



- Passos:
 - Basta retornar o elemento armazenado na primeira posição do *heap*.

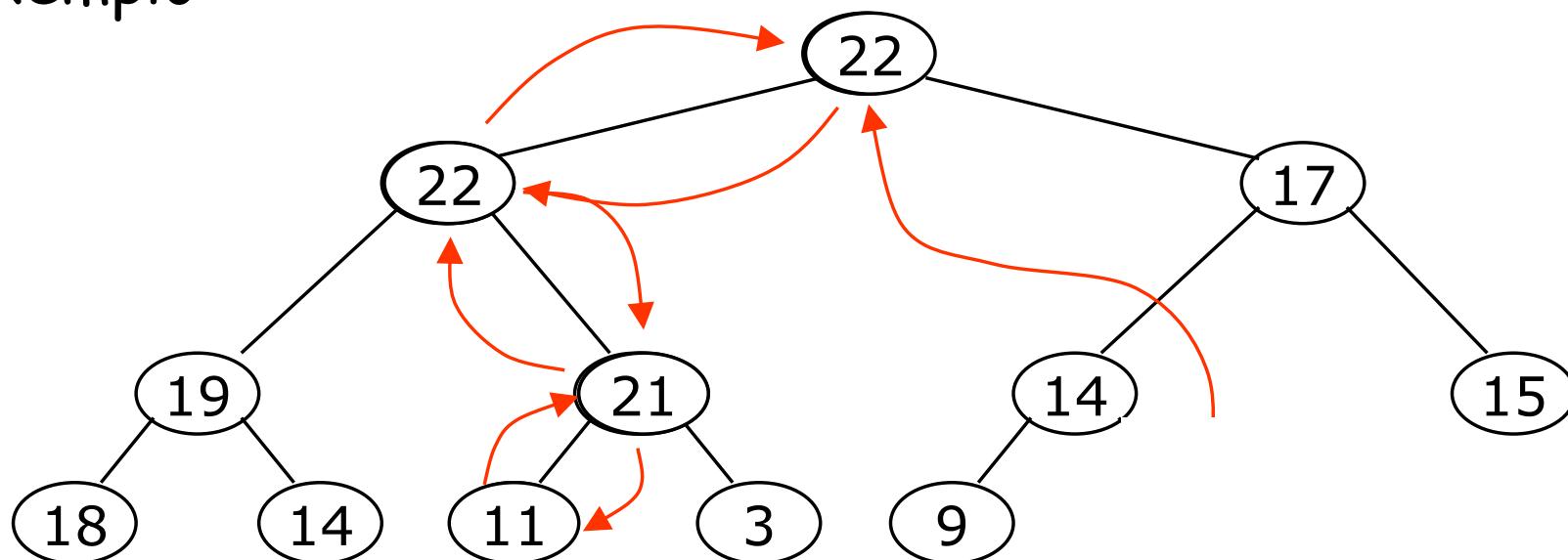
```
Max () {  
    return v[1];  
}
```

Tempo: *constante*

Um *heap* com operação *Min* é análogo.

Remoção da raiz

- Remover a raiz equivale a extrair o elemento de prioridade máxima presente no *heap*.
- Em seguida, colocaremos em seu lugar a última folha e recuperaremos a propriedade estrutural com a aplicação de vários *Sifts*.
- Exemplo:



Operação *ExtractMax*

- Passos:
 - 1) Substituir a raiz pelo último elemento do *heap*.
 - 2) Decrementar o seu tamanho atual.
 - 3) Chamar *Sift* desde a raiz.
- ```
ExtractMax () {
 if (size < 1)
 Erro("heap underflow") ;
 else {
 max = v[1] ;
 v[1] = v[size--] ;
 Sift(1, size) ;
 return max ;
 }
}
```
- Tempo: *logarítmico*

# Operação *Modify(k, x)*

## ■ Passos:

- 1) Modificar a prioridade da posição k.
- 2) Se a propriedade estrutural for perdida, ir trocando os elementos para cima ou para baixo da árvore, até “consertar” o heap.

```
Modify(k, x) {
 if (k > size || k < 1)
 Erro("Index error");
 else {
 v[k] = x;
 while (k > 1 && v[$\lfloor k/2 \rfloor$] < v[k]) { //conserta para cima
 aux = v[k];
 v[k] = v[$\lfloor k/2 \rfloor$];
 v[$\lfloor k/2 \rfloor$] = aux;
 k = $\lfloor k/2 \rfloor$;
 }
 Sift(k, size); // ou conserta para baixo
 }
}
```

Tempo: *logarítmico*

# Operação *Insert(x)*

---

- Passos:
  - 1) Aumentar uma posição no final do *heap*.
  - 2) Chamar *Modify* nessa posição, que equivale a inserir um elemento com a prioridade *x*.

```
Insert(x) {
 Modify(++size, x);
}
```

Tempo: *logarítmico*

# Sumário

---

- A tabela abaixo indica as complexidades de tempo das operações de uma fila de prioridades implementada com um *heap* :

| <u>Operação</u>                   | <u>Tempo</u> |
|-----------------------------------|--------------|
| <i>Build</i>                      | Linear       |
| <i>Max (ou Min)</i>               | Constante    |
| <i>ExtractMax (ou ExtractMin)</i> | Logarítmico  |
| <i>Modify</i>                     | Logarítmico  |
| <i>Insert</i>                     | Logarítmico  |

# Exercícios

---

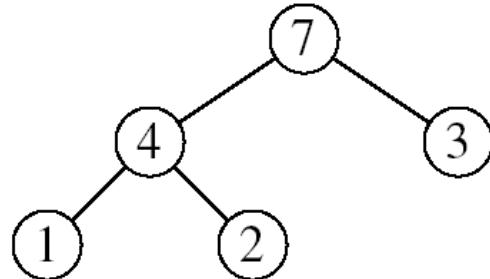
- Implemente uma fila de prioridades utilizando lista ligada e calcule a complexidade de tempo de cada uma das suas operações.
- Compare esses resultados com a implementação que utiliza *heap*.

# *HeapSort* (Williams, 1964)

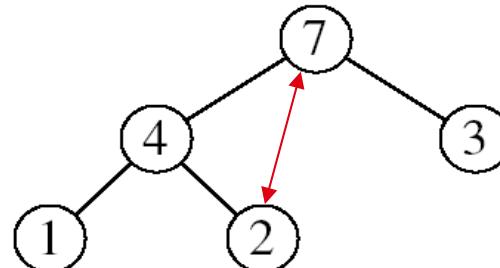
- A estrutura de *heap* permite a elaboração de um eficiente algoritmo de ordenação.
- Ideia:
  - 1) Transformar o vetor inicial em um *heap*.
  - 2) Laço de repetição:
    - a) Trocar a raiz (elemento de valor máximo) com o último elemento do *heap*.
    - b) Desconsiderar esse último elemento.
    - c) Chamar *Sift* para os demais elementos do vetor.

# Exemplo

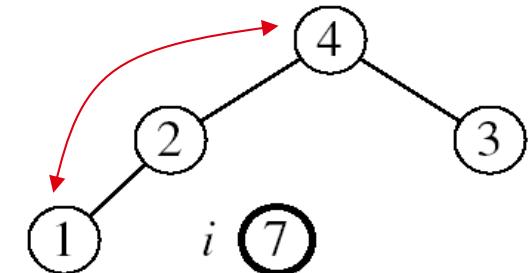
Heap já construído:



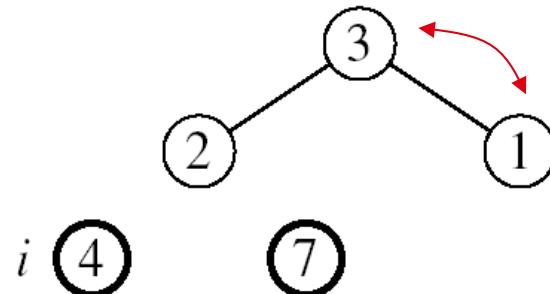
|   |   |   |   |   |
|---|---|---|---|---|
| 7 | 4 | 3 | 1 | 2 |
|---|---|---|---|---|



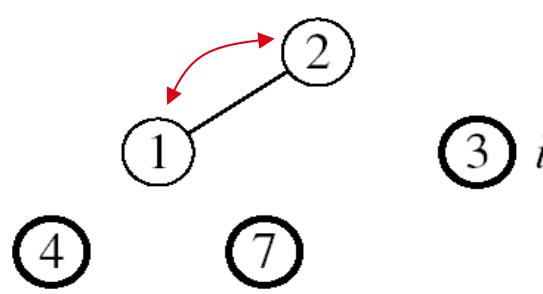
Sift(1, 4)



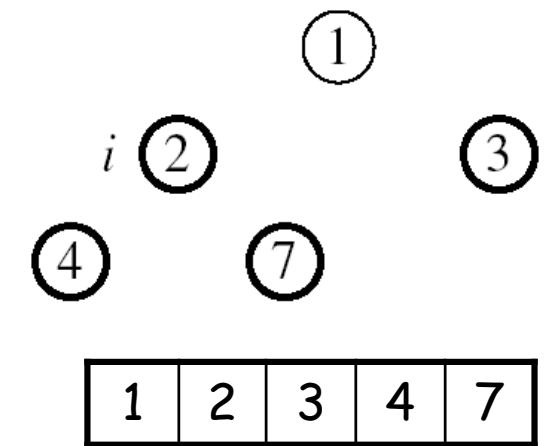
Sift(1, 3)



i 4  
Sift(1, 2)



i 7  
Sift(1, 1)



|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |
|---|---|---|---|---|

# Algoritmo HeapSort

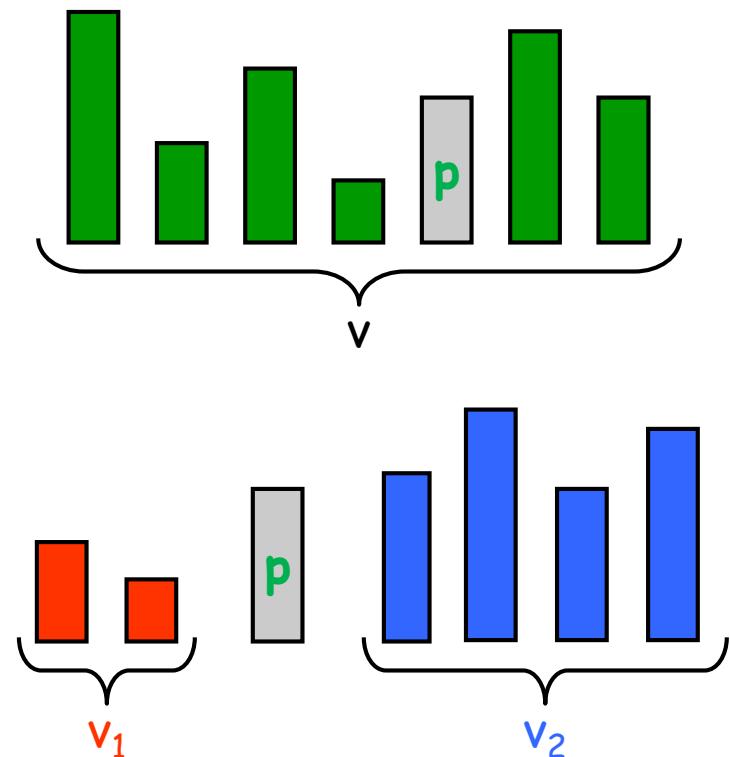
```
HeapSort(v) {
 Build(v);
 for (i=n; i>1; i--) {
 aux = v[i];
 v[i] = v[1];
 v[1] = aux;
 Sift(1, i-1);
 }
}
```

- Tempo de pior caso:
  - *Build*:  $\Theta(n)$
  - $n-1$  *Sifts*:  $\Theta(n \cdot \log n)$
- Total:  $\Theta(n \cdot \log n)$

Dentre os algoritmos de ordenação baseados em comparação, *HeapSort* é ótimo em termos de complexidade de tempo e em termos de complexidade de espaço extra

# QuickSort (Hoare, 1961)

- Na prática, QuickSort é o algoritmo de ordenação mais rápido.
- Também segue o paradigma da *Divisão-e-Conquista*.
- Divisão:
  - 1) Escolha um elemento  $p$  para ser o pivô em  $v$ .
  - 2) Particione  $v - \{p\}$  em dois grupos distintos:
    - $v_1 = \{x \in v - \{p\} \mid x < p\}$
    - $v_2 = \{x \in v - \{p\} \mid x \geq p\}$
- Conquista: ordene recursivamente  $v_1$  e  $v_2$ .
- Combinação: junte  $v_1$ ,  $p$  e  $v_2$  para obter  $v$  ordenado.



# Algoritmo básico para o *QuickSort*

---

```
QuickSort(min, max) {
 if (min < max) {
 p = Partition(min, max);
 Quicksort(min, p-1);
 Quicksort(p+1, max);
 }
}
```

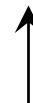
- Pontos-chave: a escolha do pivô e o algoritmo de particionamento.
- Há várias técnicas eficientes.

# Um possível particionamento

- Escolha como pivô o primeiro elemento do vetor.
- Começando da esquerda, encontre o primeiro elemento do vetor igual ou maior que o pivô (ou seja, um valor que deverá ir para o lado direito do vetor).
- Vindo do direita, encontre o primeiro elemento menor que o pivô (idem: deverá ir para o lado esquerdo).
- Troque esses dois elementos.
- Continue o mesmo procedimento até que os pontos de busca se encontrem em alguma posição do vetor.
- No final, troque o pivô com o último valor encontrado vindo da direita.

# Exemplo

- Escolha do pivô:      **4 3 6 9 2 4 3 1 2 1 4 9 3 5 6**
- Busca:                    **4 3 6 9 2 4 3 1 2 1 4 9 3 5 6**
- Troca:                    **4 3 3 9 2 4 3 1 2 1 4 9 6 5 6**
- Busca:                    **4 3 3 9 2 4 3 1 2 1 4 9 6 5 6**
- Troca:                    **4 3 3 1 2 4 3 1 2 9 4 9 6 5 6**
- Busca:                    **4 3 3 1 2 4 3 1 2 9 4 9 6 5 6**
- Troca:                    **4 3 3 1 2 2 3 1 4 9 4 9 6 5 6**
- Busca:                    **4 3 3 1 2 2 3 1 4 9 4 9 6 5 6**
- Troca com o pivô:       **1 3 3 1 2 2 3 4 4 9 4 9 6 5 6**



Posição do pivô

# Algoritmo para particionamento

```
int Partition(left, right) {
 pivot = v[left];
 l = left + 1;
 r = right;
 while (true) {
 while (l < right && v[l] < pivot) l++;
 while (r > left && v[r] >= pivot) r--;
 if (l >= r) break;
 aux = v[l];
 v[l] = v[r];
 v[r] = aux;
 }
 v[left] = v[r];
 v[r] = pivot;
 return r;
}
```

Tempo:  $\Theta(n)$

Mostrar animação  
(o pivô será o último elemento)

# Análise de tempo do *QuickSort*

---

- $T(n)$ : tempo do *QuickSort* para ordenar  $v[1..n]$
- $T(n) = \Theta(n) + T(i) + T(n-i-1)$ , onde  $0 \leq i < n$  (obs.:  $i = |v_1|$ )
- Melhor caso:  $i = n/2$  (balanceamento perfeito)
  - $T(n) = T(n/2) + T((n/2)-1) + \Theta(n) \approx 2T(n/2) + \Theta(n)$
  - $T(n) = \Theta(n \cdot \log n)$
- Pior caso:  $i = 0$  ou  $i = n-1$ 
  - $T(n) = T(n-1) + \Theta(n)$
  - $T(n) = \Theta(n^2)$
- No pior caso, o *QuickSort* é quadrático!!
- Comprove essas complexidades resolvendo as recorrências.

# Casos práticos

---

- Embora haja casos em que o desempenho do QuickSort seja quadrático, seu tempo médio é  $O(n \cdot \log n)$ .
- Além disso, as constantes são tão boas que o QuickSort é o melhor algoritmo de ordenação conhecido.
- No mundo real, a grande maioria das ordenações é realizada através deste algoritmo, principalmente quando  $n$  é grande.
- Para se encontrar um particionamento ótimo, seria preciso escolher a mediana como pivô. E para encontrar a mediana, seria necessário ordenar o vetor...
- Na verdade, em determinadas condições é possível encontrar a mediana em tempo  $\Theta(n)$ , mas com um algoritmo nada trivial. Isso garantiria tempo total de pior caso  $\Theta(n \cdot \log n)$ .

# Mediana de três



- Uma alternativa mais simples é encontrar a chamada *mediana de três*.
- Comparam-se três elementos do vetor: o primeiro, o central e o último:
  - O pivô será a mediana entre os três.
  - Este valor é trocado com o que estava na posição inicial e o particionamento é feito do mesmo modo anterior.
- Quando esta técnica é utilizada, tornam-se muito raros os casos em que o *QuickSort* gasta tempo quadrático.

# Pilha de execução

- Nos piores casos do *QuickSort* (vetor ordenado, por exemplo), devido às chamadas recursivas, a pilha de execução chega a exigir espaço  $\Theta(n)$ .
  - Isso ocorre porque pode haver até  $n$  chamadas pendentes.
- Dependendo do tamanho do vetor, esse espaço pode se esgotar, e o programa será abortado...
- Através de uma pequena alteração no código do *QuickSort*, é possível eliminar uma das chamadas recursivas:
  - A ideia é chamar a recursão apenas na menor metade de cada subvetor.
  - Desse modo, cada subvetor tratado na pilha de execução será menor que a metade do subvetor imediatamente anterior.
  - Isso garante que a pilha de execução tenha tamanho  $O(\log n)$ .

# QuickSort com uma única recursão

```
QuickSort(min, max) {
 while (min < max) {
 p = Partition(min, max);
 if (p-min < max-p) {
 QuickSort(min, p-1);
 min = p+1;
 }
 else {
 QuickSort(p+1, max);
 max = p-1;
 }
 }
}
```

- O tamanho da pilha de execução passa a ser  $O(\log n)$ : o caso que consome mais espaço é aquele em que o vetor é sempre quebrado em duas metades iguais, gastando tempo  $\Theta(n \cdot \log n)$ .
- Por outro lado, nos casos em que o algoritmo gasta tempo  $\Theta(n^2)$ , a pilha de execução cresce apenas  $\Theta(1)$ .

# Análise do caso médio

- Por questão de simplicidade, supomos que:
  - todos os elementos do vetor são distintos;
  - todos os particionamentos do vetor  $(0:n-1, 1:n-2, \dots, n-2:1, n-1:0)$  são equiprováveis.
- Portanto, a probabilidade de que ocorra cada um desses particionamentos será  $1/n$ .
- Desse modo, o tempo médio esperado será:

$$T(n) = \frac{1}{n} \sum_{k=0}^{n-1} [T(k) + T(n-1-k)] + \Theta(n)$$

$$= \frac{2}{n} \sum_{k=0}^{n-1} T(k) + \Theta(n)$$

# Análise do caso médio

$$T(n) = \frac{2}{n} \sum_{k=0}^{n-1} T(k) + \Theta(n)$$

- Vamos provar por indução em  $n$  que  $T(n) \leq a.n.\lg n + b$ 
  - A base da indução é  $n = 1$ :  $T(1) = \Theta(1) \leq b$   
*( $T(0)=0$ ; basta escolher constantes convenientes)*
  - Hipótese indutiva: suporemos válido para  $k < n$
  - Passo da indução (demonstrar para  $n$  usando a hipótese indutiva):

$$\begin{aligned} T(n) &\leq \frac{2}{n} \sum_{k=0}^{n-1} (ak \lg k + b) + \Theta(n) \\ &\leq \frac{2}{n} \left[ b + \sum_{k=1}^{n-1} (ak \lg k + b) \right] + \Theta(n) \quad \text{Extraindo o caso } k=0 \end{aligned}$$

# Análise do caso médio

$$T(n) \leq \frac{2}{n} \left[ b + \sum_{k=1}^{n-1} (ak \lg k + b) \right] + \Theta(n)$$

$$= \frac{2}{n} \sum_{k=1}^{n-1} (ak \lg k + b) + \frac{2b}{n} + \Theta(n)$$

*Separando as parcelas*

$$= \frac{2}{n} \sum_{k=1}^{n-1} (ak \lg k + b) + \Theta(n)$$

*2b/n é inferior a Θ(n)*

$$= \frac{2}{n} \sum_{k=1}^{n-1} ak \lg k + \frac{2}{n} \sum_{k=1}^{n-1} b + \Theta(n)$$

*Distribuindo as somatórias*

# Análise do caso médio

$$T(n) \leq \frac{2}{n} \sum_{k=1}^{n-1} ak \lg k + \frac{2}{n} \sum_{k=1}^{n-1} b + \Theta(n)$$

$$= \frac{2a}{n} \sum_{k=1}^{n-1} k \lg k + \frac{2b}{n} (n-1) + \Theta(n)$$

$$b+b+\dots+b = b(n-1)$$

$$\leq \frac{2a}{n} \boxed{\sum_{k=1}^{n-1} k \lg k} + 2b + \Theta(n)$$

Como  $n-1 < n$ , então  
 $2b(n-1)/n < 2b$

$$\leq \frac{2a}{n} \left( \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + 2b + \Theta(n)$$

Será provado  
mais tarde...

# Análise do caso médio

$$T(n) \leq \frac{2a}{n} \left( \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + 2b + \Theta(n)$$

$$= an \lg n - \frac{a}{4} n + 2b + \Theta(n)$$

$$= an \lg n + b + \left( \Theta(n) + b - \frac{a}{4} n \right)$$

$$\leq an \lg n + b$$

*Escolhemos uma constante a suficientemente grande para que  $an/4$  seja maior que  $\Theta(n)+b$*

# Análise do caso médio

- Ainda falta calcular aquela somatória...

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k$$

*Quebramos a somatória em duas partes*

$$\leq \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg n$$

*$\lg k$  na segunda somatória é limitado por  $\lg n$*

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

*Tiramos  $\lg n$  para fora da somatória*

# Análise do caso médio

$$\sum_{k=1}^{n-1} k \lg k \leq \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

$$\leq \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg(n/2) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

*lg k na primeira somatória é limitado por lg n/2*

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k(\lg n - 1) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

*lg n/2 = lg n - 1*

$$= (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

*Tiramos (lg n - 1) para fora da somatória*

# Análise do caso médio

$$\sum_{k=1}^{n-1} k \lg k \leq (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

$$= \lg n \sum_{k=1}^{\lceil n/2 \rceil - 1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Distribuição do  $(\lg n - 1)$

$$= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

Combinamos duas somatórias

$$= \lg n \left( \frac{(n-1)n}{2} \right) - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

Somatória de P.A.

# Análise do caso médio

$$\sum_{k=1}^{n-1} k \lg k \leq \left( \frac{(n-1)n}{2} \right) \lg n - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

$$\leq \frac{1}{2} [n(n-1)] \lg n - \sum_{k=1}^{n/2-1} k$$

*Troca de limite superior na  
segunda somatória*

$$\leq \frac{1}{2} [n(n-1)] \lg n - \frac{1}{2} \left( \frac{n}{2} \right) \left( \frac{n}{2} - 1 \right)$$

*Somatória de P.A.*

$$\leq \frac{1}{2} (n^2 \lg n - n \lg n) - \frac{1}{8} n^2 + \frac{n}{4}$$

# Análise do caso médio

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2} (n^2 \lg n - n \lg n) - \frac{1}{8} n^2 + \frac{n}{4}$$

$$\leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \quad \text{quando } n \geq 2$$

- Demonstramos que  $T(n) \leq a.n.\lg n + b$
- Portanto, o tempo médio do *Quicksort* é  $O(n.\log n)$

# *QuickSort versus MergeSort*

---

- Tanto o *QuickSort* como o *MergeSort* :
  - são  $O(n \cdot \log n)$  no caso médio;
  - possuem duas chamadas recursivas.
- Além disso, há casos em que o *QuickSort* é quadrático, enquanto o *MergeSort* é sempre  $\Theta(n \cdot \log n)$ .
- Por que o *QuickSort* é mais rápido que o *MergeSort* ?
  - Seu laço interno consiste apenas de incrementos, decrementos, comparações e algumas atribuições (são operações rápidas).
  - Não possui tanto processamento como o *MergeSort*, que manipula subvetores.

# Exercícios

```
Partition2(left, right) {
 x = v[right];
 i = left - 1;
 for (j=left; j<right; j++)
 if (v[j] <= x) {
 aux = v[++i];
 v[i] = v[j];
 v[j] = aux;
 }
 aux = v[i+1];
 v[i+1] = v[right];
 v[right] = aux;
 return i+1;
}
```

- O que faz este algoritmo de particionamento do *QuickSort*?
- Ele funciona?

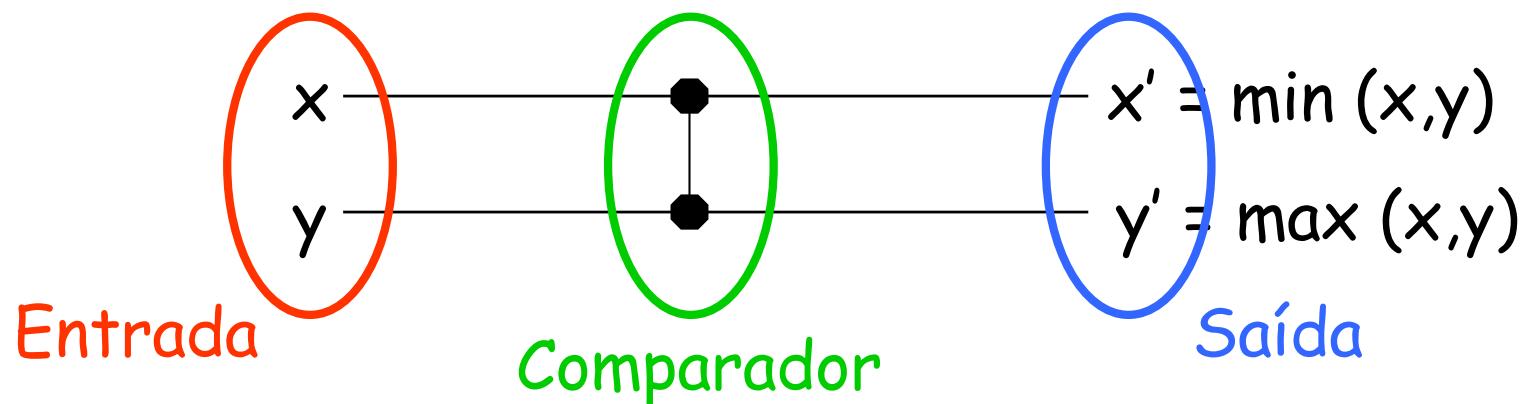
# Exercícios

---

- Implemente todos os algoritmos apresentados:  
*BubbleSort, SelectionSort, InsertionSort, MergeSort, RadixSort, HeapSort e QuickSort.*
- Crie um arquivo com milhares de números inteiros gerados aleatoriamente, e através dele compare os tempos de execução desses algoritmos.

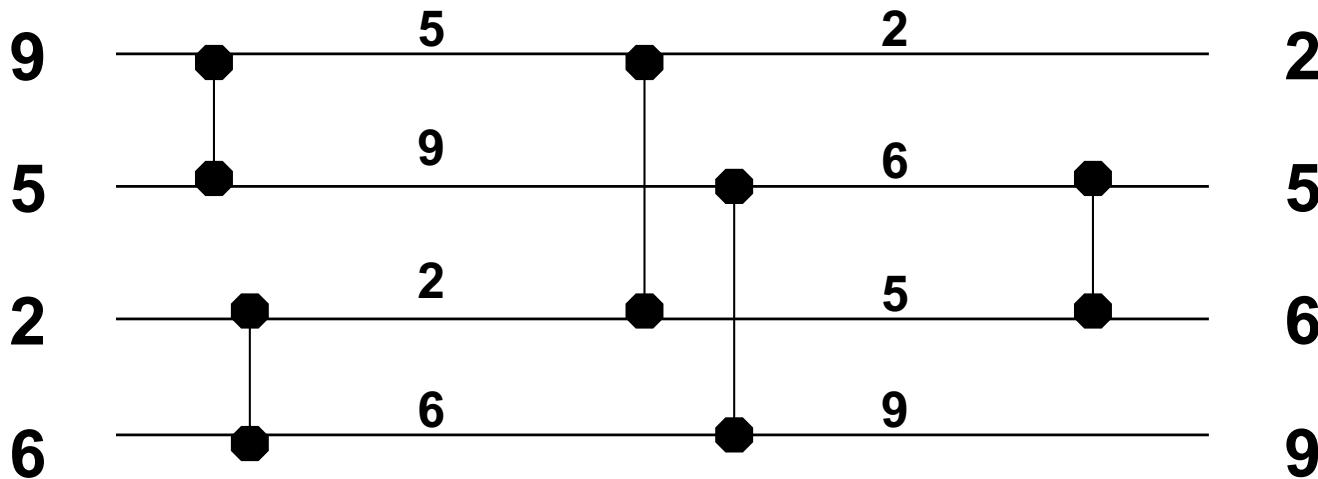
# Redes de comparação

- Uma rede (ou circuito) de comparação é composta de ligações e comparadores.
- As ligações transmitem valores de um lugar para outro.
- Os comparadores recebem duas entradas e produzem duas saídas.
- Uma rede de comparação que sempre ordena suas entradas é chamada de *rede de ordenação*.
- Exemplo:



# Redes de ordenação

- Um exemplo:



- Evidentemente, nem todas as redes de comparação são de ordenação.
- Considerando que determinadas comparações possam ser feitas em paralelo, o nível de profundidade de uma rede é o número de passos necessários para que a saída seja calculada.

# *Princípio Zero-Um*



- Se uma rede de comparação com  $n$  entradas ordena corretamente todas as  $2^n$  possíveis sequências formadas por  $n$  bits, então também ordenará qualquer sequência com  $n$  números reais.
- Este princípio facilita a elaboração de redes de ordenação, pois basta considerar apenas as entradas binárias.
- Concretamente, vamos utilizá-lo para apresentar a rede de ordenação bitônica (*Batcher*, 1968).

# Demonstração do Princípio Zero-Um

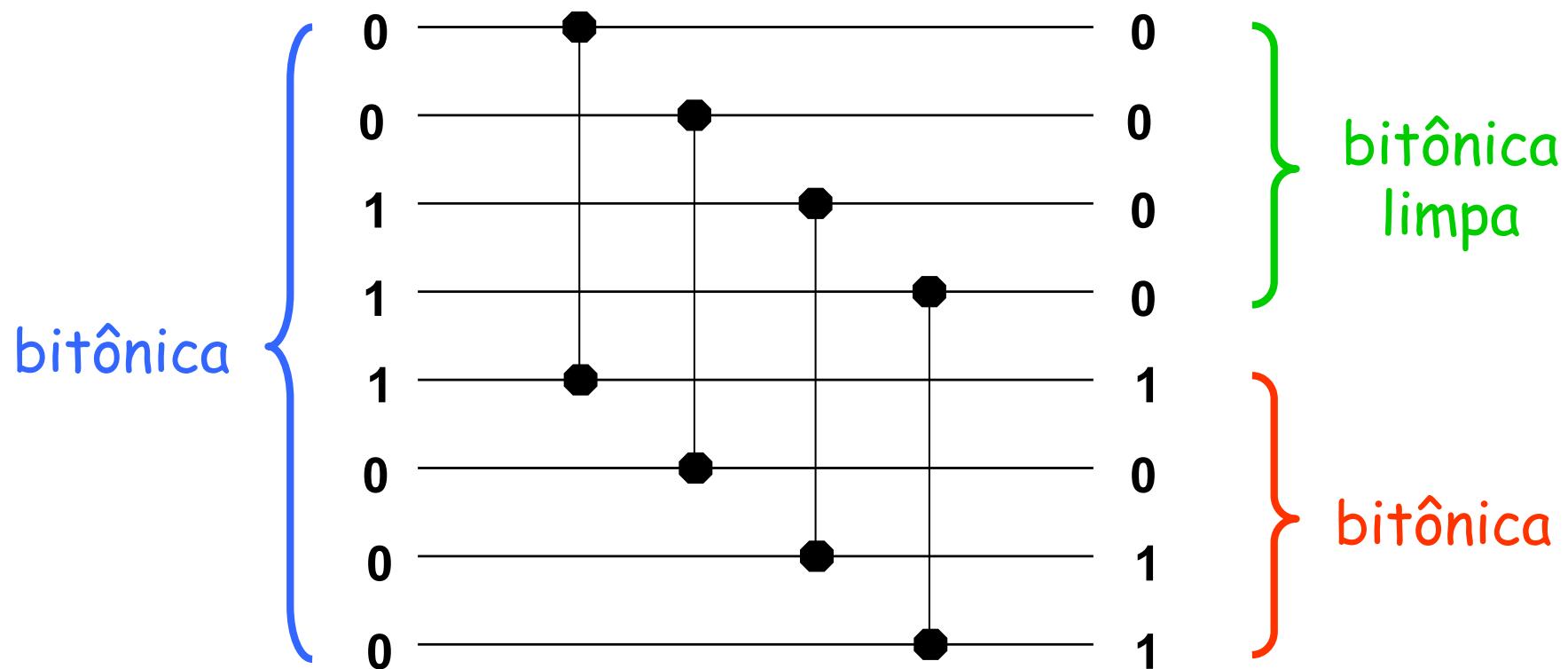
- Numa função  $f$  monotônica crescente, se  $x \leq y$  então  $f(x) \leq f(y)$ .
- Neste caso,  $\min \{f(x), f(y)\} = f(\min \{x, y\})$  e  $\max \{f(x), f(y)\} = f(\max \{x, y\})$ .
- Portanto, se uma determinada rede de ordenação recebe como entrada a sequência  $\langle a_1, a_2, \dots, a_n \rangle$  e produz a saída  $\langle b_1, b_2, \dots, b_n \rangle$ , então essa mesma rede, se tiver como entrada  $\langle f(a_1), f(a_2), \dots, f(a_n) \rangle$ , irá produzir a saída  $\langle f(b_1), f(b_2), \dots, f(b_n) \rangle$ , onde  $f$  é uma função monotônica crescente.
  - Isso pode ser demonstrado por indução na profundidade da rede, usando os resultados para min e max mostrados acima.
- Suponhamos que o *Princípio Zero-Um* não seja válido. Em outras palavras, essa rede ordena corretamente todas as sequências de  $n$  bits, mas existe uma sequência não binária  $\langle a_1, a_2, \dots, a_n \rangle$ , com  $a_i < a_j$ , que, se for recebida como entrada, irá produzir uma saída onde  $a_j$  vem antes de  $a_i$ .
- Seja a função monotônica crescente definida como  $f_i(x) = 0$  se  $x \leq a_i$ , e  $f_i(x) = 1$  se  $x > a_i$ . Portanto, essa rede não ordenará corretamente a entrada  $\langle f_i(a_1), f_i(a_2), \dots, f_i(a_n) \rangle$ : contradição!

# Sequência binária bitônica

- Uma sequência binária é chamada *bitônica* se cresce monotonicamente e depois decresce, ou vice-versa.
- Únicos casos possíveis:
  - $0^i 1^j 0^k$ , com  $i, j, k \geq 0$
  - $1^i 0^j 1^k$ , com  $i, j, k \geq 0$
- Uma sequência binária bitônica pode ser vista como a juxtaposição de duas sequências binárias ordenadas, sendo que a segunda está na ordem contrária da primeira.

# Meio-limpador

- Um meio-limpador (*half-cleaner*) é uma rede de comparação com  $n$  entradas e um único nível de profundidade, onde a linha  $i$  é comparada com a linha  $i + n/2$ , para  $i = 1, 2, \dots, n/2$ . Assume-se que  $n$  é par.
- Meio-limpador de tamanho 8:

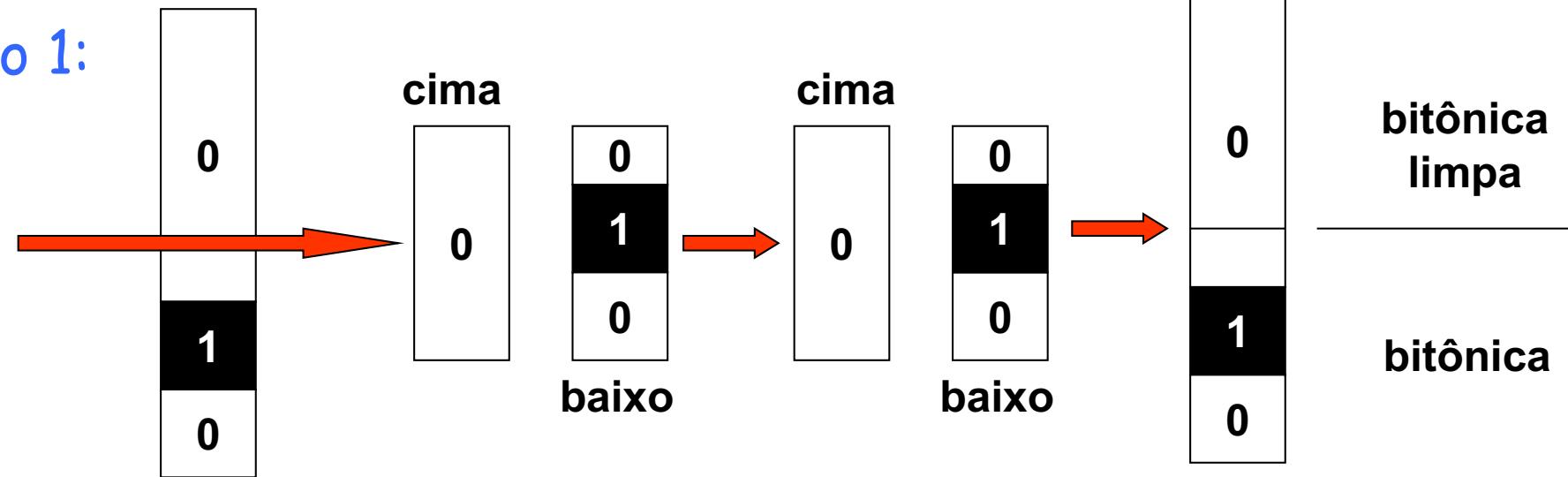


# Propriedades de um meio-limpador

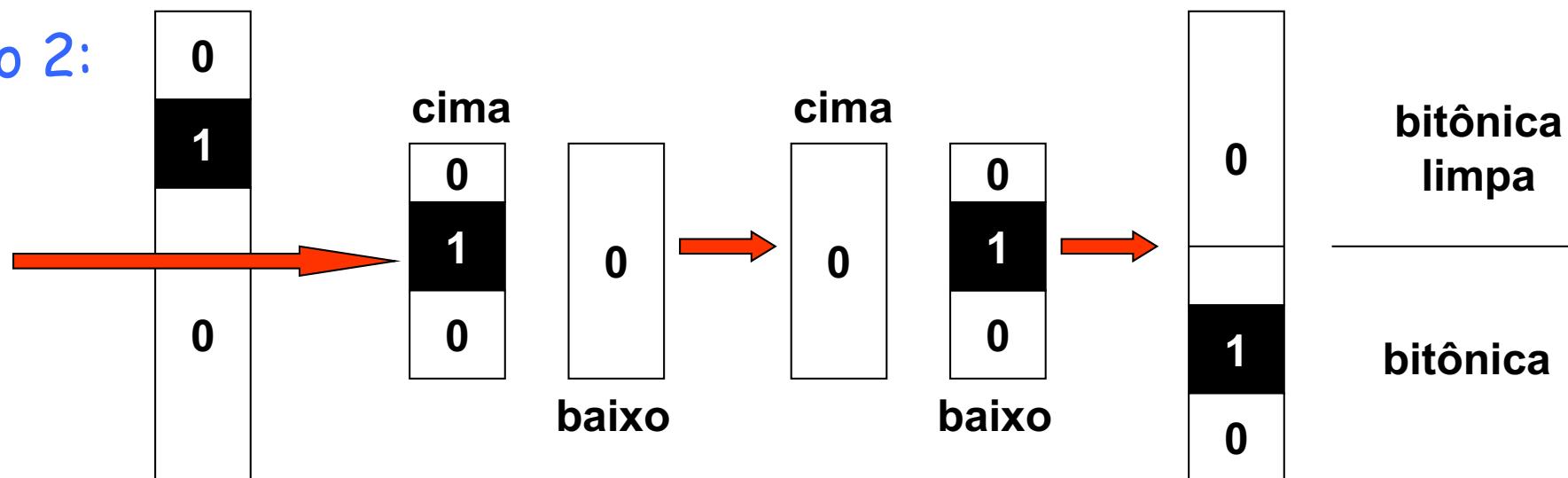
- Se um meio-limpador recebe como entrada uma sequência binária bitônica, então sua saída satisfará três propriedades:
  - 1) Pelo menos uma metade será limpa (somente 0's ou 1's).
  - 2) As metades de cima e de baixo serão bitônicas.
  - 3) Todos os elementos da metade de cima serão menores ou iguais aos elementos da metade de baixo.
- Demonstração:
  - Há 8 possíveis casos. Veremos a seguir os 4 casos que correspondem a  $0^i 1^j 0^k$ , com  $i, j, k \geq 0$ .
  - Os casos  $1^i 0^j 1^k$ , com  $i, j, k \geq 0$ , são análogos.
- As duas últimas propriedades sugerem um algoritmo de ordenação *Divisão-e-Conquista*.

# Demonstração (casos $0^i 1^j 0^k$ )

Caso 1:

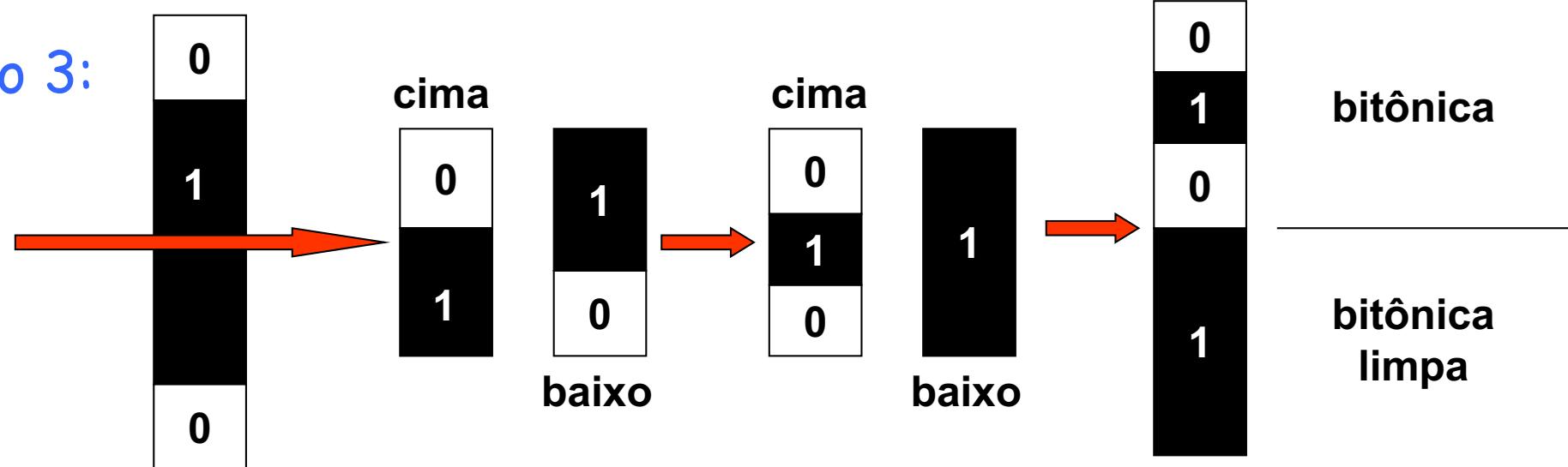


Caso 2:

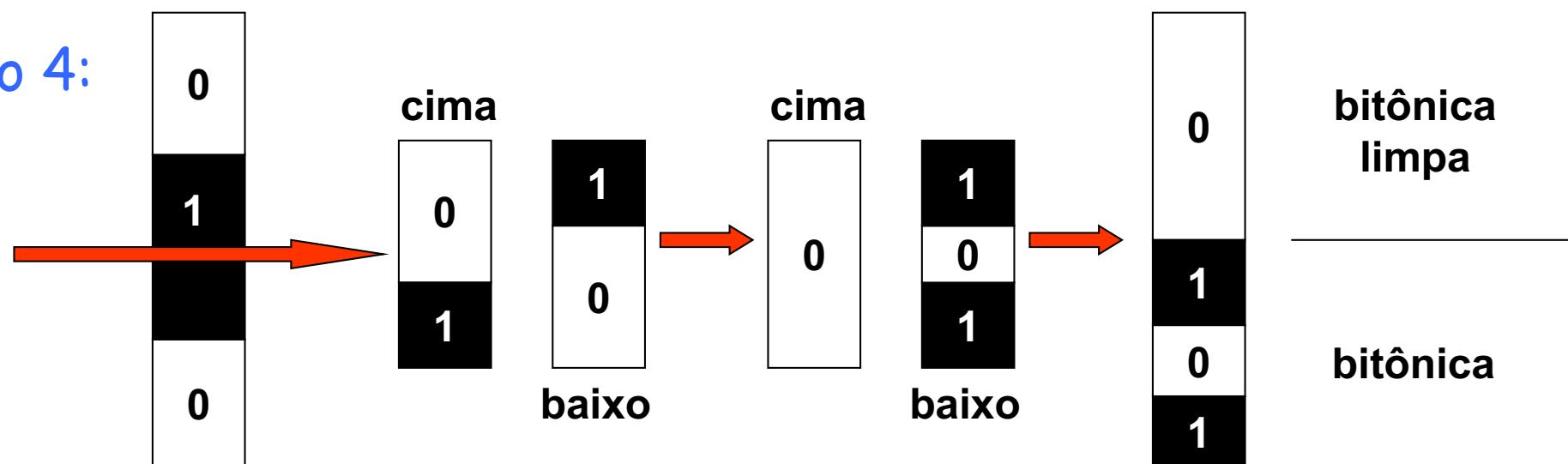


# Demonstração (casos $0^i 1^j 0^k$ )

Caso 3:



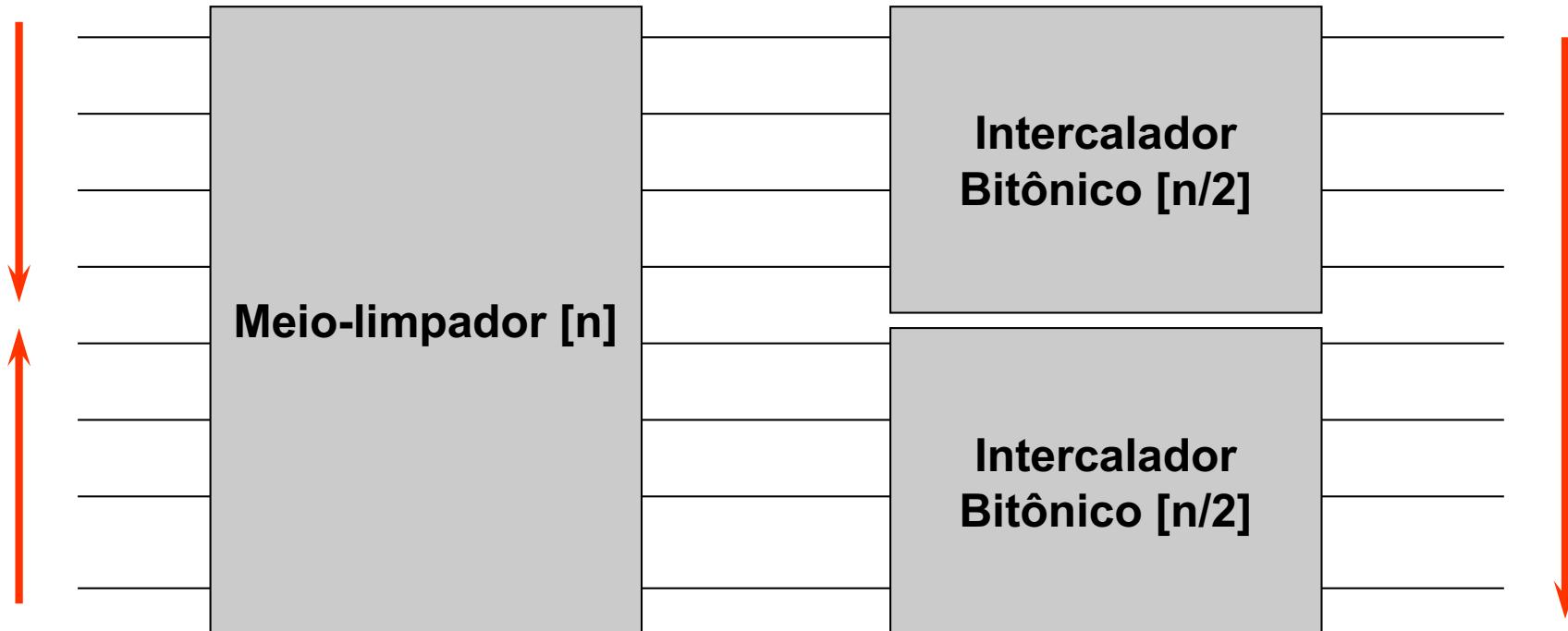
Caso 4:



# Intercalador bitônico (*bitonic merger*)

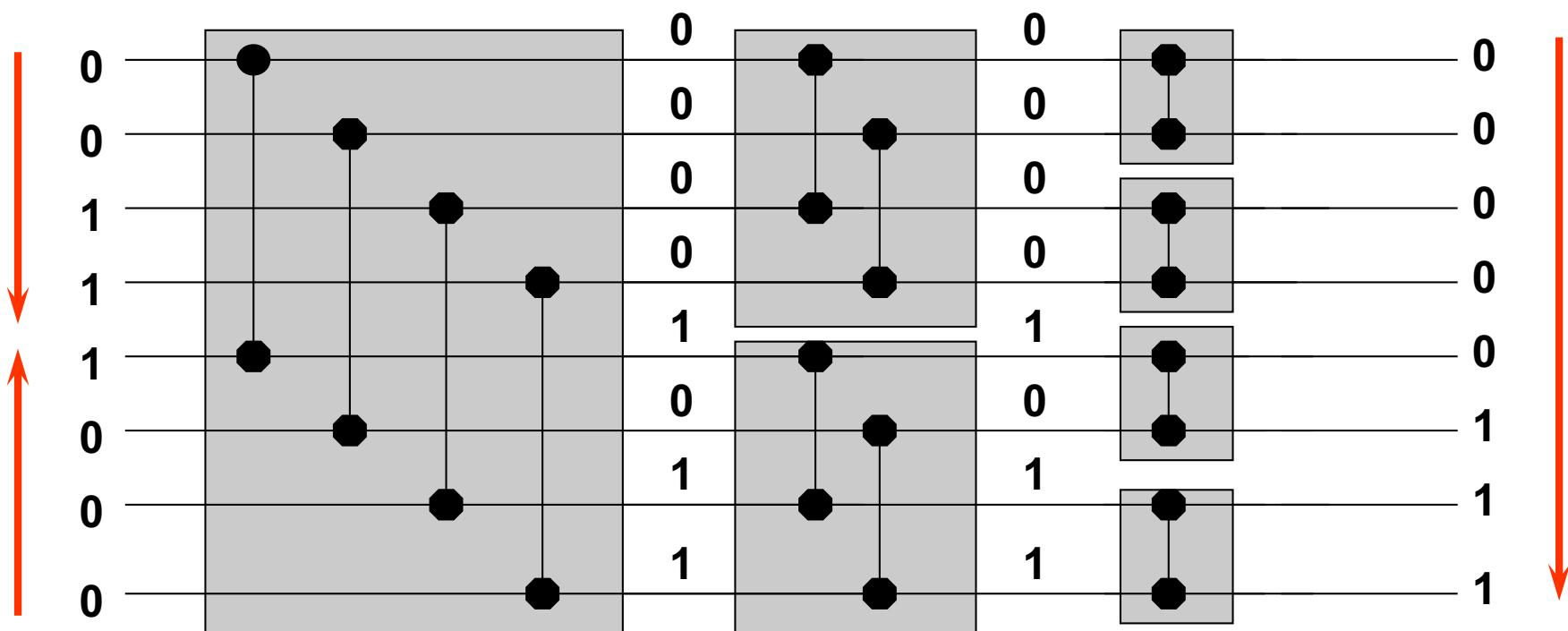
- Um intercalador (*merger*) é um circuito que recebe duas sequências ordenadas e gera uma nova sequência, também ordenada, com todos os elementos das duas primeiras.
- Através de combinações recursivas de meio-limpadores, é possível construir um intercalador bitônico com  $n$  entradas:
  - Primeiro estágio: um meio-limpador de tamanho  $n$ .
  - Fase seguinte: uso recursivo de um intercalador bitônico com  $n/2$  entradas.

# Esquema de um intercalador bitônico [n]



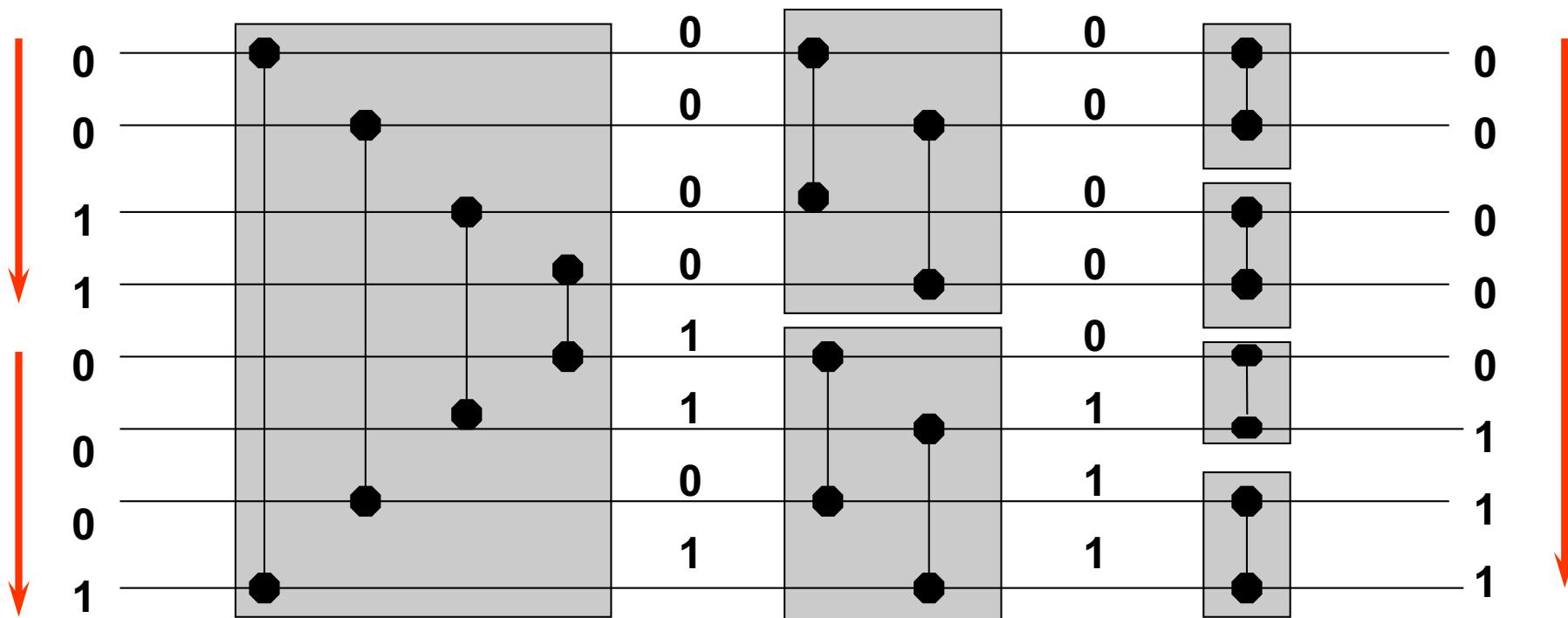
- $D(n)$ : profundidade do intercalador bitônico [n]
  - $D(1) = 0, D(2) = 1$
  - $D(n) = 1 + D(n/2)$ , se  $n = 2^k$  e  $k \geq 1$
  - Logo,  $D(n) = k = \lg n$ : este seria o tempo da intercalação.

# Intercalador bitônico [8]



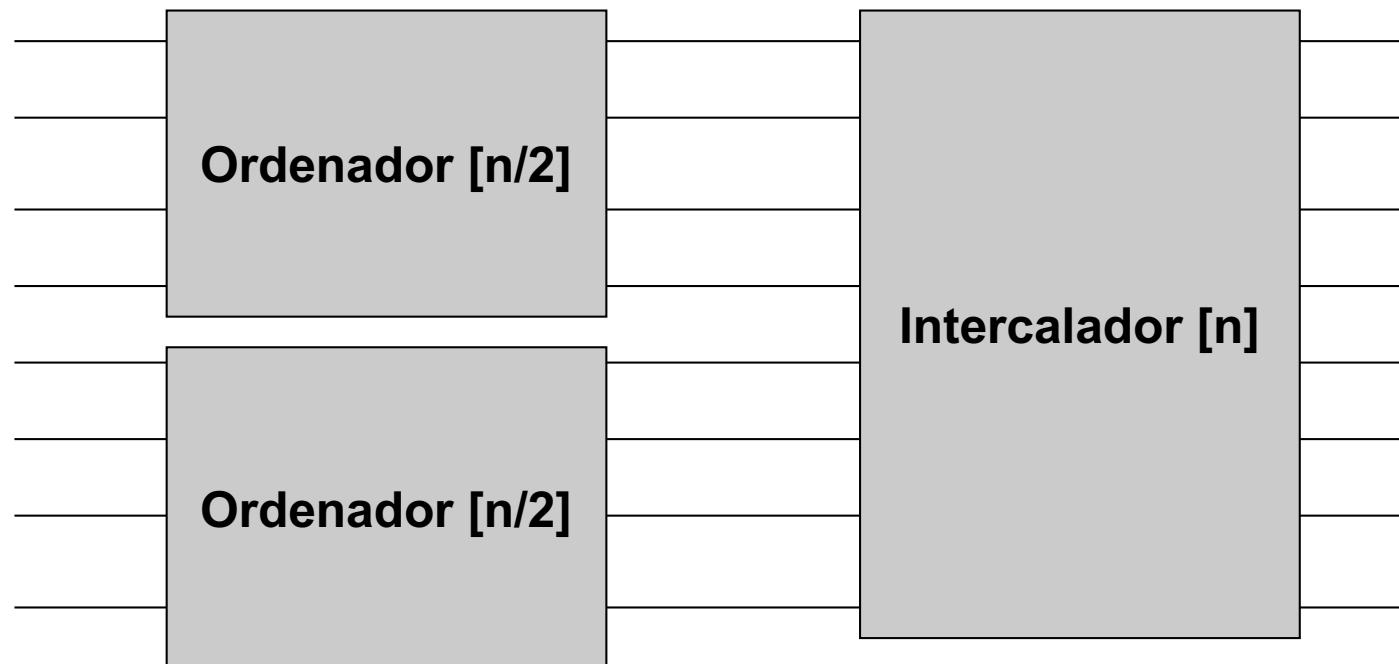
# Intercalador [8]

- É fácil transformar este circuito em um simples intercalador (*merger*).



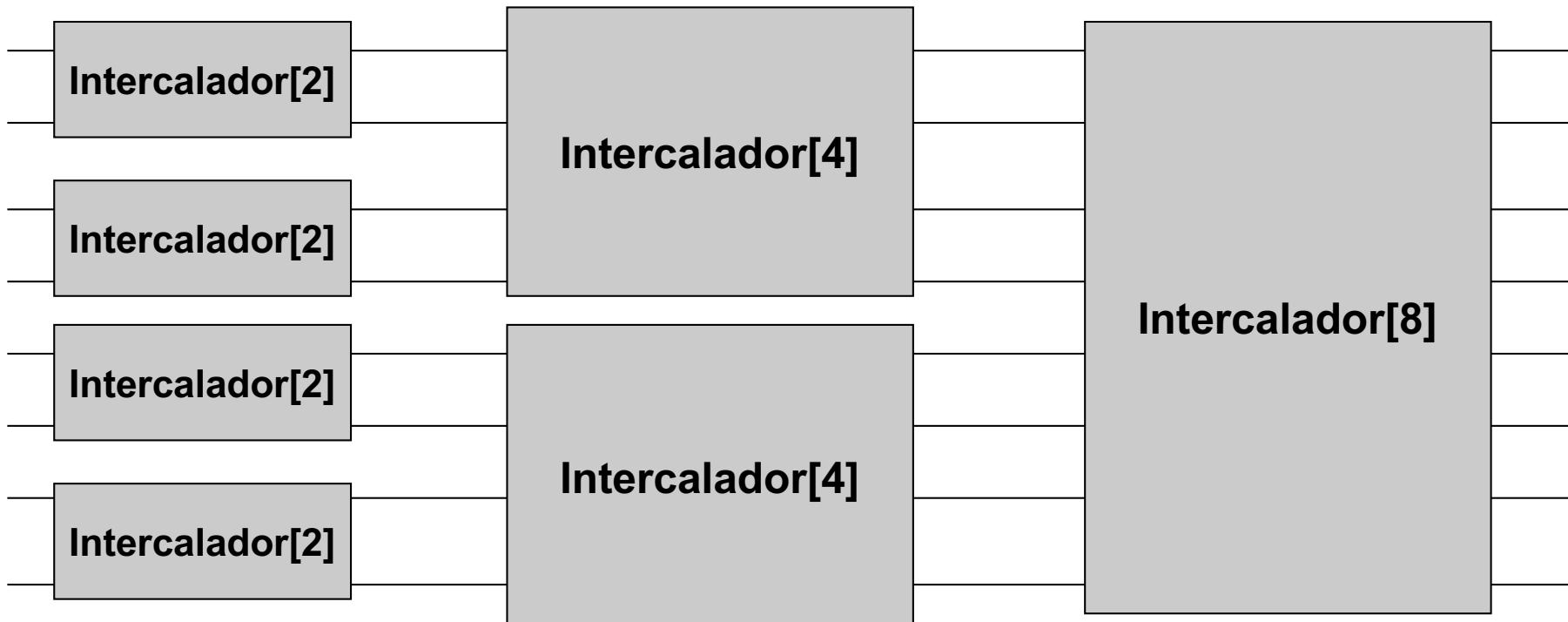
# Uma rede de ordenação

- Também de modo recursivo, podemos construir um Ordenador  $[n]$ :

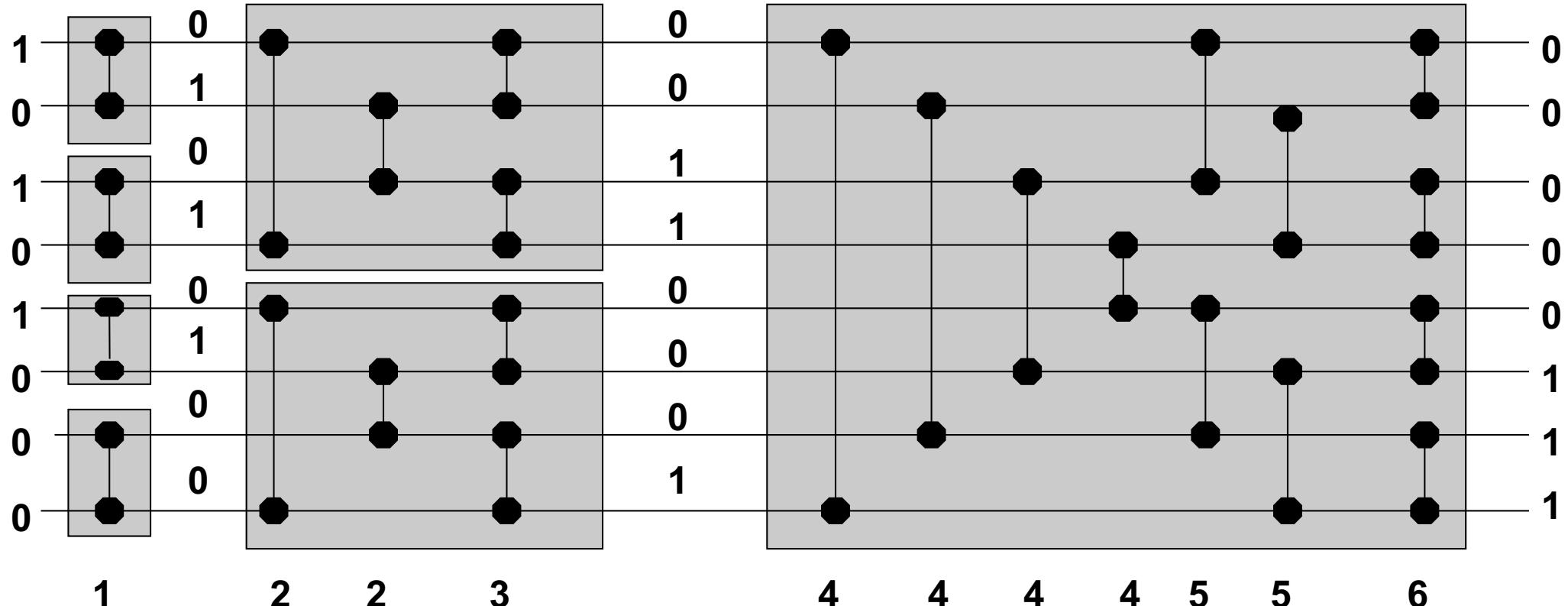


- Tempo de ordenação:  $T(n) = T(n/2) + \lg n$
- $T(n) = \Theta(\log^2 n)$

# Ordenador para n=8



# Exemplo para n=8



## ■ Exercícios:

- Desenhar uma rede de ordenação com 16 entradas.
- Verificar que essas redes também ordenam números reais.