# Métodos de Ordenação
## Parte 4

**Introdução à Ciência de Computação II**

Prof. Diego Raphael Amancio

Baseado no material do Prof. Rudinei Goularte e Prof. Thiago A. S. Pardo

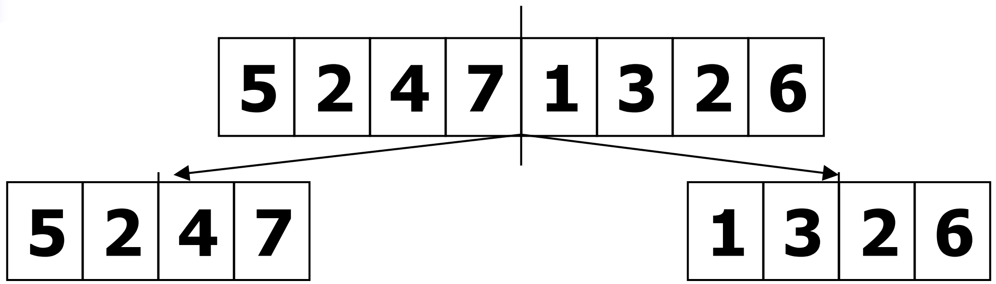
# Ordenação por Intercalação

- Revisando...
  - Também chamado merge-sort
  - Idéia básica: dividir para conquistar
    - Um vetor v é dividido em duas partes, recursivamente
    - Cada metade é ordenada e ambas são intercaladas formando o vetor ordenado
    - Usa um vetor auxiliar para intercalar

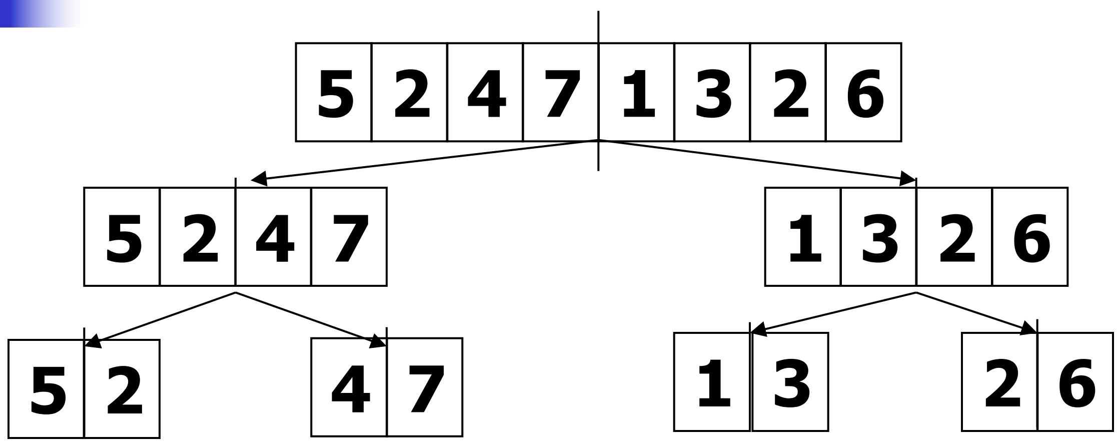
# Ordenação por Intercalação

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **5** | **2** | **4** | **7** | **1** | **3** | **2** | **6** |

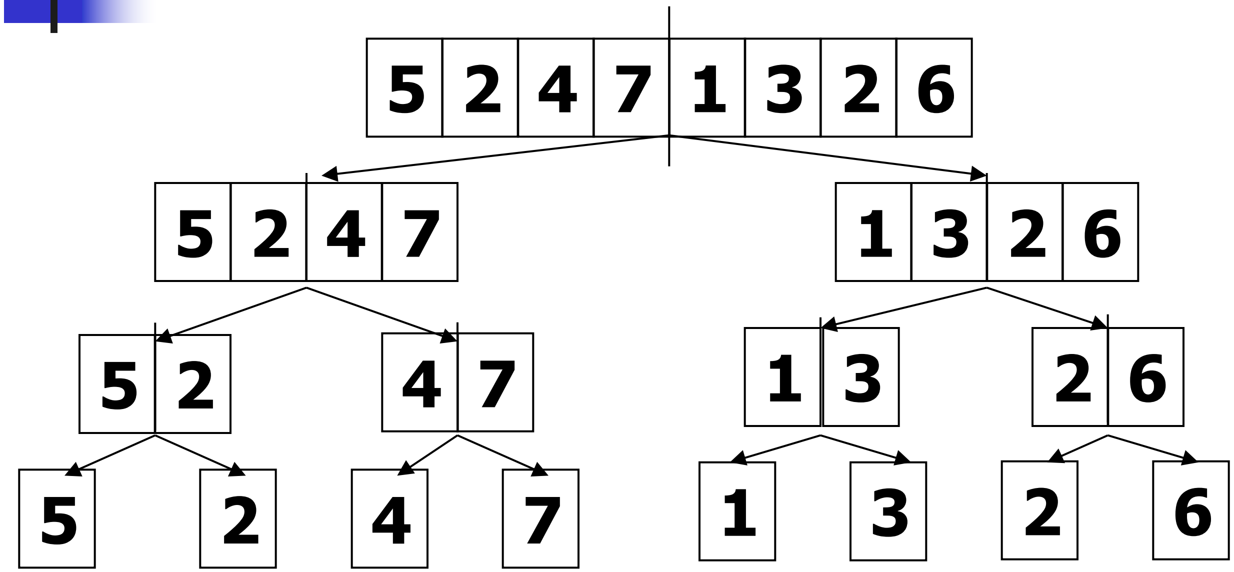
# Ordenação por Intercalação



# Ordenação por Intercalação

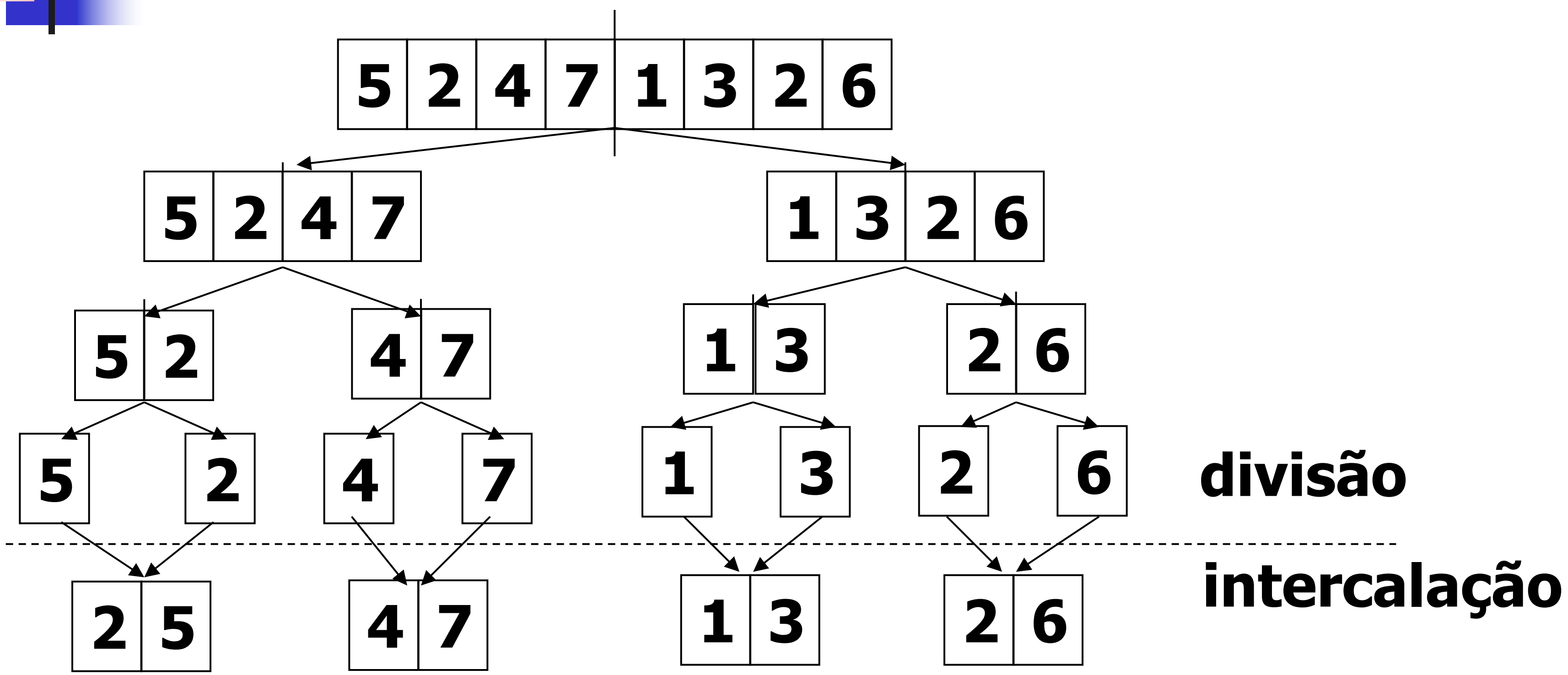


# Ordenação por Intercalação

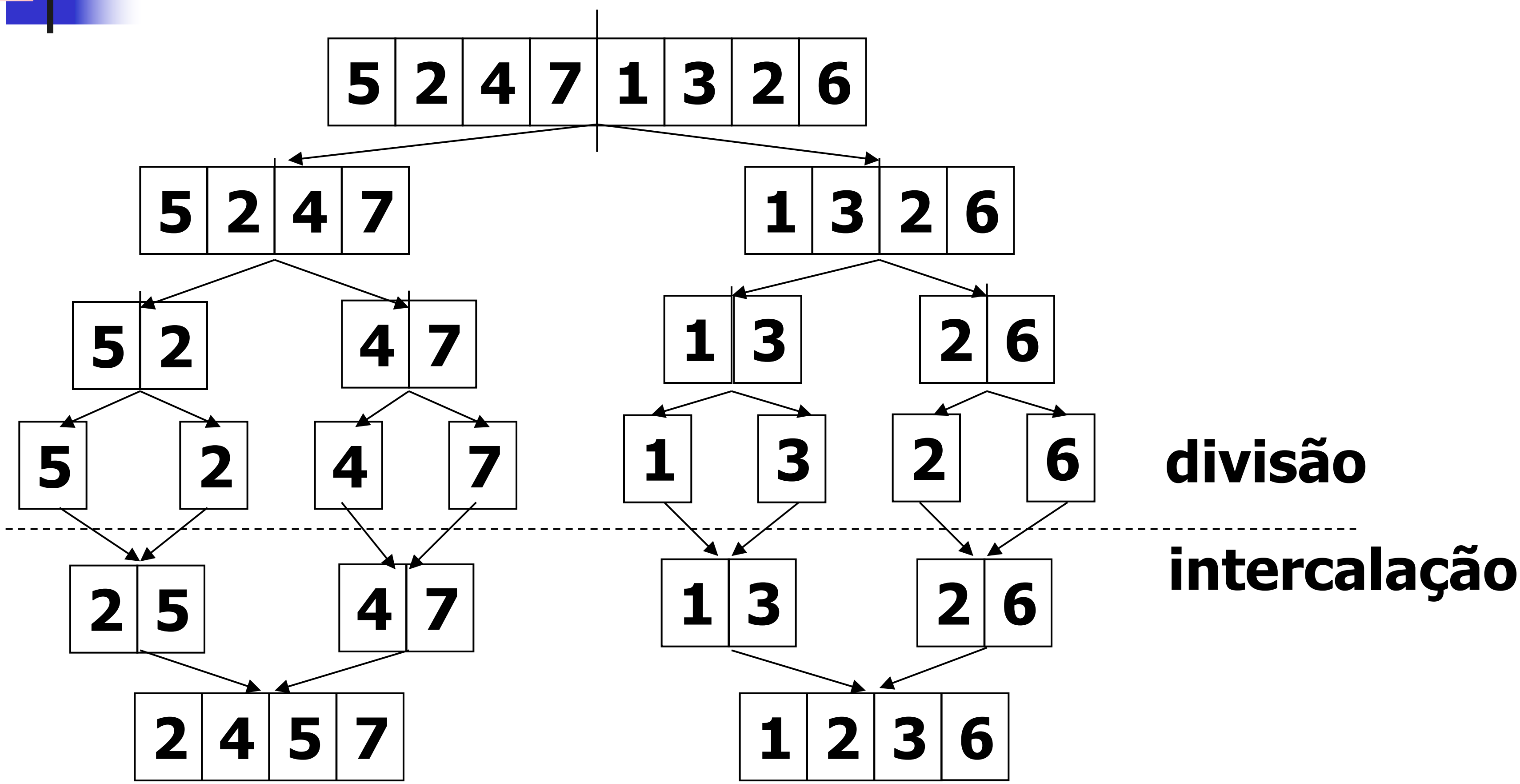



**divisão**

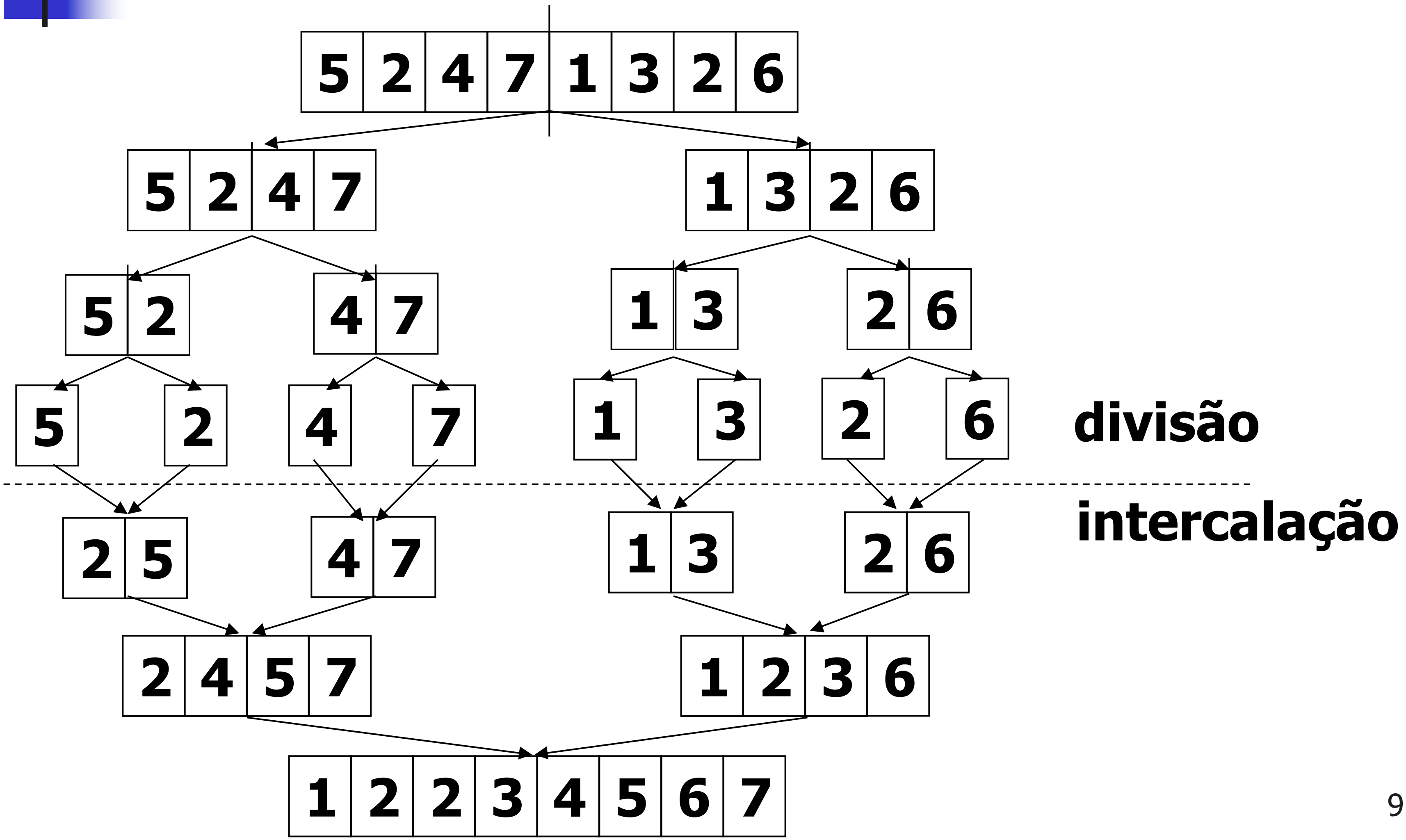
# Ordenação por Intercalação



# Ordenação por Intercalação



# Ordenação por Intercalação



# Ordenação por Intercalação

- Qual a complexidade de tempo?

- E a complexidade de espaço?

# Ordenação por Contagem de Menores

- Idéia básica: se soubermos quantos são os elementos menores que um determinado valor, saberemos a posição que o mesmo deve ocupar no arranjo ordenado
  - Por exemplo, se há 5 valores menores do que o elemento 7, o elemento 7 será inserido na sexta posição do arranjo

- Usa-se um arranjo auxiliar para manter a contagem de menores e um outro para montar o arranjo ordenado

# Ordenação por Contagem de Menores

- Exemplo

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 7 | 9 | 8 | 3 | 0 | 5 |

Arranjo original A desordenado

# Ordenação por Contagem de Menores

- 1º. Passo: criar arranjo auxiliar

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 7 | 9 | 8 | 3 | 0 | 5 |

Arranjo original A desordenado

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

Arranjo auxiliar X, em que X[i]=número de elementos no arranjo A que são menores que A[i] → indicam a posição correta de A[i] no arranjo ordenado

# Ordenação por Contagem de Menores

- 1º. Passo: criar arranjo auxiliar

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 7 | 9 | 8 | 3 | 0 | 5 |

Arranjo original A desordenado

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 1 | 3 | 7 | 9 | 8 | 3 | 0 | 6 |

Arranjo auxiliar X, em que X[i]=número de elementos no arranjo A que são menores que A[i] → indicam a posição correta de A[i] no arranjo ordenado

# Ordenação por Contagem de Menores

- 2º. Passo: montar arranjo final ordenado

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 7 | 9 | 8 | 3 | 0 | 5 |

Arranjo original A desordenado

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 1 | 3 | 7 | 9 | 8 | 3 | 0 | 6 |

Arranjo auxiliar X, em que X[i]=número de elementos no arranjo A que são menores que A[i] → indicam a posição correta de A[i] no arranjo ordenado

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

Arranjo final B ordenado: A[i] vai para a posição X[i] de B

→ Atenção com elemento 3 duplicado

# Ordenação por Contagem de Menores

- 2º. Passo: montar arranjo final ordenado

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 7 | 9 | 8 | 3 | 0 | 5 |

Arranjo original A desordenado

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 1 | 3 | 7 | 9 | 8 | 3 | 0 | 6 |

Arranjo auxiliar X, em que X[i]=número de elementos no arranjo A que são menores que A[i] → indicam a posição correta de A[i] no arranjo ordenado

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 3 | 4 | 5 | 7 | 8 | 9 |

Arranjo final B ordenado: A[i] vai para a posição X[i] de B
→ Atenção com elemento 3 duplicado

# Ordenação por Contagem de Menores

- Exercício
  - Implementar em C a função de ordenação por contagem de menores
  - Calcular complexidade

# Ordenação por Contagem de Menores

- Implementação

```c
void contagem_de_menores(int v[], int n)
{
    int X[n], B[n], i, j;

    for (i=0; i<n; i++)  //inicializando arranjo auxiliar
        X[i]=0;

    for (i=1; i<n; i++)  //contando menores
        for (j=i-1; j>=0; j--)
            if (v[i]<v[j])
               X[j]++;
            else X[i]++;

    for (i=0; i<n; i++)  //montando arranjo final
        B[X[i]]=v[i];

    for (i=0; i<n; i++)  //copiando arranjo final para original
        v[i]=B[i];
}
```

# Ordenação por Contagem de Menores

- Complexidade de tempo?
- Complexidade de espaço?

# Ordenação por Contagem de Menores

- Complexidade de tempo?
  - $O(n^2)$

- Complexidade de espaço?
  - $O(3n)$

# Ordenação por Contagem de Menores

- Exercício: executar algoritmo para o vetor abaixo

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 16 | 2 | 1 | 8 | 2 |

# Ordenação por Contagem de Tipos

- Também chamado counting-sort
- Idéia básica: conta-se o número de vezes que cada elemento ocorre no arranjo; se há k elementos antes dele, ele será inserido na posição k+1 do arranjo ordenado
  - Restrição: os **elementos devem estar contidos em um intervalo [min, max] do conjunto de números inteiros positivos**
- Usa-se um arranjo auxiliar para manter a contagem de tipos e um outro para montar o arranjo ordenado

# Ordenação por Contagem de Tipos

- Exemplo

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 2 | 7 | 6 | 3 | 1 | 2 | 1 | 7 |

Arranjo original A desordenado
→ min=1, max=7

# Ordenação por Contagem de Tipos

- Exemplo

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 2 | 7 | 6 | 3 | 1 | 2 | 1 | 7 |

Arranjo original A desordenado
→ min=1, max=7

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| | | | | | | |

Arranjo auxiliar X, em que X[i] indica o número de elementos i no vetor original A

# Ordenação por Contagem de Tipos

- Exemplo

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 2 | 7 | 6 | 3 | 1 | 2 | 1 | 7 |

Arranjo original A desordenado
→ min=1, max=7

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 3 | 2 | 2 | 0 | 0 | 1 | 2 |

Arranjo auxiliar X, em que X[i] indica o número de elementos i no vetor original A
→ há 3 elementos 1, que ocuparão as posições 0, 1 e 2 do vetor ordenado
→ há 2 elementos 2, que ocuparão as posições livres seguintes (3 e 4) ...

# Ordenação por Contagem de Tipos

- Exemplo

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 2 | 7 | 6 | 3 | 1 | 2 | 1 | 7 |

Arranjo original A desordenado
→ min=1, max=7

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 3 | 2 | 2 | 0 | 0 | 1 | 2 |

Arranjo auxiliar X, em que X[i] indica o número de elementos i no vetor original A
→ há 3 elementos 1, que ocuparão as posições 0, 1 e 2 do vetor ordenado
→ há 2 elementos 2, que ocuparão as posições livres seguintes (3 e 4) ...

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

Arranjo final B ordenado

# Ordenação por Contagem de Tipos

- Exemplo

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 2 | 7 | 6 | 3 | 1 | 2 | 1 | 7 |

Arranjo original A desordenado
→ min=1, max=7

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 3 | 2 | 2 | 0 | 0 | 1 | 2 |

Arranjo auxiliar X, em que X[i] indica o número de elementos i no vetor original A
→ há 3 elementos 1, que ocuparão as posições 0, 1 e 2 do vetor ordenado
→ há 2 elementos 2, que ocuparão as posições livres seguintes (3 e 4) ...

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 2 | 3 | 3 | 6 | 7 | 7 |

Arranjo final B ordenado

# Ordenação por Contagem de Menores

- Implementação

# Implementação

```
void contagem_de_tipos(int v[], int n)
{
    int B[n], i, j, max;

    max=v[0];  //determinando max
    for (i=1; i<n; i++)
        if (v[i]>max)
           max=v[i];
    int X[max+1];

    //inicializando arranjo auxiliar
    for (i=0; i<max+1; i++)
        X[i]=0;

    //contando tipos
    for (i=0; i<n; i++)
        X[v[i]]++;

    //montando arranjo final
    j=0;
    for (i=0; i<max+1; i++)
        while (X[i]!=0) {
            B[j]=i;
            j++;
            X[i]--;
        }

    //copiando arranjo
    //final para original
    for (i=0; i<n; i++)
        v[i]=B[i];
}
```

# Ordenação por Contagem de Tipos

- Complexidade de tempo?

- Complexidade de espaço?

# Ordenação por Contagem de Tipos

- Complexidade de tempo?
  - O(n), se max<=n
    - Por que é “tão melhor” do que outros métodos?

- Complexidade de espaço?
  - O(3n), se max<=n

# Ordenação por Contagem de Tipos

- Complexidade de tempo?
  - O(n), se max<=n
    - A ordenação não é por comparação
- Complexidade de espaço?
  - O(3n), se max<=n

# Ordenação de raízes

- Também chamado radix-sort
- Idéia básica: os números são ordenados por seus dígitos, dos menos significativos para os mais significativos
  - Por exemplo, ordenar os números 236 e 235 implica comparar os últimos dígitos 6 e 5 dos dois; se não bastar, comparam-se os dígitos do meio; por fim, comparam-se os mais significativos
- Baseado na forma de funcionamento das antigas perfuradoras de cartões

# Ordenação de raízes

- Utilizam-se listas
  - Uma fila para cada dígito
  - Os números vão sendo inseridos na fila de acordo com o dígito sendo avaliado
  - A cada iteração, os números estão mais próximos da ordenação final

# Ordenação de raízes: exemplo (1ª. iteração)

Arquivo original

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |

Filas baseadas no dígito menos significativo.

| | Início | Final |
|---|---|---|
| fila [0] | | |
| fila [1] | | |
| fila [2] | 12 | 92 |
| fila [3] | 33 | |
| fila [4] | | |
| fila [5] | 25 | |
| fila [6] | 86 | |
| fila [7] | 57 | 37 |
| fila [8] | 48 | |
| fila [9] | | |

Depois da primeira passagem.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 12 | 92 | 33 | 25 | 86 | 57 | 37 | 48 |

# Ordenação de raízes: exemplo (2ª. iteração)

12 92 33 25 86 57 37 48

Filas baseadas no dígito mais significativo.

| | Início | Final |
|---|---|---|
| fila [0] | | |
| fila [1] | 12 | |
| fila [2] | 25 | |
| fila [3] | 33 | 37 |
| fila [4] | 48 | |
| fila [5] | 57 | |
| fila [6] | | |
| fila [7] | | |
| fila [8] | 86 | |
| fila [9] | 92 | |

Arquivo classificado: 12 25 33 37 48 57 86 92

# Ordenação de raízes

- Para os números do arranjo terem o mesmo número de dígitos, pode-se completá-los com zeros à esquerda
  - Por exemplo, 026 e 235

- Quantas iterações são necessárias para ordenar o arranjo?

# Ordenação de raízes

- Para os números do arranjo terem o mesmo número de dígitos, pode-se completá-los com zeros à esquerda
  - Por exemplo, <u>0</u>26 e 235

- Quantas iterações são necessárias para ordenar o arranjo?
  - São necessárias m iterações no máximo, sendo m o número de dígitos do maior número

# Ordenação de raízes

- Algoritmo

```
for (k=digito menos signif; k<=digito mais signif; k++){
    for (i=0; i<n; i++){
        j=kesimo digito de x[i];
        posiciona x[i] no final da fila[j];
    }
    for (f=0; f<10; f++)
        coloca elems da fila[f] na prox posicao de x;
}
```

# Ordenação de raízes

- Exercício: ordene o vetor abaixo

**(44 , 55 , 112 , 42 , 94 , 18 , 6 , 67)**

# Ordenação de raízes

- Qual a complexidade de tempo do método?

- Qual a complexidade de espaço?

# Ordenação de raízes

- Qual a complexidade de tempo do método?
  - O(m*n)
    - Se m pequeno, O(n)

- Qual a complexidade de espaço?
  - Além do vetor, devem-se contar os espaços para as filas

# Comparação entre os métodos mais conhecidos

- Ordem aleatória dos elementos
  - O mais rápido recebe valor 1 e o restante é recalculado em função disso

| | 500 | 5.000 | 10.000 | 30.000 |
|---|---|---|---|---|
| Inserção | 11,3 | 87 | 161 | – |
| Seleção | 16,2 | 124 | 228 | – |
| *Shellsort* | 1,2 | 1,6 | 1,7 | 2 |
| *Quicksort* | 1 | 1 | 1 | 1 |
| *Heapsort* | 1,5 | 1,6 | 1,6 | 1,6 |

Ziviani, 2007

# Comparação entre os métodos mais conhecidos

- Ordem ascendente dos elementos (já ordenado)

| | 500 | 5.000 | 10.000 | 30.000 |
|---|---|---|---|---|
| Inserção | 1 | 1 | 1 | 1 |
| Seleção | 128 | 1.524 | 3.066 | – |
| *Shellsort* | 3,9 | 6,8 | 7,3 | 8,1 |
| *Quicksort* | 4,1 | 6,3 | 6,8 | 7,1 |
| *Heapsort* | 12,2 | 20,8 | 22,4 | 24,6 |

# Comparação entre os métodos mais conhecidos

- Ordem descendente dos elementos

| | 500 | 5.000 | 10.000 | 30.000 |
|---|---|---|---|---|
| Inserção | 40,3 | 305 | 575 | – |
| Seleção | 29,3 | 221 | 417 | – |
| *Shellsort* | 1,5 | 1,5 | 1,6 | 1,6 |
| *Quicksort* | 1 | 1 | 1 | 1 |
| *Heapsort* | 2,5 | 2,7 | 2,7 | 2,9 |

# Comparação entre os métodos mais conhecidos

- Quick-sort é o mais rápido para todos os arranjos com elementos aleatórios
- Heap-sort e quick-sort têm uma diferença constante, sendo o heap-sort mais lento
- Para arranjos pequenos, shell-sort é melhor do que o heap-sort
- O método da inserção direta é mais rápido para arranjos ordenados
- O método da inserção direta é melhor do que o método da seleção direta para arranjos com elementos aleatórios
- ====
- Shell-sort e quick-sort são sensíveis em relação às ordenações ascendentes e descendentes
- Heap-sort praticamente não é sensível em relação às ordenações ascendentes e descendentes

# Métodos de ordenação

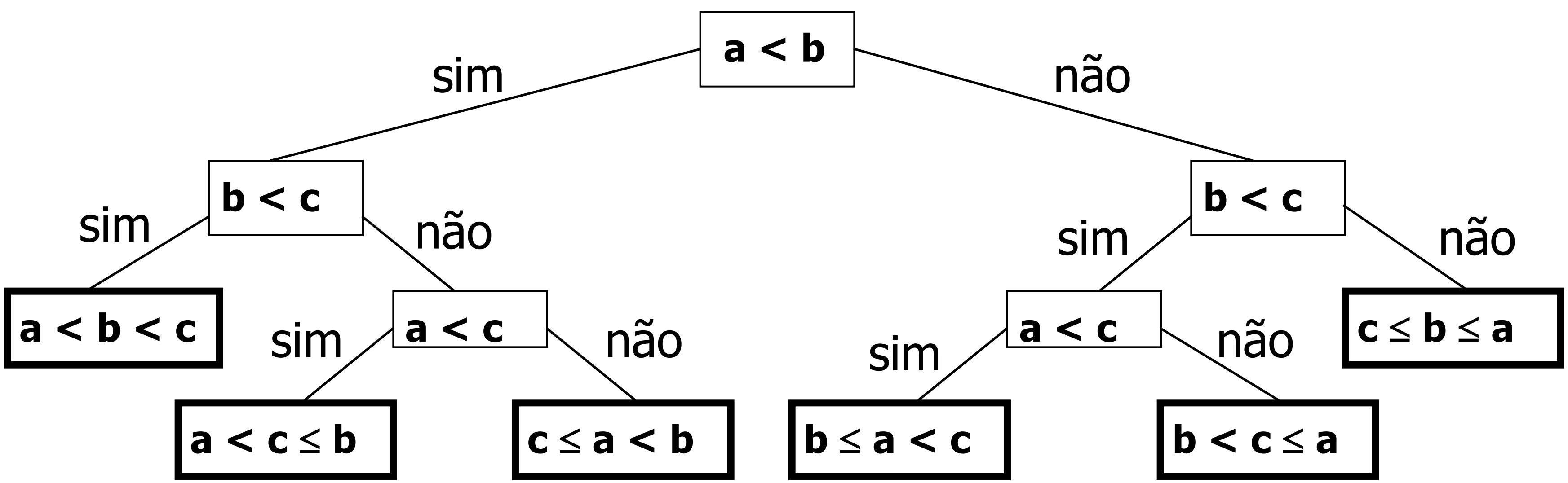
- Outros métodos: tarefa para casa
  - Shake-sort ou método da coqueteleira
    - Melhoramento do bubble-sort
  - Tree-sort ou método da árvore binária
  - Bucket-sort ou método do balde

# Cota Inferior para Ordenação

- Cota (ou limite) inferior do problema
  - Prova-se que é impossível resolver o problema em menos que C(n) passos para uma entrada de tamanho n
  - Algoritmo ótimo: resolve problema em tempo igual à cota inferior

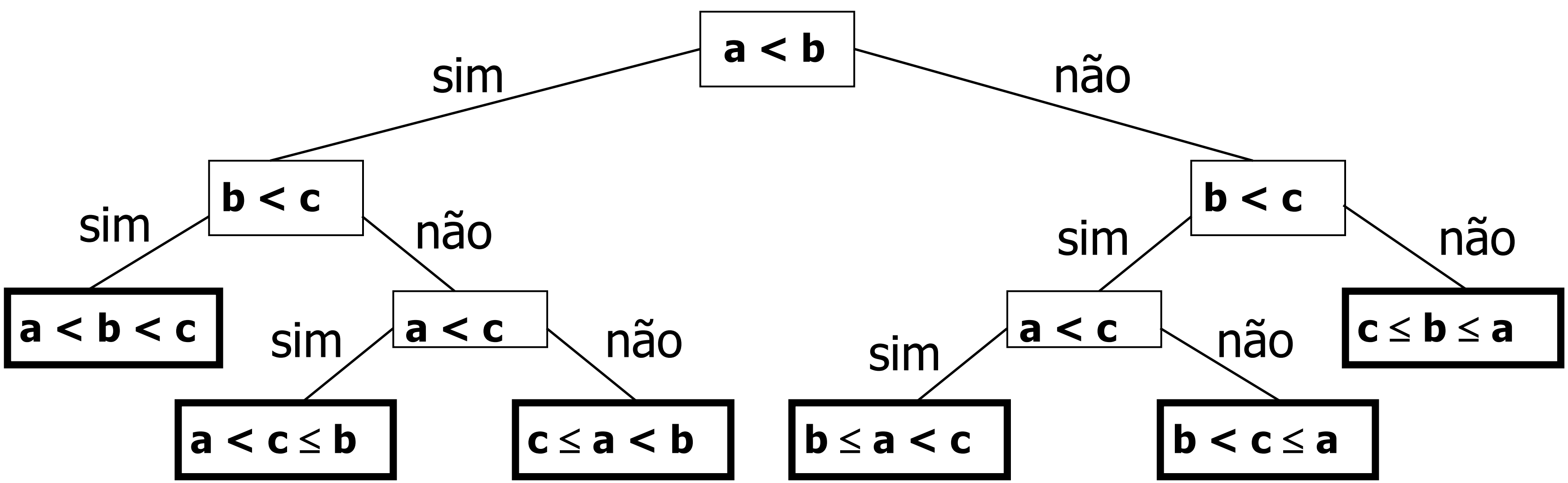
# Cota Inferior para Ordenação

- Cota inferior dos métodos de <u>ordenação por comparação</u> de elementos
  - Pode-se montar uma <u>árvore de decisão</u> para representar o problema
    - Exemplo: ordenação de três elementos a,b e c



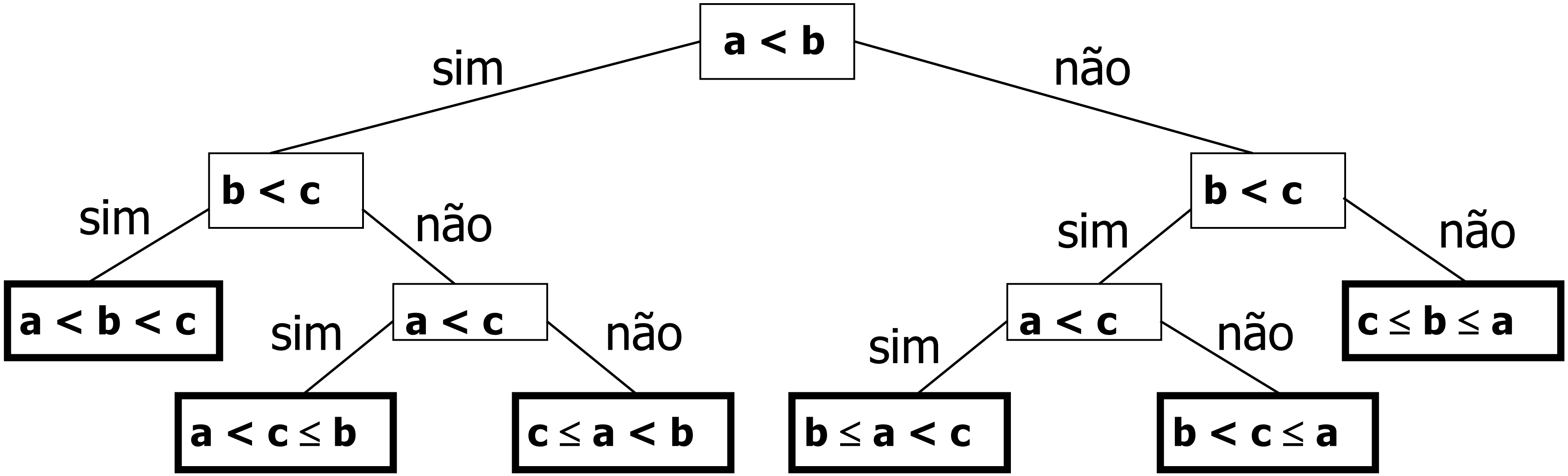
# Cota Inferior para Ordenação

- No mínimo, quantas folhas existem nessa árvore, assumindo um arranjo de tamanho n?
  - Ou: quantas possibilidades de ordenação existem?



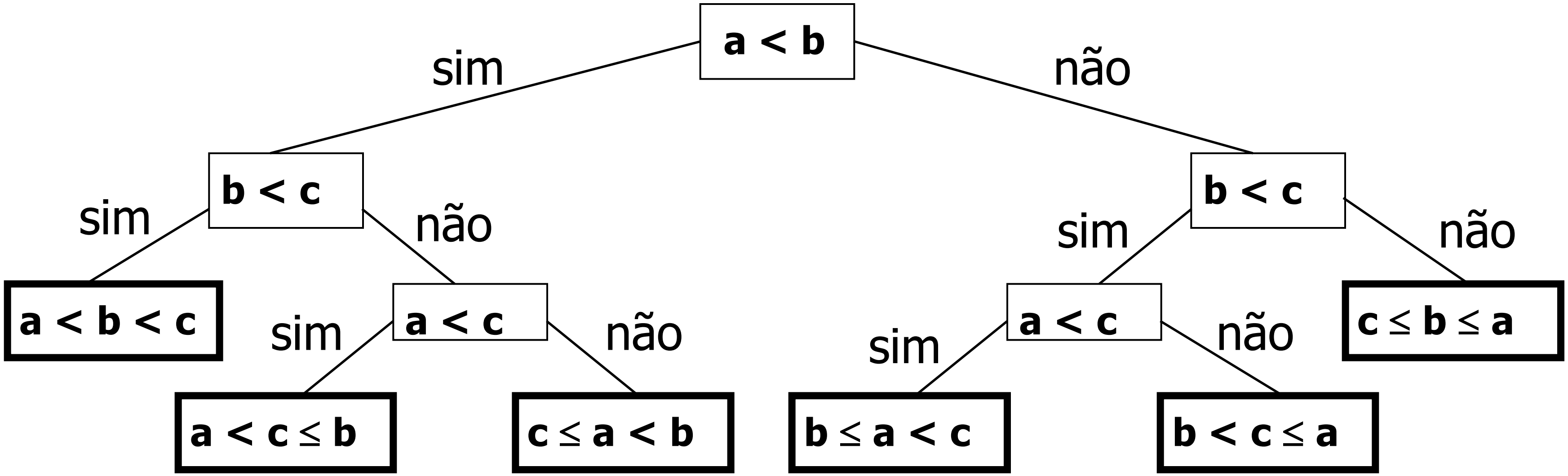
# Cota Inferior para Ordenação

- No mínimo, quantas folhas existem nessa árvore, assumindo um arranjo de tamanho n?
  - Ou: quantas possibilidades de ordenação existem?
    - **n!**



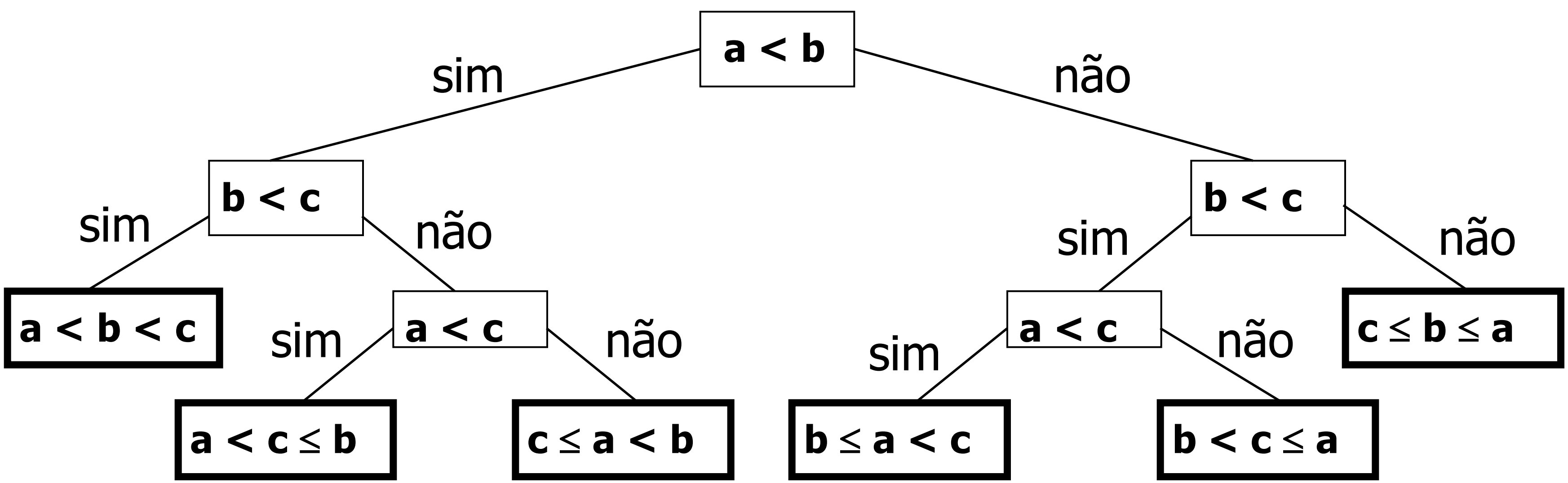
# Cota Inferior para Ordenação

- Quantas comparações devem ser feitas para ordenar n elementos?



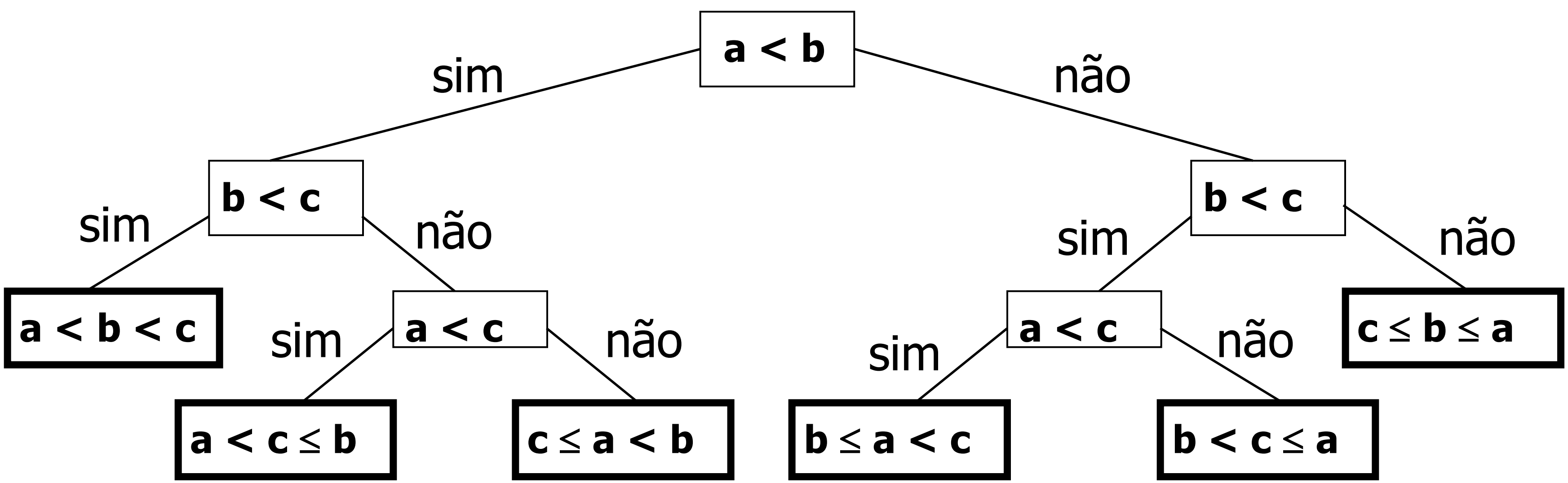
# Cota Inferior para Ordenação

- Quantas comparações devem ser feitas para ordenar n elementos?
  - A altura máxima da árvore de decisão, aproximadamente



# Cota Inferior para Ordenação

- Sabe-se que uma árvore binária de altura **h** não tem mais do que $2^h$ folhas
  - Altura 3: $2^3 \rightarrow$ 8 folhas no máximo



# Cota Inferior para Ordenação

- Então se sabe que
  - Número máximo de folhas de uma árvore binária de altura h: $2^h$
  - Número de folhas de uma árvore de decisão para ordenação por comparação: n!
- Portanto:
  - $2^h \geq n!$

# Cota Inferior para Ordenação

- Então se sabe que
  - Número máximo de folhas de uma árvore binária de altura h: $2^h$
  - Número de folhas de uma árvore de decisão para ordenação por comparação: n!
- Portanto:
  - $2^h \geq n!$
  - Representando via logaritmo: $h \geq \log(n!)$

# Cota Inferior para Ordenação

- Então se sabe que
  - Número máximo de folhas de uma árvore binária de altura h: $2^h$
  - Número de folhas de uma árvore de decisão para ordenação por comparação: n!
- Portanto:
  - $2^h \geq n!$
  - Representando via logaritmo: $h \geq \log(n!)$
  - Aproximação de Stirling: $\log(n!) = O(n \log(n))$

# Cota Inferior para Ordenação

- Então se sabe que
  - Número máximo de folhas de uma árvore binária de altura h: $2^h$
  - Número de folhas de uma árvore de decisão para ordenação por comparação: n!
- Portanto:
  - $2^h \geq n!$
  - Representando via logaritmo: $h \geq \log(n!)$
  - Aproximação de Stirling: $\log(n!) = O(n \log(n))$
  - Resultando que $h \geq \log(n!) \rightarrow h \geq n \log(n) \rightarrow h = \Omega(n \log(n))$

# Cota Inferior para Ordenação

- Resultado
  - Métodos de ordenação por comparação de elementos <u>não podem ser melhores</u> do que O(n log(n))