SCC-601– Introdução à Ciência da Computação II

Ordenação e Complexidade – Parte 4

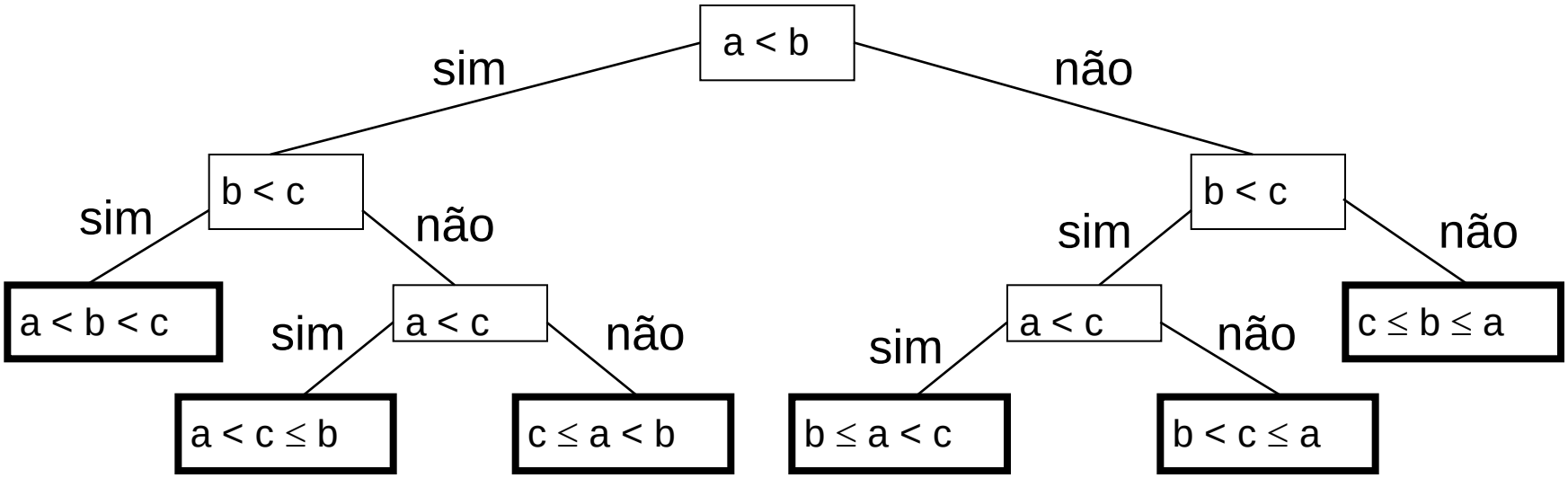
Lucas Antiqueira

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

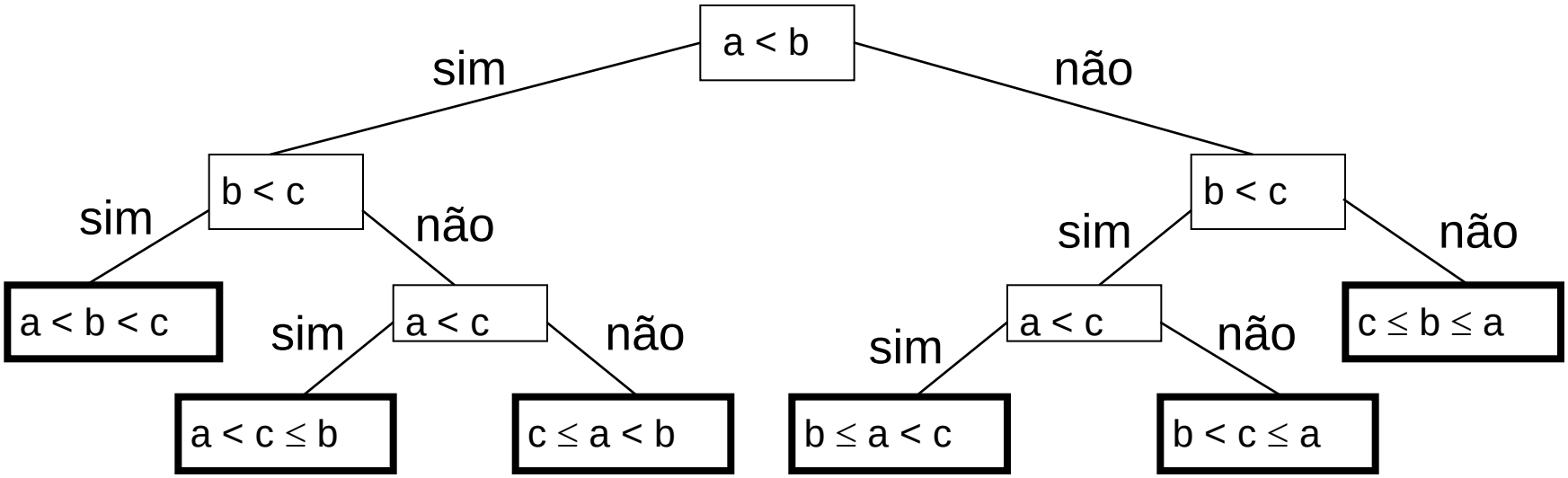
Cota Inferior para Ordenação

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



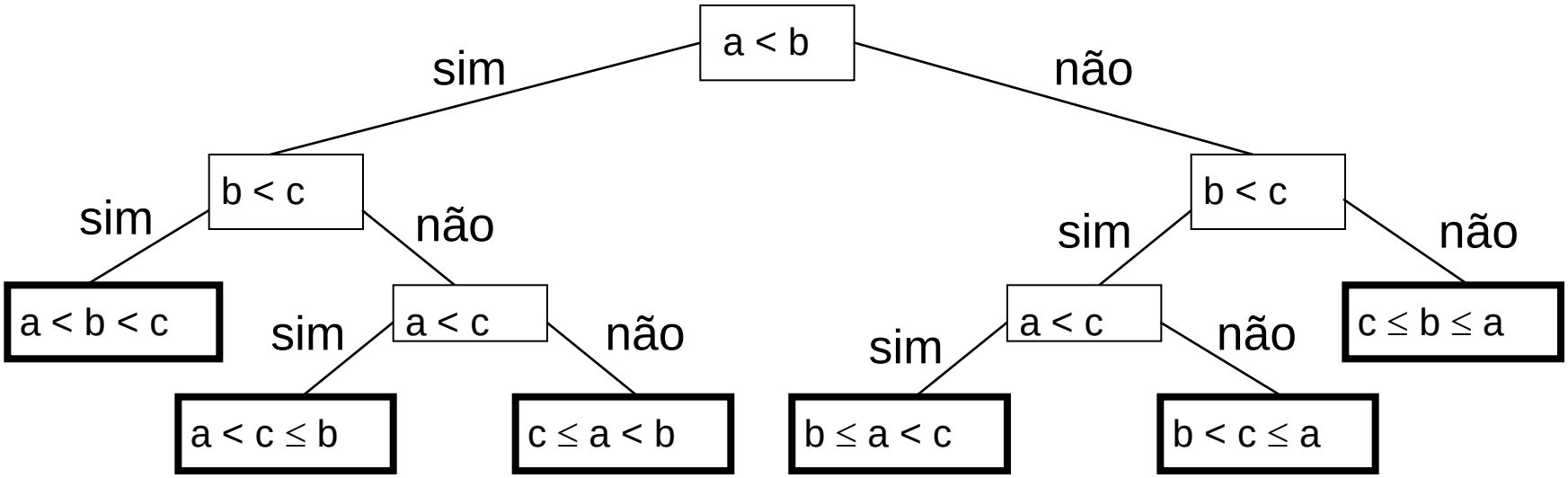
Cota Inferior para Ordenação

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



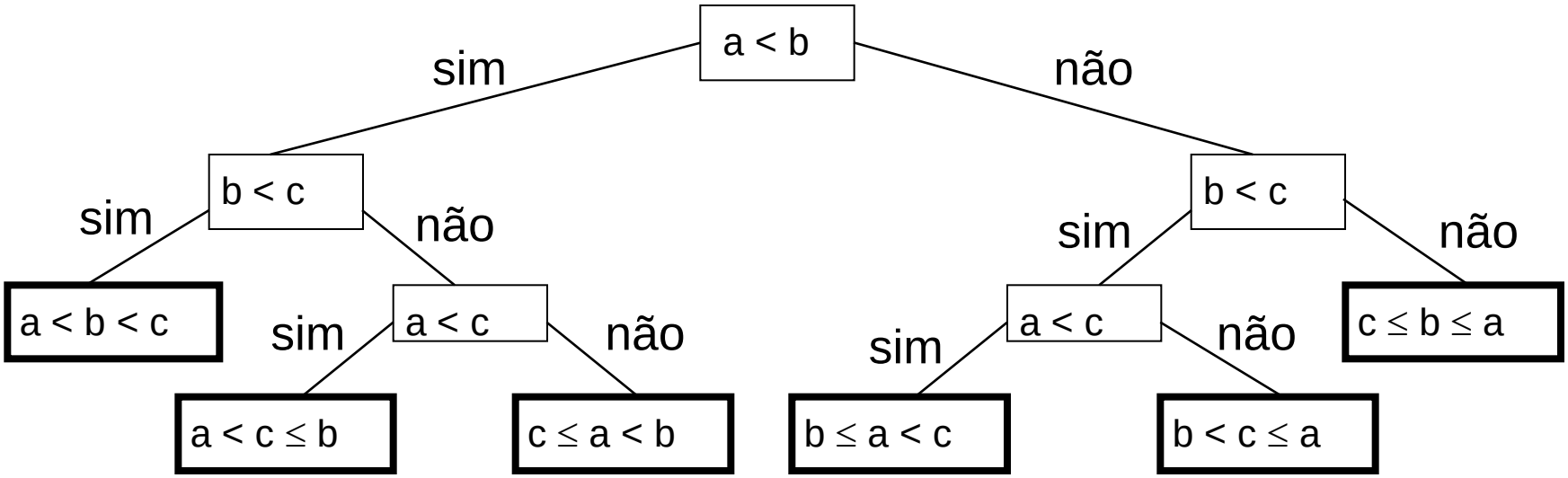
Cota Inferior para Ordenação

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



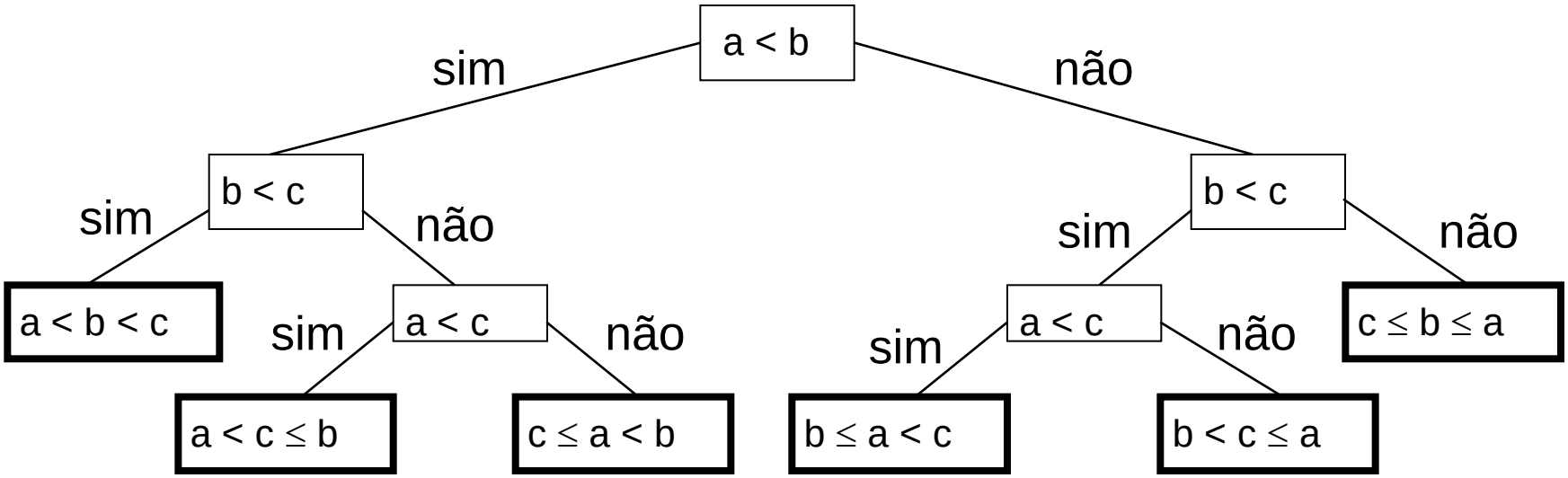
Cota Inferior para Ordenação

- Quantas comparações devem ser feitas para ordenar n elementos?



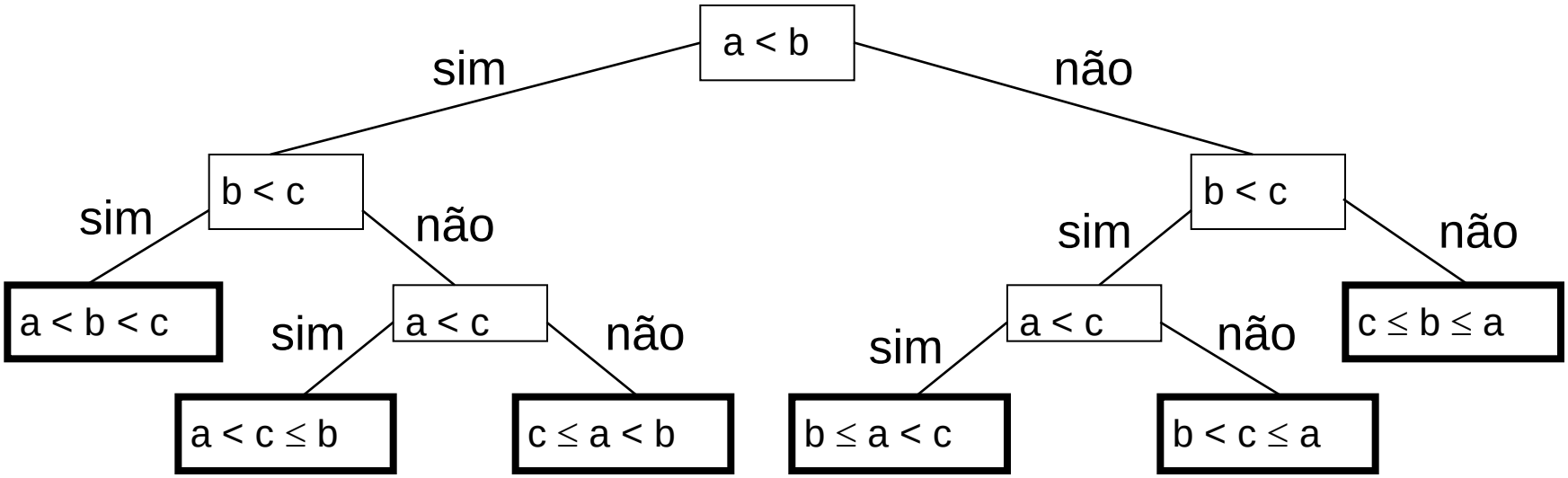
Cota Inferior para Ordenação

- Quantas comparações devem ser feitas para ordenar n elementos?
  - Igual à 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
  - Por exemplo: altura h=3:
    $2^3 = 8$ 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:
  - $2^h \geq n!$
  - Aplicando logaritmo: $h \geq \log_2(n!)$

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:
  - $2^h \geq n!$
  - Aplicando logaritmo: $h \geq \log_2(n!)$
  - Usando a aproximação de Stirling: $\log(n!) \approx n \log_2(n)$

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:
  - $2^h \geq n!$
  - Aplicando logaritmo: $h \geq \log_2(n!)$
  - Usando a aproximação de Stirling: $\log(n!) \approx n \log_2(n)$
  - Resultando que o número de comparações h é:
    $h \geq \log_2(n!) \rightarrow h \geq n \log_2(n) \rightarrow h = \Omega(n \log_2(n))$

Cota Inferior para Ordenação

- Consequentemente:
  - Métodos de ordenação por comparação de elementos não podem ser melhores do que $n \log_2(n)$
  - Limitante inferior: $\Omega(n \log_2(n))$
  - Se um método de ordenação for $O(n \log_2(n))$, ele é ótimo, pois seu limitante inferior é $\Omega(n \log_2(n))$ e, portanto, será $\Theta(n \log_2(n))$.

Cota Inferior para Ordenação

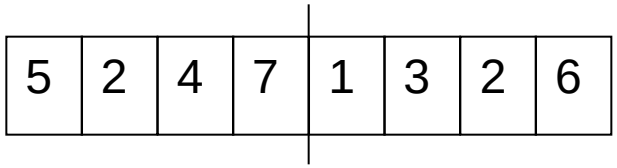
- Quais métodos de ordenação ótimos já vimos?
- Existem outros?

Ordenação: MergeSort

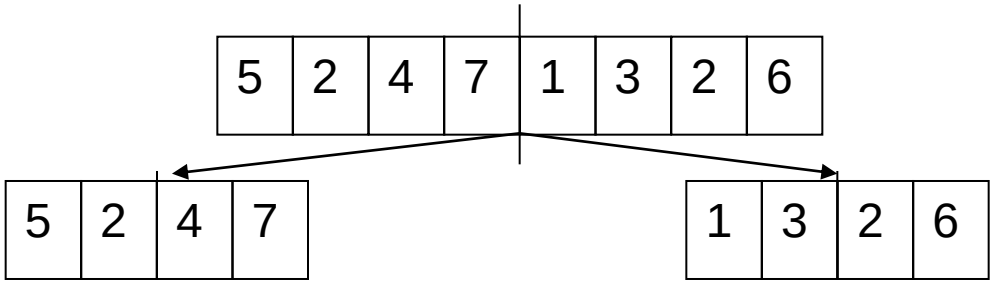
MergeSort

- Também chamado *Ordenação por Intercalação*
- Idéia básica: dividir para conquistar
  - Um vetor v é dividido em duas partes, recursivamente
  - Cada metade é ordenada, e a seguir ambas são intercaladas formando o vetor ordenado
  - Usa vetor auxiliar para intercalar

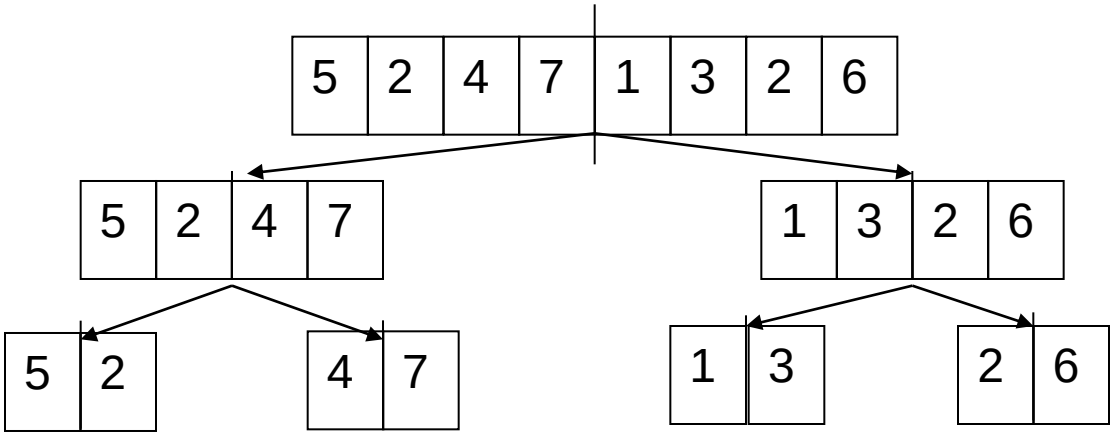
MergeSort



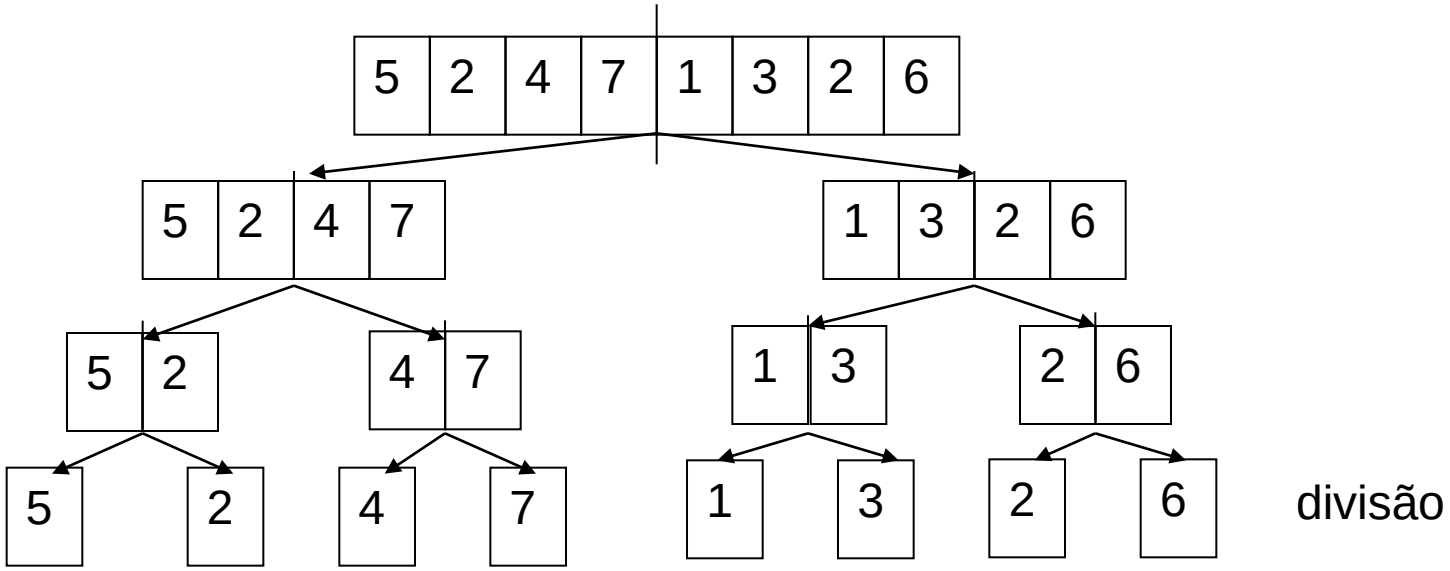
MergeSort



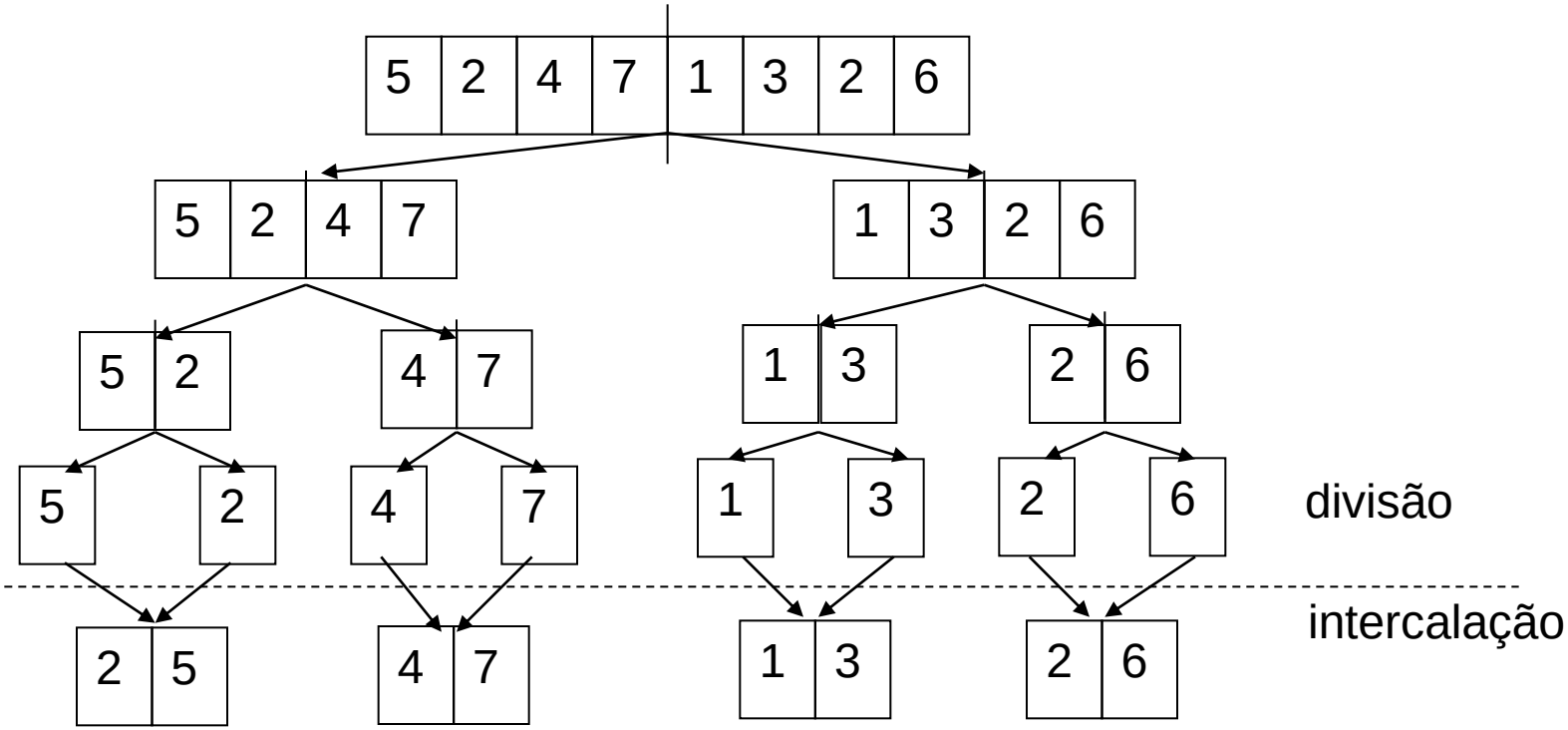
MergeSort



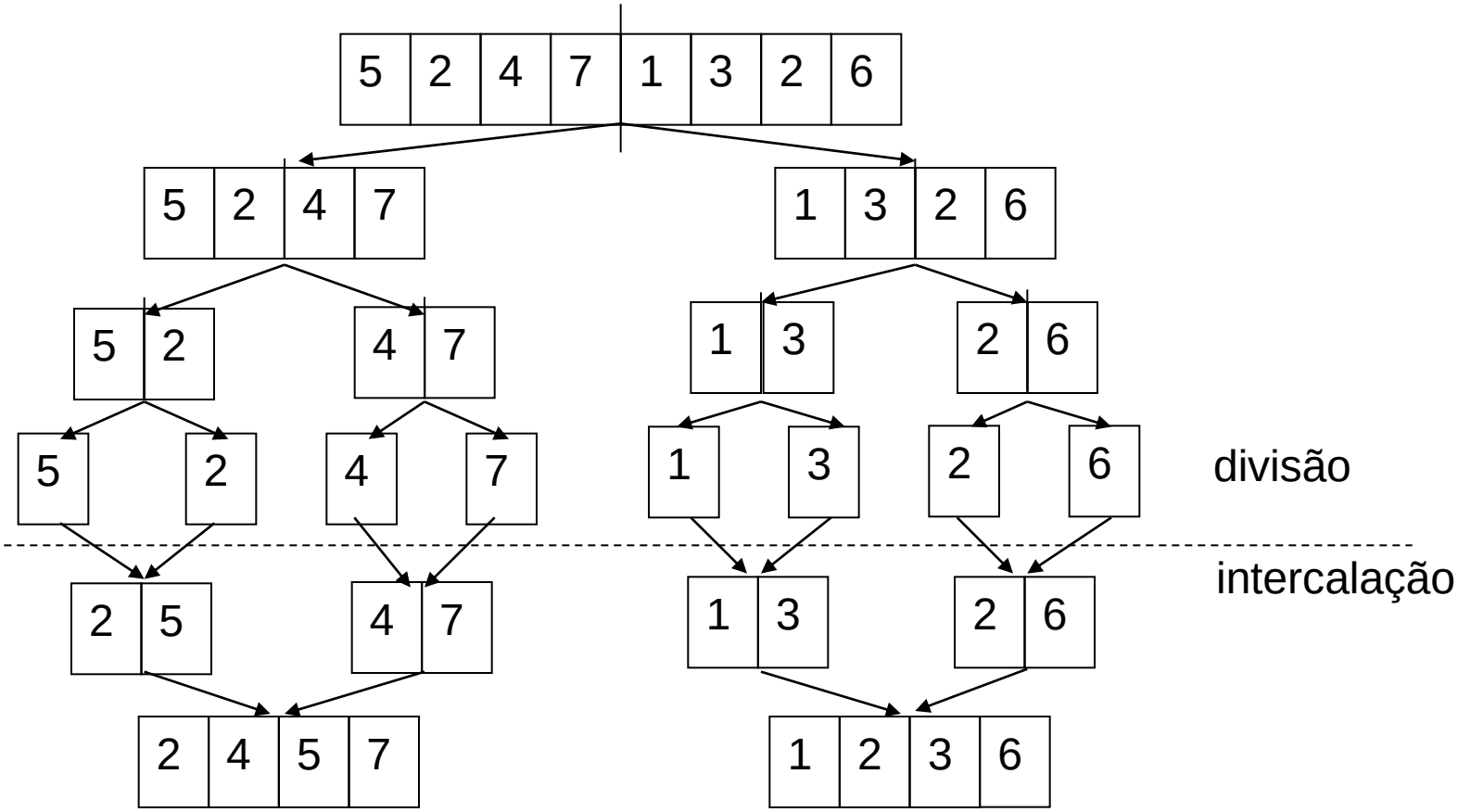
MergeSort



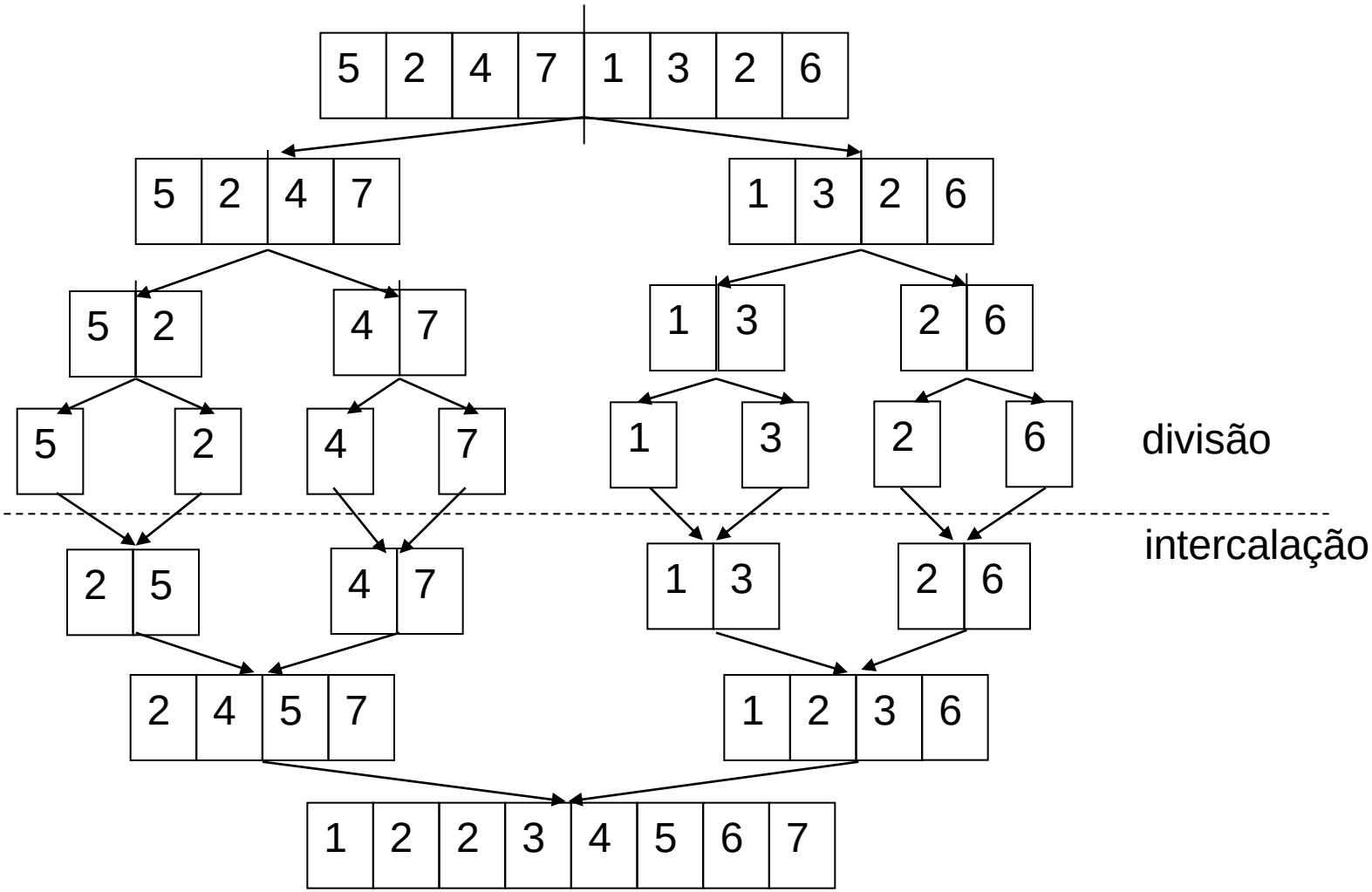
MergeSort


5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
divisão
intercalação
2 5
4 7
1 3
2 6

MergeSort

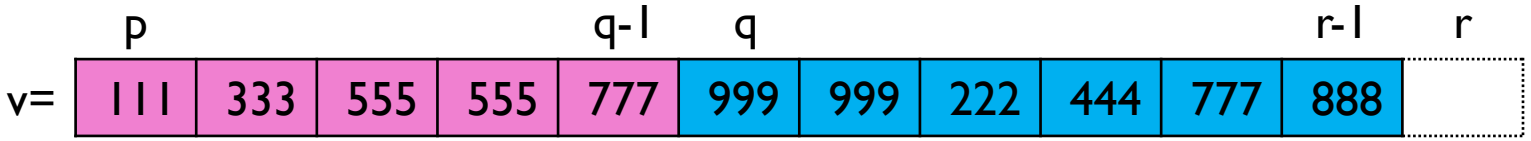

5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
divisão
intercalação
2 5
4 7
1 3
2 6
2 4 5 7
1 2 3 6

MergeSort


5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
divisão
intercalação
2 5
4 7
1 3
2 6
2 4 5 7
1 2 3 6
1 2 2 3 4 5 6 7

MergeSort

```
void mergesort(int v[], int p, int r) {
        int q;
        if (p < r - 1) {
                q = (p + r) / 2;
                mergesort(v, p, q);
                mergesort(v, q, r);
                intercala(v, p, q, r);
        }
}
```



Baseado em http://www.ime.usp.br/~pf/algoritmos/aulas/mrgsrt.html

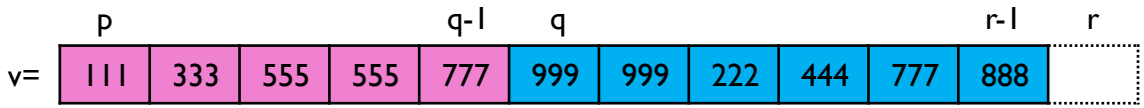
MergeSort

```
1.  void intercala(int v[], int p, int q, int r) {
2.        int i=p, j=q, k=0, *w;
3.        w = malloc((r - p)*sizeof(int));
4.        while (i < q && j < r) {
5.                if (v[i] <= v[j])
6.                        w[k++] = v[i++];
7.                else
8.                        w[k++] = v[j++];
9.        }
10.       while (i < q)
11.               w[k++] = v[i++];
12.       while (j < r)
13.               w[k++] = v[j++];
14.       for (i = p; i < r; ++i)
15.               v[i] = w[i-p];
16.       free(w);
17. }
```

Intercala em um vetor auxiliar w

e

copia de volta ao vetor v



MergeSort

- Complexidade
  - Se n=1, ordenação não é necessária: tempo constante O(c)
  - Se n>1
    - O problema é inicialmente dividido em subproblemas: tempo constante O(c)
    - Os subproblemas são processados: 2 subproblemas, sendo que cada um tem metade do tamanho original: 2T(n/2)
    - As soluções são combinadas: O(n)

MergeSort

- Equações de complexidade do algoritmo

$T(n)=O(c)$, se $n=1$

$T(n)=2T(n/2) + \underbrace{O(c) + O(n)}_{O(n)}$, se $n>1$

Pois $O(n) = \max(O(c),O(n))$

MergeSort

- Equações de complexidade do algoritmo

  $T(n)=O(c)$, se $n=1$

  $T(n)=2T(n/2) + O(n)$, se $n>1$

MergeSort

- Equações de complexidade do algoritmo

  $T(n)=1$, se $n=1$

  $T(n)=2T(n/2) + n$, se $n>1$

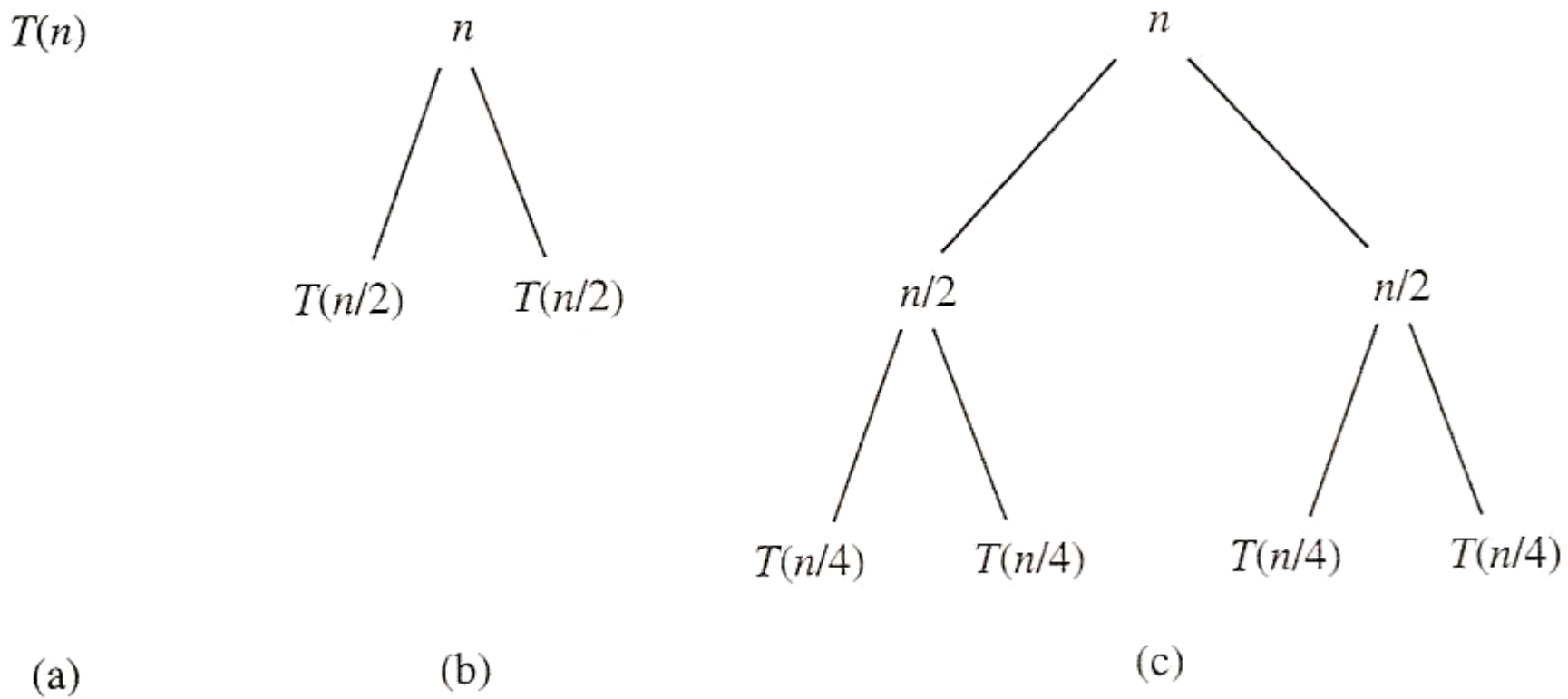
MergeSort

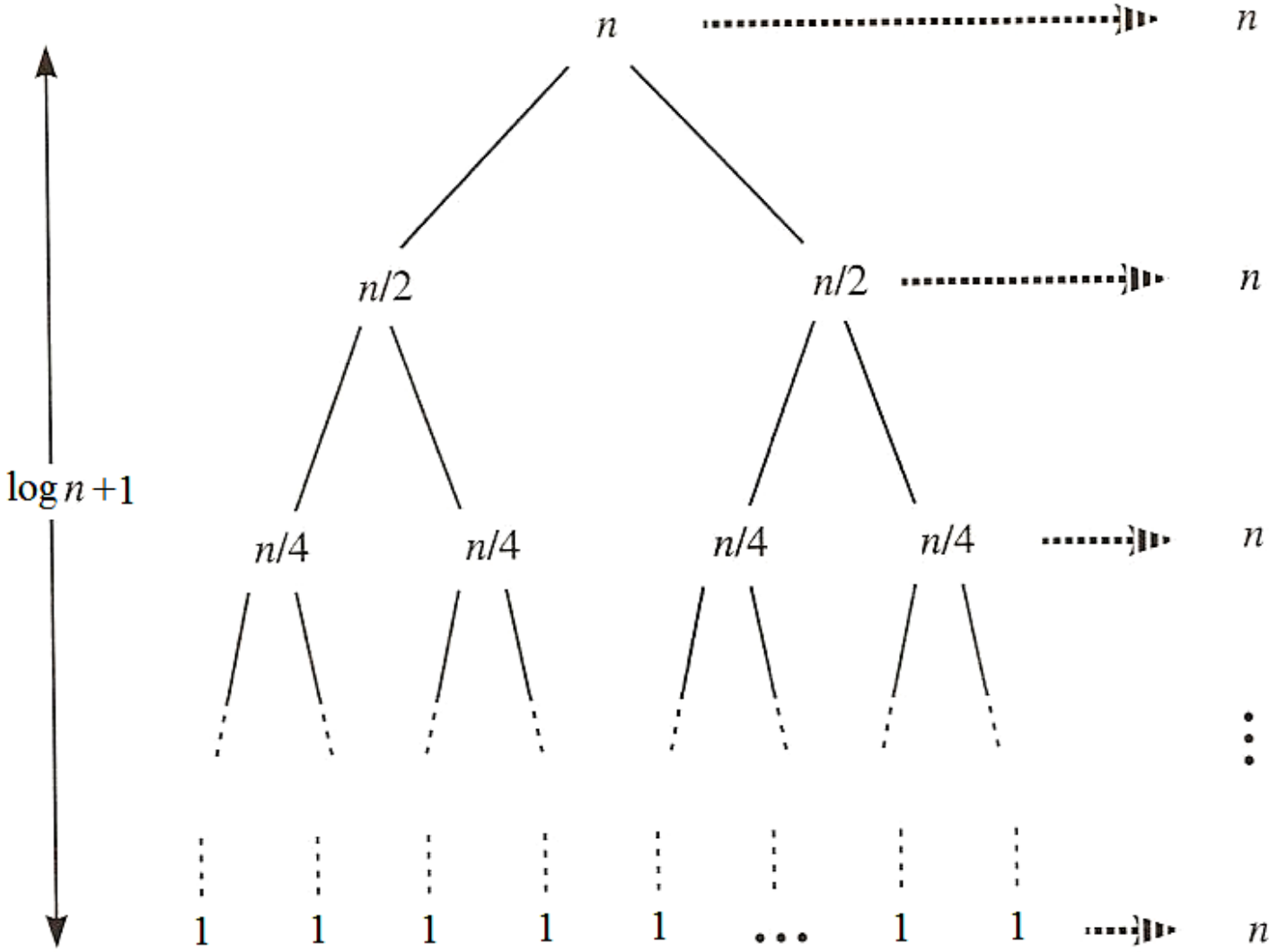
- Equações de complexidade do algoritmo

  $T(n)=1$, se $n=1$

  $T(n)=2T(n/2) + n$, se $n>1$

EQUAÇÃO DE RECORRÊNCIA, podendo ser resolvida via árvore de recorrência


T(n)
n
T(n/2)
T(n/2)
n
n/2
n/2
T(n/4)
T(n/4)
T(n/4)
T(n/4)
(a)
(b)
(c)



$n(\log n + 1)$

Total: $n \log n + n$

(d)

MergeSort

- Tem-se que:
  - Na parte (a), há T(n) ainda não expandido
  - Na parte (b), T(n) foi dividido em árvores equivalentes representando a recorrência com custos divididos (T(n/2) cada uma), sendo n o custo no nível superior da recursão (fora da recursão e, portanto, associado ao nó raiz)
  - No fim, nota-se que a altura da árvore corresponde a (log n)+1, o qual multiplica os valores obtidos em cada nível da árvore, os quais, nesse caso, são iguais a n
    - Como resultado, tem-se n logn + n, ou seja, O(n log n)

MergeSort

- $O(n \log_2 n)$ → melhor caso, caso médio e pior caso
- Memória → ?

MergeSort

- $O(n \log_2 n)$ → melhor caso, caso médio e pior caso
- Memória → $O(n)$

MergeSort

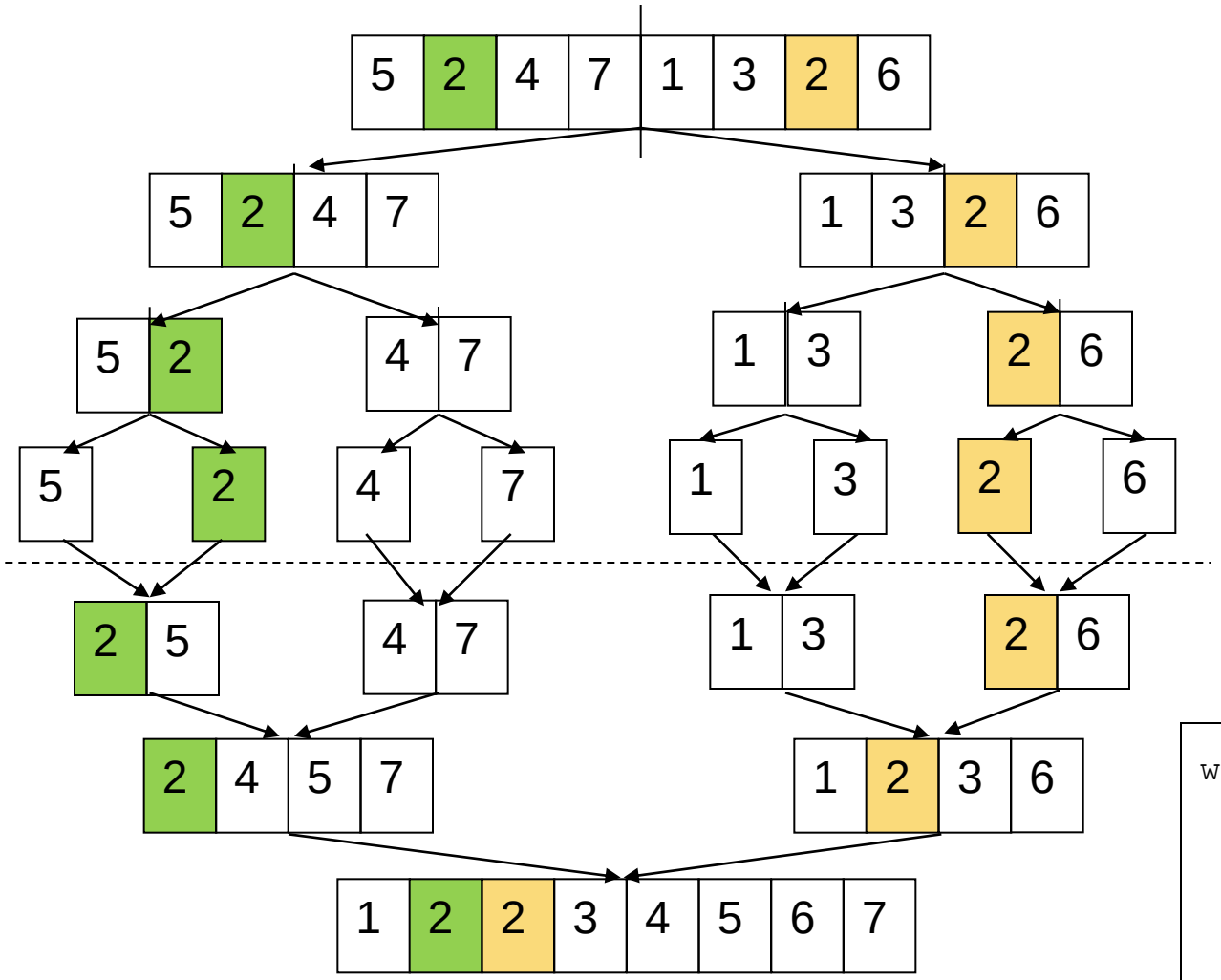
- $O(n \log_2 n)$ → melhor caso, caso médio e pior caso
- Memória → $O(n)$

É comportamento assintótico.
Na prática, usa (no máximo) o dobro de memória que os métodos sem vetor auxiliar.

MergeSort

- É estável?

MergeSort



MergeSort é estável

```
while (i < q && j < r) {
    if (v[i] <= v[j])
            w[k++] = v[i++];
    else
            w[k++] = v[j++];
}
```