

# TÉCNICAS DE PROGRAMAÇÃO: PROGRAMAÇÃO DINÂMICA MERGE SORT TREE

# O que é programação Dinâmica?

- Programação dinâmica trata-se de uma técnica de otimização para resolução de problemas complexos.
- Divide o problema em subproblemas menores
- Armazena resultados intermediários para evitar recomputações
- Ideal para problemas com:
  - Subestrutura ótima
  - Sobreposição de subproblemas

# Problemas Clássicos

- Fibonacci
- Problema da Mochila
- Cortes de Barra
- LCS (Longest Common Subsequence)
- Matrix Chain Multiplication

# Técnicas para resolver um problema de programação dinâmica

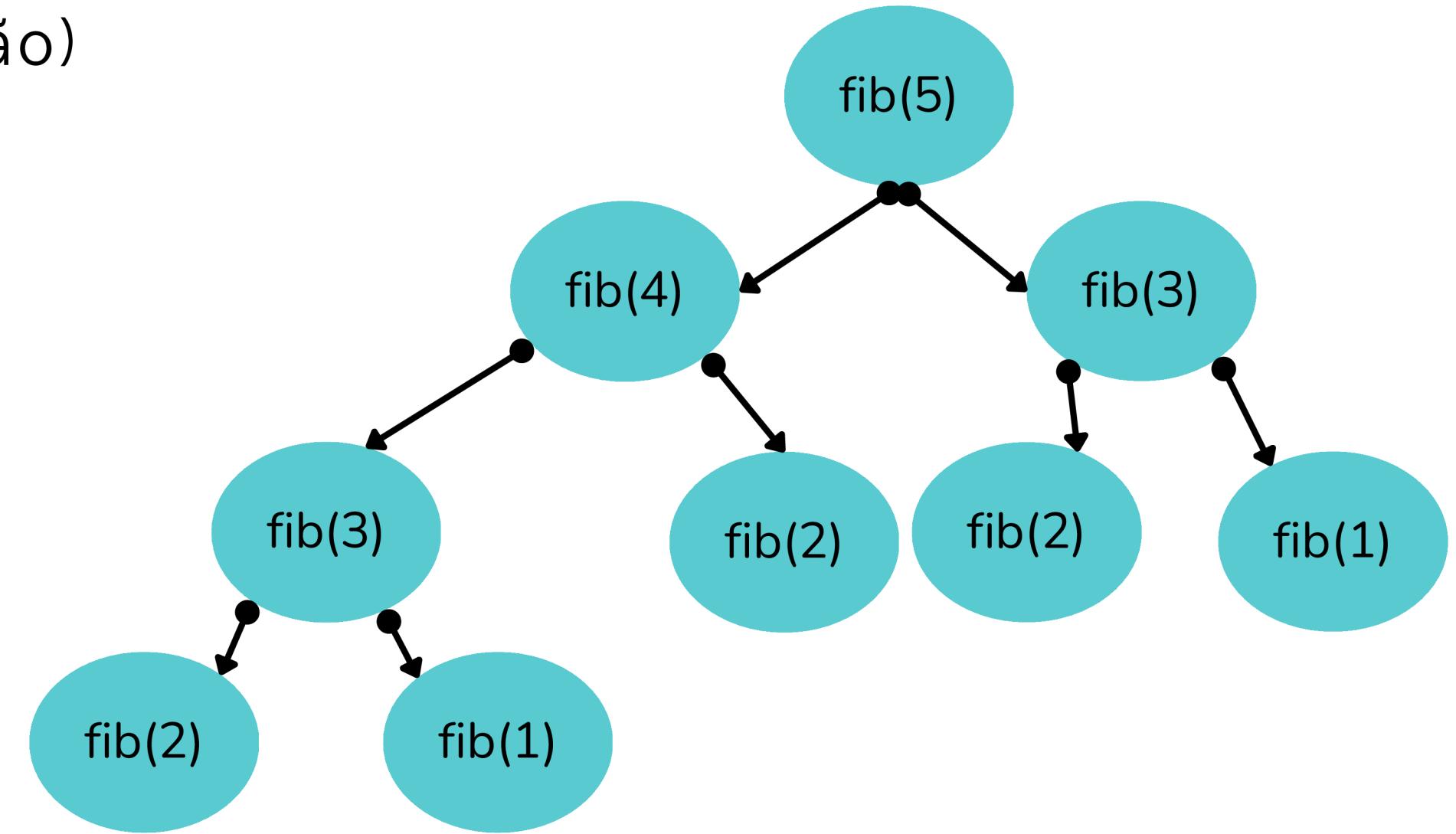
- Recursão.
- Memoization (Top-Down): Recursão + Cache de Dados.
- Tabulação (Bottom-Up): Preenchimento iterativo de tabela.

Atenção: Memoization  $\neq$  Memorization

# Exemplo

- Fibonacci sem otimização (recursão)

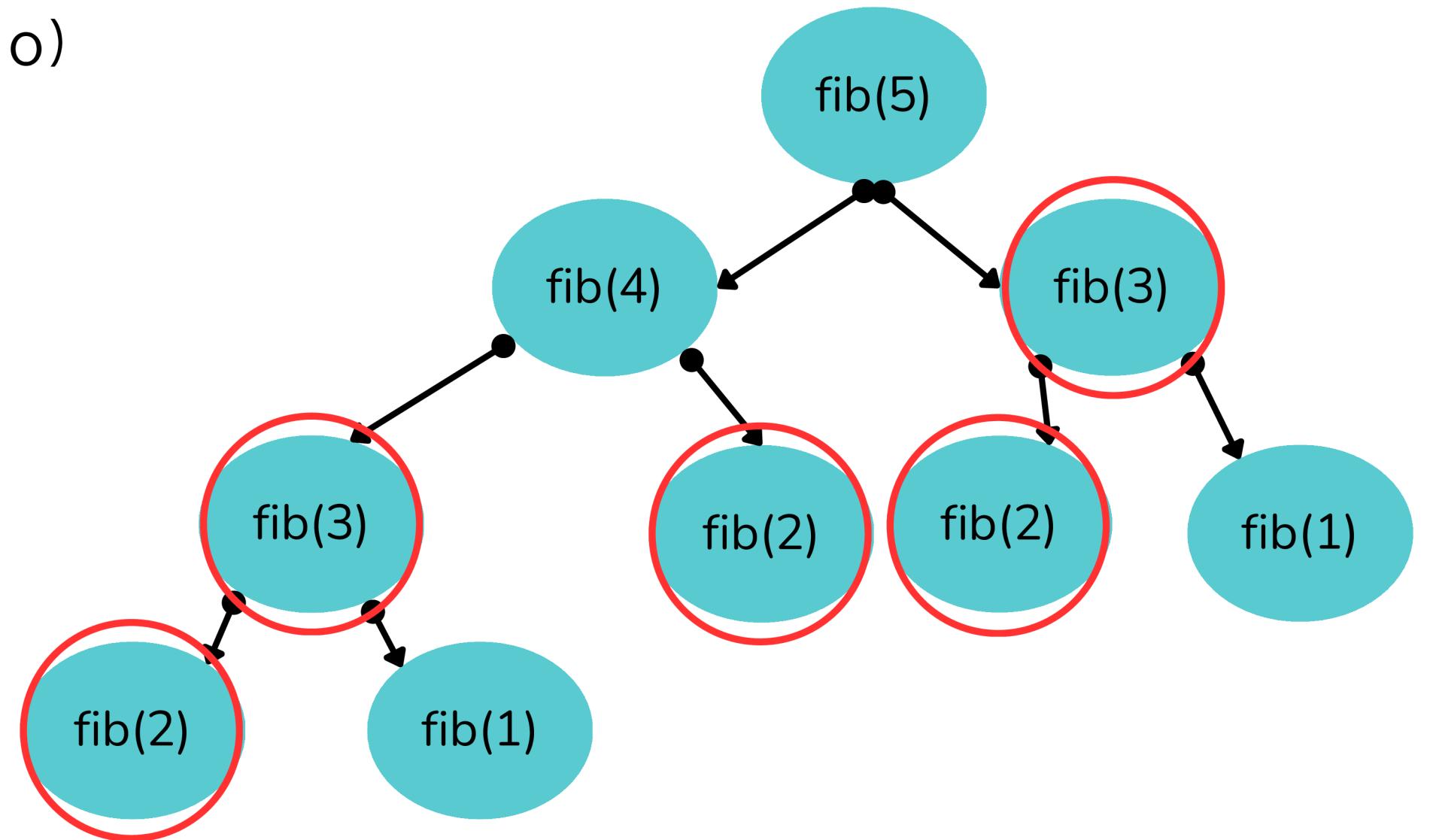
```
def fib(n):  
    if n == 1 or n == 2:  
        result = 1  
    else:  
        return = fib(n-1) + fib(n-2)  
  
    return result
```



# Exemplo

- Fibonacci sem otimização (recursão)

```
def fib(n):  
    if n == 1 or n == 2:  
        result = 1  
    else:  
        return fib(n-1) + fib(n-2)  
  
    return result
```

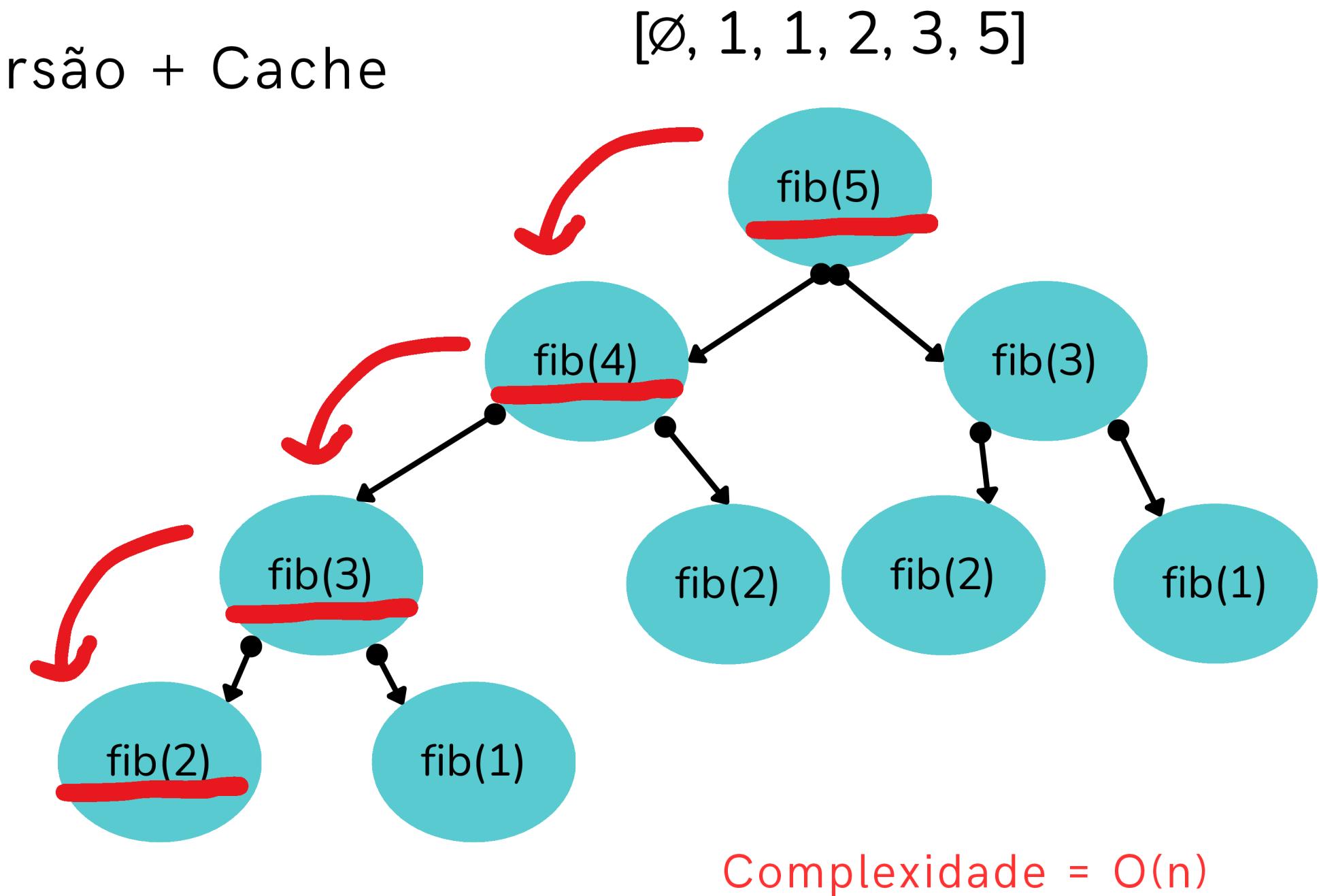


Complexidade =  $O(e^n)$

# Exemplo

- Fibonacci com Memoization: Recursão + Cache de Dados

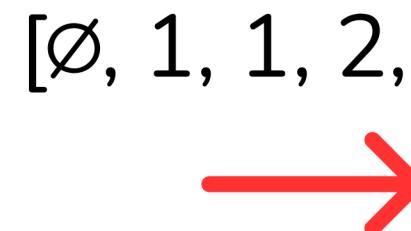
```
def fib(n, memo):  
    if memo[n] != null:  
        if n == 1 or n == 2:  
            result = 1  
        else:  
            return = fib(n-1) + fib(n-2)  
  
    memo[n] = result  
    return result
```



# Exemplo

- Fibonacci com Tabulação: Preenchimento iterativo do array.

```
def fib_bottom_up(n):
    if n == 1 or n == 2:
        result = 1
    bottom_up = new int[n+1]
    bottom_up[1] = 1
    bottom_up[2] = 1
    [∅, 1, 1, 2, 3, 5]
    for i from 3 upto n:
        bottom_up[i] = bottom_up[i-1] + bottom_up[i-2]
    return result
```



Complexidade = O(n)

# Exemplo notebook

- Fibonacci sem otimização
- Fibonacci com Memoization
- Fibonacci com Tabulação

# Resumo

- Evita recomputações desnecessárias
- Útil em problemas com recursividade pesada
- Abordagens: Top-Down e Bottom-up
- Melhora na performance

# MERGE SORT TREE

# Definição

Estrutura de dados que armazena informações de intervalos de arrays por meio de árvores.

É uma ótima estrutura para fazer buscas de alcances, assim como é eficiente o suficiente para realizar modificações na árvore.

É uma combinação de uma Árvore de Segmentos e Merge Sort.

Algumas utilizações comuns dessa estrutura:

- Contar elementos em um intervalo que são menores/maiores que um valor
- Encontrar o k-ésimo menor/maior elemento em um intervalo
- Contar elementos em um intervalo que estão dentro de um limite específico

# Relembrando

## Merge Sort

- Divide o array em duas metades recursivamente até chegar em arrays de 1 elemento. Então une as metades ordenadas comparando elementos.
- Complexidade:  $O(n \log n)$  no pior caso.
- Estável e eficiente para grandes volumes de dados.

## Segment Tree

- Estrutura de dados usada para responder consultas (como soma, mínimo, máximo) em intervalos de um array.
- Construção:  $O(n)$ , consultas e atualizações:  $O(\log n)$ .
- Cada nó é uma informação, como a soma do array, mínimo, etc. E forma uma árvore com os filhos feitos pela divisão do array, como no merge sort

# Complexidade

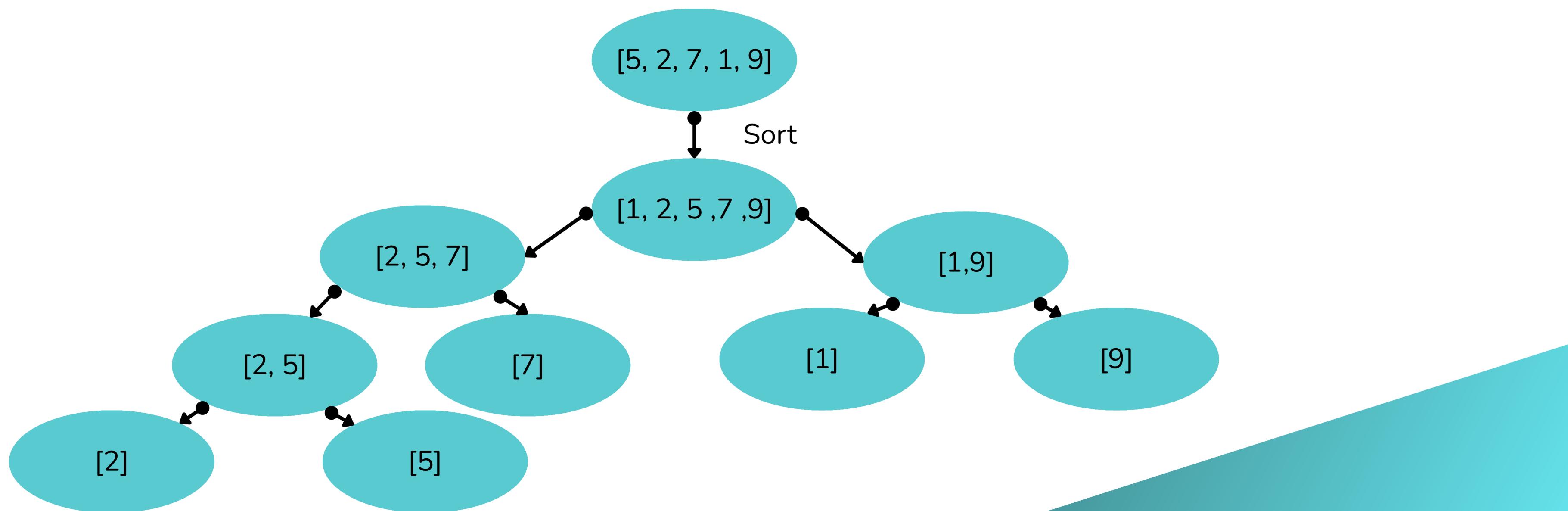
- **Construção:**  $O(n \log n)$ 
  - $n$  nós no nível das folhas, merge em  $O(n)$  em cada nível,  $\log n$  níveis
- **Espaço:**  $O(n \log n)$ 
  - Cada elemento é armazenado uma vez em cada nível
- **Consulta:**  $O(\log^2 n)$ 
  - $O(\log n)$  para percorrer a árvore
  - $O(\log n)$  para busca binária em cada nó visitado

# Estrutura

Cada nó armazena um array ordenado dos elementos em seu intervalo:

- **Nó raiz:** array inteiro ordenado
- **Nós folha:** elementos individuais
- **Nós internos:** merge de seus filhos (como no Merge Sort)

\*Esse algoritmo usa a posição original como base, por isso o 1 e 9 vão para a direita.



# Criação e consulta

```
import bisect

# Constructs a segment tree and stores sTree[]
def buildTree(idx, ss, se, a, sTree):
    # leaf node
    if ss == se:
        sTree[idx] = a[ss]
        return

    mid = (ss + se) // 2

    # building left subtree
    buildTree(2 * idx + 1, ss, mid, a, sTree)

    # building right subtree
    buildTree(2 * idx + 2, mid + 1, se, a, sTree)

    # merging left and right child in sorted order
    sTree[idx] = sorted(sTree[2 * idx + 1] + sTree[2 * idx + 2])
```

```
# Recursive function to count smaller elements from row
# a[ss] to a[se] and value smaller than or equal to k.
def queryRec(node, start, end, ss, se, k, a, sTree):
    # If out of range return 0
    if ss > end or start > se:
        return 0

    # if inside the range return count
    if ss <= start and se >= end:
        # binary search over the sorted list to return count >= k
        return bisect.bisect_right(sTree[node], k) - start

    mid = (start + end) // 2

    # searching in left subtree
    p1 = queryRec(2 * node + 1, start, mid, ss, se, k, a, sTree)

    # searching in right subtree
    p2 = queryRec(2 * node + 2, mid + 1, end, ss, se, k, a, sTree)
    if p1 + p2 < 0:
        return 0

    # adding both the result
    return p1 + p2
```

# Exemplo

Array: [5, 2, 7, 1, 9]

Consulta: Quantos elementos  $\leq 6$  no intervalo [1, 3]?

- Intervalo [1, 3] corresponde aos valores [2, 7, 1]
- Segmentos na árvore que cobrem [1, 3]:
  - [1, 2]  $\rightarrow$  nó com array ordenado [2, 7]
  - [3]  $\rightarrow$  nó com array ordenado [1]
- Contagem de elementos  $\leq 6$  em [2, 7]: 1
- Contagem de elementos  $\leq 6$  em [1]: 1
- Total: 2 elementos

[5, 2, 7, 1, 9]  
[0, 1, 2, 3, 4]

# Fontes

- <https://www.geeksforgeeks.org/dynamic-programming/>
- <https://www.youtube.com/watch?v=vYquumk4nWw>
- [https://cp-algorithms.com/data\\_structures/segment\\_tree.html#toc-tgt-12](https://cp-algorithms.com/data_structures/segment_tree.html#toc-tgt-12)
- <https://www.geeksforgeeks.org/merge-sort-tree-smaller-or-equal-elements-in-given-row-range/>