

# Design and Analysis of Algorithm 303105218

**Dr. Meghana Harsh Ghogare**

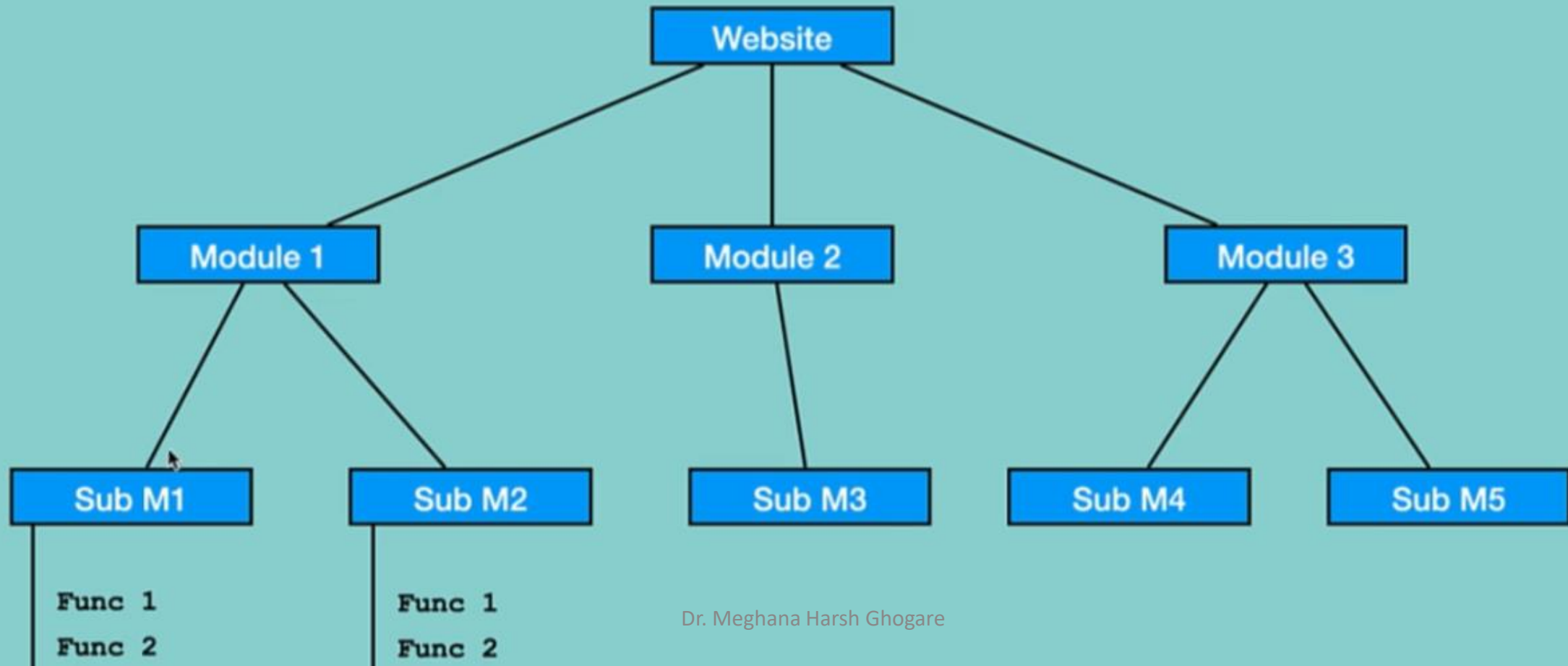
# UNIT 2 Divide and Conquer Algorithms

- Structure of divide-and-conquer algorithms
- Binary search
- Quick sort
- Merge sort
- Strassen Multiplication
- Max-Min problem

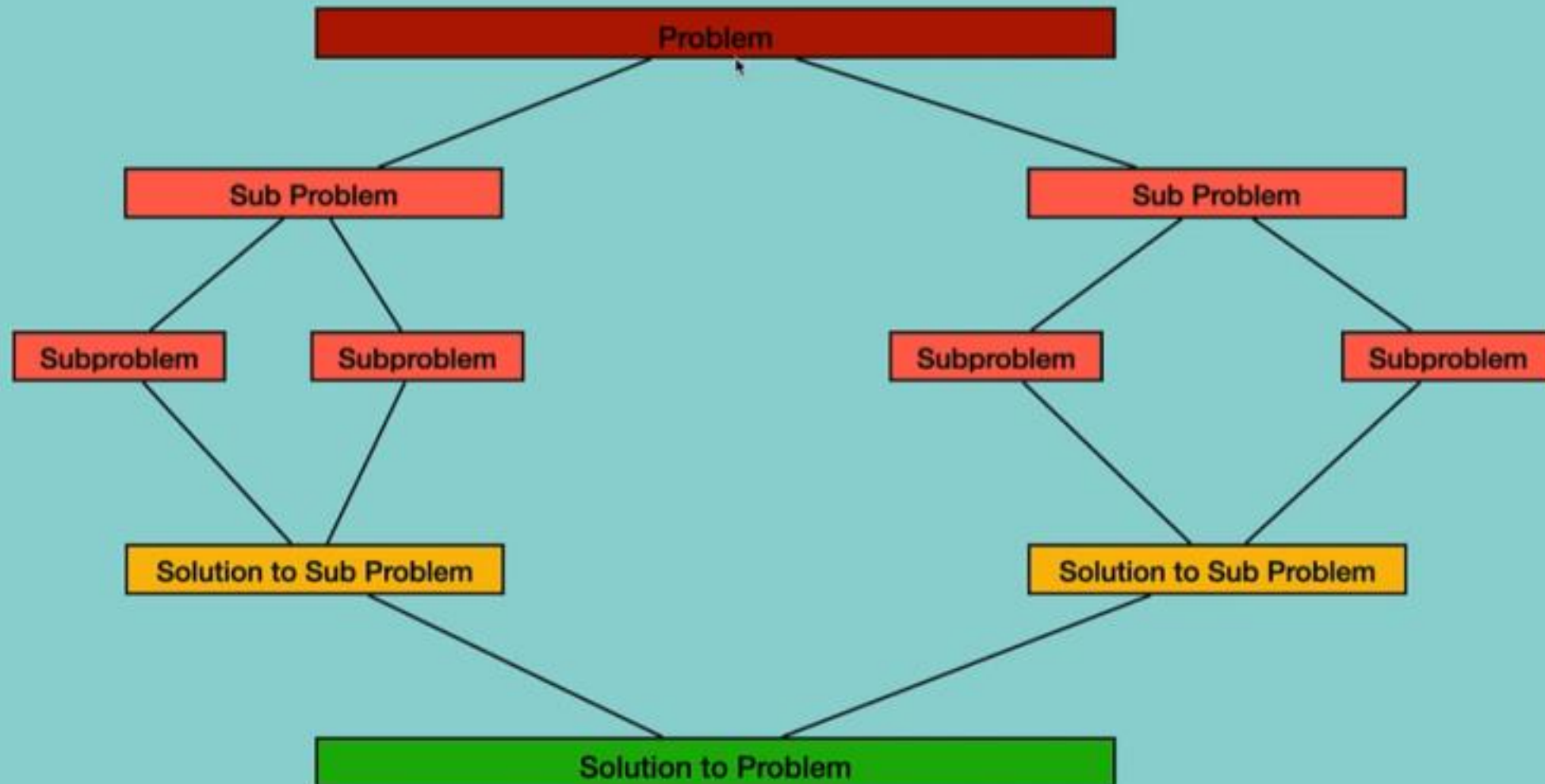
# What is Divide and Conquer Algorithm?

Divide and conquer is an algorithm design paradigm which works by recursively breaking down a problem into subproblems of similar type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.

## Example : Developing a website



## What is Divide and Conquer Algorithm?



# Property of Divide and Conquer Algorithm

## Optimal Substructure:

If any problem's overall optimal solution can be constructed from the optimal solutions of its subproblem then this problem has optimal substructure

Example:  $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

## Why we need it?

It is very effective when the problem has optimal substructure property.

If you can solve a big problem by solving smaller problems and putting their solutions together, the big problem has optimal substructure.

# Divide and Conquer Algorithms

- **Structure of Divide and Conquer Algorithms**

**1.Divide:** Break the problem into smaller sub-problems of the same type.

**2.Conquer:** Solve the sub-problems recursively.

**3.Combine:** Merge the solutions of the sub-problems to get the solution to the original problem.

# Binary Search

- **Algorithm**

**1.Divide:** Find the middle element of the array.

**2.Conquer:**

1. If the middle element is the target, return the index.
2. If the target is smaller than the middle element, repeat the process on the left sub-array.
3. If the target is larger than the middle element, repeat the process on the right sub-array.

**3.Combine:** The solution is directly obtained from the sub-array that contains the target.

```
def binary_search(arr, target):  
    def search(left, right):  
        if left > right:  
            return -1  
  
        mid = (left + right) // 2  
        if arr[mid] == target:  
            return mid  
        elif arr[mid] > target:  
            return search(left, mid - 1)  
        else:  
            return search(mid + 1, right)  
    return search(0, len(arr) - 1)
```



# Eg of left > right

```
arr = [2, 4, 6, 8, 10]  
target = 5
```

## 1. Initial Call: `search(0, 4)`

- `left` = 0
- `right` = 4
- `mid` =  $(0 + 4) // 2 = 2$
- `arr[mid]` = 6

Since `6` is greater than `5`, we search the left half: `search(0, 1)`

## 2. Second Call: `search(0, 1)`

- `left` = 0
- `right` = 1
- `mid` =  $(0 + 1) // 2 = 0$
- `arr[mid]` = 2

Since `2` is less than `5`, we search the right half: `search(1, 1)`

### 3. Third Call: `search(1, 1)`

- `left` = 1
- `right` = 1
- `mid` =  $(1 + 1) // 2 = 1$
- `arr[mid]` = 4

Since `4` is less than `5`, we search the right half: `search(2, 1)`

### 4. Fourth Call: `search(2, 1)`

- `left` = 2
- `right` = 1

Now, `left` is greater than `right` ( $2 > 1$ ). This means we have exhausted the search space without finding the target. Therefore, we return `-1`.

## Traceout

For array [1, 3, 5, 7, 9], target = 5:

1. mid = 2, arr[mid] = 5 (found at index 2)

## Complexity

- Time:  $O(\log n)$
- Space:  $O(\log n)$  (due to recursion stack) or  $O(1)$  (iterative version)

# Quick Sort

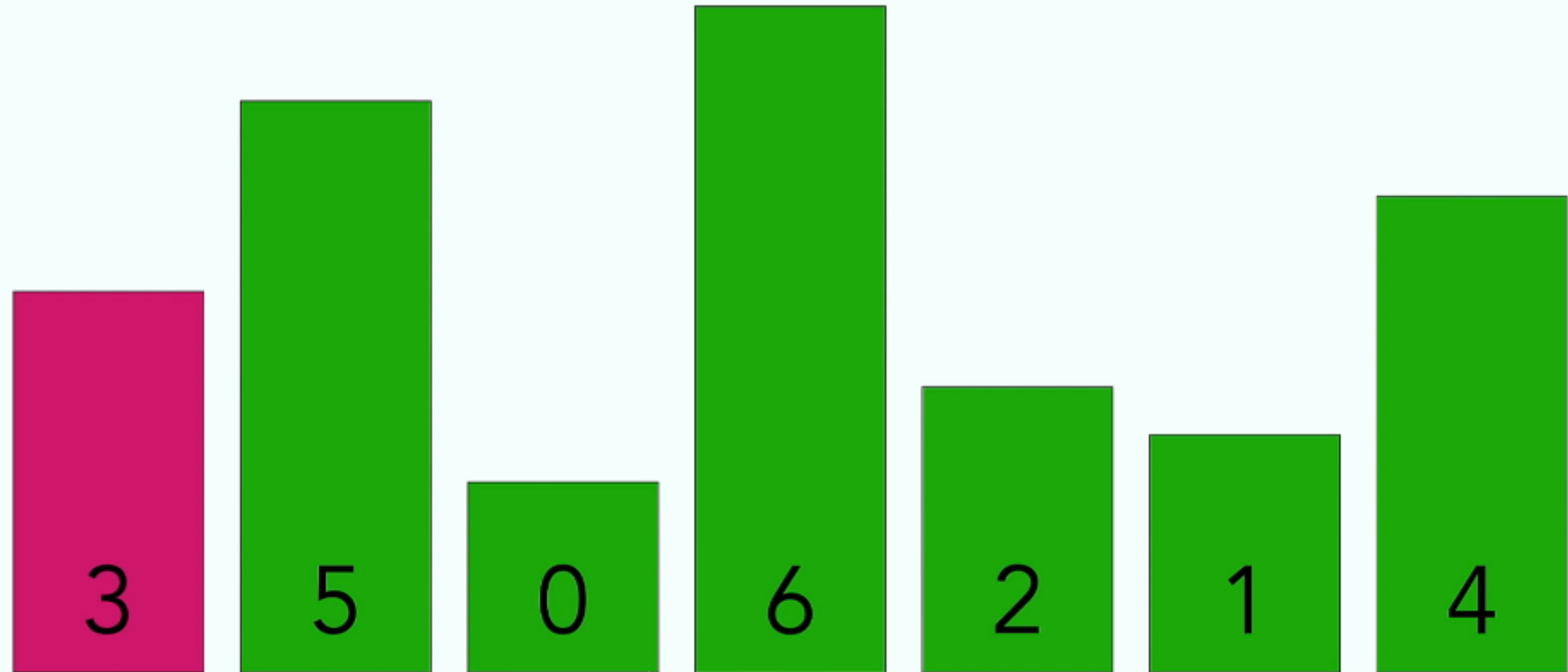
- **Algorithm**

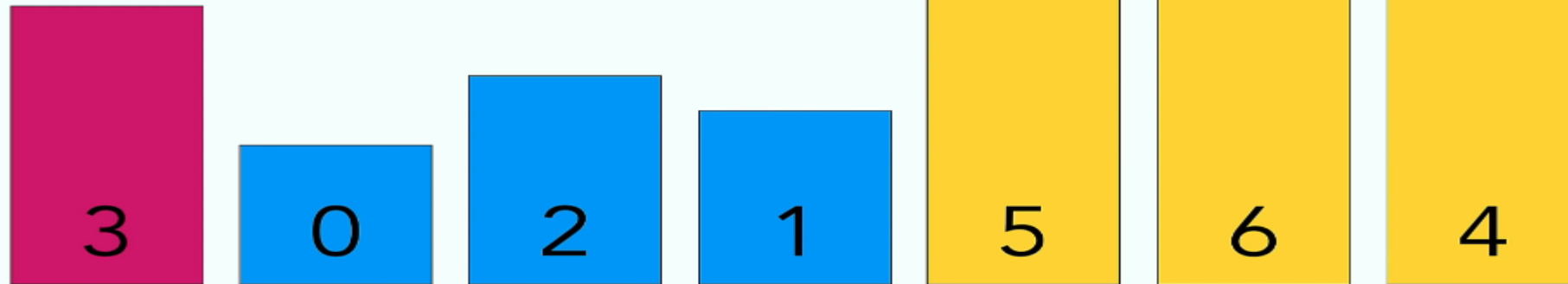
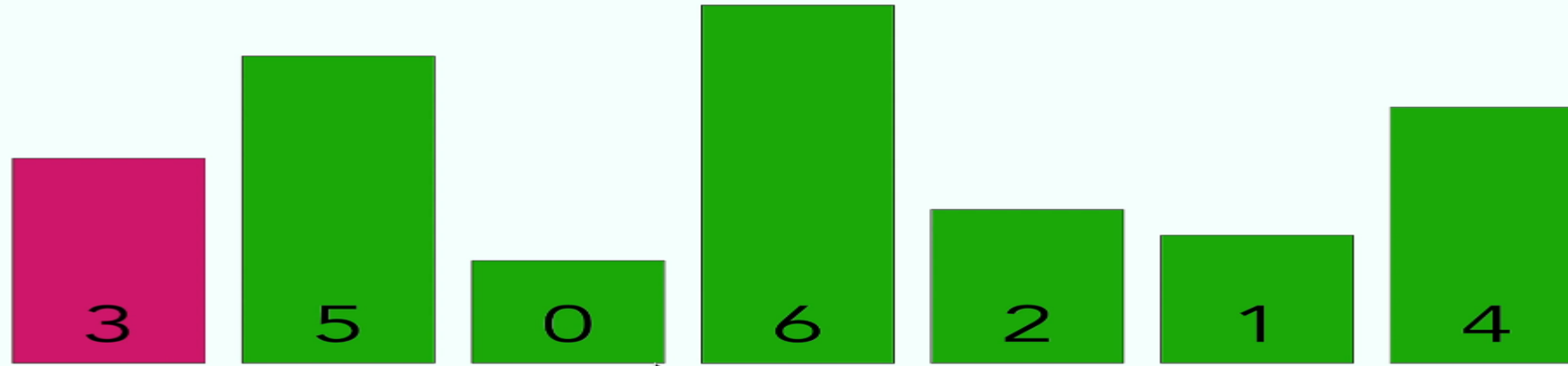
**1.Divide:** Choose a pivot element from the array.

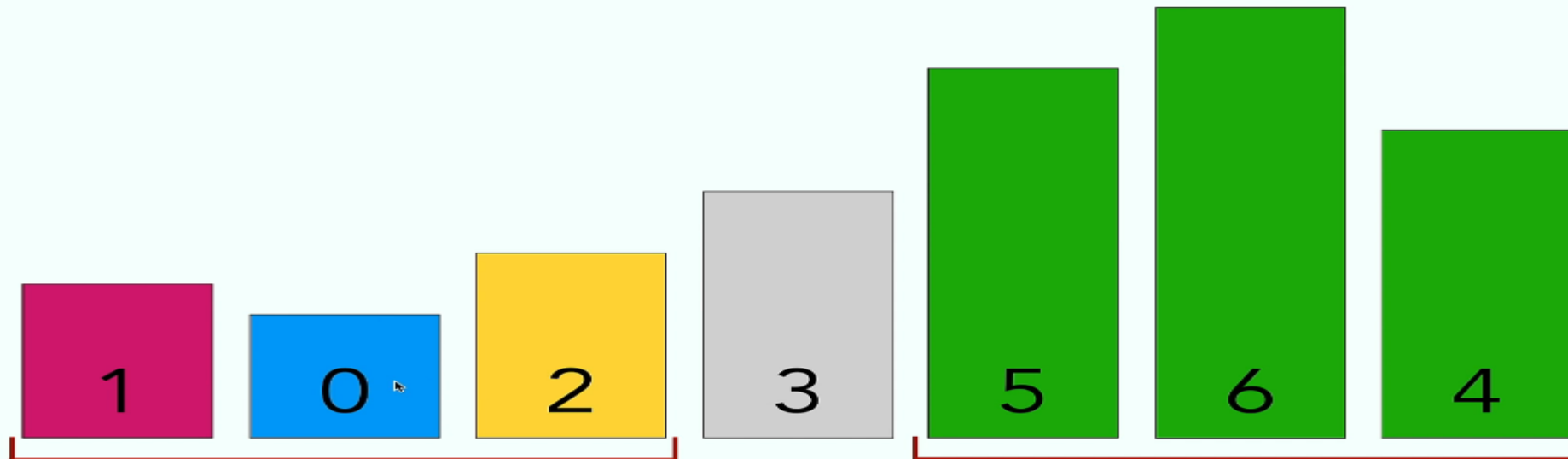
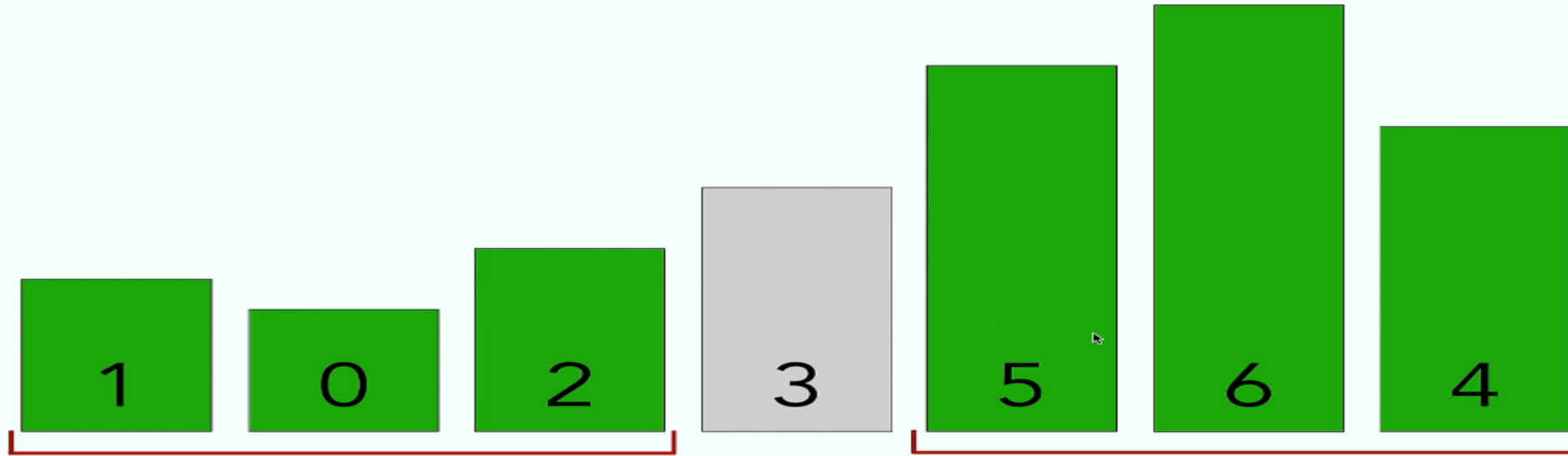
**2.Conquer:** Partition the array into two sub-arrays such that elements less than the pivot are on the left, and elements greater than the pivot are on the right.

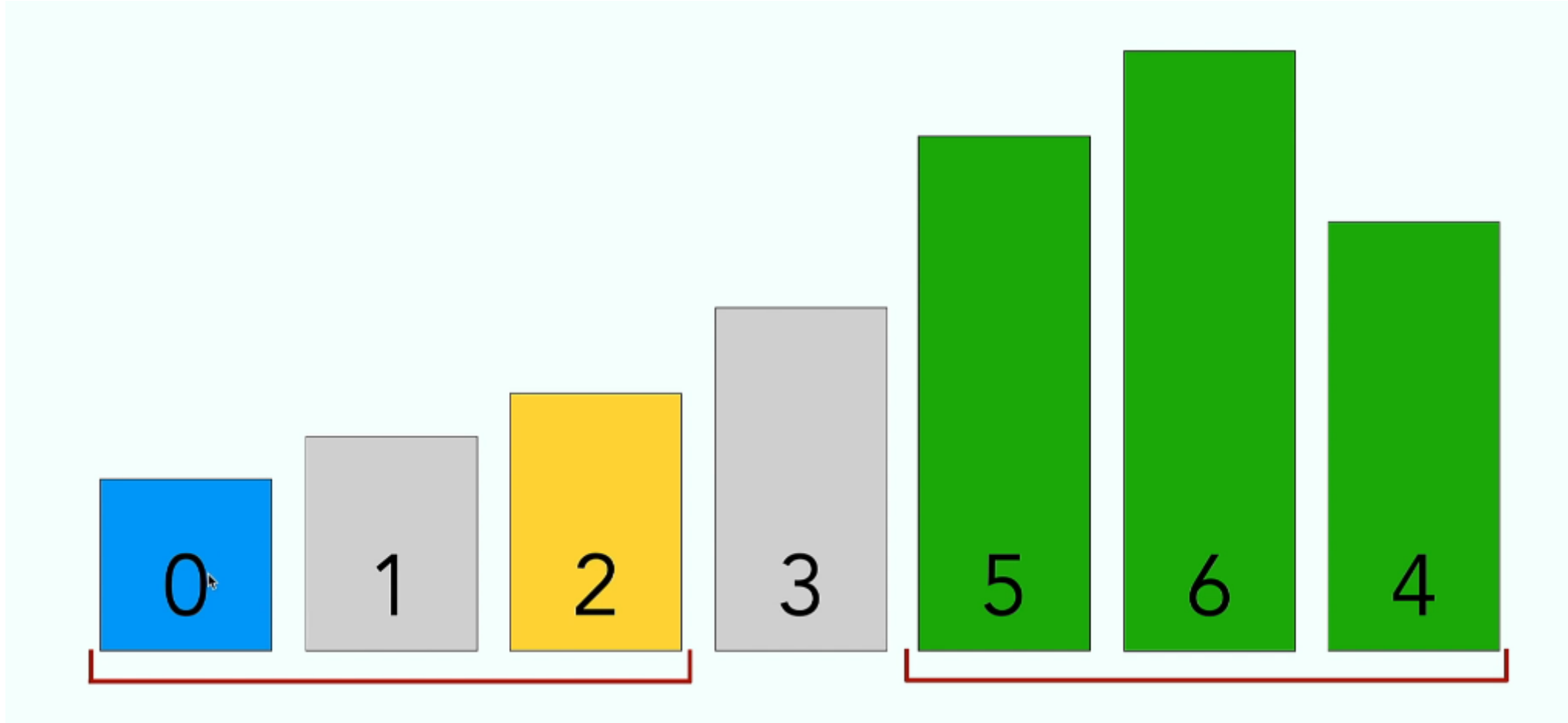
**3.Combine:** Recursively apply the above steps to the sub-arrays.

## Quick Sort

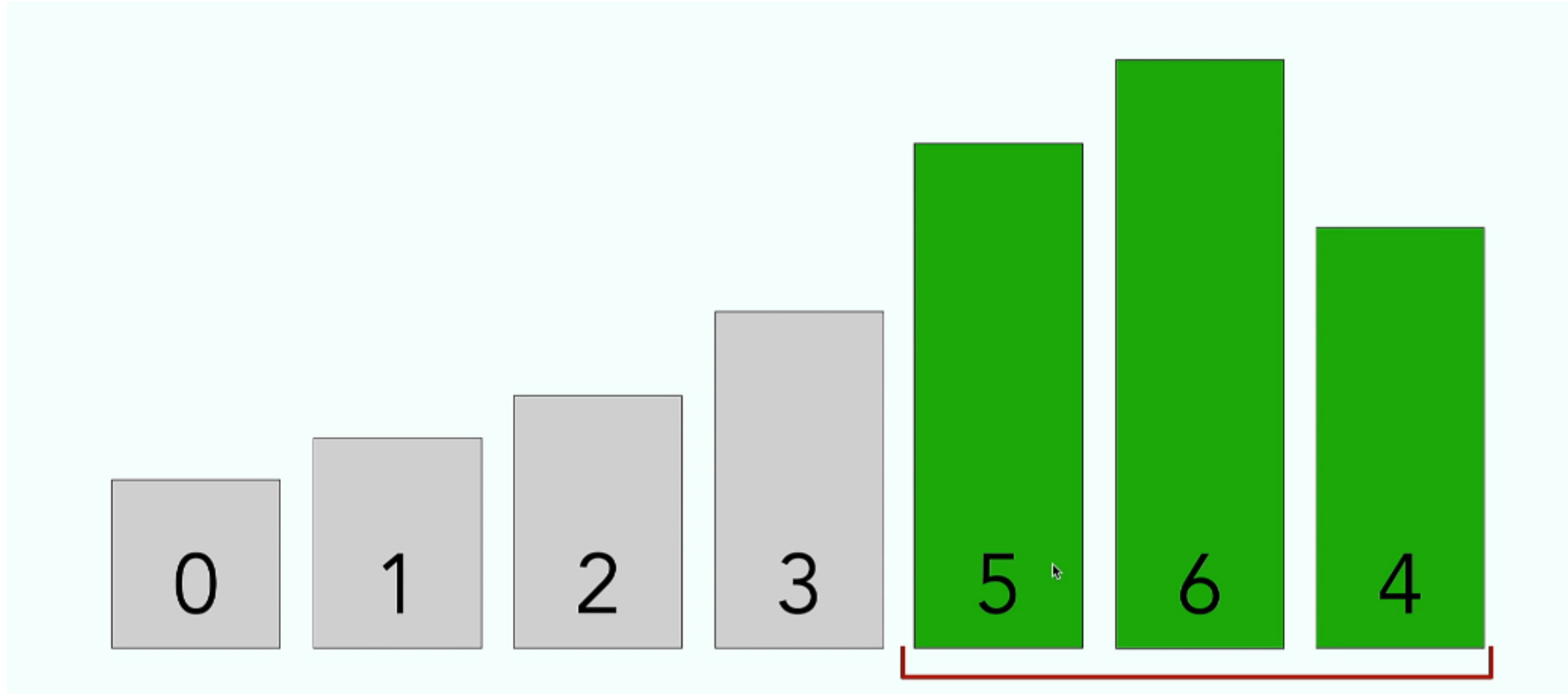


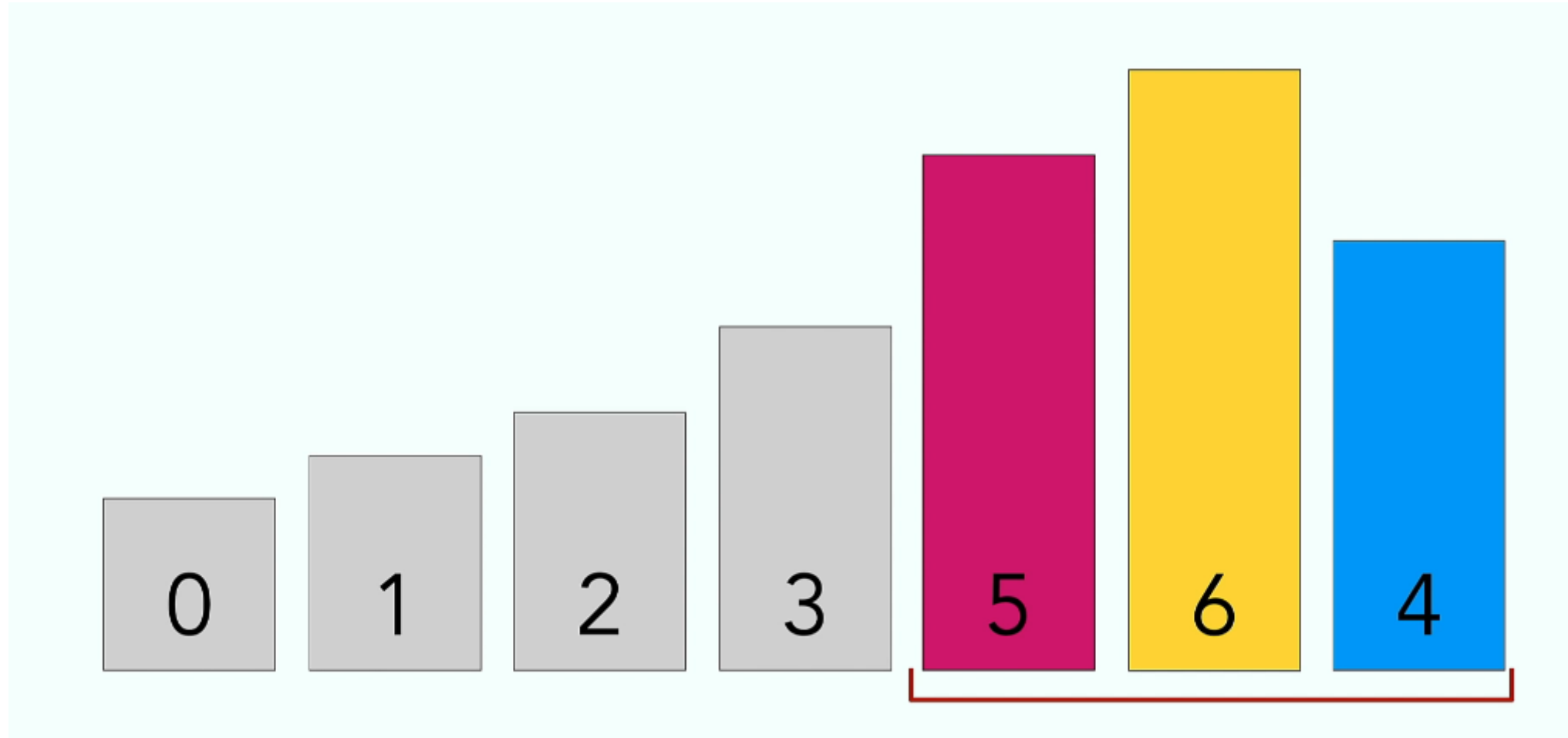


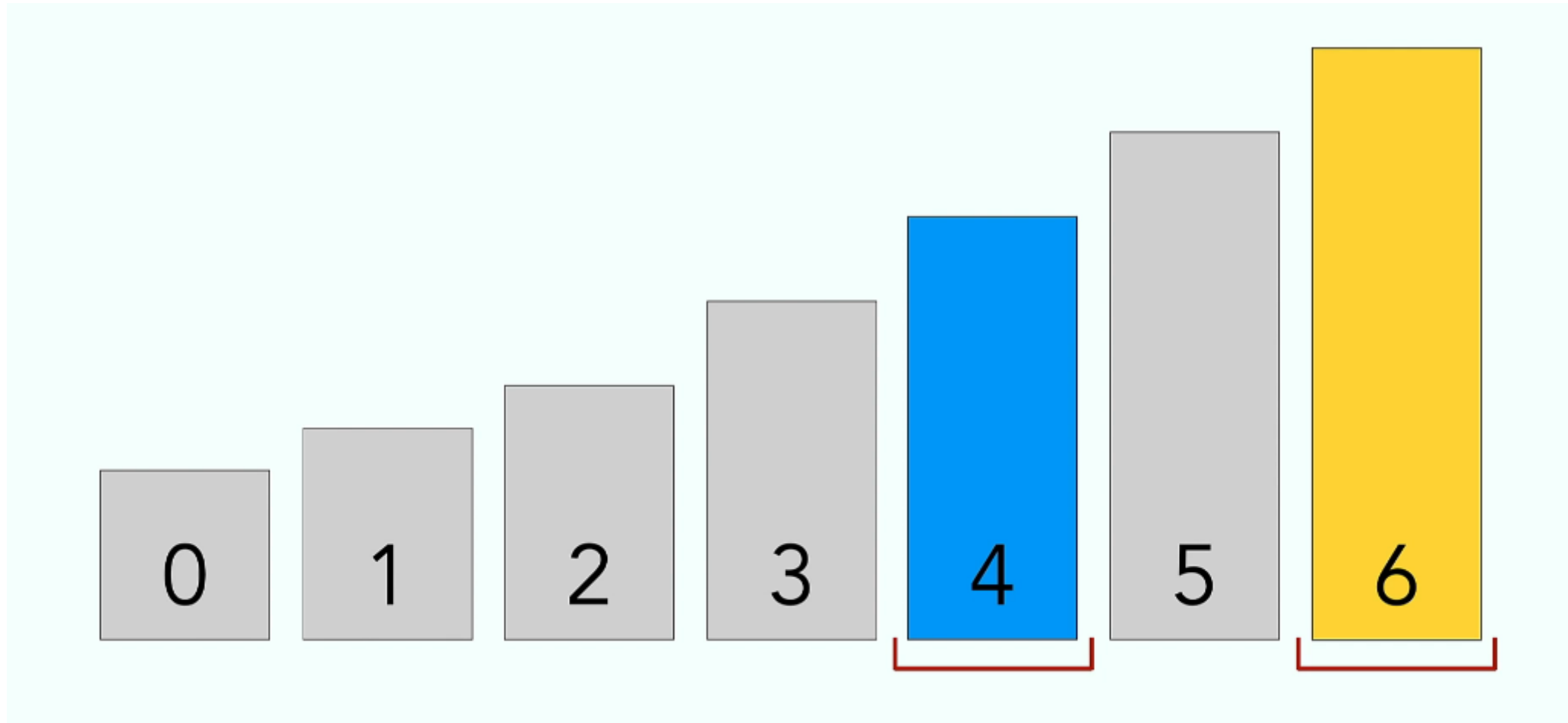


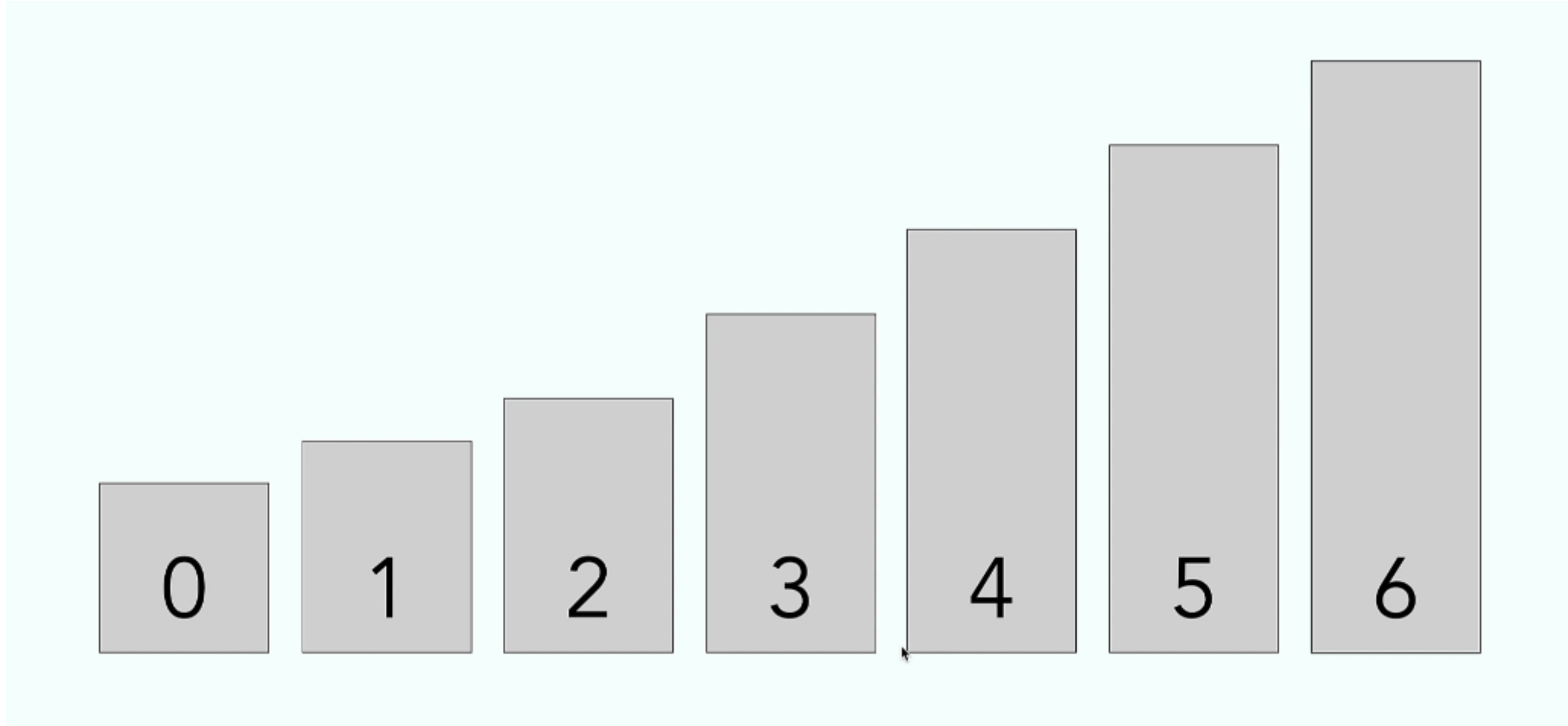












```
def quick_sort(arr):  
    if len(arr) <= 1:  
        return arr  
  
    pivot = arr[len(arr) // 2]  
    left = [x for x in arr if x < pivot]  
    middle = [x for x in arr if x == pivot]  
    right = [x for x in arr if x > pivot]  
    return quick_sort(left) + middle + quick_sort(right)
```

## Traceout

For array [3, 6, 8, 10, 1, 2, 1]:

1. Pivot = 8
2. Left = [3, 6, 1, 2, 1], Middle = [8], Right = [10]
3. Recursively sort Left and Right sub-arrays.

## Complexity

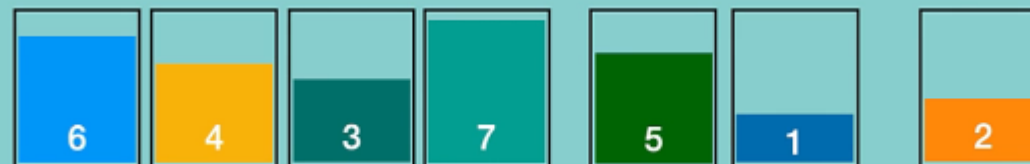
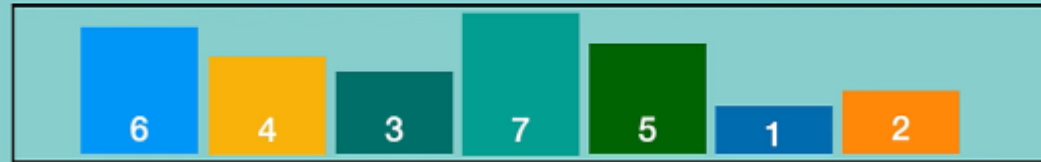
- Average Time:  $O(n \log n)$
- Worst Time:  $O(n^2)$
- Space:  $O(\log n)$  (due to recursion stack)

# Merge Sort

- **Divide:** Split the array into two halves.
- **Conquer:** Recursively sort each half.
- **Combine:** Merge the two sorted halves to produce the sorted array.

# Common Divide and Conquer Algorithms

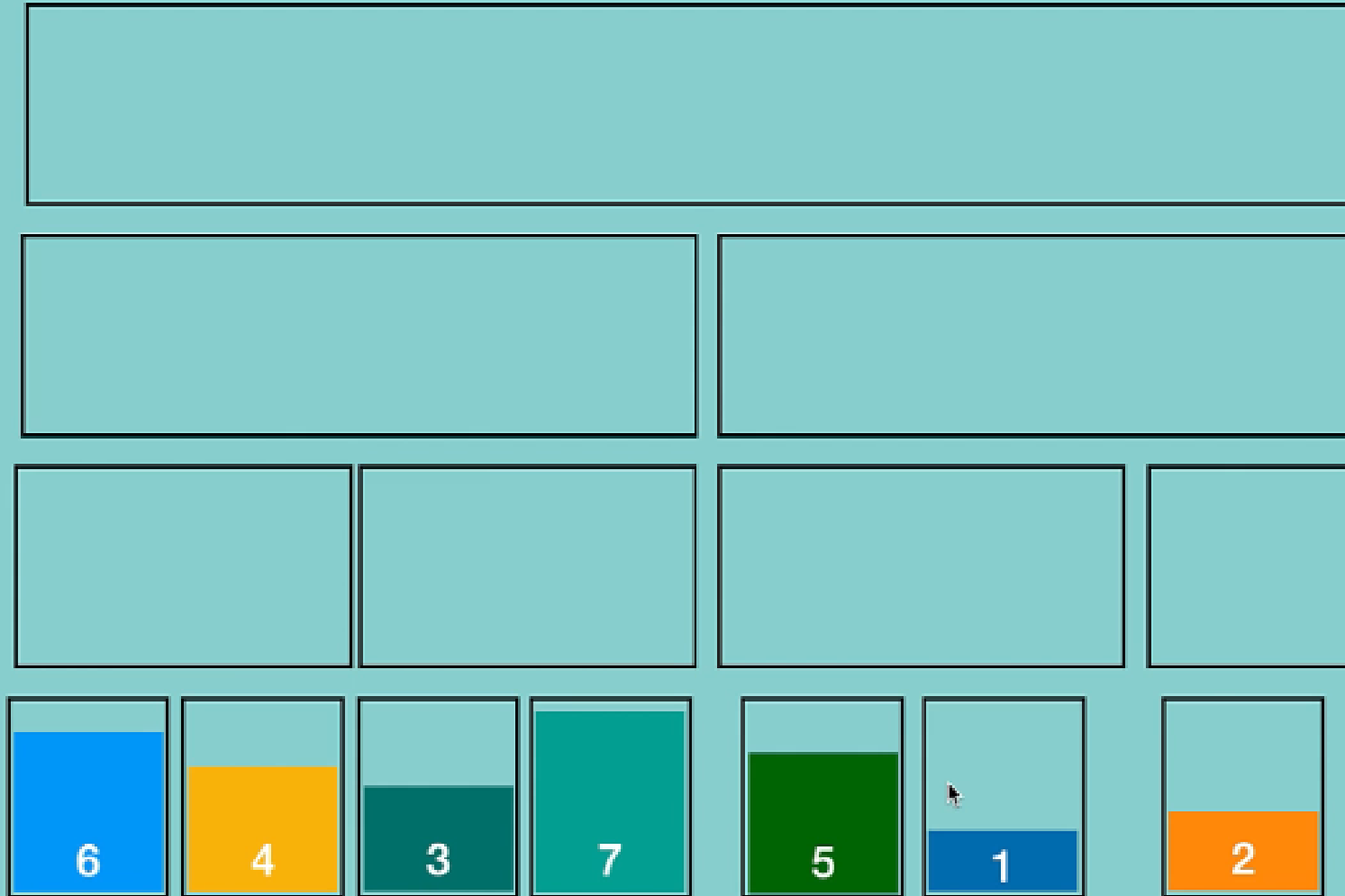
## Merge Sort





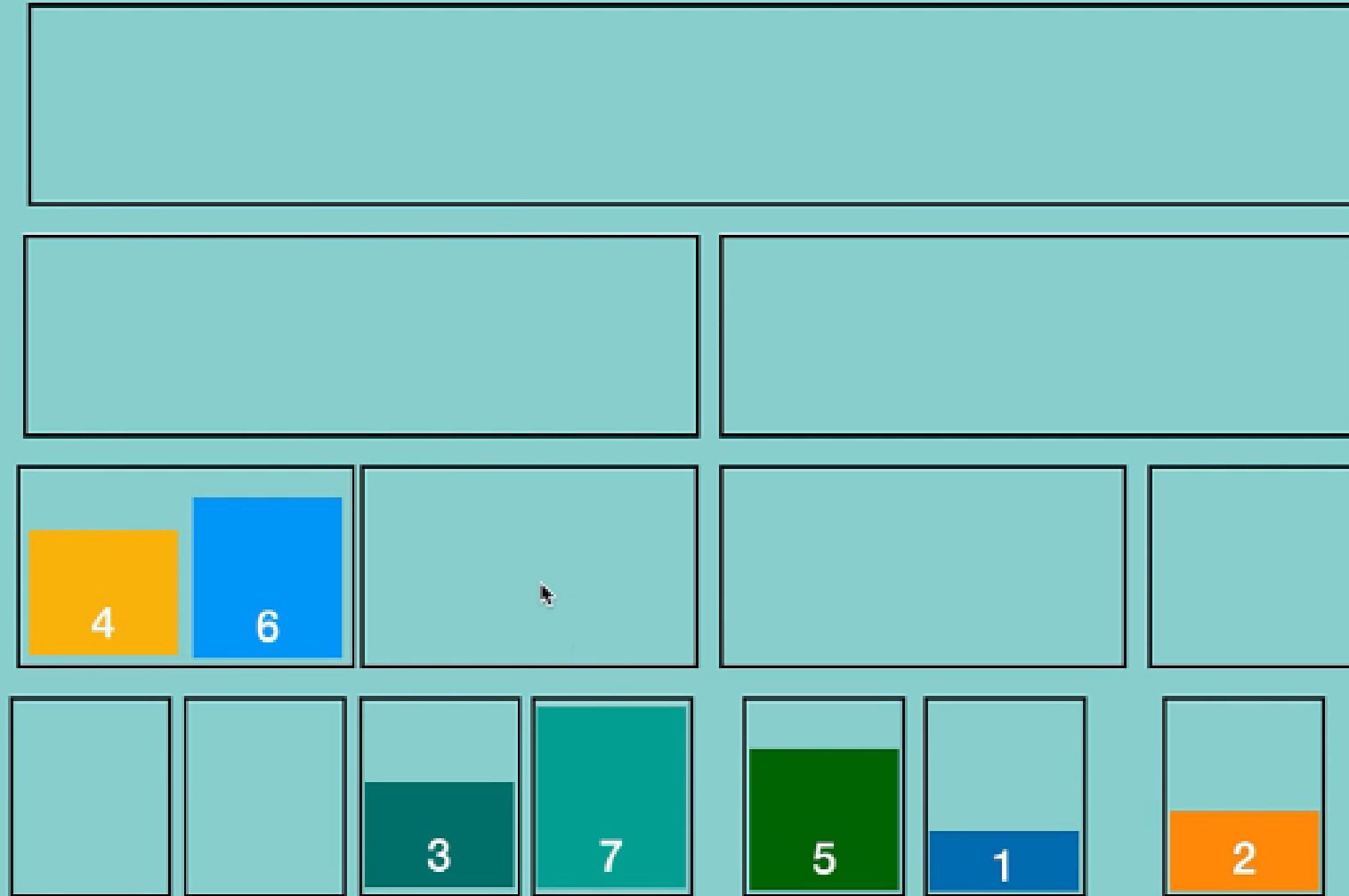
# Common Divide and Conquer Algorithms

## Merge Sort



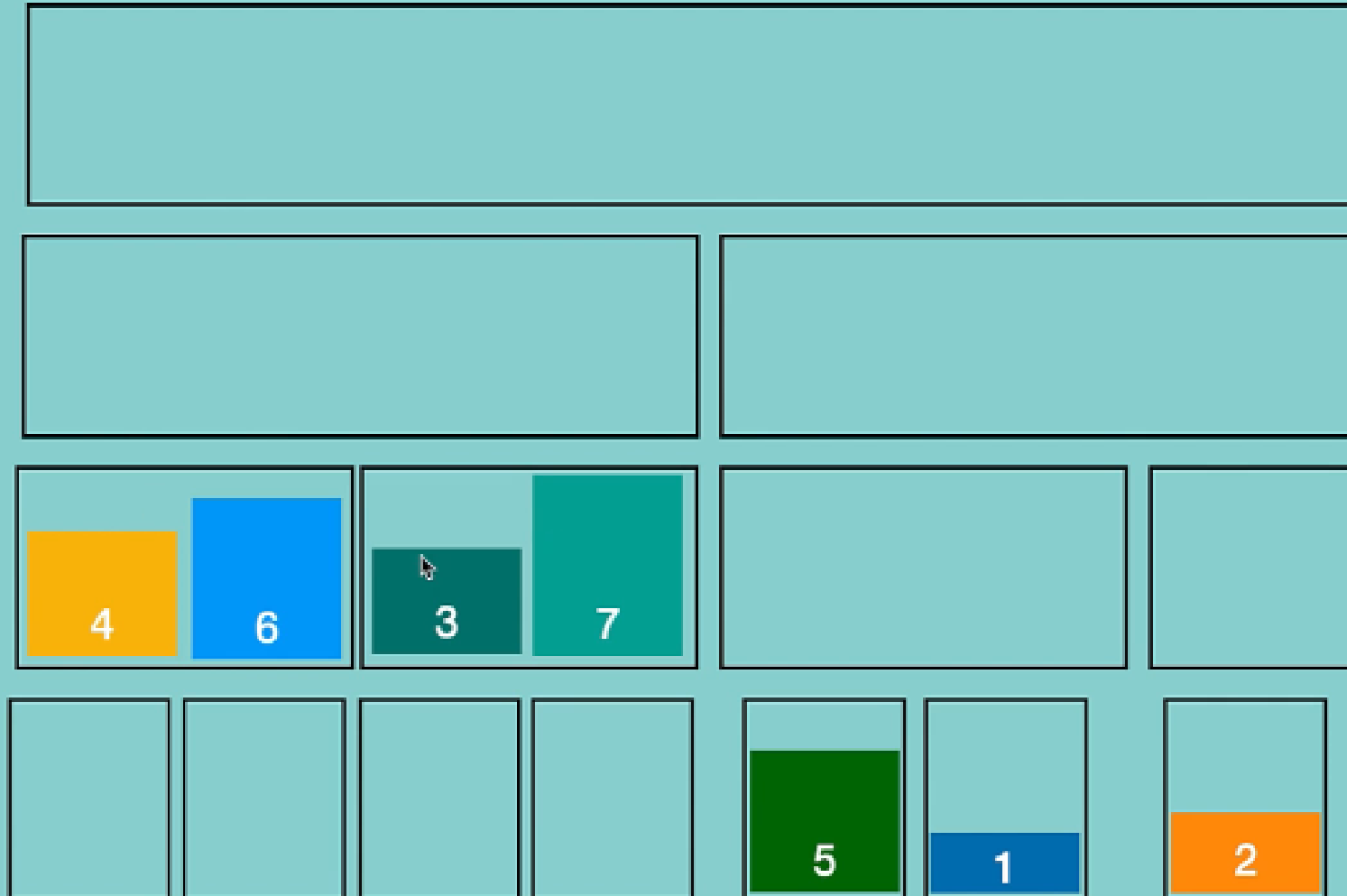
# Common Divide and Conquer Algorithms

## Merge Sort



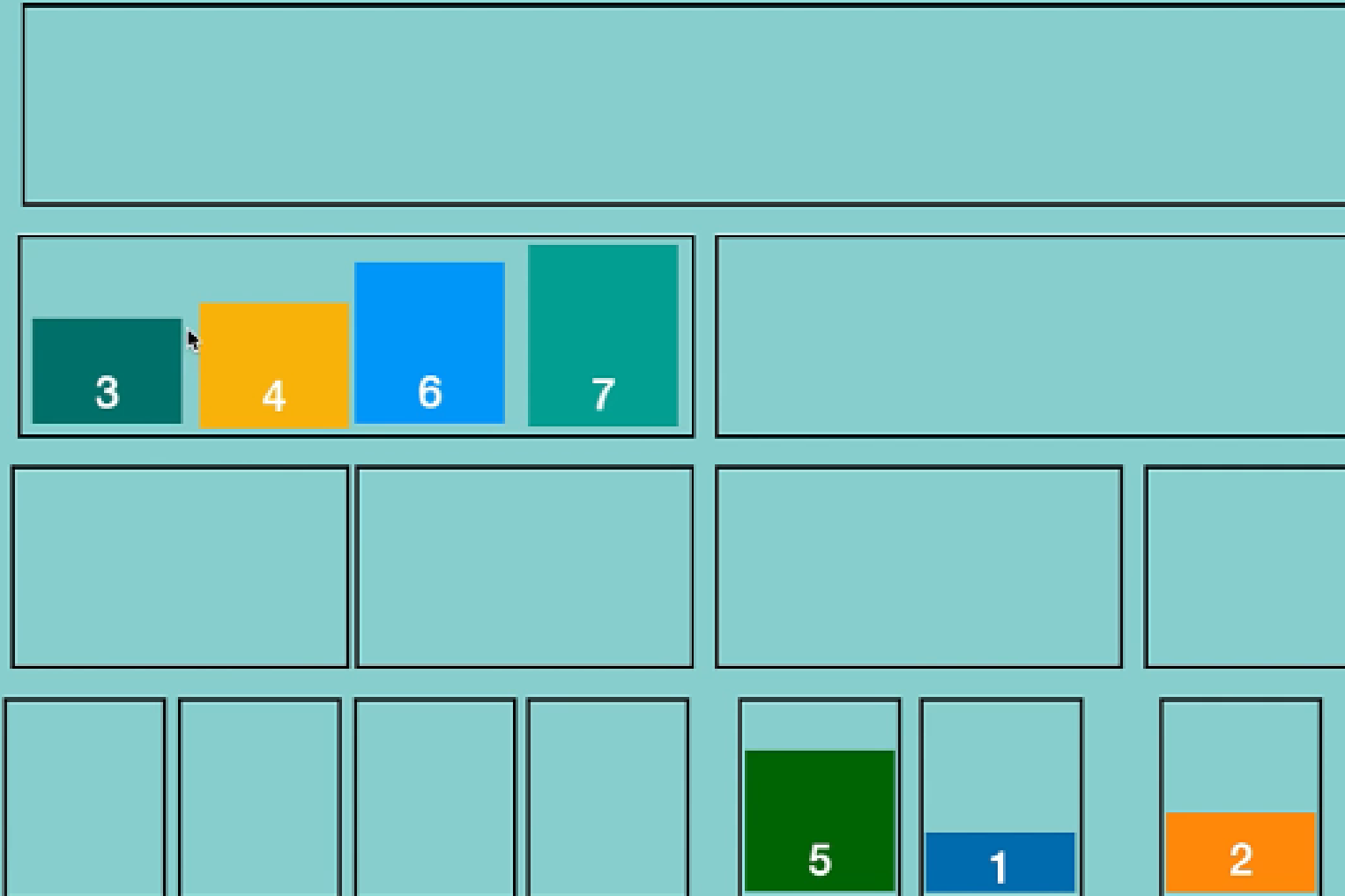
# Common Divide and Conquer Algorithms

## Merge Sort



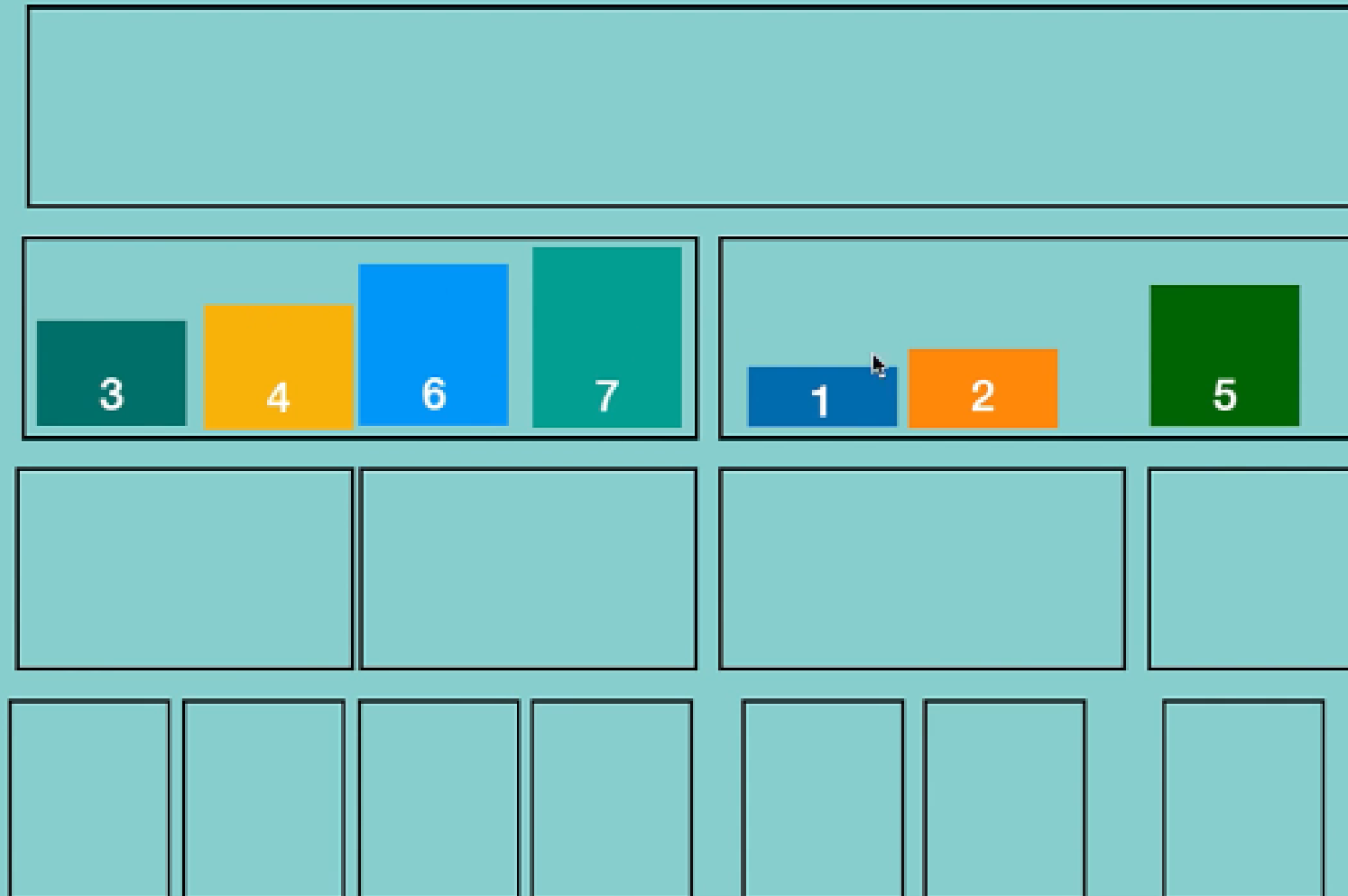
# Common Divide and Conquer Algorithms

## Merge Sort



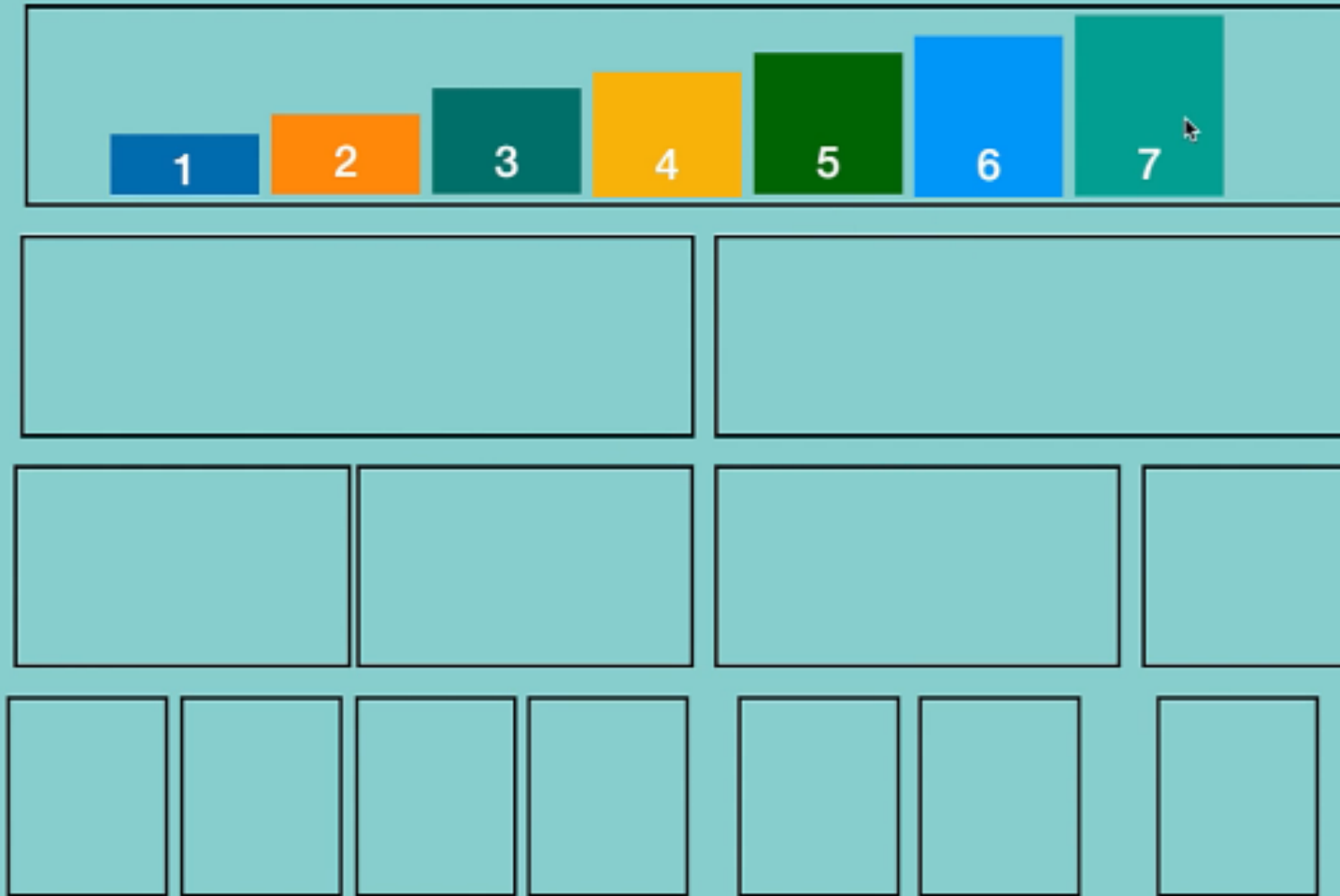
# Common Divide and Conquer Algorithms

## Merge Sort




# Common Divide and Conquer Algorithms

## Merge Sort



```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Recursively apply merge\_sort to the left half of   
Merge the two sorted halves together and return the result.

merge that takes two sorted lists left and right as inputs

Append the smaller element from left to result

Otherwise, append the smaller element from right to result  
and move the index j to the next position.

Add any remaining elements from left to result.

Add any remaining elements from right to result.

Loop while there are elements in both left and right lists to be merged.

If the current element in left is smaller than the current element in right:

Append the smaller element from left to result and move the index i to the next position.

Otherwise, append the smaller element from right to result  
and move the index j to the next position.

## Traceout

For array [38, 27, 43, 3, 9, 82, 10]:

1. Split into [38, 27, 43] and [3, 9, 82, 10]
2. Further split into [38], [27, 43], [3, 9], [82, 10]
3. Merge step by step to get the sorted array.

## Complexity

- Time:  $O(n \log n)$
- Space:  $O(n)$



# Strassen Multiplication

- Algorithm
- Divide: Split each matrix into four sub-matrices.
- Conquer: Perform recursive multiplications on the sub-matrices using Strassen's formulas.
- Combine: Combine the results to get the final matrix.

# Divide and Conquer :

Following is simple Divide and Conquer method to multiply two square matrices.

- Divide matrices A and B in 4 sub-matrices of size  $N/2 \times N/2$  as shown in the below diagram.
- Calculate following values recursively.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A                      B                      C

A, B and C are square matrices of size  $N \times N$

a, b, c and d are submatrices of A, of size  $N/2 \times N/2$

e, f, g and h are submatrices of B, of size  $N/2 \times N/2$

# Example

Array A =>

1	1	1	1
2	2	2	2
3	3	3	3
2	2	2	2

Array B =>

1	1	1	1
2	2	2	2
3	3	3	3
2	2	2	2

# Result Array =>

8	8	8	8
16	16	16	16
24	24	24	24
16	16	16	16

Suppose we have two 2x2 matrices  $A$  and  $B$ :

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

1. Split matrices  $A$  and  $B$  into sub-matrices:

$A$ :

$$A_{11} = [1], A_{12} = [2], A_{21} = [3], A_{22} = [4]$$

$B$ :

$$B_{11} = [5], B_{12} = [6], B_{21} = [7], B_{22} = [8]$$

# Strassen's Formula

$$P1 = (a11 + a22) * (b11 + b22)$$

$$P2 = (a21 + a22) * b11$$

$$P3 = (b12 - b22) * a11$$

$$P4 = (b21 - b11) * a22$$

$$P5 = (a11 + a12) * b22$$

$$P6 = (a21 - a11) * (b11 + b12)$$

$$P7 = (a12 - a22) * (b21 + b22)$$

$$C00 = P1 + P4 - P5 + P7$$

$$C01 = P3 + P5$$

$$C10 = P2 + P4$$

$$C11 = P1 + P3 - P2 + P6$$

Here, C00, C01, C10, and C11 are the elements of the 2\*2 matrix.

3. Combine the sub-matrices to get the final result:

$$C_{11} = M_1 + M_4 - M_5 + M_7 = 65 + 8 - 24 - 30 = 19$$

$$C_{12} = M_3 + M_5 = -2 + 24 = 22$$

$$C_{21} = M_2 + M_4 = 35 + 8 = 43$$

$$C_{22} = M_1 + M_3 - M_2 + M_6 = 65 - 2 - 35 + 22 = 50$$

Combine  $C_{11}$ ,  $C_{12}$ ,  $C_{21}$ ,  $C_{22}$  to form the result matrix  $C$ :

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

# Strassen's Formula for Multiplication

$$p1 = a(f - h)$$

$$p3 = (c + d)e$$

$$p5 = (a + d)(e + h)$$

$$p7 = (a - c)(e + f)$$

$$p2 = (a + b)h$$

$$p4 = d(g - e)$$

$$p6 = (b - d)(g + h)$$

The A x B can be calculated using above seven multiplications.

Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A                      B                      C

A, B and C are square matrices of size N x N

a, b, c and d are submatrices of A, of size N/2 x N/2

e, f, g and h are submatrices of B, of size N/2 x N/2

p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2



# Complexity Analysis for $N \times N$ matrix

## Naive Matrix Multiplication

- **Process:** Multiplies each element of the row of the first matrix by each element of the column of the second matrix.
- **Time Complexity:**
  - Requires  $N^3$  multiplications and  $N^2$  additions for two  $N \times N$  matrices.
  - Therefore, the time complexity is  $O(N^3)$ .

### Total Multiplications for $2 \times 2$ Matrices

- For  $c_{11}$ : 2 multiplications
- For  $c_{12}$ : 2 multiplications
- For  $c_{21}$ : 2 multiplications
- For  $c_{22}$ : 2 multiplications

Total number of multiplications:  $2 + 2 + 2 + 2 = 8$

## Strassen's Matrix Multiplication

- **Improvement:** Reduces the number of multiplications by splitting matrices into submatrices.
- **Process:**
  - Divides  $N \times N$  matrices into  $N/2 \times N/2$  submatrices.
  - Uses 7 multiplications and 18 additions/subtractions on  $N/2 \times N/2$  submatrices.
- **Time Complexity:**
  - The recurrence relation for Strassen's algorithm is  $T(N) = 7T(N/2) + O(N^2)$ .
  - Applying the Master Theorem, we get  $T(N) = O(N^{\log_2 7}) \approx O(N^{2.81})$ .

```
def strassen_multiply(A, B):
    if len(A) == 1:
        return [[A[0][0] * B[0][0]]]
    # Split A and B into sub-matrices
    mid = len(A) // 2
    A11, A12, A21, A22 = split_matrix(A)
    B11, B12, B21, B22 = split_matrix(B)

    # Strassen's 7 multiplications
    M1 = strassen_multiply(add(A11, A22), add(B11, B22))
    M2 = strassen_multiply(add(A21, A22), B11)
    M3 = strassen_multiply(A11, subtract(B12, B22))
    M4 = strassen_multiply(A22, subtract(B21, B11))
    M5 = strassen_multiply(add(A11, A12), B22)
    M6 = strassen_multiply(subtract(A21, A11), add(B11, B12))
    M7 = strassen_multiply(subtract(A12, A22), add(B21, B22))

    # Combine sub-matrices
    C11 = add(subtract(add(M1, M4), M5), M7)
    C12 = add(M3, M5)
    C21 = add(M2, M4)
    C22 = add(subtract(add(M1, M3), M2), M6)

    return combine_matrix(C11, C12, C21, C22)
```

## Traceout

For matrices  $A$  and  $B$ :

1. Split into sub-matrices and perform 7 multiplications.
2. Combine the results according to Strassen's formulas.

## Complexity

- Time:  $O(n^{\log_2 7}) \approx O(n^{2.81})$
- Space:  $O(n^2)$

# Max-Min Problem

- **Algorithm**

**1.Divide:** Split the array into two halves.

**2.Conquer:** Recursively find the maximum and minimum in each half.

**3.Combine:** Compare the maximums and minimums of the two halves to get the overall maximum and minimum.

# Finding Max and Min without DAC

```
for i = 2 to n do
  if numbers[i] >= max then
    max := numbers[i]
  else
    if numbers[i] <= min then
      min := numbers[i]
return (max, min)
```

20	3	40	2	60	12
----	---	----	---	----	----

Max = 20

Min = 20

Best Case (n-1) eg: 1,2,3,4

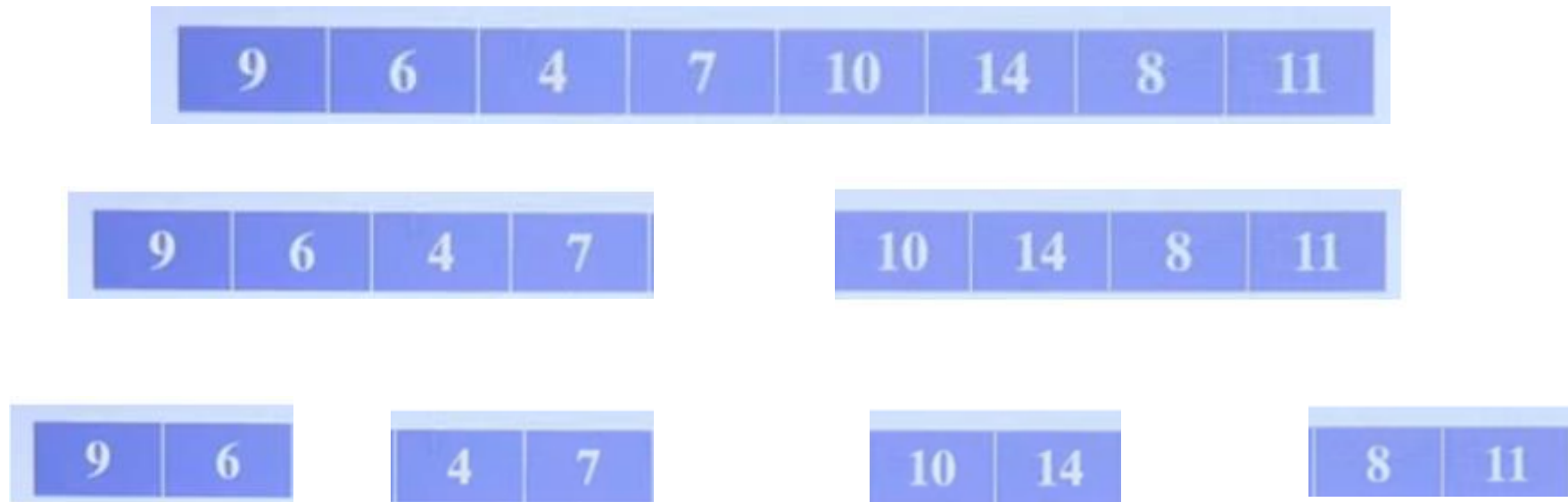
Worst Case 2(n-1) eg: 4,3,2,1

where n is number of elements

```
function findMaxMin(arr):  
    if arr is empty:  
        return (None, None)  
  
    max_value = arr[0]  
    min_value = arr[0]  
  
    for each element in arr:  
        if element > max_value:  
            max_value = element  
        if element < min_value:  
            min_value = element  
  
    return (max_value, min_value)
```

# Max-Min Problem

- The Max-Min problem involves finding both the maximum and minimum values in an array of numbers.





```
function find_max_min(arr, low, high):  
    if low == high:  
        return arr[low], arr[low]  
  
    if high - low == 1:  
        if arr[low] < arr[high]:  
            return arr[low], arr[high]  
        else:  
            return arr[high], arr[low]  
  
    mid = (low + high) / 2  
    max1, min1 = find_max_min(arr, low, mid)  
    max2, min2 = find_max_min(arr, mid + 1, high)  
  
    max_val = max(max1, max2)  
    min_val = min(min1, min2)  
  
    return max_val, min_val
```

## Traceout

For array [4, 3, 2, 1, 5, 6]:

1. Split into [4, 3, 2] and [1, 5, 6]
2. Recursively find max and min in each half.
3. Compare results from each half to get final max and min.

## Complexity

- Time:  $O(n)$
- Space:  $O(\log n)$  (due to recursion stack)

## Complexity Analysis

- **Time Complexity:** The time complexity of the Divide and Conquer approach for finding both maximum and minimum elements in an array is  $O(n)$ , where  $n$  is the number of elements in the array. This is because each element is processed a constant number of times during the recursive calls, and combining results takes constant time.
- **Space Complexity:** The space complexity is  $O(\log n)$  due to the recursive calls, which consume memory on the call stack proportional to the depth of recursion.

Thank You!

