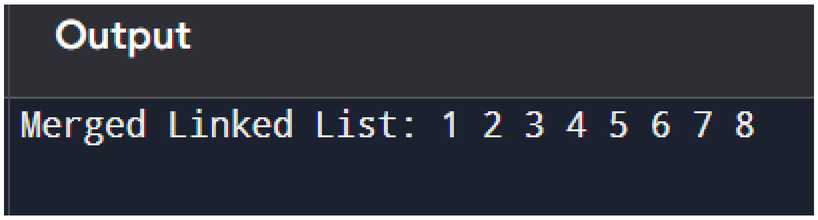
MERGE SORT

```cpp
#include <iostream>
using namespace std;

// Node class for the linked list
class Node {
private:
    int data;
    Node* next;

public:
    // Constructor
    Node(int val) : data(val), next(nullptr) {}

    // Setter for data
    void setData(int val) {
        data = val;
    }

    // Getter for data
    int getData() const {
        return data;
    }

    // Setter for next pointer
    void setNext(Node* nextNode) {
        next = nextNode;
    }

    // Getter for next pointer
    Node* getNext() const {
        return next;
    }
};

// LinkedList class
class LinkedList {
private:
    Node* head;

public:
    // Constructor
    LinkedList() : head(nullptr) {}

    // Getter for head
    Node* getHead() const {
        return head;
    }

    // Setter for head
    void setHead(Node* newHead) {
        head = newHead;
    }

    // Function to append a value to the linked list
    void append(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->getNext()) {
            temp = temp->getNext();
        }
        temp->setNext(newNode);
    }

    // Function to display the linked list
    void display() const {
        Node* temp = head;
        while (temp) {
            cout << temp->getData() << " ";
            temp = temp->getNext();
        }
        cout << endl;
    }
};

// Function to merge two sorted linked lists into a third list

void mergeSortedLists(LinkedList& list1, LinkedList& list2, LinkedList& list3) {

    Node* ptr1 = list1.getHead(); // Pointer to traverse list1
    Node* ptr2 = list2.getHead(); // Pointer to traverse list2

    // Traverse both lists and insert nodes in sorted order

    while (ptr1 != nullptr && ptr2 != nullptr) {
        if (ptr1->getData() < ptr2->getData()) {
            list3.append(ptr1->getData()); // Add the smaller value to list3
            ptr1 = ptr1->getNext();       // Move to the next node in list1
        }
        else {
            list3.append(ptr2->getData()); // Add the smaller value to list3
            ptr2 = ptr2->getNext();       // Move to the next node in list2
        }
    }

    // If there are remaining nodes in list1, add them to list3

    while (ptr1 != nullptr) {
        list3.append(ptr1->getData());
        ptr1 = ptr1->getNext();
    }

    // If there are remaining nodes in list2, add them to list3

    while (ptr2 != nullptr) {
        list3.append(ptr2->getData());
        ptr2 = ptr2->getNext();
    }
}

// Driver code
int main() {
    LinkedList list1, list2, list3;

    // Insert values into the first sorted list
    list1.append(1);
    list1.append(3);
    list1.append(5);
    list1.append(7);

    // Insert values into the second sorted list
    list2.append(2);
    list2.append(4);
    list2.append(6);
    list2.append(8);

    // Merge the two sorted lists into the third list
    mergeSortedLists(list1, list2, list3);

    // Display the merged list
    cout << "Merged Linked List: ";
    list3.display();

    return 0;
}
```

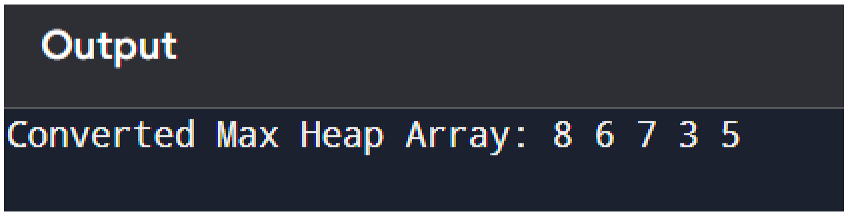


```
Output

Merged Linked List: 1 2 3 4 5 6 7 8
```

HEAP

```
#include <iostream>
using namespace std;

void convertToMaxHeapAndPrint(int arr[], int n) {

    // Convert the min heap to max heap starting from last non-leaf node

    for (int i = n / 2 - 1; i >= 0; i--) {
        int largest = i;
        int current = i;

        while (true) {
            int left = 2 * current + 1; // Left child index
            int right = 2 * current + 2; // Right child index

            // Find the largest element among current, left child, and right child

            if (left < n && arr[left] > arr[largest]) {
                largest = left;
            }
            if (right < n && arr[right] > arr[largest]) {
                largest = right;
            }

            // If current node is the largest, then break

            if (largest == current) break;

            // Swap and continue heapifying downwards

            swap(arr[current], arr[largest]);
            current = largest;
        }
    }
```

```cpp

        // Print the result as a max heap array
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    // Input min heap array
    int minHeap[] = {3, 5, 7, 6, 8};
    int n = sizeof(minHeap) / sizeof(minHeap[0]);

    cout << "Converted Max Heap Array: ";
    convertToMaxHeapAndPrint(minHeap, n);

    return 0;
}
```



```
Output

Converted Max Heap Array: 8 6 7 3 5
```

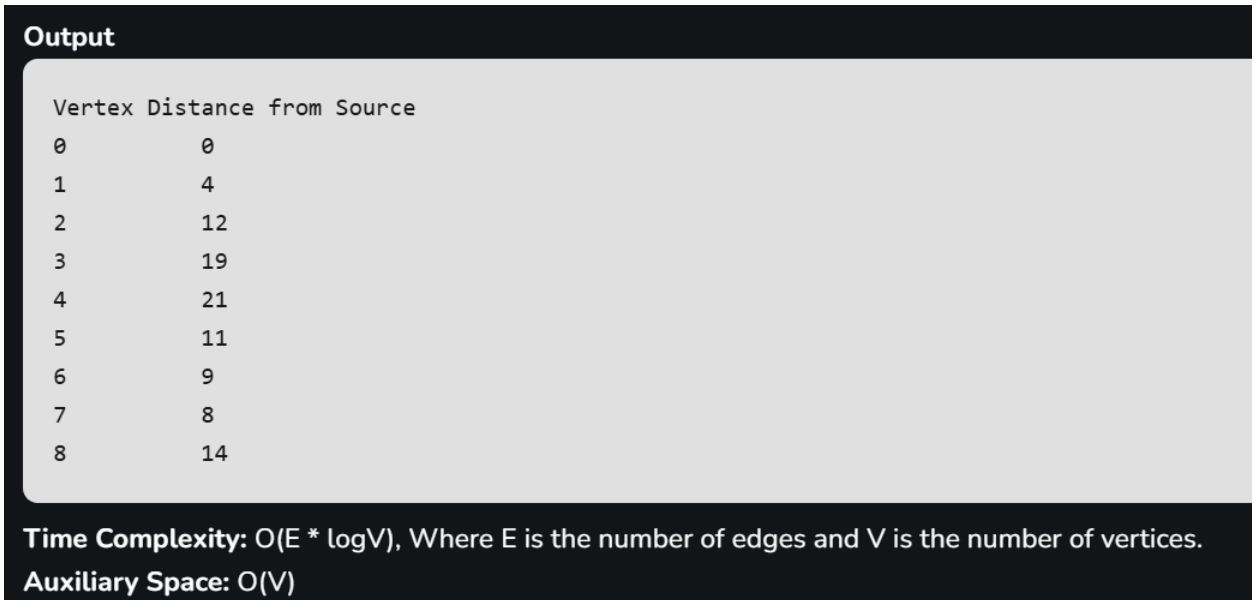
➢ Dijkstra Shortest Path

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

```cpp
#include <bits/stdc++.h>
using namespace std;

// Define INF as a large value to represent infinity
#define INF 0x3f3f3f3f

// iPair ==> Integer Pair

typedef pair<int, int> iPair;

// Class representing a graph using adjacency list representation

class Graph {
    int V; // Number of vertices
    list<iPair> *adj; // Adjacency list

public:
    Graph(int V); // Constructor

    void addEdge(int u, int v, int w); // Function to add an edge
    void shortestPath(int s); // Function to print shortest path from source
};

// Constructor to allocate memory for the adjacency list
Graph::Graph(int V) {
    this->V = V;
    adj = new list<iPair>[V];
```

```
 }

 // Function to add an edge to the graph
 void Graph::addEdge(int u, int v, int w) {
     adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w)); // Since the graph is undirected
}

// Function to print shortest paths from source

void Graph::shortestPath(int src) {

    // Create a priority queue to store vertices being processed

    priority_queue<iPair, vector<iPair>, greater<iPair>> pq;

    // Create an array to store distances and initialize all distances as INF

    int dist[V];
    for (int i = 0; i < V; ++i)
        dist[i] = INF;

    // Insert source into priority queue and initialize its distance as 0

    pq.push(make_pair(0, src));
    dist[src] = 0;

    // Process the priority queue

    while (!pq.empty()) {

        // Get the vertex with the minimum distance

        int u = pq.top().second;
        pq.pop();

        // Iterate through all adjacent vertices of the current vertex

        for (auto &neighbor : adj[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            // If a shorter path to v is found

            if (dist[v] > dist[u] + weight) {

                // Update distance and push new distance to the priority queue

                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    // Print the shortest distances
    cout << "Vertex Distance from Source" << endl;
    for (int i = 0; i < V; ++i)
        cout << i << " \t\t " << dist[i] << endl;
}

// Driver's code
int main() {
    int V = 9; // Number of vertices
    Graph g(V);
```

```
     // Add edges to the graph
     g.addEdge(0, 1, 4);
     g.addEdge(0, 7, 8);
     g.addEdge(1, 2, 8);
     g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    // Call the shortestPath function
    g.shortestPath(0);

    return 0;
}
```



Output

```
Vertex Distance from Source
0           0
1           4
2           12
3           19
4           21
5           11
6           9
7           8
8           14
```

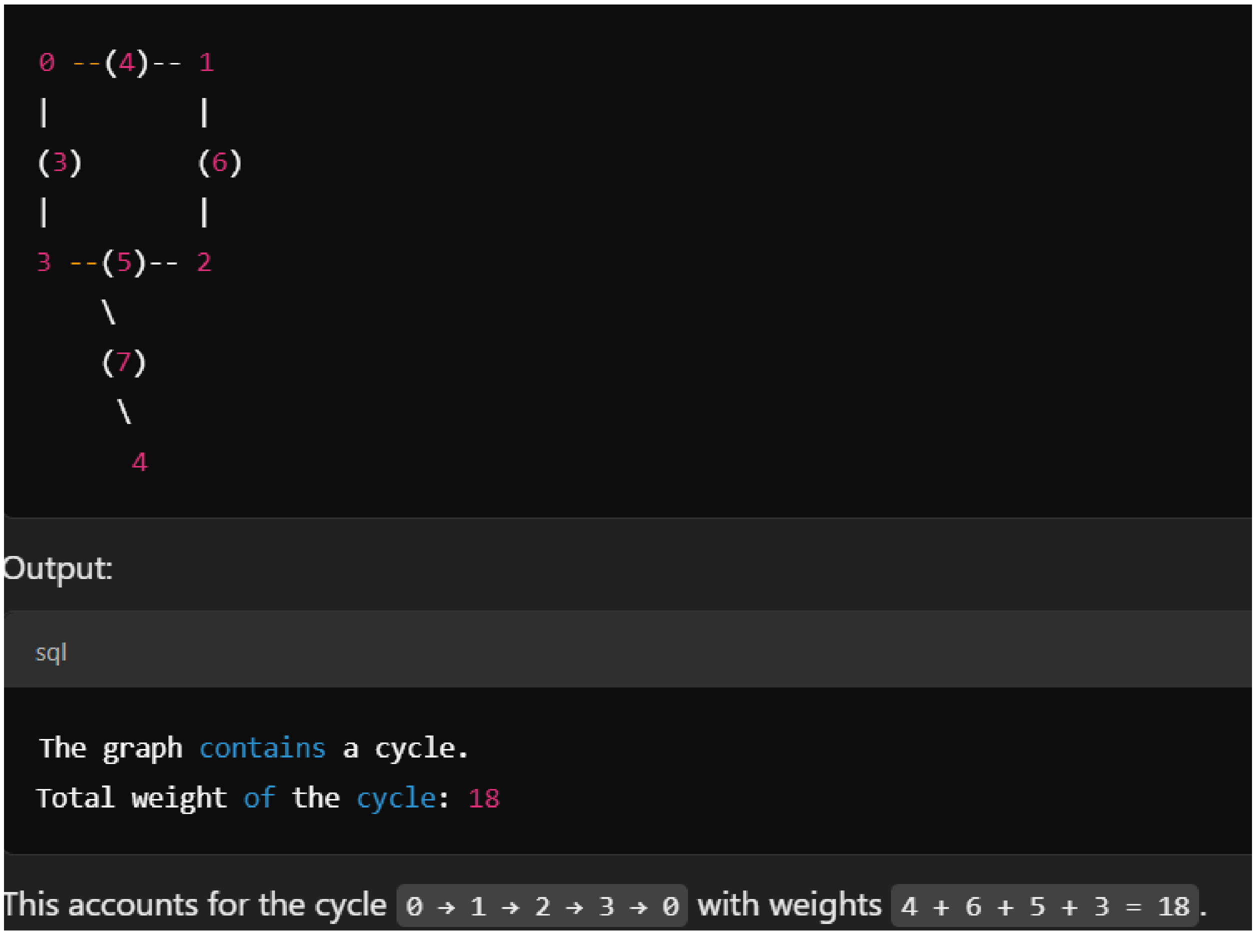
Time Complexity: O(E * logV), Where E is the number of edges and V is the number of vertices.
Auxiliary Space: O(V)

- To check if a **graph contains cycles** and calculates the total weight of the edges involved, we need to traverse the graph. For this, we can use **DFS (Depth First Search)** or **Union-Find (Disjoint Set Union)** methods depending on whether the graph is directed or undirected.

Here is the code for:

1. **Detecting cycles in a weighted graph** using DFS.
2. **Returning the sum of edge weights** if a cycle is detected.

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <cstring>
using namespace std;

// Edge structure to store graph edges
struct Edge {
    int u, v, weight; // Two vertices and the weight
};

// Class representing a weighted graph
class Graph {
    int V;                       // Number of vertices
    list<pair<int, int>>* adj; // Adjacency list (node, weight)

public:
    Graph(int V);
    void addEdge(int u, int v, int weight);
    bool hasCycle(int node, int parent, bool visited[], int& cycleWeight);
};

// Constructor to initialize the adjacency list
Graph::Graph(int V) {
    this->V = V;
    adj = new list<pair<int, int>>[V];
}

// Function to add an edge to the graph
void Graph::addEdge(int u, int v, int weight) {
    adj[u].push_back({v, weight});
    adj[v].push_back({u, weight}); // Since the graph is undirected
}

// Recursive DFS(Depth First Search) function to detect cycles and calculate weight

bool Graph::hasCycle(int node, int parent, bool visited[], int& cycleWeight) {
    visited[node] = true;

    for (auto& neighbor : adj[node]) {
        int v = neighbor.first;
        int weight = neighbor.second;

        // If neighbor is not visited, recurse

        if (!visited[v]) {
            cycleWeight += weight; // Add weight of the edge

            if (hasCycle(v, node, visited, cycleWeight)) {
                return true;
            }
        }
        // If visited and not the parent, it's a cycle

        else if (v != parent) {
            cycleWeight += weight; // Add weight of the edge in the cycle

            return true;
        }
    }
    return false;
}

int main() {
    int V = 5; // Number of vertices
    Graph g(V);

    // Add edges to the graph
    g.addEdge(0, 1, 4);
    g.addEdge(1, 2, 6);
    g.addEdge(2, 3, 5);
    g.addEdge(3, 0, 3);
   g.addEdge(3, 4, 7);

   bool visited[V];
   memset(visited, false, sizeof(visited));
   int cycleWeight = 0;

   // Check for cycles and calculate weights
   bool hasCycle = false;
   for (int i = 0; i < V; ++i) {
       if (!visited[i] && g.hasCycle(i, -1, visited, cycleWeight)) {
           hasCycle = true;
           break;
       }
   }

   if (hasCycle) {
       cout << "The graph contains a cycle.\n";
       cout << "Total weight of the cycle: " << cycleWeight << endl;
   } else {
       cout << "The graph does not contain a cycle.\n";
   }

   return 0;
}
```



```
0 --(4)-- 1
|         |
(3)       (6)
|         |
3 --(5)-- 2
    \
    (7)
     \
      4
```

Output:

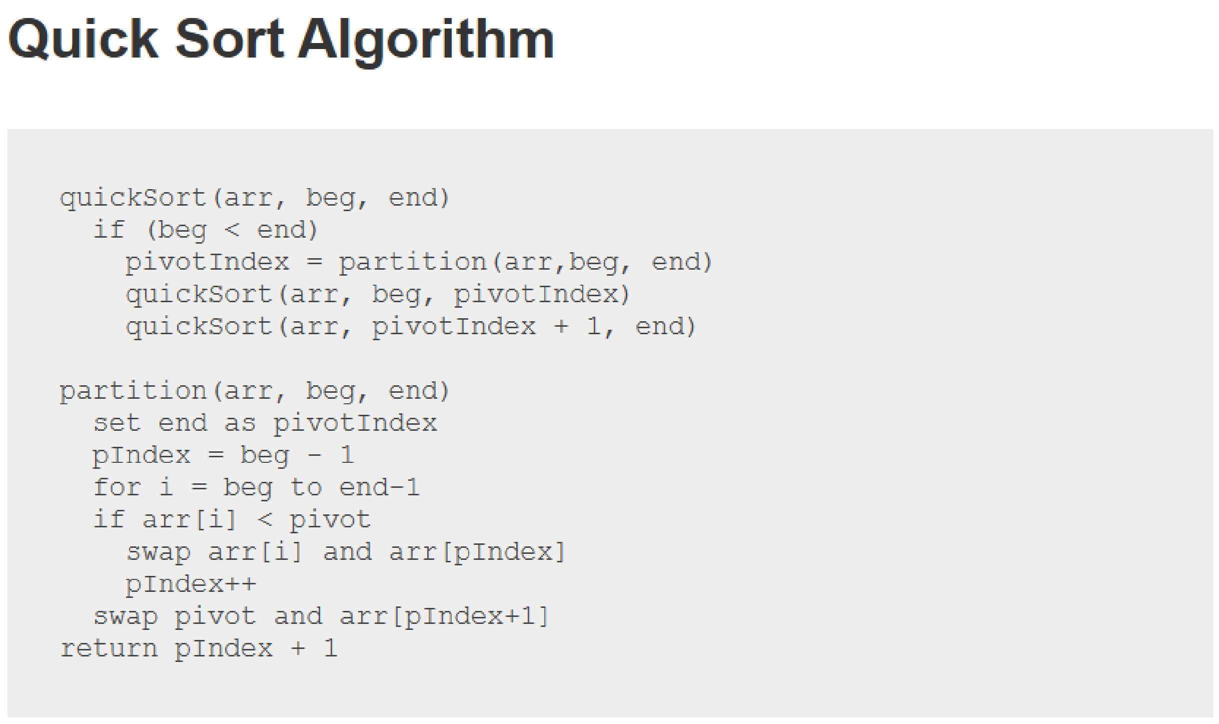
```sql
The graph contains a cycle.
Total weight of the cycle: 18
```

This accounts for the cycle `0 → 1 → 2 → 3 → 0` with weights `4 + 6 + 5 + 3 = 18`.

Quick Sort Pseudocode

- We are given with an input array
- Choose pivot, here we are choosing the last element as our pivot
- Now partition the array as per pivot
    - Keep a partitioned index say p and initialize it to -1
    - Iterate through every element in the array except the pivot
    - If an element is less than the pivot element then increment p and swap the elements at index p with the element at index i.

    - Once all the elements are traversed, swap pivot with element present at p+1 as this will the same position as in the sorted array
    - Now return the pivot index
- Once partitioned, now make 2 calls on quicksort
    - One from beg to p-1
    - Other from p+1 to n-1



Quick Sort Algorithm

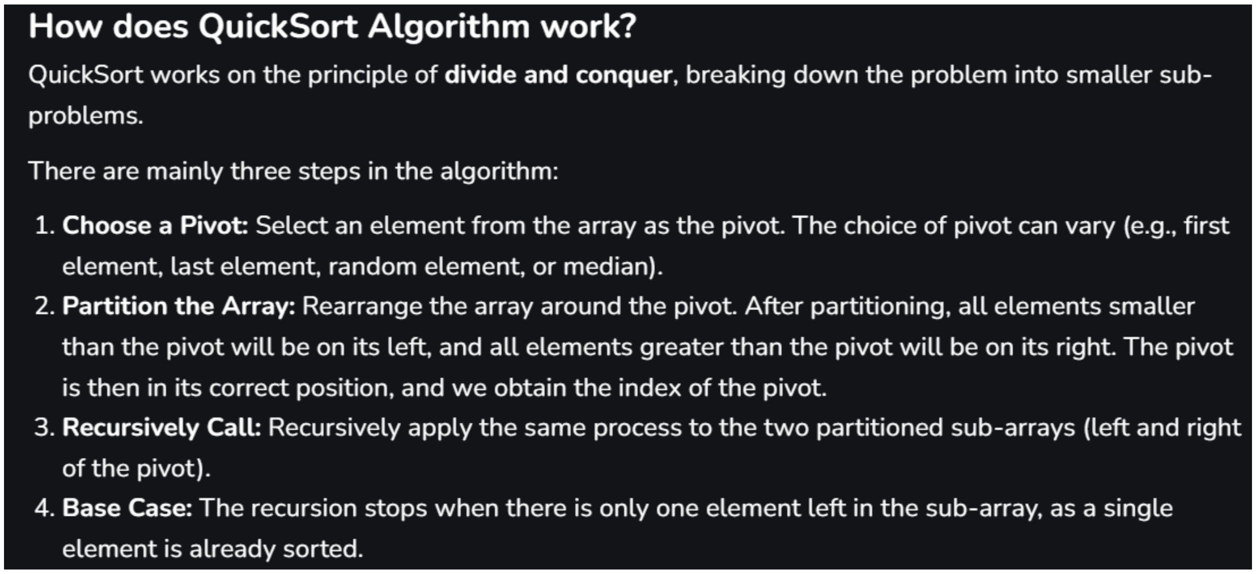
```
quickSort(arr, beg, end)
  if (beg < end)
    pivotIndex = partition(arr,beg, end)
    quickSort(arr, beg, pivotIndex)
    quickSort(arr, pivotIndex + 1, end)

partition(arr, beg, end)
  set end as pivotIndex
  pIndex = beg - 1
  for i = beg to end-1
  if arr[i] < pivot
    swap arr[i] and arr[pIndex]
    pIndex++
  swap pivot and arr[pIndex+1]
return pIndex + 1
```

Quick Sort Time Complexity

- Partition of elements take n time
- And in quicksort problem is divide by the factor 2
- Best Time Complexity : O(nlogn)
- Average Time Complexity : O(nlogn)
- Worst Time Complexity : O(n^2)
- Worst Case will happen when array is sorted

Quick Sort Space Complexity

- O(n) : basic approach
- O(logn) : modified approach



How does QuickSort Algorithm work?

QuickSort works on the principle of **divide and conquer**, breaking down the problem into smaller sub-problems.

There are mainly three steps in the algorithm:

1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. **Partition the Array:** Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
4. **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.