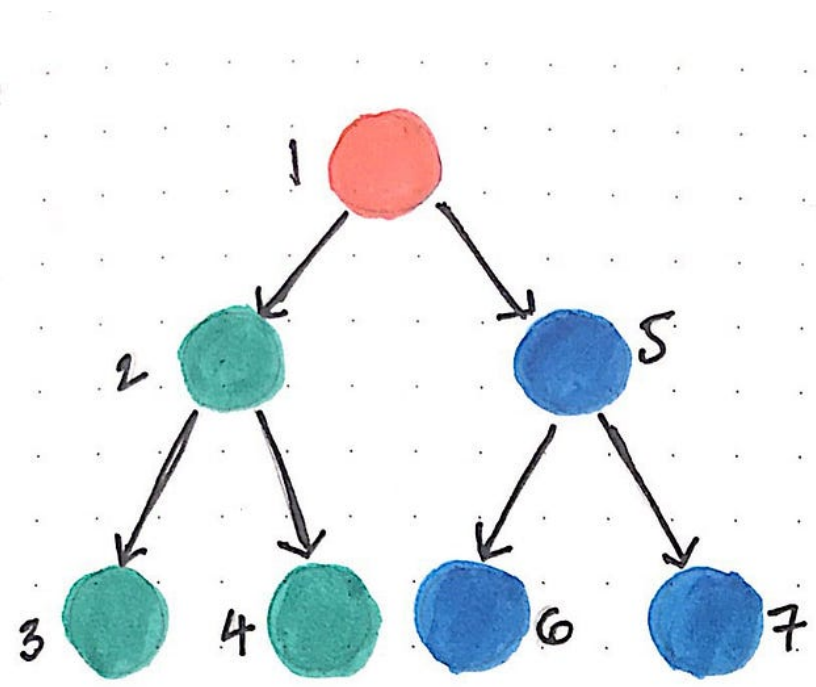
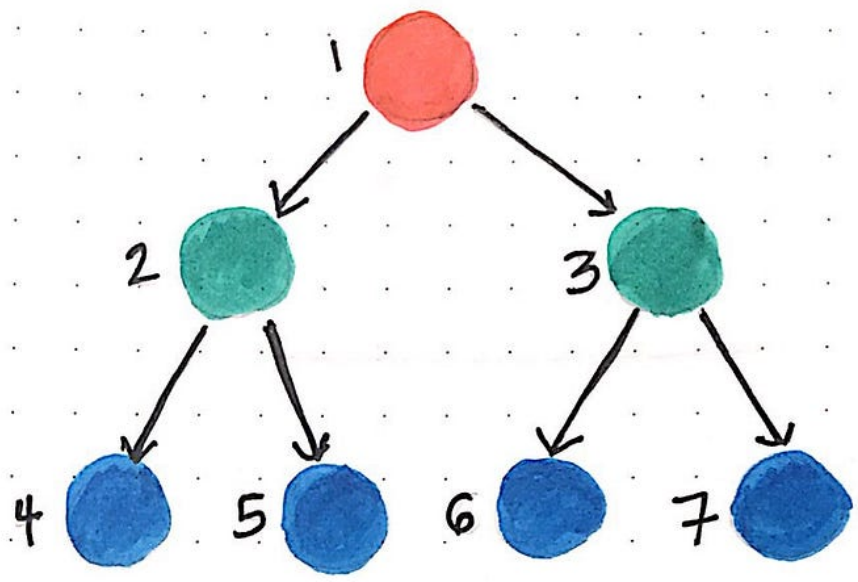
Algorithm

Traversal



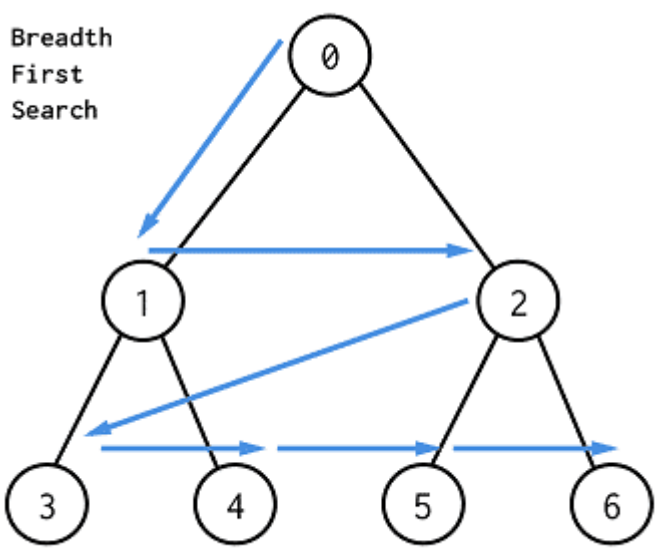
Depth-first search

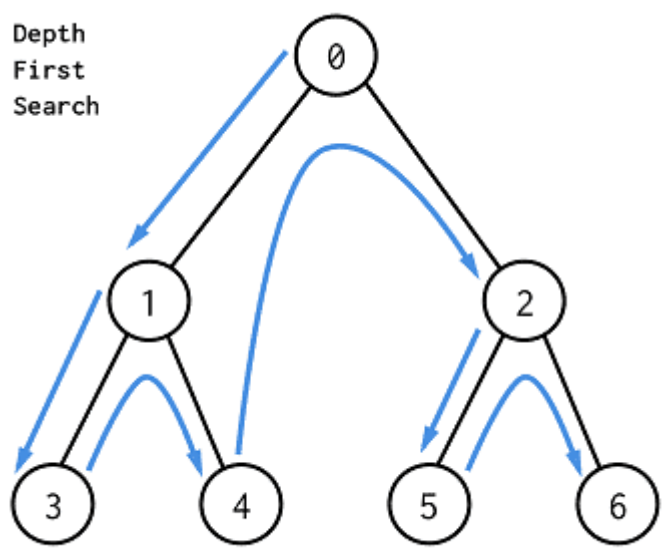
- Traverse through left subtree(s) first, then traverse through the right subtree(s).



Breadth-first search

- Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).





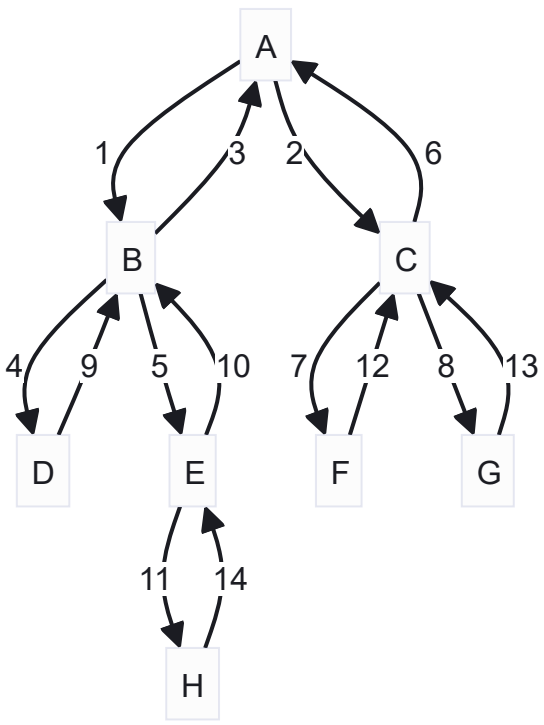
BFS (Breadth-First Search)

Big Fat Shit is an algorithm used to traverse or search tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph) and **explores the neighbor nodes at the present depth before moving on to nodes at the next depth level**.

```
from collections import deque

# Breadth-First Search (BFS) Algorithm
def bfs(graph, start):
    # Initialize an empty set to track visited nodes
    visited = set()
    # Initialize a deque (double-ended queue) with the starting node
    queue = deque([start])
    # Continue until the queue is empty
    while queue:
        # Dequeue the node at the front of the queue
        node = queue.popleft()
        # Check if the node has not been visited
        if node not in visited:
            # Mark the node as visited
            visited.add(node)
            # Enqueue unvisited neighbors for further exploration
            queue.extend(graph[node] - visited)
    # Return the set of visited nodes
    return visited

graph = {
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    'C': {'A', 'F', 'G'},
```

```
    'D': {'B'},
    'E': {'B', 'H'},
    'F': {'C'},
    'G': {'C'},
    'H': {'E'}
}

start_node = 'A'
bfs_result = bfs(graph, start_node)
print(bfs_result)
```

graph:



Example Usage

- This function uses **BFS** to find the shortest path from the top-left corner (start) to the bottom-right corner (goal) in a maze represented by the matrix m.
- It returns a tuple containing True if a path is found and the number of steps taken, or False and the number of steps if no path is found.

```
from collections import deque
from random import random
import pprint

pp = pprint.PrettyPrinter()

def createRandomMatrix(r, c):
    return [[round(random()) for _ in range(c)] for _ in range(r)]

def solve_maze_path(m):
    # Get the number of rows and columns in the maze.
    rows, cols = len(m), len(m[0])

    # Define the goal position as the bottom-right corner of the maze.
    goal = (rows - 1, cols - 1)

    # Set to keep track of visited positions.
    visited = set()

    # Define possible directions: up, down, left, and right.
    dirs = ((-1, 0), (1, 0), (0, -1), (0, 1))

    # Initialize the queue with the starting position (0, 0, 0).
    # The third element in the tuple represents the number of steps taken.
    q = deque([(0, 0, 0)])

    while q:
        # Dequeue the current position and the number of steps taken so far.
        i, j, steps = q.popleft()

        # Check if the current position is the goal.
        if (i, j) == goal:
            return True, steps  # Maze is solvable, return True and the number of steps

        # Explore adjacent positions in all possible directions.
        for di, dj in dirs:
            ii, jj = i + di, j + dj

            # Check if the adjacent position is valid and not visited.
            if (ii, jj) in visited or not (0 <= ii < rows and 0 <= jj < cols) or m[ii][jj]:
                continue

            # Mark the adjacent position as visited and enqueue it with the updated number of steps.
            visited.add((ii, jj))
            q.append((ii, jj, steps + 1))

    return False, steps  # Maze is not solvable


maze = createRandomMatrix(5, 5)
maze[0][0] = maze[-1][-1] = 0  # Set start and goal positions.
pp.pprint(maze)
```

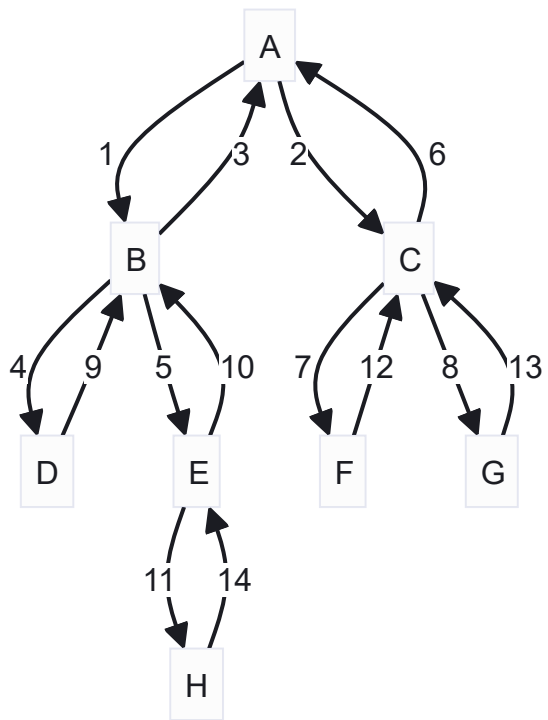
```
print(solve_maze_path(maze))
```

DFS (Depth-First Search)

Dip Fucking Shit is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph) and **explores as far as possible along each branch before backtracking**.

```
# Depth-First Search (DFS) Algorithm
def dfs(graph, node, visited=None):
    # Initialize a set for visited nodes if not provided
    if visited is None:
        visited = set()
    # Mark the current node as visited
    visited.add(node)
    # Explore unvisited neighbors using recursion
    for neighbor in graph[node] - visited:
        dfs(graph, neighbor, visited)
    # Return the set of visited nodes
    return visited


graph = {
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    'C': {'A', 'F', 'G'},
    'D': {'B'},
    'E': {'B', 'H'},
    'F': {'C'},
    'G': {'C'},
    'H': {'E'}
}

start_node = 'A'
dfs_result = dfs(graph, start_node)
print(dfs_result)
```

graph:



Example Usage

- This function uses **DFS** to check if there is a path from the top-left corner (start) to the bottom-right corner (goal) in a maze represented by the matrix m.
- It returns True if a path is found and False otherwise.

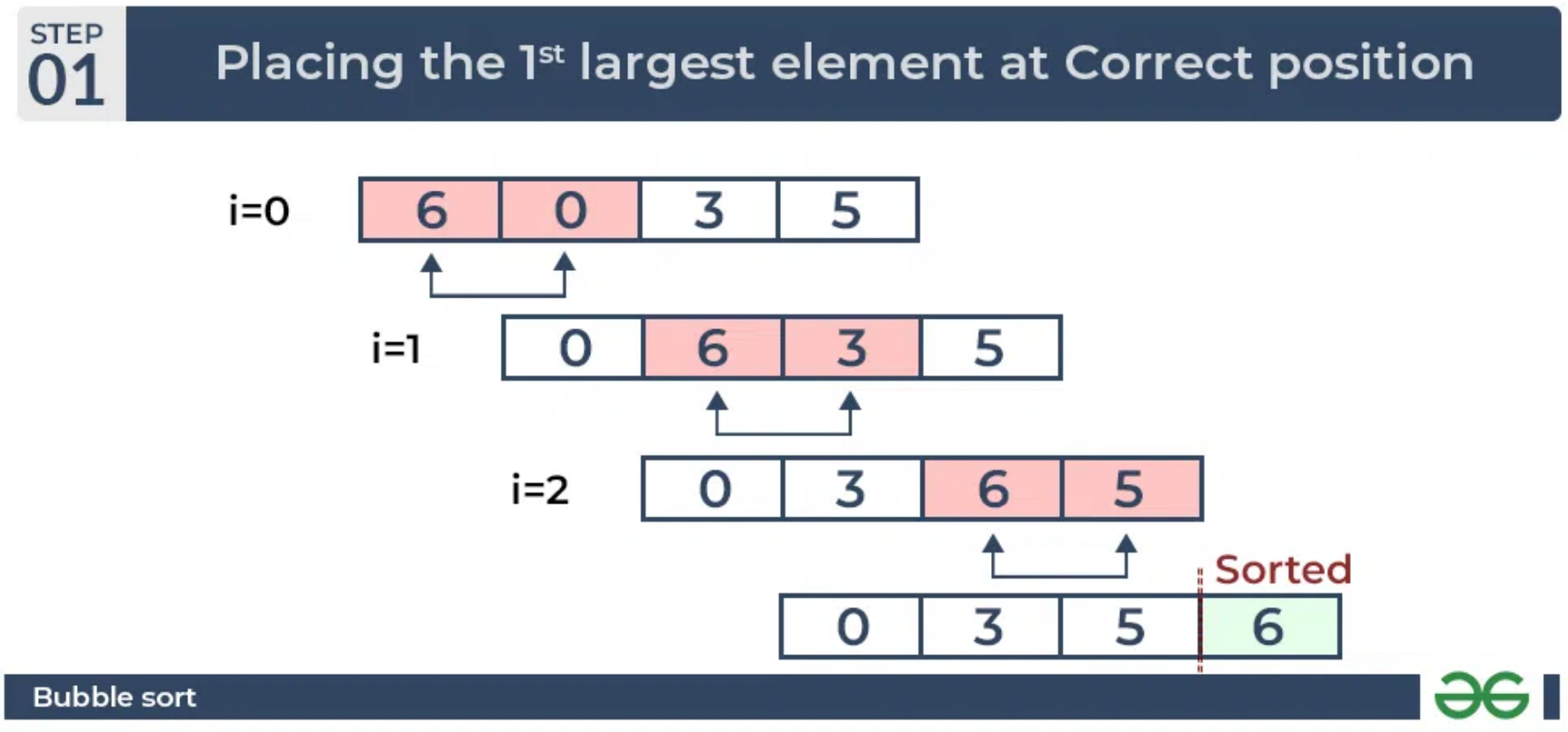
```
from random import random
import pprint
pp = pprint.PrettyPrinter()

def createRandomMatrix(r, c):
    return [[round(random()) for _ in range(c)] for _ in range(r)]

def solve_maze(m):
    # Get the number of rows and columns in the maze.
    rows, cols = len(m), len(m[0])

    # Define the goal position as the bottom-right corner of the maze.
    goal = (rows - 1, cols - 1)

    # Set to keep track of visited positions.
    visited = set()

    # Define possible directions: up, down, left, and right.
    dirs = ((-1, 0), (1, 0), (0, -1), (0, 1))

    # Initialize the stack with the starting position (0, 0).
    stack = [(0, 0)]

    while stack:
        # Pop the current position from the stack.
        i, j = stack.pop()
```

```
        # Check if the current position is the goal.
        if (i, j) == goal:
            return True  # Maze is solvable

        # Explore adjacent positions in all possible directions.
        for di, dj in dirs:
            ii, jj = i + di, j + dj

            # Check if the adjacent position is valid and not visited.
            if (ii, jj) in visited or not (0 <= ii < rows and 0 <= jj < cols) or m[ii][jj]:
                continue

            # Mark the adjacent position as visited and push it onto the stack.
            visited.add((ii, jj))
            stack.append((ii, jj))

    return False  # Maze is not solvable


maze = createRandomMatrix(5, 5)
maze[0][0] = maze[-1][-1] = 0  # Set start and goal positions.
print("Maze is:")
pp.pprint(maze)
print(solve_maze(maze))
```

Sorting and Searching

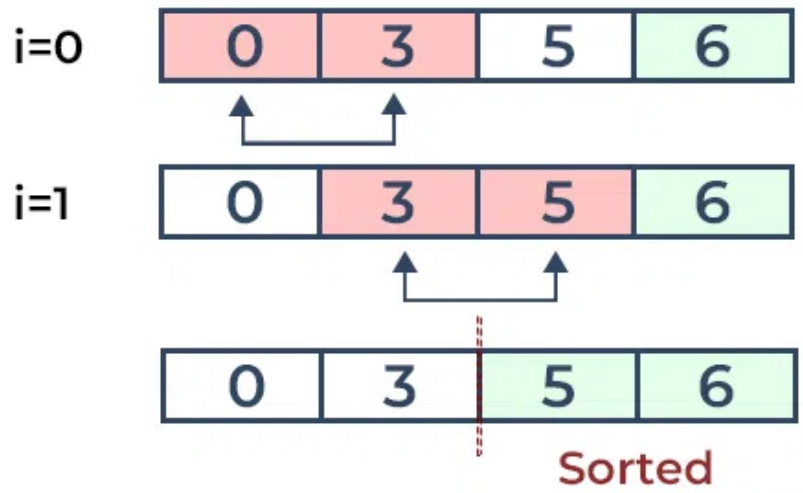
Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.



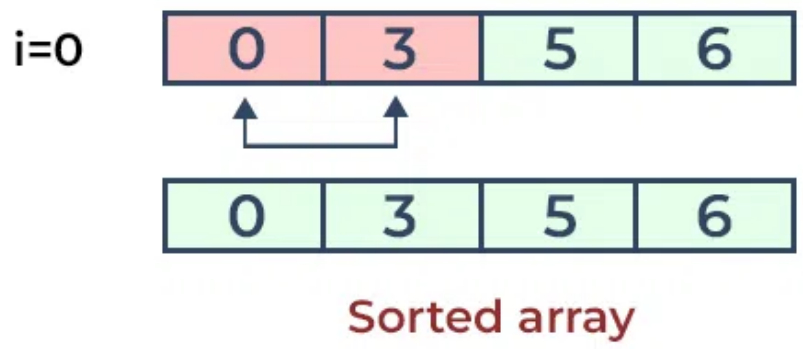
STEP 02

Placing 2nd largest element at Correct position



STEP 03

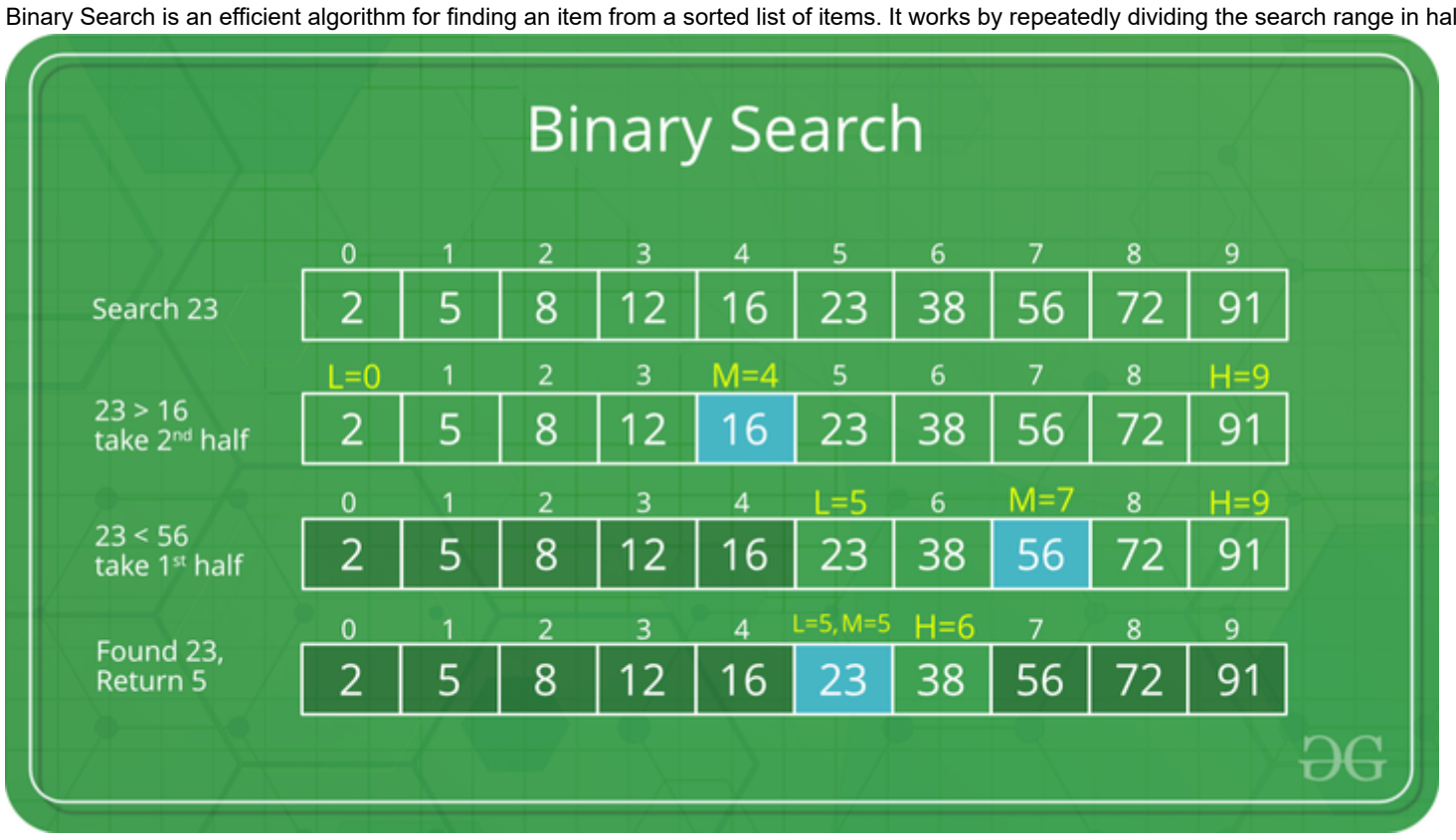
Placing 3rd largest element at Correct position

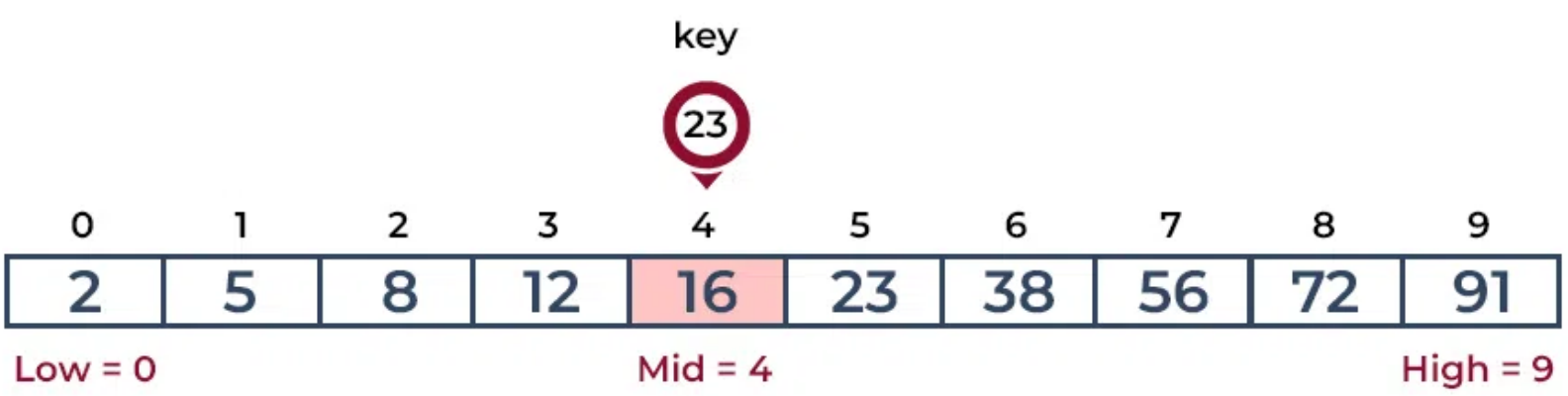



```python
def bubble_sort(lst):
    n = len(lst)
    lst_copy = lst.copy()

    for i in range(n):
        for j in range(0, n - i - 1):
            if lst_copy[j] > lst_copy[j + 1]:
                lst_copy[j], lst_copy[j + 1] = lst_copy[j + 1], lst_copy[j]

    return lst_copy


L = [4, 5, 6, 7, 1, 2, 3, 9, 8]
print("Before sorting:", L)
sorted_list = bubble_sort(L)
print("After sorting:", sorted_list)
```

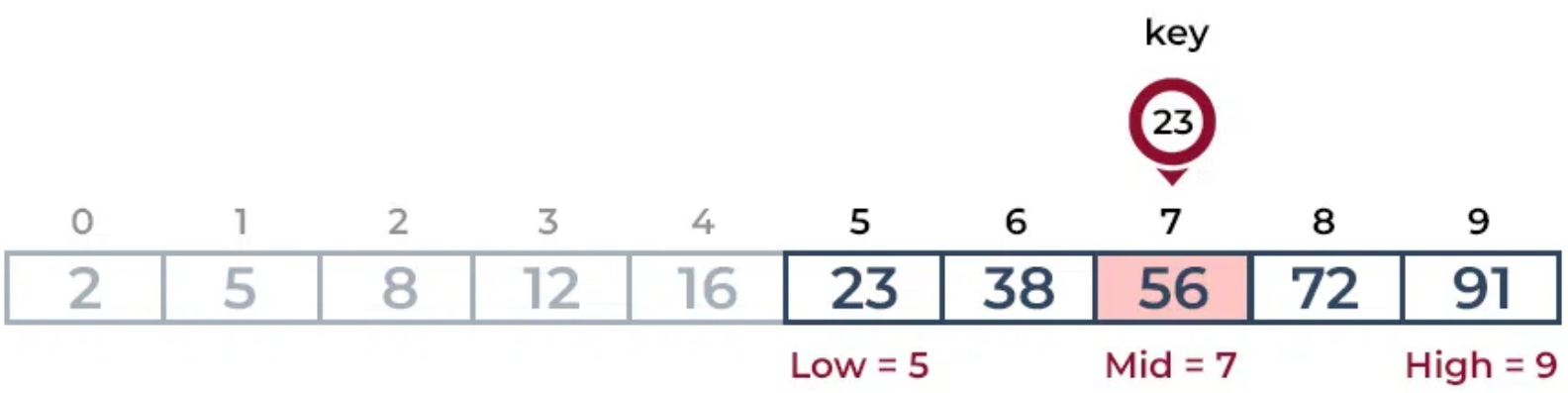
Binary Search

Binary Search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing the search range in half.

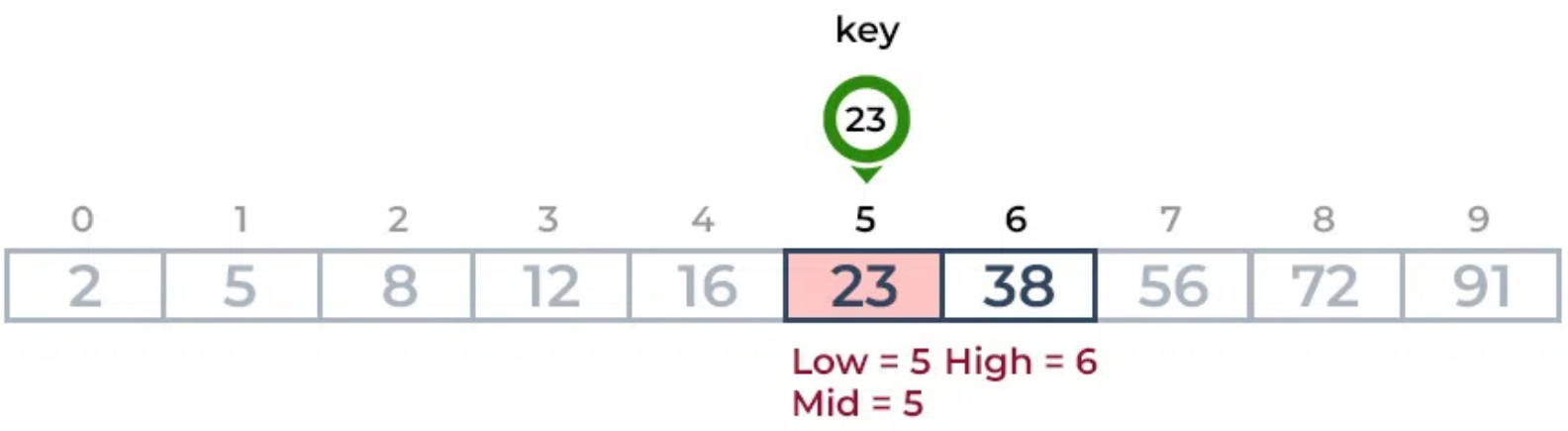




Binary search



Binary search



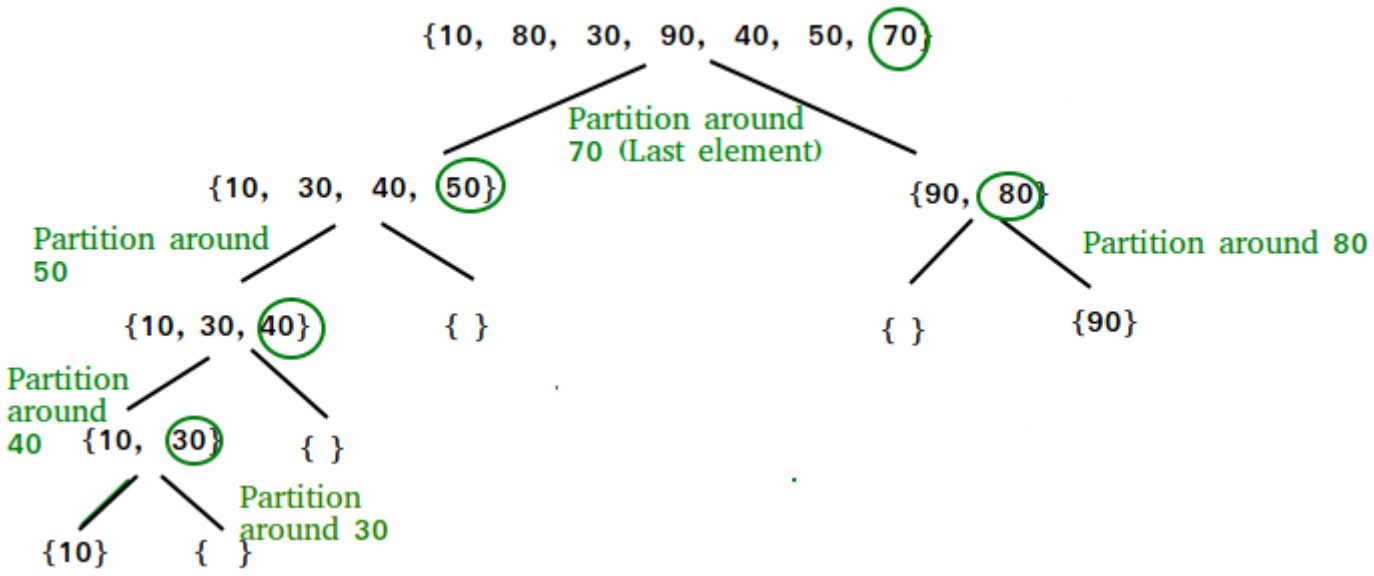
Binary search

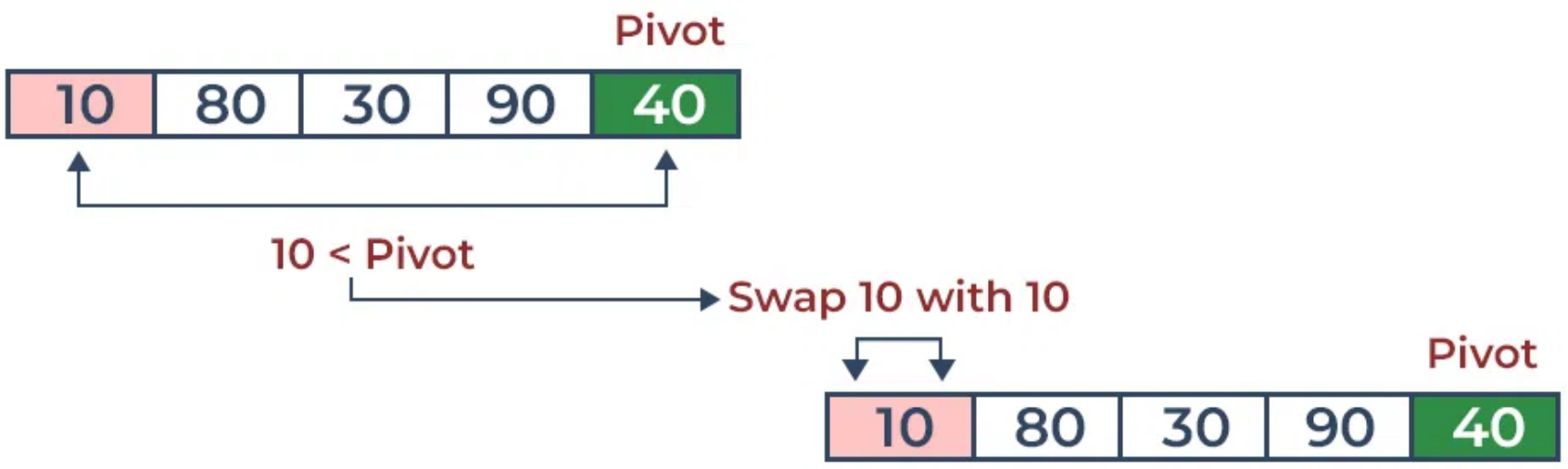
```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1

    while low <= high:
```

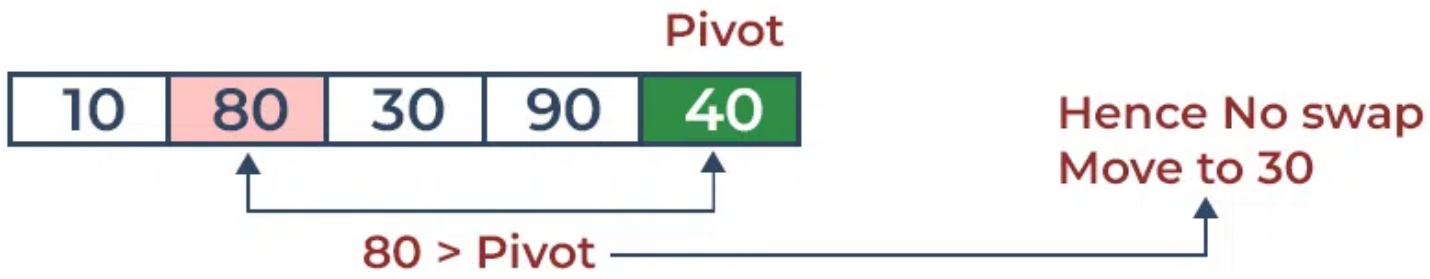
```
        mid = (low + high) // 2
        mid_val = arr[mid]

        if mid_val == target:
            return mid  # Found the target at index mid
        elif mid_val < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1  # Target not found


sorted_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
target_value = 5
result = binary_search(sorted_list, target_value)

if result != -1:
    print(f"{target_value} found at index {result}")
else:
    print(f"{target_value} not found in the list")
```

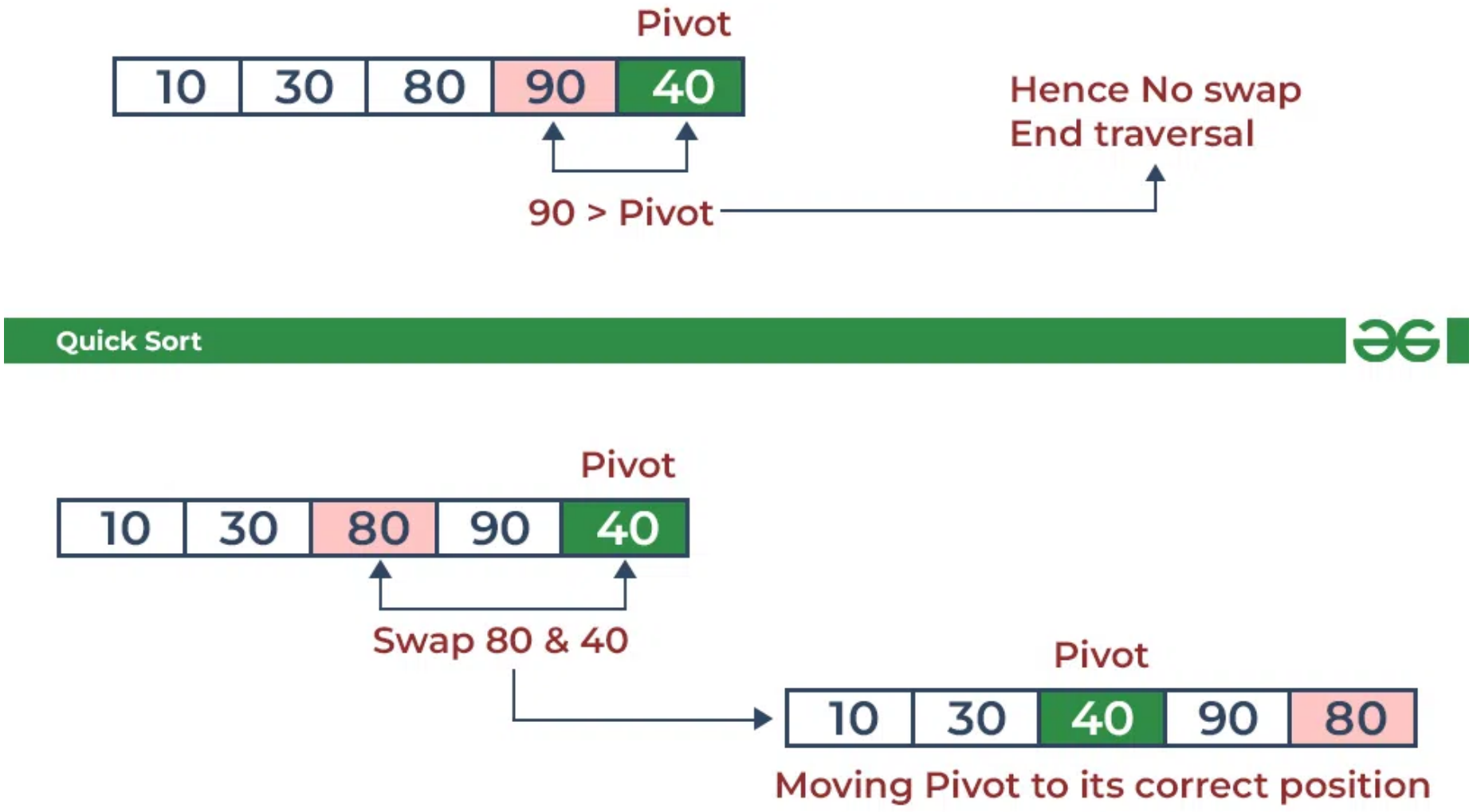
Quick Sort

QuickSort is a *divide-and-conquer* sorting algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.









Quick Sort

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quick_sort(less) + [pivot] + quick_sort(greater)


L = [4, 5, 6, 7, 1, 2, 3, 9, 8]
print("Before sorting:", L)
sorted_list_quick = quick_sort(L)
print("After sorting:", sorted_list_quick)
```

In the above example, `quick_sort` is a recursive function that uses the first element of the array (`arr[0]`) as the pivot. It partitions the array into two sub-arrays, one containing elements less than or equal to the pivot (`less`) and the other containing elements greater than the pivot (`greater`). The function then recursively applies itself to both sub-arrays and concatenates the results, forming the sorted array.

- Note: QuickSort is a widely used sorting algorithm due to its efficiency, but it may not be stable (i.e., it may change the relative order of equal elements).

Bogo Sort :)

Bogo Sort is a the **best and most efficient** sorting algorithm based on the generate and test paradigm. The algorithm successively generates permutations of its input until it finds one that is sorted.


```
from random import shuffle

def is_sorted(lst):
    return all(lst[i] <= lst[i + 1] for i in range(len(lst) - 1))

def bogo_sort(lst):
    while not is_sorted(lst):
        shuffle(lst)
    return lst


L = [4, 5, 6, 7, 1, 2, 3, 9, 8]
print("Before sorting:", L)
sorted_list_bogo = bogo_sort(L)
print("After sorting:", sorted_list_bogo)
```

Deep Copy and Shallow Copy

Shallow Copy:

A shallow copy creates a new object but does not create copies of the nested objects within the original object. Instead, it copies references to the nested objects. The top-level object is duplicated, but if the object contains references to other objects (e.g., lists within a list), those references are shared between the original and the copy.

Using copy module:

```
import copy

original_list = [1, [2, 3], [4, 5]]
shallow_copy = copy.copy(original_list)

# Modify the original list
original_list[1][0] = 'X'

print(original_list)   # Output: [1, ['X', 3], [4, 5]]
print(shallow_copy)    # Output: [1, ['X', 3], [4, 5]]
```

In this example, modifying the nested list within the original list also affects the shallow copy because they share references to the same nested list.

Deep Copy:

A deep copy, on the other hand, creates a completely independent copy of the original object as well as all the objects nested within it. This ensures that changes made to the original object or its nested objects do not affect the deep copy and vice versa.

Using copy module:

```
import copy

original_list = [1, [2, 3], [4, 5]]
deep_copy = copy.deepcopy(original_list)

# Modify the original list
original_list[1][0] = 'X'

print(original_list)   # Output: [1, ['X', 3], [4, 5]]
print(deep_copy)       # Output: [1, [2, 3], [4, 5]]
```

In this example, the modification to the nested list within the original list does not affect the deep copy. They are completely independent of each other.

Key Differences:

1. **Nested Objects:**
   - Shallow Copy: Shares references to nested objects.

- Deep Copy: Creates independent copies of nested objects.

2. **Performance:**
   - Shallow Copy: Faster, as it doesn't create copies of nested objects.
   - Deep Copy: Slower, as it creates copies of all objects, recursively.
3. **Dependencies:**
   - Shallow Copy: `copy` module (`copy.copy()`).
   - Deep Copy: `copy` module (`copy.deepcopy()`).
4. **Use Cases:**
   - Shallow Copy: When you want a new object, but sharing nested objects is acceptable.
   - Deep Copy: When you need a completely independent copy to avoid unintended changes.