**Quick Sort: Explanation, Algorithm, and Complexity**

**Quick sort** is a highly efficient and widely used sorting algorithm. Unlike merge sort, quick sort doesn't require extra memory, making it memory-efficient. It uses the **divide-and-conquer** approach, where an array is partitioned into smaller subarrays around a pivot element. The algorithm then recursively sorts the subarrays. On average, its time complexity is **O(n log n)**, though it can degrade to **O(n²)** in the worst case.

**1. What is Quick Sort?**

Quick sort divides the array into three parts:

* All elements smaller than a **pivot** element.
* The pivot element itself.
* All elements larger than the pivot.

The key operation is the **partitioning** step, where the array is rearranged so that the pivot is in its final sorted position. The elements smaller than the pivot come before it, and those larger come after it. This partitioning is done efficiently in **O(n)** time, where **n** is the number of elements.

Once partitioned, the pivot is correctly placed in the array, and the process is repeated recursively for the subarrays formed on either side of the pivot.

**2. Quick Sort Algorithm:**

**Steps in the Quick Sort Algorithm:**

1. **Base Case**: If the partition is small (size ≤ threshold), apply insertion sort.
2. **Partitioning**:
   * Select a **pivot** from the partition.
   * Rearrange the array such that all elements smaller than the pivot are to its left, and all elements larger are to its right.
   * The pivot is now in its correct position.
3. **Recursive Case**:
   * Recursively apply quick sort to the left and right parts of the array (excluding the pivot).

**Key Concepts:**

* **Pivot Selection**: The pivot can be chosen in various ways (e.g., randomly or using the first/last element). Randomly selecting a pivot helps avoid worst-case scenarios.
* **Partitioning**: The partitioning algorithm rearranges the array such that smaller elements come before the pivot, and larger elements come after.

**3. Quick Sort in Detail:**

**Partitioning Algorithm:**

The partitioning step divides the array into three sections (smaller than the pivot, the pivot itself, and larger than the pivot).

**Steps**:

1. **Choose a pivot**: Randomly select a pivot element from the partition.
2. **Move the pivot** to the end of the partition to avoid interference during sorting.
3. **Rearrange the array**: Start scanning the array from the left and swap elements as needed to ensure that all elements smaller than the pivot come before it.
4. **Place the pivot** in its correct position.
5. **Repeat** for the two subarrays (elements smaller and elements larger than the pivot).

**Quick Sort Algorithm Outline:**

1. **Base Case**: If the array or subarray is small (typically fewer than 16-32 elements), use insertion sort.
2. **Partition** the array around a pivot.
3. **Recursively** sort the left and right subarrays formed by partitioning.

**4. Python Implementation:**

**Quick Sort Function:**

The main function initializes the recursive quick sort process.

python

Copy code

def quick\_sort(mylist):

recursive\_quick\_sort(mylist, 0, len(mylist) - 1)

**Recursive Quick Sort Function:**

This function recursively sorts the array using the quick sort algorithm. The base case is when the partition size is small enough for insertion sort.

python

Copy code

def recursive\_quick\_sort(mylist, left, right, THRESHOLD=32):

if right - left <= THRESHOLD:

insertion\_sort(mylist, left, right) # Use insertion sort for small partitions

else:

pivot\_position = partition(mylist, left, right) # Partition the array

recursive\_quick\_sort(mylist, left, pivot\_position - 1) # Sort left part

recursive\_quick\_sort(mylist, pivot\_position + 1, right) # Sort right part

**Insertion Sort Function:**

Insertion sort is used to handle small subarrays. It's efficient for small input sizes and serves as the base case for quick sort.

python

Copy code

def insertion\_sort(mylist, left, right):

for i in range(left + 1, right + 1):

curr = mylist[i]

j = i

while j > left and mylist[j - 1] > curr: # Shift values to the right

mylist[j] = mylist[j - 1]

j -= 1

mylist[j] = curr # Place the current element in the correct position

**Partition Function:**

This is the core of the quick sort algorithm, where the array is partitioned based on the pivot element.

python

Copy code

import random

def partition(mylist, left, right):

# Choose a random pivot

pivot\_location = random.randint(left, right)

pivot = mylist[pivot\_location]

# Move the pivot to the end

mylist[pivot\_location], mylist[right] = mylist[right], mylist[pivot\_location]

end\_of\_smaller = left - 1

# Partition the array

for j in range(left, right):

if mylist[j] <= pivot:

end\_of\_smaller += 1

mylist[end\_of\_smaller], mylist[j] = mylist[j], mylist[end\_of\_smaller]

# Move the pivot to its correct position

mylist[end\_of\_smaller + 1], mylist[right] = mylist[right], mylist[end\_of\_smaller + 1]

return end\_of\_smaller + 1 # Return the pivot's final position

**5. Time Complexity of Quick Sort:**

* **Average Case**: The average-case time complexity is **O(n log n)** because the partitioning divides the array into two roughly equal halves, and each partitioning step takes **O(n)** time.
* **Worst Case**: The worst-case time complexity is **O(n²)**. This occurs when the pivot divides the array in a highly unbalanced way (e.g., when the smallest or largest element is always chosen as the pivot).

**6. Space Complexity of Quick Sort:**

Quick sort operates **in place**, so it uses **O(log n)** space for the recursion stack (in the average case). In the worst case, it uses **O(n)** space due to deep recursion when the array is split unevenly.

**7. Example Walkthrough:**

Let's walk through an example of quick sort on a small list.

**Input List:**

[8, 3, 1, 7, 0, 10, 2]

1. **Choose a pivot**: Let's say we pick 7 as the pivot.
2. **Partition** the list: All values smaller than 7 go to the left, and all values larger go to the right.
   * [3, 1, 0, 2] (smaller than 7), [7] (pivot), [8, 10] (larger than 7)
3. **Recursively apply** quick sort to the left part [3, 1, 0, 2].
4. **Choose a pivot** for the left part: Suppose we pick 1 as the pivot.
5. **Partition** the left part: [0] (smaller than 1), [1], [3, 2].
6. **Recursively apply** quick sort to [3, 2] and sort it.
7. Repeat the process for the right part [8, 10] until the entire array is sorted.

After all recursive steps, the sorted array becomes:  
[0, 1, 2, 3, 7, 8, 10]

**8. How to Explain Quick Sort in a Quiz:**

If asked to explain quick sort's **time complexity** and **space complexity** in a quiz, here’s how you can approach it:

**Question**: Explain the time and space complexity of quick sort.

**Answer**:

Quick sort is a divide-and-conquer algorithm that sorts an array by partitioning it into smaller subarrays around a pivot. The partitioning step rearranges the elements so that all elements smaller than the pivot come before it and all larger elements come after it. The pivot is then in its final sorted position.

1. **Time Complexity**:
   * **Best and Average Case**: In the best and average cases, the pivot splits the array into two roughly equal parts, and the time complexity is **O(n log n)**. This is because partitioning takes **O(n)** time, and there are **log n** recursive levels.
   * **Worst Case**: In the worst case, when the array is split unevenly (e.g., when the smallest or largest element is always chosen as the pivot), the time complexity becomes **O(n²)**.
2. **Space Complexity**:
   * **Average Case**: Quick sort is an **in-place** sorting algorithm, meaning it requires **O(log n)** space for the recursion stack.
   * **Worst Case**: In the worst case, the recursion depth can reach **O(n)**, leading to **O(n)** space complexity.

**Conclusion**: Quick sort is an efficient and commonly used sorting algorithm with an average time complexity