CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Quick Sort

Department of Computer Science
University of Maryland, College Park

Introduction to Quick Sort

- Quick Sort is a **divide-and-conquer** sorting algorithm.
- It was developed by **Tony Hoare** in 1959.
- It works by:
  - **Choosing a pivot**
  - **Partitioning** the array into two sub-arrays:
    - Elements **less than** the pivot
    - Elements **greater than** the pivot
  - Recursively sorting the sub-arrays

Why Use Quick Sort?

- **Fast in practice**: Average time complexity is **O(n log n)**
- **In-place sorting**: Requires only **O(log n)** space on average for recursion
- **Commonly used** in real-world applications (e.g., Java's built-in sort for primitives)

Algorithm	Time (Average)	Time (Worst)	Space
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(1)$

The High-Level Algorithm

```
quickSort(arr, low, high):
    if low < high:
        pivotIndex = partition(arr, low, high)
        quickSort(arr, low, pivotIndex - 1)
        quickSort(arr, pivotIndex + 1, high)
```

- Sorts elements between low and high.
- Partition puts the pivot in its correct place.
- Recursively sort left and right sub-arrays.
- **Partitioning (Hard Split)**: The work-intensive part of Quick Sort is the partitioning step, where elements are rearranged around a pivot.
- **Recursion (Easy Merge)**: After partitioning, the sub-arrays are recursively sorted. No merging is required; the array elements are "sorted in place" as the recursion works.

Partitioning Explained

- Choose a pivot (e.g., last element).
- Use two pointers:
  - i tracks position for smaller elements.
  - j iterates from low to high - 1.
  - Swap arr[i] and arr[j] when arr[j] < pivot.
  - Finally, place the pivot at i + 1.
- **After Partitioning:** Once the partitioning step is complete, the **pivot** will be in its **correct and final position**. This means that the pivot element will never move again.
- **Smaller Elements on the Left:** All elements smaller than the pivot will be placed in the **left half** of the array. They are **not necessarily in sorted order**, but they are all smaller than the pivot.
- **Larger Elements on the Right:** Similarly, all elements larger than the pivot will be placed in the **right half** of the array. These elements are also **not necessarily in sorted order**, but they are all larger than the pivot.
- **What Happens Next:** After the partitioning, the array is divided into two sub-arrays (left and right of the pivot). Each sub-array is then treated as a new input to the **same algorithm**. This is the magic of **recursion**—you are solving the same problem, just with **smaller, simpler sub-problems**!
- **Recursion's Power:** The same Quick Sort algorithm is applied to these sub-arrays, which are now smaller and simpler. The process repeats until the sub-arrays become so small (typically a single element) that they are trivially sorted.

Step-by-Step Example

Input: arr = [10, 3, 7, 2, 6] //**Pivot:** Select the last element as the pivot. **Pivot = 6**.

- Start with i = -1 (index of last element smaller than the pivot).

1. **i = -1**, **j = 0** (element = 10):
   1. 10 is greater than 6, so do nothing. i stays at -1.
2. **i = -1**, **j = 1** (element = 3):
   1. 3 is smaller than 6, so **increment i to 0**, and **swap arr[0] and arr[1]**.
      1. New array: [3, 10, 7, 2, 6].
3. **i = 0**, **j = 2** (element = 7):
   1. 7 is greater than 6, so do nothing. i stays at 0.
4. **i = 0**, **j = 3** (element = 2):
   1. 2 is smaller than 6, so **increment i to 1**, and **swap arr[1] and arr[3]**.
      1. New array: [3, 2, 7, 10, 6].
5. **i = 1**, **j = 4** (pivot = 6):
   1. Now place the pivot (6) in its correct position by swapping arr[i + 1] and the pivot.
      1. **Swap arr[2] and arr[4]**.
      2. New array: [3, 2, 6, 10, 7].

- **Key Points:**
  - **Pivot (6)** is now in its correct position.
  - **Left sub-array** contains elements smaller than the pivot, and **right sub-array** contains elements larger than the pivot.
  - The process repeats recursively, working with smaller and simpler sub-arrays.

Time Complexity

Case	Time Complexity
Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	$O(n^2)$

- **Worst case happens when:**
  - Pivot is always the smallest or largest element
  - Example: Already sorted input with naive pivot choice.
- **Ways to improve:**
  - **Randomized Pivot:** Pick a random element as pivot to avoid worst-case
  - **Median-of-Three Pivot:** Use the median of first, middle, and last
  - **Switch to Insertion Sort** for small sub-arrays (e.g., size < 10)
- **Tradeoffs:**
  - Slightly more complex code
  - Better average performance

Proving Quick Sort is O(n log n) for 50-50 Split

- Let's assume that at each recursion step, the array is split into two roughly equal halves (50-50 split).

1. **Initial Array (n elements):**
   1. Total work at the first level of recursion: **n** (all elements need to be compared to the pivot).
2. **Second Level of Recursion:**
   1. After partitioning, the two sub-arrays each contain **n/2** elements.
   2. Total work at this level: **n/2** for each sub-array, **n/2 + n/2 = n** total.
3. **Third Level of Recursion:**
   1. The array is split into four sub-arrays, each containing **n/4** elements.
   2. Total work at this level: **n/4 + n/4 + n/4 + n/4 = n**.
4. **Subsequent Levels:**
   1. This pattern continues with the array being split into more and more sub-arrays, each time halving the size.
   2. Each level of recursion requires **n** comparisons in total.

Proving Quick Sort is O(n log n) for 50-50 Split

- At each level of recursion, we split the array into two, and the total work done at each level is **O(n)**. The number of levels of recursion is **$\log_2 n$**, as the array size keeps halving.
- So, the total work done is the sum of all levels:
- **Work per level:** O(n)
- **Number of levels:** O(log n)
- Therefore, the **overall time complexity** is:

  `O(n) * O(log n) = O(n log n)`
- **`Note #1:`**
- **Perfect 50-50 Split:** Optimal scenario for **O(n log n)**.
- **Other Splits:** Quick Sort still maintains **O(n log n)** for non-perfect but relatively balanced splits (e.g., 60-40, 70-30).
- **`Note #2:`**
- QuickSort has to be **at least n log n slow** in the best case, because that's the **least amount of work <u>any comparison-based sorting algorithm</u>** can possibly do. Formally, this is written as saying that QuickSort is in **Ω(n log n)**.

Space Complexity of Quick Sort

- **In-place sorting:**
  QuickSort does **not use extra arrays** — all swaps happen inside the original array.
- **What about recursion?**
  - QuickSort is a **divide-and-conquer** algorithm.
  - It recursively processes subarrays.
  - **Each recursive call** adds a frame to the **call stack**.
- **Best and Average Case:**
  - If each pivot splits the array **roughly in half**:
    - The depth of the recursion is $\log_2 n$
    - So the **call stack only grows to O(log n)**
- **Worst Case**
- If the pivot always picks the smallest or largest value (bad splits):
    - Recursion depth becomes **n**
    - So space usage becomes **O(n)**
- **In summary:**
  - In the **average case**, QuickSort only needs **O(log n)** space — that's just the memory for recursion, not for extra arrays!

See: QuickSort Example. If time allows, make it generic so that it works with other types, such as Strings.