Introduction to Quicksort

Quicksort is a widely used and highly efficient **comparison-based sorting algorithm** that follows the **divide-and-conquer** paradigm. It was invented by British computer scientist **C.A.R. (Tony) Hoare** in 1959, while he was working on a machine translation project in the Soviet Union. It has since become one of the most commonly used sorting techniques due to its **simplicity, speed, and in-place sorting capability**.
At its core, Quicksort works by selecting a special element called the **pivot**, and then **reorganizing (partitioning)** the elements in the array so that:

- All the elements **smaller than the pivot** come **before it**,
- All the elements **greater than the pivot** come **after it**.

After partitioning, it recursively applies the same logic to the subarrays formed on either side of the pivot. This continues until the base case is reached (i.e., the array has zero or one element, which is already sorted).

Key Characteristics of Quicksort

In-Place Sorting

Quicksort does not require significant extra memory. It sorts the elements **within the same array** by swapping them in place. This makes it more **space-efficient** than algorithms like Merge Sort, which require additional arrays for merging.

Unstable Sorting

Quicksort is **not a stable sort**. This means that it does **not preserve the relative order of equal elements**. For example, if two records have the same key, their order may be changed after sorting.

Time Complexity

- **Best Case:** $O(n \log n)$
  Occurs when the pivot divides the array into two nearly equal halves.
- **Average Case:** $O(n \log n)$
  The most common scenario due to randomized or median-of-three pivot selection.
- **Worst Case:** $O(n^2)$
  Happens when the pivot ends up being the smallest or largest element every time (e.g., sorted or reverse-sorted arrays), leading to highly unbalanced partitions.

However, with **optimized pivot selection**, such as choosing a **random pivot** or the **median of the first, middle, and last element**, the worst-case scenario can usually be avoided in practice.

Space Complexity

- **Auxiliary Space:** $O(\log n)$ due to the recursive call stack.
- It does **not** require a separate array like Merge Sort, making it **ideal for memory-constrained systems**.

Why Use Quicksort?

- Quicksort is **fast in practice** for large datasets, especially when compared to algorithms like Bubble Sort or Insertion Sort.
- It makes **efficient use of memory**, as it does not require additional storage.
- It is **widely implemented** in standard libraries (e.g., `qsort()` in C, `Arrays.sort()` in Java for primitive types).

Quicksort Algorithm with Hoare's Partitioning

The Quicksort algorithm is a classic divide-and-conquer approach to sorting. It works in the following steps:

1. **Choose a Pivot:** Select a pivot element from the array. In Hoare's partitioning scheme, the **first element** is commonly chosen as the pivot.
2. **Partition the Array:** Rearrange the array such that:
   - All elements less than or equal to the pivot are placed on the left side.
   - All elements greater than or equal to the pivot are placed on the right side.

   Two indices, `i` and `j`, are used to scan the array from both ends toward the center, and out-of-place elements are swapped until the pointers cross.
3. **Recursive Sorting:** Recursively apply Quicksort on the left and right subarrays. Unlike Lomuto's scheme, the pivot is not guaranteed to be at its final sorted position after partitioning.

How Quicksort Works (Step-by-Step)

Let us consider the following example:
Input:

```
arr = [9, 3, 7, 5, 6, 4, 8, 2]
```

- **Step 1: Choose Pivot**
  Use the first element as pivot: `pivot = 9`
- **Step 2: Partition**
  Rearranging using Hoare's partitioning:
  - Scan from both ends.

  - Swap elements until the left and right pointers meet.

One possible result after partitioning:

```
arr = [2, 3, 7, 5, 6, 4, 8, 9]
```

- **Step 3: Recursively Sort Subarrays**
  Now apply Quicksort recursively on the left subarray: `[2, 3, 7, 5, 6, 4, 8]` and right subarray: `[9]`

  Continue the process until all subarrays are sorted.

Quicksort Pseudocode

```
function quicksort(arr, low, high)
    if low < high then
        pivotIndex = partition(arr, low, high)
        quicksort(arr, low, pivotIndex)
        quicksort(arr, pivotIndex + 1, high)

function partition(arr, low, high)
    pivot = arr[low]
    i = low
    j = high + 1

    while true
        repeat
            i = i + 1
        until i == high or arr[i] >= pivot

        repeat
            j = j - 1
        until j == low or arr[j] <= pivot

        if i >= j then
            return j

        swap arr[i] and arr[j]
```

Example: Quicksort Using Hoare's Partitioning

Let us sort the array:
`arr = [5, 3, 8, 4, 2]`

1. **Choose pivot = 5 (first element)**
   Apply Hoare's partitioning. One possible result is:

   `arr = [2, 3, 4, 5, 8]`

   Pivot index = 3
   Left subarray: `[2, 3, 4]`
   Right subarray: `[8]`

2. **Apply on [2, 3, 4], pivot = 2**
   After partitioning:

   `[2, 3, 4]`

   Pivot index = 0
   Left subarray: `[]`
   Right subarray: `[3, 4]`

3. **Apply on [3, 4], pivot = 3**
   Already sorted.

Final Sorted Array:
`[2, 3, 4, 5, 8]`

Important Note

In the below exercises, you are required to choose a different pivot instead of the first element. If you want to use a different element as the pivot (e.g., the last or middle element), you can simply **swap that element with the first element** before calling the partition function. This makes it compatible with **Hoare's partitioning**, which always assumes the pivot is the first element.

For example:
If you want to use the last element as pivot:

```
swap(arr[low], arr[high]); // Move last element to the front
// Now call partition as usual
```

Lab exercises

Exercise 1 ....

Sorting Temperature Readings: A weather station records hourly temperature readings (in C) throughout the day. These readings, stored in a vector of floating-point numbers, need to be sorted in ascending order before being stored in the system for analysis and reporting. Your task is to implement the Quicksort algorithm to sort these temperature readings. Use the last element as the pivot during partitioning.

Input Example:

```
{12.5, -3.2, 7.0, 0.0, 22.3, 15.4, -1.0}
```

Output:

```
{-3.2, -1.0, 0.0, 7.0, 12.5, 15.4, 22.3}
```

Requirements:

- Use recursion and in-place sorting.
- Use the last element as the pivot.
- Do not use any STL sorting functions.

Exercise 2 ....

Sorting Customer Names for a Report: You are building a feature for a retail company's reporting system. Each day, the system receives a list of customer names (as `vector<string>`) who made purchases. Before generating the daily report, the names must be sorted lexicographically (dictionary order) for easier lookup and cleaner formatting. You are required to sort this list using the Quicksort algorithm. The sort must be case-sensitive, which means `"Alice"` comes before `"bob"` due to ASCII order.

Input Example:

```
{"Zara", "ali", "Blake", "chris"}
```

Output:

```
{"Blake", "Zara", "ali", "chris"}
```

Requirements:

- Implement recursive QuickSort using the last element as pivot.
- Sorting must respect case-sensitive lexicographical order.
- Do not use built-in sorting functions.

Exercise 3 . . . .

Ranking Daily Sales Amounts: An e-commerce analytics team needs to generate a daily sales report. The report must list all transaction amounts in descending order, so the largest sales appear first. You are given a list of transaction amounts (as `vector<float>`) recorded throughout the day.

You are required to implement a Quicksort algorithm that sorts the list from highest to lowest. Instead of using the last element as the pivot, choose the middle element for partitioning to improve performance on varied datasets.

Input Example:

```
{99.99, 120.5, 35.75, 85.0, 150.25}
```

Output:

```
{150.25, 120.5, 99.99, 85.0, 35.75}
```

Requirements:

- Implement QuickSort using the **middle element** as the pivot.
- Modify the algorithm to sort in **descending order**.
- **Do not** use built-in sorting functions.
- Use only `vector<float>` and basic logic.

Exercise 4 . . . .

Sorting Product Names by Length: A retail company is preparing to generate a catalog of product names for its online store. They need to list the products in order of name length (shortest first). In case two products have the same length, they should be sorted alphabetically.

You are tasked with implementing a QuickSort algorithm that will sort the product names according to these two criteria:

- By length of the name (shortest first).
- If two names are the same length, sort them alphabetically.

Input Example:

```
{"Apple", "Banana", "Orange", "Peach", "Pineapple"}
```

Output:

```
{"Apple", "Peach", "Banana", "Orange","Pineapple"}
```

Requirements:

- Implement QuickSort using a **custom comparator** during partitioning.
- Ensure two-level sorting: **first by length**, then **alphabetically**.
- **Do not** use built-in sorting functions.
- Use only `vector<string>` for the input.