

**Name: Yogendra Pokhrel**

Student ID: 617585

**1. Show all steps of QuickSort in sorting the array [1, 6, 2, 4, 3, 5]. Use leftmost values as pivots at each step.**

**Step 1: First Partitioning**

- Array: [1, 6, 2, 4, 3, 5]
- Pivot: 1 (leftmost element)
- We compare all elements with 1 and place smaller elements to the left and larger ones to the right.
- Since all elements are greater than 1, the array remains the same.
- Partitioned Array: [1] | [6, 2, 4, 3, 5]
- Left subarray is sorted ([1]), apply QuickSort on [6, 2, 4, 3, 5].

**Step 2: Second Partitioning**

- Array: [6, 2, 4, 3, 5]
- Pivot: 6 (leftmost element)
- All elements are smaller than 6, so they stay in place.
- Partitioned Array: [1] | [2, 4, 3, 5] | [6]
- Right subarray [6] is sorted, apply QuickSort on [2, 4, 3, 5].

**Step 3: Third Partitioning**

- Array: [2, 4, 3, 5]
- Pivot: 2 (leftmost element)
- All elements are greater than 2, so no swaps are needed.
- Partitioned Array: [1] | [2] | [4, 3, 5] | [6]
- Left part [2] is sorted, apply QuickSort on [4, 3, 5].

**Step 4: Fourth Partitioning**

- Array: [4, 3, 5]
- Pivot: 4
- Elements smaller than 4: [3]
- Elements greater than 4: [5]
- Partitioned Array: [1] | [2] | [3] | [4] | [5] | [6]
- All subarrays are now sorted.

**Final Sorted Array: [1, 2, 3, 4, 5, 6]**

2. In our average case analysis of QuickSort, we defined a good self-call to be one in which the pivot  $x$  is chosen so that number of elements  $< x$  is less than  $3n/4$ , and also the number of elements  $> x$  is less than  $3n/4$ . We call an  $x$  with these properties a good pivot. When  $n$  is a power of 2, it is not hard to see that at least half of the elements in an  $n$ -element array could be used as a good pivot (exactly half if there are no duplicates). For this exercise, you will verify this property for the array  $A = [5, 1, 4, 3, 6, 2, 7, 1, 3]$  (here,  $n = 9$ ). Note: For this analysis, use the version of QuickSort in which partitioning produces 3 subsequences  $L, E, R$  of the input sequence  $S$ .
- a. Which  $x$  in  $A$  are good pivots? In other words, which values  $x$  in  $A$  satisfy:
- the number of elements  $< x$  is less than  $3n/4$ , and also
  - the number of elements  $> x$  is less than  $3n/4$
- b. Is it true that at least half the elements of  $A$  are good pivots?

### Step 1: Understanding the Condition for a Good Pivot

A number  $x$  is a good pivot if:

- The number of elements less than  $x$  is less than  $3n/4$
- The number of elements greater than  $x$  is less than  $3n/4$ .

Since  $n = 9$ , we compute:

$$3n/4 = (3 \times 9) / 4 = 6.75$$

This means a pivot  $x$  is good if:

- The number of elements less than  $x$  is at most 6.
- The number of elements greater than  $x$  is at most 6.

### Step 2: Counting Elements Less Than and Greater Than Each $x$ in $A$

We evaluate each unique number in  $A = [5, 1, 4, 3, 6, 2, 7, 1, 3]$ :

$x$	Count of elements $< x$	Count of elements $> x$	Good Pivot?
1	0	7	Too many $> x$

2	2	6	$\leq 6$ in both cases
3	4	4	$\leq 6$ in both cases
4	5	3	$\leq 6$ in both cases
5	6	2	$\leq 6$ in both cases
6	7	1	Too many $< x$
7	8	0	Too many $< x$

### Step 3: Counting Good Pivots

From the table, the good pivots are: 2, 3, 4, and 5.

Total unique elements in A: 7

Total good pivots: 4

Since 4 out of 7 elements (more than half) are good pivots, the claim that at least half of the elements are good pivots holds true for this array.

### Conclusion:

(a) Good pivots are: {2, 3, 4, 5}

(b) Yes, at least half the elements in A are good pivots.

### 3. Give an $o(n)$ (“little-oh”) algorithm for determining whether a sorted array A of distinct integers contains an element m for which $A[m] = m$ . You must also provide a proof that your algorithm runs in $o(n)$ time.

#### $O(n)$ (“Little-oh”) Algorithm for Finding an Index m Where $A[m] = m$

We are given a sorted array A of distinct integers, and we need to check whether there exists an index m such that:

$$A[m] = m$$

We want an algorithm that runs in  $o(n)$  time, meaning it must run strictly faster than linear time, so an  $O(n)$  solution is not sufficient. A binary search-based approach will achieve  $O(\log n)$  time complexity, which satisfies the requirement.

### Algorithm: Binary Search Approach

Since  $A$  is sorted and contains distinct integers, we can leverage binary search to find if there exists an index  $m$  where  $A[m] = m$ .

#### Steps:

1. **Initialize:** Set  $low = 0$  and  $high = n - 1$ .
2. **Binary Search:**
  - Compute  $mid = (low + high) / 2$  (integer division).
  - If  $A[mid] == mid$ , return  $mid$  (found).
  - If  $A[mid] > mid$ , search the left half ( $high = mid - 1$ ).
  - If  $A[mid] < mid$ , search the right half ( $low = mid + 1$ ).
3. **Repeat until  $low > high$ .** If no match is found, return false.

#### Proof of $O(\log n)$ Complexity

1. **Binary search repeatedly halves the search space.**
  - At each step, we eliminate half of the remaining elements.
  - The number of steps required is at most  $\log_2(n)$  since after  $\log_2(n)$  splits, we reduce the array to size 1.
2. **Time Complexity Calculation:**
  - Let  $T(n)$  be the runtime of our algorithm.
  - Since each step reduces the problem size by half:
$$T(n) = T(n/2) + O(1)$$
    - This recurrence solves to  $O(\log n)$  using the recurrence tree method or the Master Theorem.
3.  **$O(\log n)$  is strictly faster than  $o(n)$  ("little-oh"):**
  - We require  $o(n)$ , which means the function must grow strictly slower than  $O(n)$ .
  - Since  $O(\log n)$  grows significantly slower than  $O(n)$ , our algorithm satisfies this condition.

#### Conclusion:

- Algorithm: Use binary search to check if  $A[m] = m$ .
- Time Complexity:  $O(\log n)$ , which is  $o(n)$ .
- Correctness: Since the array is sorted and contains distinct elements, binary search efficiently narrows the search space.

#### 4. Devise a pivot-selection strategy for QuickSort that will guarantee that your new QuickSort has a worst-case running time of $O(n \log n)$ .

To ensure QuickSort always runs in  $O(n \log n)$  worst-case time, we must select pivots in a way that avoids worst-case unbalanced partitions (which lead to  $O(n^2)$  complexity). The median-of-medians algorithm is an effective strategy that ensures balanced partitions.

##### **Pivot Selection Strategy: Median-of-Medians**

Instead of picking a random or fixed-position pivot, we select the pivot strategically using the following steps:

##### **Step 1: Divide the Array into Groups**

- Divide the array of  $n$  elements into groups of 5 elements (or a constant small size  $k$ ).
- If  $n$  is not a multiple of 5, the last group may contain fewer elements.

##### **Step 2: Find the Median of Each Group**

- Sort each group of 5 (which takes  $O(1)$  time per group).
- Find the median of each group (the middle element after sorting).

##### **Step 3: Recursively Find the Median of Medians**

- Collect all the medians from the groups into a new array.
- Recursively apply the median-of-medians algorithm to find the median of these medians.
- This median is chosen as the pivot.

##### **Step 4: Partition the Array Around the Pivot**

- Perform the standard QuickSort partitioning step using this pivot.
- Since the pivot is close to the true median, it ensures that both partitions are at least 30% and at most 70% of the original size.

##### **Why Does This Guarantee $O(n \log n)$ Worst-Case?**

###### **1. Balanced Partitions:**

- The median-of-medians pivot always splits the array into two reasonably balanced parts, ensuring that neither partition is too small.
- Each partition is at least 30% of the original size, avoiding worst-case  $O(n^2)$  splits (such as when selecting the smallest or largest element).

###### **2. Time Complexity Breakdown:**

- **Step 1:** Grouping elements into sets of 5  $\rightarrow O(n)$

- **Step 2:** Sorting each group  $\rightarrow O(n)$  (since sorting groups of 5 is constant time per group)
- **Step 3:** Finding the median of medians recursively  $\rightarrow T(n/5)$
- **Step 4:** Partitioning the array around the pivot  $\rightarrow O(n)$
- **Final recurrence relation:**

$$T(n) = T(n/5) + T(7n/10) + O(n)$$

- Using the recursion tree method or the Master Theorem, this solves to  $O(n \log n)$ .

## Conclusion

Using the median-of-medians pivot selection strategy, we ensure that:

Each partition remains balanced, preventing worst-case  $O(n^2)$  behavior.

The worst-case time complexity is  $O(n \log n)$ .

This strategy makes QuickSort a robust deterministic sorting algorithm with guaranteed worst-case efficiency.

**5. Show the steps performed by QuickSelect as it attempts to find the median of the array [1, 12, 8, 7, -2, -3, 6]. (The median is the element that is less than or equal to  $n/2$  of the elements in the array. Since  $n$  is odd in this case, it is the element whose position lies exactly in the middle. Hint: The median is 6.) For pivots, always use the leftmost element of the current array.**

### QuickSelect Algorithm to Find the Median of [1, 12, 8, 7, -2, -3, 6]

We need to find the median, which is the 4th smallest element (since the array has 7 elements, and the median is at position  $(n+1)/2 = 4$  when indexed from 1).

We will use QuickSelect, always choosing the leftmost element as the pivot.

### Step-by-Step Execution of QuickSelect

#### Step 1: Initial Array and First Pivot

**Array:** [1, 12, 8, 7, -2, -3, 6]

**Pivot:** 1 (leftmost element)

**Partitioning Around Pivot (1):**

- Elements less than 1: [-2, -3]
- Elements equal to 1: [1]
- Elements greater than 1: [12, 8, 7, 6]

**Partitioned Array:** [-2, -3] [1] [12, 8, 7, 6]

**Rank of pivot (1) in sorted order: 3rd**

- Since we need the 4th smallest element, it must be in the right partition [12, 8, 7, 6].
- Recursive call on [12, 8, 7, 6] to find the 1st smallest element in this subarray (since we now need the element at index  $4 - 3 = 1$ ).

**Step 2: Recursive Call on [12, 8, 7, 6]**

**Array:** [12, 8, 7, 6]

**Pivot:** 12 (leftmost element)

**Partitioning Around Pivot (12):**

- Elements less than 12: [8, 7, 6]
- Elements equal to 12: [12]
- Elements greater than 12: []

**Partitioned Array:** [8, 7, 6] [12] []

**Rank of pivot (12) in sorted order: 4th**

- Since we need the 1st smallest element, we recurse on [8, 7, 6], looking for the 1st smallest element.

**Step 3: Recursive Call on [8, 7, 6]**

**Array:** [8, 7, 6]

**Pivot:** 8 (leftmost element)

**Partitioning Around Pivot (8):**

- Elements less than 8: [7, 6]
- Elements equal to 8: [8]
- Elements greater than 8: []

**Partitioned Array:** [7, 6] [8] []

**Rank of pivot (8) in sorted order: 3rd**

- Since we need the 1st smallest element, we recurse on [7, 6], looking for the 1st smallest element.

#### Step 4: Recursive Call on [7, 6]

**Array:** [7, 6]

**Pivot:** 7 (leftmost element)

#### Partitioning Around Pivot (7):

- Elements less than 7: [6]
- Elements equal to 7: [7]
- Elements greater than 7: []

**Partitioned Array:** [6] [7] []

**Rank of pivot (7) in sorted order: 2nd**

- Since we need the 1st smallest element, we recurse on [6], looking for the 1st smallest element.

#### Step 5: Recursive Call on [6]

Since the array has only one element (6), it is the answer.

**Median of [1, 12, 8, 7, -2, -3, 6] is 6.**

#### Conclusion:

1. **Pivot = 1**, partitions: [-2, -3] [1] [12, 8, 7, 6], recurse on [12, 8, 7, 6] for 1st smallest.
2. **Pivot = 12**, partitions: [8, 7, 6] [12] [], recurse on [8, 7, 6] for 1st smallest.
3. **Pivot = 8**, partitions: [7, 6] [8] [], recurse on [7, 6] for 1st smallest.
4. **Pivot = 7**, partitions: [6] [7] [], recurse on [6] for 1st smallest.
5. **Found 6**, which is the median.

#### Time Complexity Analysis:

- Each step reduces the problem size by approximately half.
- Worst-case time complexity:  $O(n)$  (linear), but expected runtime for random input is  $O(\log n)$ .

QuickSelect successfully finds the median in  $O(n)$  time using the leftmost pivot.