Lab 5

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

Phase 1: Starting Configuration

* Beginning array: [1, 6, 2, 4, 3, 5]
* Selected pivot: 1 (first element)
* Partitioning outcome: Since all other elements exceed 1, no exchanges necessary; pivot 1 remains correctly positioned
* Resulting sections: Empty left portion, right portion containing [6, 2, 4, 3, 5]

Phase 2: Processing Right Section [6, 2, 4, 3, 5]

* Selected pivot: 6 (first element)
* Partitioning outcome: Elements below 6 value: [2, 4, 3, 5]; pivot 6 shifts to end position
* Resulting sections: Left portion with [2, 4, 3, 5], empty right portion
* Current state: [2, 4, 3, 5, 6]

Phase 3: Processing Left Section [2, 4, 3, 5]

* Selected pivot: 2 (first element)
* Partitioning outcome: Elements exceeding 2: [4, 3, 5]; pivot 2 maintains correct position
* Resulting sections: Empty left portion, right portion with [4, 3, 5]
* Current state: [2, 4, 3, 5]

Phase 4: Processing Secondary Right Section [4, 3, 5]

* Selected pivot: 4 (first element)
* Partitioning outcome: Elements below 4: [3]; elements above 4: [5]; pivot 4 moves to appropriate position
* Resulting sections: Left portion [3], right portion [5]
* Current state: [3, 4, 5]

Phase 5: Final Processing

* Both [3] and [5] contain single elements and require no further sorting
* Complete sorted sequence: [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:**

**i. the number of elements < x is less than 3n/4, and also**

**ii. 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?**

a.

3n/4 equals (3\*9)/4 which calculates to 6.5, approximately 7.

Pivot Selection Requirements: Requirement 1: Elements less than pivot value must number fewer than 7. Requirement 2: Elements greater than pivot value must number fewer than 7.

| **Value** | **Elements < Value (Count)** | **Elements > Value (Count)** | **Meets Requirement 1** | **Meets Requirement 2** | **Fulfills Both Requirements** |
| --- | --- | --- | --- | --- | --- |
| 1 | [] (0) | [2,3,3,4,5,6,7] (7) | Yes | No | No |
| 2 | [1,1] (2) | [3,3,4,5,6,7] (6) | Yes | Yes | Yes |
| 3 | [1,1,2] (3) | [4,5,6,7] (4) | Yes | Yes | Yes |
| 4 | [1,1,2,3,3] (5) | [5,6,7] (3) | Yes | Yes | Yes |
| 5 | [1,1,2,3,3,4] (6) | [6,7] (2) | Yes | Yes | Yes |
| 6 | [1,1,2,3,3,4,5] (7) | [7] (1) | No | Yes | No |
| 7 | [1,1,2,3,3,4,5,6] (8) | [] (0) | No | Yes | No |

Conclusion: Optimal pivot values are {2, 3, 4, 5}

b. The good pivots are {2,3,4,5} = 4

4/9 (0.44) is less than 0.5 hence false

* 1. **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**

**Algorithm: Binary Search Technique**

**Setup Phase:** Initialize lower bound = 0 and upper bound = n - 1 (representing array boundaries).

**Iterative Search Process:** While lower bound ≤ upper bound:

* Calculate central position: central index = (lower bound + upper bound) ÷ 2
* Evaluate array value at central position:
  + When A[central index] equals central index: Return Success (found matching position where A[m]=m)
  + When A[central index] < central index: Shift search rightward by setting lower bound = central index + 1 (all leftward elements will fail due to sorted unique properties)
  + When A[central index] > central index: Shift search leftward by setting upper bound = central index - 1 (all rightward elements will fail for identical reasons)

**Termination Condition:** If iteration completes without finding any position where A[m]=m, return Failure.

**Practical Illustration:** Input sequence: A=[-3,-1,1,3,5] Initial state: lower bound = 0, upper bound = 4, central index = (0 + 4) ÷ 2 = 2 A[2]=1 is below 2. Adjust search rightward (lower bound = central index + 1 = 3) Next iteration: lower bound = 3, upper bound = 4, central index = (3 + 4) ÷ 2 = 3 A[3]=3 matches 3. Return Success.

**Mathematical Validation:** Invariant property: Each iteration halves the search region while preserving the invariant that any A[m]=m must exist within the remaining section. Properties of uniqueness and ordering: The array's sorted and distinct nature ensures that A[central index]<central index or A[central index]>central index consistently guides the search toward the appropriate half.

* 1. **Devise a pivot-selection strategy for QuickSort that will guarantee that your new QuickSort has a worst-case running time of O(nlog n).**

Here's a rephrased version of the Median-of-Medians information:

**Median-of-Medians: Advanced Pivot Selection Technique**

The Median-of-Medians approach ensures pivots approximate the true median value, creating balanced partitioning and securing O(n log n) worst-case efficiency for QuickSort operations.

**Implementation Procedure**

1. **Array Segmentation**: Separate the input sequence into groups containing 5 elements each. (The final group may contain fewer elements if the total count isn't divisible by 5.)
2. **Group Median Identification**: Arrange each 5-element group and identify its middle value. This operation requires constant O(1) time since group size remains fixed.
3. **Median Collection**: Construct a separate array containing all identified group medians.
4. **Master Median Calculation**: Apply the Median-of-Medians technique recursively to the collection of medians to determine the "median of medians." This value becomes the designated pivot.
5. **Array Division**: Utilize the selected pivot to separate the original array into three sections: elements below, equal to, and above the pivot value.
6. **Recursive Processing**: Apply QuickSort recursively to the lower and higher partitions.

**Performance Guarantee Explanation**

1. **Balanced Division**: The median-of-medians selection method guarantees an effective pivot by positioning it near the actual median value. This ensures partitioning where each section contains at most 75% of the original elements.
2. **Worst-Case Prevention**: Unlike simpler selection methods (such as selecting first or last elements), this approach prevents problematic scenarios where QuickSort degrades to O(n²) performance.

**Key Benefits**

* **Assured O(n log n) Complexity**: Prevents highly unbalanced partitioning scenarios
* **Predictable Performance**: Functions independently of randomization or data distribution assumptions
* **Resilience**: Maintains effectiveness even with deliberately challenging inputs designed to trigger worst-case behavior

While Median-of-Medians guarantees optimal worst-case efficiency, it does introduce computational overhead during pivot selection, making it typically slower than standard QuickSort implementations for average cases. Its primary value lies in applications where consistent worst-case performance guarantees are essential.

* 1. **Show the steps performed by Quick Select 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.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **STEPS** | **CURRENT SUBARRAY** | **Pivot** | **Elements < Pivot** | **Elements > Pivot** | **Array after partition** | **Pivot Position** | **ACTION** |
| 1 | [1,12,8,7,-2,-3,6] | 1 | [-2,-3] | [12,8,7,6] | [-2,-3,1,12,8,7,6] | 2 | k=3>2 so, Search in the right subarray [12,8,7,6] |
| 2 | [12,8,7,6] | 12 | [8,7,6] | [] | [-2,-3,1,8,7,6,12] | 6 | k=3< 6 so, Search in the left subarray [8,7,6] |
| 3 | [8,7,6] | 8 | [7,6] | [] | [-2,-3,1,7,6,8,12] | 5 | k=3<5 so, Search in the left subarray [7,6] |
| 4 | [7,6] | 7 | [6] | [] | [-2,-3,1,6,7,8,12] | 4 | K =3<4 search in the left subarray |
| 5 | [6] | 6 | [] | [] | [-2,-3,1,6,7,8,12] | 3 | K=3=PIVOT POSTN **FOUND THE MEDIAN** |