Assignment 5

Divide and Conquer

Ehu Shubham Shaw
Joe Johnson
March 29, 2025

Question 1

Solution:

I'll use a binary search approach to find the median with O(log n) queries.
Let's denote the two databases as A and B. The algorithm works as follows:

Initialize search ranges for both databases: A[1 till n] and B[1 till n]
Initialize low_A = 1, high_A = n, low_B = 1, high_B = n
While the search ranges are not reduced to a single element:

Let mid_A = [(low_A + high_A)/2] and mid_B = [(low_B + high_B)/2]
Query for A[mid_A] and B[mid_B]
Count how many elements in the combined array are less than A[mid_A]:

Count_A = mid_A - 1 (elements in A less than A[mid_A])
If B[mid_B] < A[mid_A], then Count_B = mid_B, else perform binary search in B to find largest index i where B[i] < A[mid_A]
Total = Count_A + Count_B

If Total == n-1, then A[mid_A] is the median
If Total > n-1, too many elements are smaller than A[mid_A], so adjust high_A = mid_A - 1
If Total < n-1, too few elements are smaller than A[mid_A], so adjust low_A = mid_A + 1
Similarly check B[mid_B] if needed

This algorithm performs O(log n) queries because each iteration reduces the search space by approximately half, and we need to find the position in O(log n) time.

Question 2

Solution:

We can modify the merge sort algorithm that counts regular inversions to count significant inversions

```
CountSignificantInversions(A[1.till n]):
   if n = 1:
      return 0

   m = ⌊n/2⌋
   left_count = CountSignificantInversions(A[1...m])
   right_count = CountSignificantInversions(A[m+1...n])
   split_count = MergeAndCount(A[1...m], A[m+1...n])

   return left_count + right_count + split_count

function MergeAndCount(L[1...p], R[1...q]):
   count = 0
   i = 1, j = 1
```

```
    for i = 1 to p:
        while j ≤ q and L[i] > 2·R[j]:
            j++
        count += (j - 1)

    i = 1, j = 1, k = 1
    while i ≤ p and j ≤ q:
        if L[i] ≤ R[j]:
            A[k++] = L[i++]
        else:
            A[k++] = R[j++]

    while i ≤ p: A[k++] = L[i++]
    while j ≤ q: A[k++] = R[j++]

    return count
```

The time complexity is O(n log n) since we're following the same divide-and-conquer approach as merge sort, with each level of recursion doing O(n) work and the recursion depth being O(log n).

Question 3

Solution:

We can solve this using a divide-and-conquer approach similar to finding a majority element:

```
FindMajorityEquivalent(Cards[1...n]):
    if n = 1:
        return Cards[1]

    m = [n/2]
    left_majority = FindMajorityEquivalent(Cards[1...m])
    right_majority = FindMajorityEquivalent(Cards[m+1...n])

    if left_majority = "no majority":
        candidate = right_majority
    else if right_majority = "no majority":
        candidate = left_majority
    else if EquivalenceTester(left_majority, right_majority):
        candidate = left_majority
    else:
        left_count = CountEquivalent(Cards, left_majority)
        right_count = CountEquivalent(Cards, right_majority)
        if left_count > right_count:
            candidate = left_majority
        else:
            candidate = right_majority


    if CountEquivalent(Cards, candidate) > n/2:
```

```
        return candidate
    else:
        return "no majority"

function CountEquivalent(Cards[1...n], card):
    count = 0
    for i = 1 to n:
        if EquivalenceTester(Cards[i], card):
            count++
    return count
```

The time complexity is O(n log n) because: The recursive calls create O(log n) levels at each level, we perform at most O(n) equivalence tests therefore, total complexity is O(n log n).

Question 4

Solution:

We can find a local minimum by following a path downward from the root:

Start at the root node if the current node is a local minimum, return it
Otherwise, among the children of the current node, move to the one with the minimum value repeat steps 2-3 until a local minimum is found

```
FindLocalMinimumInTree(T):
    current = root of T
    while true:
        val = probe(current)
        if current is a leaf:
            return current

        left_child = left child of current
        left_val = probe(left_child)
        right_child = right child of current
        right_val = probe(right_child)
        if val < left_val and val < right_val:
            return current
        if left_val < right_val:
            current = left_child
        else:
            current = right_child
```

This algorithm makes O(log n) probes because

Each iteration moves one level down in the tree
The height of a complete binary tree with n nodes is O(log n)
We make a constant number of probes per level

Question 5

Solution:

We can use a divide-and-conquer approach to find a local minimum in O(n) Probes probe the middle column of the grid then find the minimum value m in this column check if m is a local minimum by probing its left and right neighbors if m is a local minimum, return it otherwise, one of its neighbors has a smaller value. Go to the half of the grid containing this smaller neighbor repeat the process on the new sub grid

```
FindLocalMinimumInGrid(G, n):
    return FindMinHelper(G, 1, 1, n, n)

function FindMinHelper(G, x1, y1, x2, y2):
    if x1 = x2 and y1 = y2:
        return (x1, y1)

    mid_x = [(x1 + x2)/2]

    min_val = infinity
    min_y = -1
    for y = y1 to y2:
        val = probe(G, mid_x, y)
        if val < min_val:
            min_val = val
            min_y = y

    is_local_min = true

    if mid_x > x1:
        left_val = probe(G, mid_x-1, min_y)
        if left_val < min_val:
            is_local_min = false
            go_left = true

    if mid_x < x2:
        right_val = probe(G, mid_x+1, min_y)
        if right_val < min_val:
            is_local_min = false
            go_left = false

    if min_y > y1:
        top_val = probe(G, mid_x, min_y-1)
        if top_val < min_val:
            return FindMinHelper(G, mid_x, y1, mid_x, min_y-1)

    if min_y < y2:
        bottom_val = probe(G, mid_x, min_y+1)
        if bottom_val < min_val:
            return FindMinHelper(G, mid_x, min_y+1, mid_x, y2)
```

```
    if is_local_min:
        return (mid_x, min_y)

    if go_left:
        return FindMinHelper(G, x1, min_y, mid_x-1, min_y)
    else:
        return FindMinHelper(G, mid_x+1, min_y, x2, min_y)
```

This algorithm makes O(n) probes because at each level of recursion, we probe O(n) elements
the recursion depth is O(log n) but since the problem size decreases by more than half each time, we can prove that the total number of probes is bounded by O(n),he recurrence relation is $T(n) = T(n/2) + O(n)$, which resolves to O(n).