Problem 1 (Divide and Conquer the Peak, 30 pts)

Suppose we have an array A of n distinct integers and moreover are guaranteed that the array has the following property: up to some index $1 \leq i \leq n$, A is increasing, i.e., $A[1] < A[2] < \ldots < A[i]$, and then after index i, A is decreasing, i.e., $A[i] > A[i+1] > \ldots > A[n]$. In this array, we call $A[i]$ the *peak* of A. For example, consider the array $[1, 4, 7, 8, 6, 2]$, which has peak 8.

Create an algorithm that finds the peak of an input array A in sublinear time (i.e. $o(n)$). Justify the correctness and time complexity of your proposed algorithm.

Solution

The key observation is that since the array is first increasing and then decreasing, a binary search can be employed to efficiently find the peak. Let low and high denote the current bounds of the search interval. At each step, compute

$$\text{mid} = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor.$$

Then, compare $A[\text{mid}]$ with $A[\text{mid} + 1]$:

- If $A[\text{mid}] < A[\text{mid} + 1]$, then the array is still increasing at mid, so the peak must lie in the right half. Set low = mid + 1.
- Otherwise, if $A[\text{mid}] > A[\text{mid} + 1]$, then the peak is either at mid or to its left, so set high = mid.

The loop terminates when low = high, and at that point, $A[\text{low}]$ is the peak.

Pseudocode:

FindPeak(A, n)

```
L ← 1
R ← n
while L < R do
    mid ← ⌊(L + R)/2⌋
    if A[mid] < A[mid + 1] then
        L ← mid + 1
    else
        R ← mid
    end if
end while
return A[L]
```

Correctness:
The algorithm maintains the invariant that the peak is always within the interval [low, high]. At each iteration:

- If $A[\text{mid}] < A[\text{mid}+1]$, the increasing property guarantees that the peak lies in the interval $[\text{mid}+1, \text{high}]$.
- If $A[\text{mid}] \geq A[\text{mid}+1]$, the peak is in $[\text{low}, \text{mid}]$ since either $A[\text{mid}]$ is the peak or the peak lies to its left.

When the loop terminates with low = high, only one element remains, and by our invariant, it must be the peak.

Time Complexity:
At each iteration, the search space is reduced by roughly half. Therefore, the number of iterations is $O(\log n)$, which is sublinear with respect to n.

Problem 2 (Two-Server Search, 35 pts)

You have (limited) access to two databases, each of which contains n numbers. For simplicity, you may assume each value across the $2n$ entries is unique. You may only query the databases in the following way: You give one of the databases an integer $1 \leq k \leq n$, and it responds with the k-th smallest number in its database.

Give a divide-and-conquer algorithm that finds the median value across all $2n$ entries using $O(\log n)$ queries to the servers. Justify the correctness and time complexity of your proposed algorithm.

Solution

We can view the two databases as two sorted arrays A and B of length n each. The median of the $2n$ entries is the n-th smallest element overall. We use a divide-and-conquer strategy to find the k-th smallest element in the union of two sorted arrays. At each step, we compare the elements at appropriately chosen positions in A and B and discard a portion of one array that cannot contain the k-th smallest element. This recursion decreases the effective search size by roughly a factor of 2 per iteration, leading to $O(\log n)$ queries.

Since we can only access each database by querying for a specific order statistic, we simulate array access by issuing queries such as $\text{Query}(A, k)$ to get the k-th smallest element in A.

Algorithm Details:
We first define a recursive subroutine KthSmallest that, given current index bounds in A and B and a number k, returns the k-th smallest element in the union of the subarrays. The base cases handle when one database is exhausted or when $k = 1$. In the recursive step, let

$$i = \min(\text{number of remaining elements in } A, \lfloor k/2 \rfloor)$$

and

$$j = \min(\text{number of remaining elements in } B, \lfloor k/2 \rfloor).$$

We query for $a = \text{Query}(A, L_A + i - 1)$ and $b = \text{Query}(B, L_B + j - 1)$ (where L_A and L_B are the current lower indices for A and B respectively). If $a < b$, then the first i elements of A can be discarded; otherwise, discard the first j elements of B. Recursively adjust k accordingly.

Pseudocode:

FindMedianTwoServers(A, B, n)

1: $L_A \leftarrow 1$, $R_A \leftarrow n$
2: $L_B \leftarrow 1$, $R_B \leftarrow n$
3: $k \leftarrow n$
4: **return** KthSmallest$(A, B, L_A, R_A, L_B, R_B, k)$

KthSmallest$(A, B, L_A, R_A, L_B, R_B, k)$

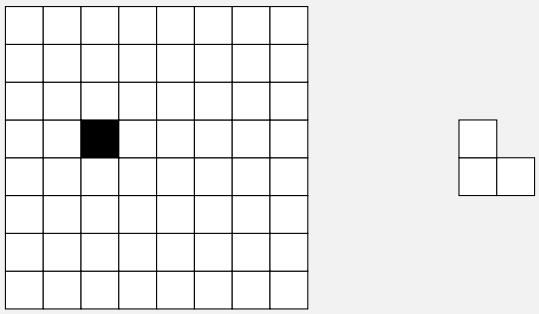
1: **if** $L_A > R_A$ **then**
2: **return** $\text{Query}(B, L_B + k - 1)$

end if
if $L_B > R_B$ **then**
  return $\text{Query}(A, L_A + k - 1)$
end if
if $k = 1$ **then**
  return $\min(\text{Query}(A, L_A), \text{Query}(B, L_B))$
end if
$i \leftarrow \min(R_A - L_A + 1, \lfloor k/2 \rfloor)$
$j \leftarrow \min(R_B - L_B + 1, \lfloor k/2 \rfloor)$
$a \leftarrow \text{Query}(A, L_A + i - 1)$
$b \leftarrow \text{Query}(B, L_B + j - 1)$
if $a < b$ **then**
  return $\text{KthSmallest}(A, B, L_A + i, R_A, L_B, R_B, k - i)$
else
  return $\text{KthSmallest}(A, B, L_A, R_A, L_B + j, R_B, k - j)$
end if

Correctness and Time Complexity:
At every recursive call, the algorithm maintains the invariant that the k-th smallest element in the union of the remaining portions of A and B is preserved. By comparing $\text{Query}(A, L_A + i - 1)$ and $\text{Query}(B, L_B + j - 1)$, we can safely discard i or j elements from one database, ensuring that the new problem size reduces k by at least $\lfloor k/2 \rfloor$. Therefore, the depth of the recursion is $O(\log n)$, and each recursive call makes only $O(1)$ queries. Consequently, the overall algorithm uses $O(\log n)$ queries.

Problem 3 (Grid Tiling, 35 pts)

For a positive integer n, consider a $2^n \times 2^n$ grid, where one of the unit squares is black and all others are white. Show that regardless of the position of the black unit square, the white area can be fully covered, without any overlapping, by L-shaped tiles consisting of 3 unit squares (rotations are allowed).



Left: an example grid with $n = 3$; Right: the L-shaped tile

(*Hint: Use a proof by construction, i.e. design a divide-and-conquer algorithm that takes as input n and (i, j) with $1 \leq i, j \leq 2^n$, and outputs a way to tile a $2^n \times 2^n$ grid with the black square at position (i, j).*)

Solution

We solve the problem by designing a divide-and-conquer algorithm. The idea is to use recursion to break the $2^n \times 2^n$ grid into four quadrants, place one L-shaped tile at the center to "simulate" a defect in the three quadrants that do not initially contain the black square, and then recursively tile each quadrant.

Algorithm Description:

1. **Base Case:** When $n = 1$, the grid is 2×2. Since one square is black, the remaining three squares form an L-shape, which can be covered by a single L-shaped tile.
2. **Recursive Step:** For $n \geq 2$:
   (a) Divide the $2^n \times 2^n$ grid into four quadrants, each of size $2^{n-1} \times 2^{n-1}$.
   (b) Identify the quadrant that contains the given black square (the defect).
   (c) Place one L-shaped tile at the center of the grid so that it covers the central unit square of the three quadrants that do *not* contain the black square. This placement introduces a new defect in each of those three quadrants.
   (d) Recursively tile each quadrant:
   - In the quadrant that originally contained the black square, use that black square.
   - In the other three quadrants, use the square covered by the central L-tile (the newly created defect) as the black square.

Pseudocode:

TileDefectiveBoard(n, i, j, r, c)

Input: n (the grid size is $2^n \times 2^n$); (i, j) is the position of the black square within the grid whose bottom-left corner is at (r, c).
Output: A tiling of the grid using L-shaped tiles.

1: **if** $n = 1$ **then**
2: **Place an L-tile** covering the three white squares in the 2×2 grid.
3: **return**
4: **end if**
5: Let $m \leftarrow 2^{n-1}$ {This is the midpoint that partitions the grid}
6: Determine the quadrant Q that contains the defective (black) square:

- **Quadrant 1 (bottom-left):** if $i \leq m$ and $j \leq m$.
- **Quadrant 2 (bottom-right):** if $i \leq m$ and $j > m$.
- **Quadrant 3 (top-left):** if $i > m$ and $j \leq m$.
- **Quadrant 4 (top-right):** if $i > m$ and $j > m$.

7: **Place one L-tile** at the center of the grid covering the central squares of the three quadrants that do not contain the original black square.
8: For each quadrant $k = 1, 2, 3, 4$:

- Let (r_k, c_k) denote the bottom-left corner of quadrant k.
- Let (i_k, j_k) be the position of the defective square in quadrant k:
  - If quadrant k contains the original defective square, then $(i_k, j_k) = (i, j)$ (translated relative to the quadrant's coordinates).
  - Otherwise, (i_k, j_k) is the position of the square covered by the central L-tile in quadrant k.
- **Recursively call** TileDefectiveBoard$(n - 1, i_k, j_k, r_k, c_k)$.

9: **return**
