1 Problem 1

1.1 Algorithm Description

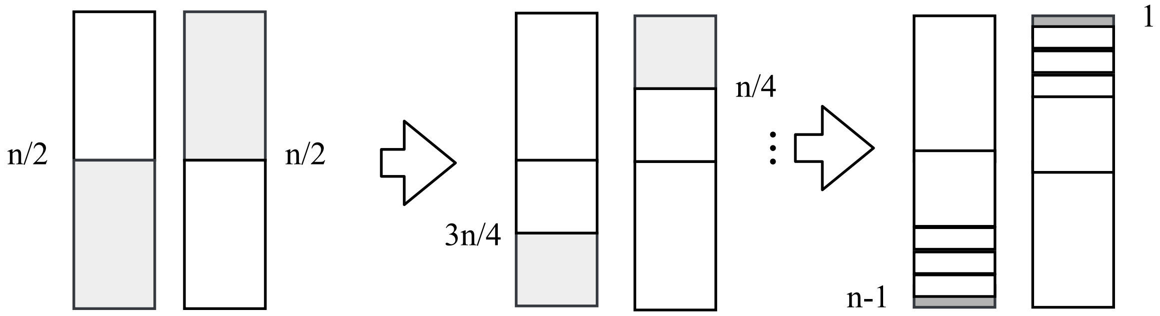
Let S_1, S_2 be the data sets in 2 separate databases. Query() takes a data set as its first input, and a number k indicating the k-th smallest number the server is supposed to return. The algorithm to solve this question is as follows.

Algorithm 1 Finding-Joint-Median

procedure GetJointMedian(S_1, S_2, n)
    $k_1 \leftarrow \frac{n}{2}$
    $k_2 \leftarrow \frac{n}{2}$
    for $i \leftarrow 2$ *to* $\log n$ **do**
        $m_1 \leftarrow Query(S_1, k_1)$
        $m_2 \leftarrow Query(S_2, k_2)$
        if $m_1 \leq m_2$ **then**
            $k_1 \leftarrow k_1 + \frac{n}{2^i}$
            $k_2 \leftarrow k_2 - \frac{n}{2^i}$
        else
            $k_1 \leftarrow k_1 - \frac{n}{2^i}$
            $k_2 \leftarrow k_2 + \frac{n}{2^i}$
    if $m_1 \leq m_2$ **then**
        return m_1
    else
        return m_2

1.2 Sub-problem Reduction Graph

Figure 1: Sub-problem reduction graph



1.3 Analysis

1.3.1 Proof of Correctness

We start by querying the $\frac{n}{2}th$ smallest number of S_1, S_2, which yields 2 medians m_1, m_2. For convenience, we assume that $m_1 \leq m_2$. It's easy to notice that the median of the joint data sets $\{S_1, S_2\}$ is located within the range $[m_1, m_2]$. We can then search for the $\frac{3n}{4}th$ smallest number of S_1 (in the range $[\frac{n}{2}, n]$) and the $\frac{n}{4}th$ smallest number of S_2 (in the range $[0, \frac{n}{2}]$), which again narrows down the search range for the actual median by half. We keep running this queries until the search range convers only 1 elemnt. After $\log n$ queries, m_1, m_2 is the nth and $(n+1)th$ smallest number in the entire data set.

1.3.2 Time Complexity

As each iteration step in the above algorithm $GetJointMedian$ takes constant time, the overall time complexity of our algorithm is the same as the iteration depth, which is $O(\log n)$.

2 Problem 2

2.1 Algorithm Description

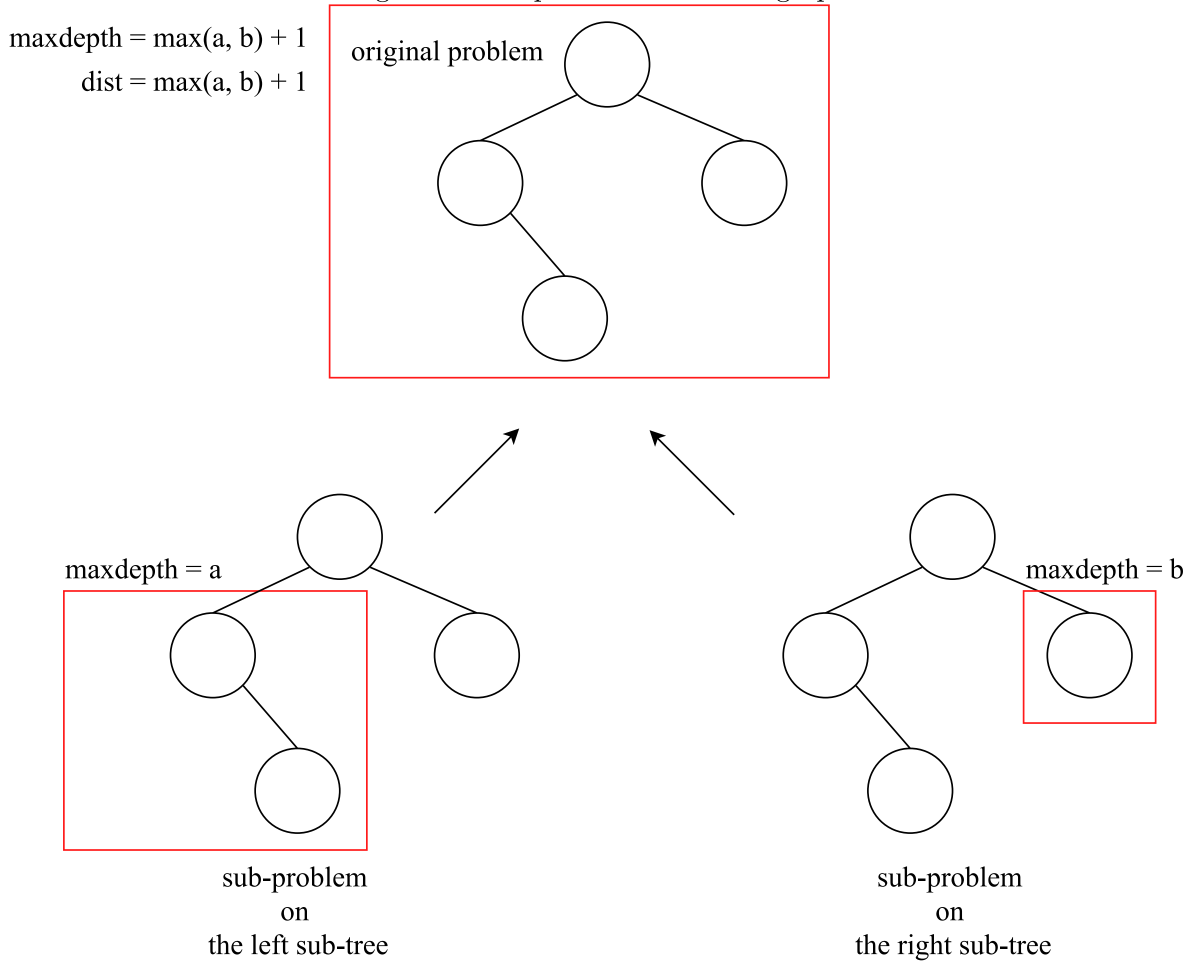
Let T denote the given binary tree, to find the maximum distance of any two node in a binary tree T_k, we introduce $MaxDepth_L, MaxDepth_R$ to represents the maximum depth of T_k's left and right sub-trees. It's easy to notice that the maximum distance of nodes in T_k that travels through its root is $MaxDepth_L + MaxDepth_R + 1$. The algorithm to solve the given problem is as follows.

Algorithm 2 Finding Maximum Distance

global variables
    $dist_{max} \leftarrow 0$
end global variables
procedure FindMaxDist(T)
    if T is a empty tree **then**
        return 0
    $MaxDepth_L \leftarrow FindMaxDist(T.left)$
    $MaxDepth_R \leftarrow FindMaxDist(T.right)$
    if $MaxDepth_L + MaxDepth_R > dist_{max}$ **then**
        $dist_{max} \leftarrow MaxDepth_L + MaxDepth_R + 1$
    return $max(MaxDepth_L, MaxDepth_R)$
Output $dist_{max} + 1$ as the final answer

2.2 Sub-problem Reduction Graph

Figure 2: Sub-problem reduction graph



2.3 Analysis

2.3.1 Proof of Correctness

$FindMaxDist$ recursively calculates the maximum depth of a sub-tree of the input binary tree from bottom to top. As we stated before, the maximum distance between nodes in the binary tree is continuously updated.

2.3.2 Time Complexsity

$T(n) = 2T(n/2) + c$, according to the Master Theorem, the time complexity of algorithm 2 is $O(n)$

3 Problem 6

3.1 Algorithm Description

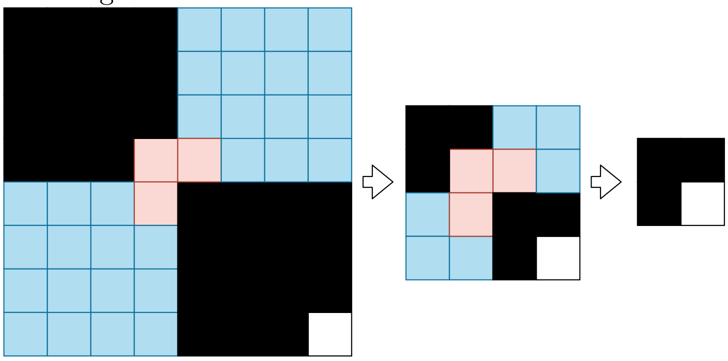
The algorithm given below returns the required filling scheme.

Algorithm 3 Filling Table

procedure LFill(2^n)
  if $n = 2$ **then**
    return $(1, 1), (1, 2), (2, 1)$
  $S \leftarrow LFill(2^{n-1})$
  $S_1 \leftarrow$ Rotate S by 90 degrees
  $S_2 \leftarrow$ Rotate S by 270 degrees
  $S_3 \leftarrow$ move S by $2^{n-1} \times 2^{n-1}$ blocks
  $mid \leftarrow (2^{n-1}, 2^{n-1}), (2^{n-1} + 1, 2^{n-1}), (2^{n-1}, 2^{n-1} + 1)$
  return $S \cup S_1 \cup S_2 \cup S_3 \cup mid$

3.2 Sub-problem Reduction Graph

Figure 3: Problem reduction of a $2^3 \times 2^3$ table



3.3 Analysis

3.3.1 Proof of Correctness

We noticed that the original problem to find the filling method for $2^n \times 2^n$ blocks (size $T(n)$) can be divided into solving a sub-problem for $2^{n-1} \times 2^{n-1}$ blocks. As is shown in figure 3 below, the problem of a $2^3 \times 2^3$ table consists of 4 $2^2 \times 2^2$ table. 2 of them is the rotation of the original solution by 90 or 270 degrees. The empty blocks left in the center of the table can then be filled by one L-shaped block. Thus, the output from algorithm 3 is the correct answer to the given problem.

3.4 Time Complexity

The rotation of the 2 sub-tables each takes $O(\frac{n}{2})$ time, the movement of the last sub-table takes $O(\frac{n}{2})$ time, thus the time complexity of this algorithm is $T(n) = T(n/2) + O(\frac{3n}{2})$. Applying the results of Master Theorem yields $T(n) = O(n)$.