Assignment 1: Divide And Conquer

Miao Hao 202328013229045

September 27, 2023

1 Question Number 1

1.1 Problem Description

Find the median of two sets, which size are both n, by querying for the k^{th} element in a set, and cost atmost $O(\log n)$ time.

1.2 Analyze

First, two sets are demoted by A and B respectively, and $S = A \cup B$. $Y[k]$ is for the k^{th} element in the set Y. *e.g.* The k_1^{th} element in set A is $A[k_1]$, and the median we want to find is $M = S[n]$. Elements in S_l are lower than the median, when elements S_g are greater than the median.

Second, we consider the simplest case of the porblem, and the value of n is 1. In this case, we should query at least 2 times to decide M.

Third, we consider a more complex situation, for $n = 2$. Note that if we want to find $M = S[2]$, there must be 1 element lower than M and 2 elements greater than M. Suppose that $k_1 - 1$ elements are from A and $k_2 - 1$ elements are from B to makes up S_l. The relation of k_1 and k_2 is declared by the following euqation:

$$(k_1 - 1) + (k_2 - 1) = n - 1 \tag{1}$$

In this case, we first let $k_1 = 1$. According to the equation, $k_2 = 2$. Then we **query** for $A[k_1]$ and $B[k_2]$.

If $A[k_1] < B[k_2]$, then we **query** for $B[k_2 - 1]$.

1. If $A[k_1] > B[k_2 - 1]$, we can make sure that there is $(k_1 - 1) + (k_2 - 1) = n - 1 = 1$ element in S which is lower than $A[k_1]$. In other words, $M = A[k_1] = A[1]$.
2. If $A[k_1] < B[k_2 - 1]$, which means that there are **at most** $(k_1 - 1) + (k_2 - 2) = 0$ elements in S lower than $A[k_1]$. The next thing to do is greater k_1 and lower k_2. Next we let $k_1' = k_1 + 1 = 2$, and $k_2' = k_2 - 1 = 1$. Then we **query** for $A[k_1']$. If $A[k_1'] < B[k_2']$, then $M = A[k_1'] = A[2]$. If $A[k_1'] > B[k_2']$, then $M = B[k_2'] = B[1]$.

If $A[k_1] > B[k_2]$, for k_1 could not be lower and k_2 could not be greater, we can make sure that there is $(k_1 - 1) + (k_2 - 1) = n - 1 = 1$ element in S which is lower than $B[k_2]$. Therefore, $M = B[k_2] = B[2]$.

Next, we consider common cases. First we prove a proposition.

Proposition 1. *Suppose that M_A, M_B is the median of A, B respectively, then the median of $S = A \cup B$, denoted by M, saticfices* $\min\{M_A, M_B\} \leq M \leq \max\{M_A, M_B\}$.

Proof.

Without loss of generality, we suppose that $M_A < M_B$.

If $M < M_A$, since M_A is the median of A, element number in S_l from A is lower than thoes from B, especially more elements which are greater than M_B would be in S_l. According to the assumption, $M < M_A < M_B$, there would be elements that greater than M in S_l, which makes a conflict. So we have proven that $M_A \leq M$.

If $M > M_B$, the same happens. So we have proven that $M \leq M_B$.

In conclusion, $M_A \leq M \leq M_M$. □

The proposition above ensures that if we query for M_A and M_B first, then M would be bounded by them. This conclusion leads to recursion. Before we give the algorithm, another proposition should be proven.

Proposition 2. *For all set A and B, $\exists!(k_1, k_2)$, which satisfies equation (1) and $S_l = \{A[1], A[2], ..., A[k_1 - 1], B[1], B[2], ..., B[k_2 - 1]\}$.*

Proof.

First, $\exists! M \in S$ obviously. Assume that $M \in A$, and is the k^{th} element. Then we let $k_1 = k$, $k_2 = n - k_1 + 1$.

Because M is unique, k is unique. Therefore, (k_1, k_2) is unique. □

According to **Proposition 1** and **Proposition 2**, we can find M by reducing the range of k_1 and k_2. The initial problem is reduced to a **search problem**, and we can use **binary search** to find k_1. Here comes the algorithm in natural language:

1. Given a search range $(begin, end)$, let $k_1 = \dfrac{begin + end}{2}$, $k_2 = n - k_1 + 1$. **Query** for $A[k_1]$ and $B[k_2]$. If $begin \geq end$, the median is found and its value is $\min\{A[k_1], B[k_2]\}$.

2. Decide the next step according to cases:

   - If $A[k_1] < B[k_2]$, **query** for $B[k_2 - 1]$, goto step 3.;

- If $A[k_1] > B[k_2]$, **query** for $A[k_1 - 1]$, goto step 4..

3. Decide the next step according to cases:
   - If $A[k_1] < B[k_2 - 1]$, goto step 1., give the range $(k_1 + 1, end)$;
   - If $A[k_1] > B[k_2 - 1]$, the median is found and its value is $A[k_1]$.
4. Decide the next step according to cases:
   - If $A[k_1 - 1] > B[k_2]$, goto step 1., give the range $(begin, k_1 - 1)$;
   - If $A[k_1 - 1] < B[k_2]$, the median is found and its value is $B[k_2]$.

1.3 Persudo Code

Algorithm 1 EX 1

```
function FindMedian(A, B, n, begin, end)
    k1 ← (begin + end) / 2
    k2 ← n − k1 + 1
    A[k1] ← query(A, k1)
    B[k2] ← query(B, k2)
    if begin ≥ end then
        return min{A[k1], B[k2]}
    end if
    if A[k1] < B[k2] then
        B[k2 − 1] ← query(B, k2 − 1)
        if A[k1] < B[k2 − 1] then
            FindMedian(A, B, n, k1 + 1, end)
        else
            return A[k1]
        end if
    else
        A[k1 − 1] ← query(A, k1 − 1)
        if A[k1 − 1] > B[k2] then
            FindMedian(A, B, n, begin, k1 − 1)
        else
            return B[k2]
        end if
    end if
end function
```

1.4 Time Complexity

In the worst case, $T(n) = T(n/2) + O(1)$, for the algorithm reduces the problem size by half each time it calls itself. The subproblem reduction graph is shown as follows:

$$
\begin{array}{c}
\text{FindMedian}(n) \cdots O(1) \\
| \\
\text{FindMedian}(n/2) \cdots O(1) \\
| \\
\text{FindMedian}(n/4) \cdots O(1) \\
| \\
\vdots \\
| \\
\text{FindMedian}(1) \cdots O(1)
\end{array}
$$

The depth of the subproblem reduction tree is $\log n$. Therefore, the time complexity of the algorithm $T(n) = O(\log n)$.

1.5 Correctness

Proof.

First, $(k_1 - 1) + (k_2 - 1) = n - 1$ is satisfied all the time.

We modify the value of k_1 to reduce. When we find $A[k_1] < B[k_2]$, if we choose $A[k_1]$ as the median, the number of element of B in S_l must be k_2. Thus, $A[k_1] > B[k_2 - 1]$ should be ensured. So if $A[k_1] > B[k_2 - 1]$, $A[k_1]$ is the median we want to find.

If $A[k_1] > B[k_2 - 1]$ is not satisfied, if we still choose $A[k_1]$ as the median, the number of elements in S_l is less than $n - 1$, which dose not meet the requirement. So we should make k_1 greater using the technique of **binary search**.

Another case is symmetrical to the case has analyzed.

Next, the size of the search range keeps reducing, so the algorithm would come to an end. $\square$

2 EX.4

2.1 Problem

Count the node number of a complete binary tree, and cost at most $O((\log(n))^2)$ time.

2.2 Analyze

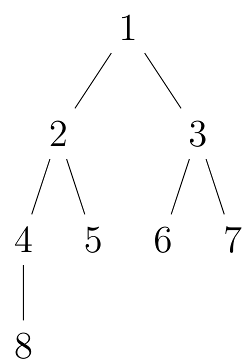
To count the node number of a complete binary tree, we should know the depth of the tree d (the depth of root node is 0) and the node number of the deepest layer of the tree n_d. Then, the node number of the tree is:

$$N_T = 2^d + n_d - 1 \tag{2}$$

So our goal is to acquire d and n_d in $O((\log(n))^2)$ time.

To get the depth of a tree, we should get the depth of the left subtree and the right subtree and choose a greater one, then add 1. For a complete binary tree, we do not need to know the depth of a tree's two shubtrees. We can go forward the left child node of each node from the root node to get the depth of a complete binary tree.

Here's an example:



We can start with the path $1 \to 2 \to 4 \to 8$ to acquire the depth of the complete binary tree above, and its depth is 4.

The second step is to get n_d. We can reduce the problem as ”to find the last node in the deepest layer of the complete binary tree” and it is a **serch prolem**. Thus, we can still ues the technique of **binary search**. Assume that there is an array with length 2^d to represent the nodes in the deepest layer of the complete binary tree. If the parent does not have a child, then use 0 to fill the array. Otherwise, 1 is used to fill the array. The array for the tree above is $[1, 0, 0, 0, 0, 0, 0, 0]$. Given an array makes up with continuous 1 and 0, we want to find the last 1 in the array. First, the array is separated into two parts: $[1, 0, 0, 0]$ and $[0, 0, 0, 0]$. Choose the part begin with 1, and do a further separation: $[1, 0]$ and $[0, 0]$. Repeat the steps, and finially we find the last 1 in the array, and it is the first element in the array.

The algorithm in natural language:

1) Given a complete binary tree T_c, go down by left to get the depth d, give T_c and range $(1, 2^d)$ to step 2);

2) Given a complete binary tree T_c and a range $(begin, end)$, get n_d by follows:

- If $begin \geq end$, n_d is found and its value is $begin$.
- Get d_l and d_r, which are the depth of subtrees of the root of T_c;
- If $d_l = d_r$, give the right subtree of root T_r and range $(end/2+1, end)$ to step 2);
- If $d_l > d_r$, give the left subtree of root T_l and range $(begin, end/2)$ to step 2).

3) Use equation(2) to calculate the node num of tree T.

2.3 Persudo Code

Algorithm 2 EX 4

```
function GetDepth(T_c)
    if T_c != NULL then
        return GetDepth(T_c.left) + 1
    end if
    return -1
end function
function GetNd(T_c, begin, end)
    if begin = end then
        return begin
    end if
    d_l ←GetDepth(T_c.left)
    d_r ←GetDepth(T_c.right)
    if d_l == d_r then
        return GetN(T_c.right, end/2 + 1, end)
    else if d_l > d_r then
        return GetN(T_c.left, begin, end/2)
    end if
end function
function GetNodeNum(T_c)
    d ←GetDepth(T_c)
    n_d ←GetN(T_c, 1, 2^d)
    return 2^d + n_d − 1
end function
```

2.4 Correctness

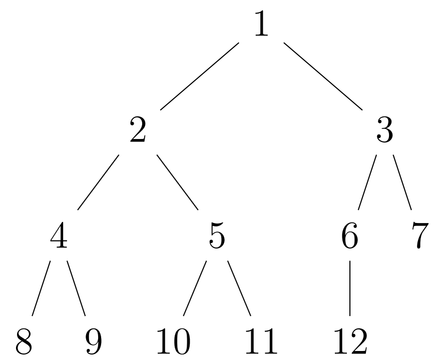
Proof. From the defination of the complete binary tree, the algorithm to get the depth of a complete binary tree is correct.

Next, to find d we use the technique of binary search. According to the properties of complete binary tree,

1) the subtrees are complete;

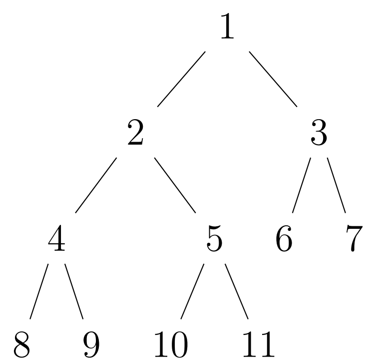
2) for each node, $d_l \geq d_r$.

When $d_l = d_r$, here is an example:



the last node is in the right subtree of root 1, so we choose the root's right child to continue our search.

When $d_l > d_r$, here is an example:



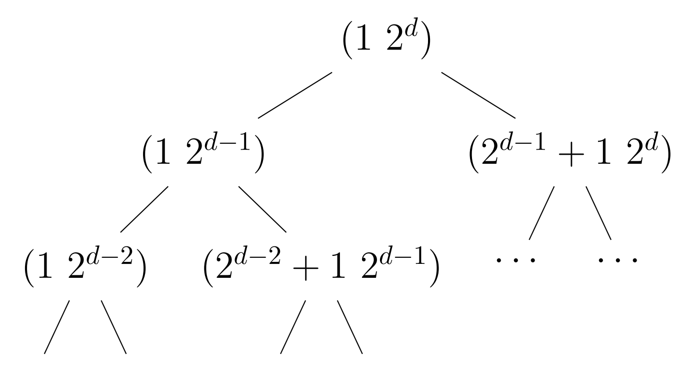
the last node is in the left subtree of root 1, so we choose the root's left child to continue our search.

Because the search range keeps decreasing, the algorighm will begin finally.

$\square$

2.5 Time Complexity

The subproblem reduction graph of this porblem is:



The time complexity $T(n) = O(\log n) \times O(\log n) = O((\log n)^2)$, in which $O(\log n)$ is the time complexity for getting the depth and binary serch.

3 EX.6

3.1 Problem

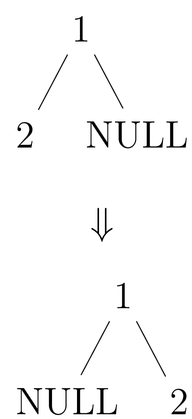
Given a binary tree T, please give an $O(n)$ algorithm to invert binary tree.

3.2 Analyze

This prolem is relatively easy. To invert a binary tree, we can exchange the left and right child of each node. Thus, this problem is reduced to a **traverse problem**.

For the simplest case, the node number of T is $n = 1$. It is so simple that the inversion of T is itself.

For $n = 2$, we exchange the left/right child with an empty node:



For common cases, we can get the inversion of T by exchange the left and child of each node in T. We can traverse the tree in any order.

The algorithm in natual language is to exchange the children of each tree during the traverse.

3.3 Persudo Code

Algorithm 3 EX 6

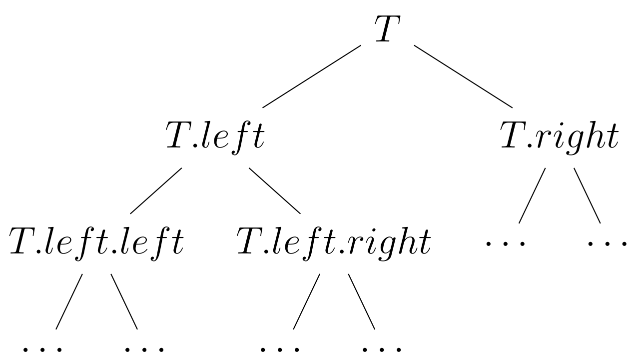
function INVERT(T)
    if $T = NULL$ **then**
        return
    end if
    INVERT($T.left$)
    INVERT($T.right$)
    $T.left \Longleftrightarrow T.right$
end function

3.4 Correctness

Proof. Since we traverse the tree, each node is visited only one time and its left and right child are exchanged, the algorithm is correct. □

3.5 Time Complexity

The subproblem reduction graph is:



The time complexity is the same as the node number, that is, $T(n) = O(n)$.