Assignment 01

肖阳 201918008629001

1

Solution:

(a) 在这个问题当中，需要求解一个旋转数组的最小值，考虑到数组在旋转之前是升序排列的，故算法只需要找到旋转之后原数组的首个元素所在位置即可求得数组的最小值。从分治算法的思路出发，设计一个递归算法，该算法的输入为待查找的数组 arr，以及数组左右两端的下标 left 和 right，通过左右下标求得数组的中间下标 mid。比较 arr[right]和 arr[mid]，若 arr[right]大于 arr[mid]则说明 mid 到 right 的数组是升序的，故继续递归查找 left 到 mid 的数组；若 arr[right]小于 arr[mid]则说明 mid 到 right 的数组是乱序的，即包含最小元素，故继续递归查找 mid+1 到 right 的数组；当算法输入左右下标相等时，返回数组该下标元素，即为最小元素。算法伪代码如下：

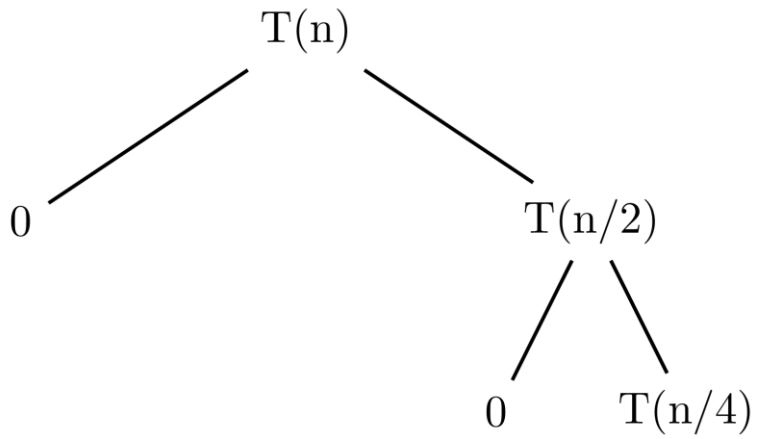
算法 1 求旋转数组最小数

输入: $Array$ 数组，n 数组大小

输出: 数组最小数

```
1: function MinElementInRotatedArray(Array, left, right)
2:     result ← Array[left]
3:     if left < right then
4:         middle ← (left + right)/2
5:         if Array[mid] < Array[right] then
6:             return MinElementInRotatedArray(Array, left, middle)
7:         end if
8:         if Array[mid] > Array[right] then
9:             return MinElementInRotatedArray(Array, middle + 1, right)
10:        end if
11:    end if
12:    return result
13: end function
14:
```

(b) 子问题递归图



(c) 算法的正确性证明

•初始化（Initialization）：在第一次递归中，对整个数组进行查询，故最小元素肯定包含在 left 到 right 的下标范围内。

•保持（Maintenance）：先假设在第 i 次递归中 Array[mid]>Array[right]。这时 mid 为第 i 次递归数组的左右下标的中位数，即 mid=(left + right)/2。考虑到数组原本是升序排列的，而下标mid ≤ right，所以在下标 mid 到 right 的区间内，数组不是升序，即旋转之后的最小元素位于下标 mid 到 right 的区间内。这时进入 i+1 次递归查询(mid , right]区间的数组，查询的数组范围缩小为原来的 1/2。反之，旋转后数组的最小元素在下标 left 到 mid 的区间内，通过第 i+1 次递归查询 [left , mid]区间的数组，同样查询数组的范围缩小到原来的 1/2。

•终止（Termination）：当查询数组的左右下标 left == right 时，说明此时待查询的数组仅剩下一个元素，根据循环不变性的性质可知，该元素即为数组的最小元素。

(d)复杂度分析

每一次递归数组的查询的规模都会减半，故有 T(n)=T(n/2)+c，根据主方法可以求得 T(n)=O($log_2 n$)。

2

Solution:

(a) 在这个问题当中需要求解一棵二叉树中的节点最大距离，每两个节点之间的距离为 1。该问题可以分为两种情况来讨论：①节点间最大距离的路径穿过根节点，从左子树的最深节点到达右子树的最深节点。②节点间最大距离的路径不穿过根节点，为左子树或者右子树中的节点最大距离的大者。最终取两种情况中的较大者作为一棵二叉树中节点最大距离。对于第一种情况，得到的距离为左子树的深度加上右子树的深度再加 2，并将特殊的空子树的深度记为-1；第二种情况可递归求解左右子树的深度。算法伪代码如下：

算法 1 求二叉树节点最大距离

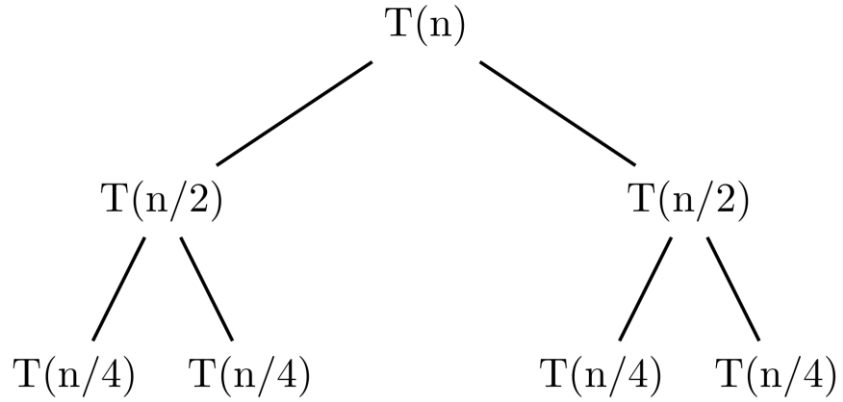
输入： $root$二叉树根节点

输出： 节点最大距离

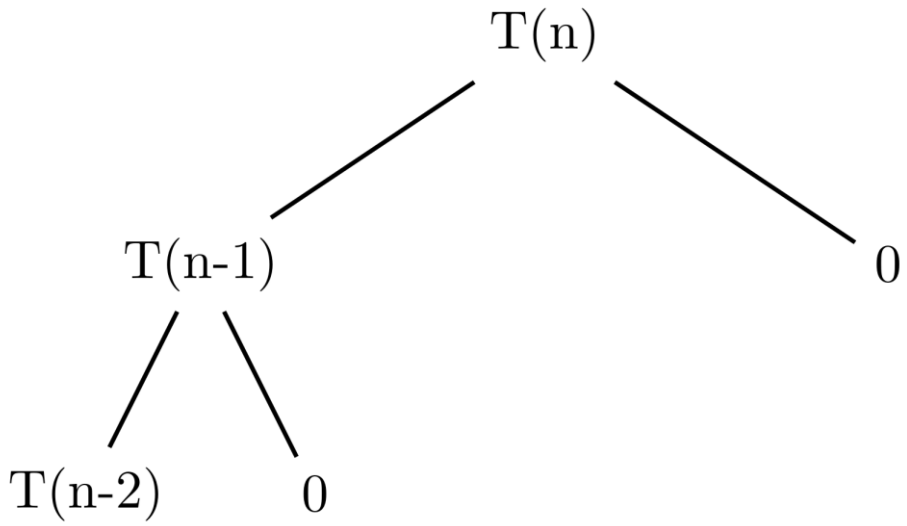
```
1: function MaxDistanceInBinaryTree(root)
2:     if root = NULL then
3:         return 0
4:     end if
5:     DepthOfLeftTree ← MaxDepthOfTree(root → left)
6:     DepthOfRightTree ← MaxDepthOfTree(root → right)
7:     MaxDistance ← DepthOfLeftTree + DepthOfRightTree + 2
8:     if root ≠ NULL then
9:         MaxLeftTreeDistance ← MaxDistanceInBinaryTree(root → left)
10:        MaxRightTreeDistance ← MaxDistanceInBinaryTree(root → right)
11:        if MaxLeftTreeDistance > MaxRightTreeDistance then
12:            MaxDistance ← max(MaxLeftTreeDistance, MaxDistance)
13:        else
14:            MaxDistance ← max(MaxRightTreeDistance, MaxDistance)
15:        end if
16:    end if
17:    return MaxDistance
18: end function
19:
20: function MaxDepthOfTree(root)
21:    if root = NULL then
22:        return -1
23:    else
24:        DepthOfLeftTree ← MaxDepthOfTree(root → left)
25:        DepthOfRightTree ← MaxDepthOfTree(root → right)
26:        if DepthOfLeftTree > DepthOfRightTree then
27:            return DepthOfLeftTree + 1
28:        else
29:            return DepthOfRightTree + 1
30:        end if
31:    end if
32: end function
```

(b) 子问题递归图

Best case:



Worst case:



(c)算法的正确性证明

• 初始化（Initialization）:当根节点为空时，该子树的最大节点距离为 0。

• 保持（Maintenance）:假设第 i 次($i \geq 2$)递归的根节点非空，并且其左右子树递归求解的最大节点距离均正确，保证了本次递归第二种情况求得的节点距离是正确的。故该子树的最大节点间距为第一种情况和第二种情况中较大者，即第 i 次递归的子树最大间距也是正确的。从而可以得到第 i-1 次递归的子树的左右子树节点最大间距是正确的。

• 终止（Termination）:自底向上返回第一次递归调用的结果也是正确的。那么原二叉树的最大节点间距就是其左右子树最大节点间距的大者与左子树最深节点到右子树最深节点距离中的较大者。

（d）复杂度分析

①最好情况下：在每一次递归中，对第一种情况求解的时间复杂度为 O(n),对第二种情况求解的时间复杂度为 2T(n/2)。故时间复杂度的递归式为 T(n)=2T(n/2)+O(n)。由主方法可得，T(n)=O(nlgn)。
②最坏情况下：在每一次递归中，对第一种情况求解的时间复杂度为 O(n),对第二种情况求解的时间复杂度为 T(n-1)。故时间复杂度的递归式为 T(n)=T(n-1)+O(n)。由主方法可得，T(n)=O(n^2)。

3

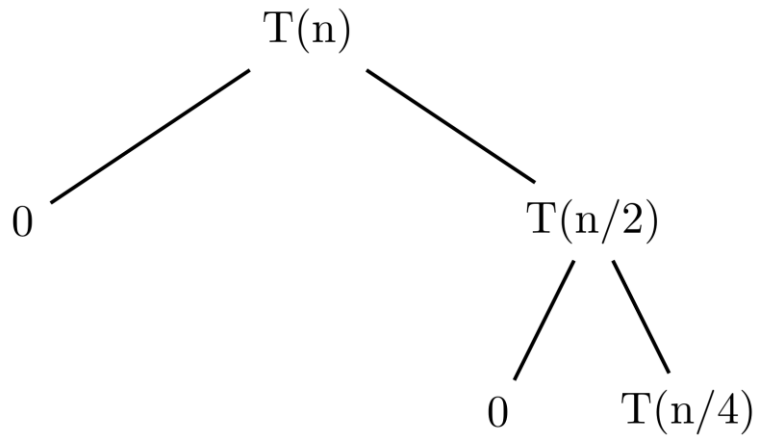
Solution:

(a)由题目可知有一棵完全二叉树，拥有 n 个节点，其中$\mathrm{n} = 2^d - 1$,即该完全二叉树为满二叉树。该二叉树每个节点都有一个实数值，并且每个节点的值均不相同。设计一个算法，对于根节点，只要根节点的值小于其叶子结点的值，即满足局部最小值的条件；对于叶子结点，只要叶子结点的值小于根节点的值，同样也满足局部最小值的条件。如果有任意一个子结点小于根结点，那么就将该子结点作为根结点递归地往下搜索局部极小值，直到达到任意一个叶子结点或者递归过程中两个子结点均大于根结点，就能找到一个局部极小值。

算法 1 求二叉树局部极小值
输入： $root$二叉树根节点
输出： 局部极小值

```
1: function LocalMinimumInBinaryTree(root)
2:     if root → left = NULL and root → right = NULL then
3:         return root → value
4:     end if
5:     if root → value > root → left → value then
6:         return LocalMinimumInBinaryTree(root → left)
7:     else if root → value > root → right → value then
8:         return LocalMinimumInBinaryTree(root → right)
9:     else
10:        return root → value
11:    end if
12: end function
```

(b)子问题递归图



(c) 算法正确性证明

- **初始化（Initialization）**:当根结点小于左右两个子结点或者根结点左右子结点均为空时，根结点即为一个局部极小值点。
- **保持（Maintenance）**:在第 i 次递归中，当有子结点 c 小于根结点时，说明根结点肯定不是局部极小值，而子结点 c 小于根结点，有成为局部极小值的可能性，于是将该子结点 c 作为根结点进行第 i+1 次递归。因为子结点 c 已经小于根结点，故递归时只需考察 c 是否小于两个子结点即可。
- **终止（Termination）**:当递归到根结点的子结点均为空时，说明该结点为叶子结点，并且其值小于其根结点，满足局部极小值条件，返回；当递归到根结点的两个子结点的值均大于根结点的值时，说明该根结点相邻结点值均小于根结点的值，满足局部极小值条件，返回。

（d）复杂度分析

该算法每次递归有两个操作，一个是比较根结点与两个子结点的值，时间复杂度为常数 c，另一个是递归的其中一个子结点，因为题目中给出的是满二叉树，故递归查找的时间复杂度为 T(n/2)。时间复杂度递归式为 T(n)=T(n/2)+c，由主方法可知，T(n)=O(lgn)。