CSC 3100 Assignment 2 Report

Ziyu XIE
121090642

October 29, 2022

1 02 Representaion

This problem asks to turn a number n into 02 representation. The source code is in A2_P1_121090642.py

Since each positive integer n corresponds to a unique binary representation, the first important thing is to represent n in binary form. In order to do further recursion, we can use function **getBiPowLst(n)** to return the binary representation list **powlst** in a power form (e.g. 137 -> [7, 3, 0]).

For each element **b** in this **powlst**, If **b** is not in binary form, we recursively do **getBiPowLst(b)** and use correct rules to put brackets until every digit used to represent the number is in binary form. And we store the result as a string **res**.

For example, getBiPowLst(7) = [2, 1, 0], then we store “2(2(2)+2+2(0))”; getBiPowLst(3) = [1, 0], then we store “2(2+2(0))”; 0 is a 02 representation, so we simply store ”2(0)”; we put them together to get “2(2(2)+2+2(0))+2(2+2(0))+2(0)” and we output it.

Finally, **res** can be the 02 representation of number **n**.

For the function getBiPowLst(n), the time complexity is O(log(n)) because we divide the number n by 2 each time.

For the whole program, for the number n, we can find 2^k > n, then k = O(log(n)), hence the recursion can be T(n) = T(log(n)) + O(log(n)), then the final time complexity is O(log(n))

2 Crossing

This problem asks to get the maximum number of crossings. The source code is in A2_P2_121090642.py

Since the crossing occurs when i < j, A[i] $\leq$ A[j], then the problem can be turned into another problem: given an array, find out the maximum number of pairs such that i < j, A[i] $\leq$ A[j].

We can use merge sort to help solving the problem. Each time when we compare and put the elements from old array A into a new array B, we focus on such pairs, and record it.

After sorting, all the pairs can be recorded, and we output the result.

Since this solution is based on merge sort, and the time complexity of comparing is constant, hence the final time complexity for the whole program is O(log(n)). It is much faster than some sorting like bubble sort or insertion sort.

3 Sierpiński carpet

This problem asks to print out the Sierpiński carpet of size $3^k \times 3^k$. The source code is in A2_P3_121090642.cpp

The core thing to do is to judge an index is whether needed to be removed or not. For each index, we use a function **whetherRemove(idxrow, idxcol, rmsize, rmrow, rmcol)** , where
1. idxrow and idxcol represents the index we are going to judge,
2. rmsize represents now which size of square we want to remove
3. rmrow and rmcol represents the index of the square we want to remove starts

to judge whether the current index is in this remove square, hence need to be removed. If yes, return True; else, if the rmsize > 1, it will also cause 8 smaller square also need to be removed. However, we do not need to see all of them, we use an algorithm to find the smaller square which has the smallest distance to the current index, i.e. we get newRmsize, newRmrow, newRmcol, then we do **whetherRemove(idxrow, idxcol, newRmsize, newRmrow, newRmcol)** recursively unitl newRmsize < 1. If now the index is still not in remove square, we output "#", otherwise " ".

Since the newRmsize = rmsize / 3, we have a recursion T(n) = T(n/3) + O(1), hence the final time complexity of the whole program is O(log(n))

4 Array Maintenance

This problem asks to maintenance an array, which can do the inserting, deleting, and sum calculating operations. The source code is in A2_P4_121090642.py

If we use simple array, the time complexity is commonly O(n), However, we can use tree to reach O(log(n)).

Firstly, we define our tree **Node** which will be like Node(value, left, right, parent, leftElement, rightElement, leftSum, rightSum), where

1. value: is just the value
2. left: the left child of the node, also a node
3. right: the right child of the node, also a node
4. leftElement: the total number of the nodes at left
5. rightElement: the total number of the nodes at right
6. leftSum: the total sum of all the nodes at left
7. rightSum: the total sum of all the nodes at right

Secondly, we need some basic information and operations in **Tree**,

1. h: the height of the tree, expected to O(log(n))
2. precessor: find the precessor of the node
3. successor: find the successor of the node
4. findMinNodeByRoot: find the minimun node from node
5. findMaxNodeByRoot: find the maximun node from node

4.1 Inserting

This operation insert a node with value after the kth element.

We start from root of the tree, say pivot. If tree.leftElement $<$ k, which means that the node should be inserted at right of the pivot, then pivot.rightElement should plus 1 and pivot.rightSum should plus value.

After this, pivot = pivot.right, else pivot = pivot.left. If k $>$ tree.leftElement, we need to say the node has been put after some nodes, hence we get the curK, which should be equal to (curK - k), which will be used for judgement of further right.

Continuously do this operation, we get the pivot == None, which is our target, to insert the node.

Since the each time the node goes deeper, the time complexity is O(h), where h is the height of the tree, namely, O(log(n)).

4.2 Deleting

This operation deletes a node which is the kth element.

We use similar method in insertion to get the node we want to delete, and now the terminating condition is not node == None but node.leftElement = k.

If this node has no child, then everything goes well, we just let its parent's one child to be None.

If this node has only left child, then we join the node and its left child' child. For only right child, we do something similar.

*Note: For all the situations above, we need to back from this node to the root, and deduce the leftElement, leftSum, rightElement, rightSum according to the correct situation.

If this node has both left and right children, we need to find the successor of the node, then renew the value of node, be careful here we still need to need to back from this node to the root, and deduce the leftElement, leftSum, rightElement, rightSum based on the difference between the old node and new one. After this, we delete the successor node using method above or recursively.

Since finding the successor requires O(h), and the backing operation requires O(h), we claim that the time complexity is O(h), namely, O(log(n)).

4.3 Calculating the sum

This operation calculating the sum from the rth element to the rth element.

We use similar method in deleting to get the node of rth and kth element. If we can calculate the sum from 1th to mth element, then do the difference, we can get the result, namely, $Sum(l, r) = Sum(0, r) - Sum(0, l - 1)$

For one node, we first get the leftSum of the node, then we use function findMinNodeByRoot to get the minimun node, the leftSum is obviously the sum from the minimun node to this node.

Then we find the precessor of the minimun node, continuously do the operation until the precessor == None. Finally the result can be calculated.

Since finding the precessor requires O(h), and the worse case of the times of finding the precessor is O(h), we have the time complexity is O(h^2), namely O($\log^2$(n)).