THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC 3100

DATA STRUCTURE

Assignment 2 Report

Author:

Ma Kexuan

Student Number:

ID 120090651

November 3, 2022

Problem 1

1. **Possible solutions:** Almost all of the solutions are based on recursion. A possible solution is to directly use the string operations and distinguish the base case by lots of if statement. Another method is more complex, by using stacks and queues, and this method has been used in my program.

2. **I solve the problem by the following logic:** First, I transfer the input integer to a stack by the algorithm of transforming decimal to binary. Then pop the elements and enqueue them into a queue. This is done in the transfer(int n) function. Then, I check the powers in the queue, whether it only contains 0 and 2, if not, the boolean check(queue power) function will return false. If true, then directly display the whole queue by display(queue q) function. Then, I use recursion to recursively get the smaller queues in order to generate the final 02 representation. Finally, due to the return value of the recursion function, I use another overloaded display(string str) function to directly concatenate the strings, and then output the result.

3. **My solution** is a little bit complex actually, but the programming paradigms in my code are clearer than the one that directly solves the problem by using a lot of if arguments.

Problem 2

1. **Possible solutions:** This crossing problem is equivalent to count the inversion pairs in the array a given by the problem. A brute-force solution is that you can use two for loops to directly count the inversions, but since the input of integer n is as most 10^5, so it is impossible to use this method to pass the Online Judge since its time complexity is $O(n^2)$. Another possible solution is to use the binary indexed tree, or we can use the algorithm similar to merge sort, both of them are in time complexity of $O(\text{nlogn})$. I use the last method in this problem.

2. **I solve the problem by the following logic:** I maintain two large arrays of size 100005, then I do operations directly on these two. Due to the mergesort algorithm, I first recursively divide the array into subarrays, and sort them and merge them one by one, the only difference is, whenever the elements in the left array is larger than the elements in the right array, since the left array is sorted, the elements on the right side of the current element in the left array all construct inversion pairs with the current element in the right array. So we should plus the amount of inversions to the total count in each layer of the recursion. When the program finishes, we can get the inversion pairs, as known as the maximum crossings in the original problem.

3. **My solution** is better than the brute-force algorithm since it has a lower

complexity, when the input n is larger, it'll be a lot more faster than the brute-force one. As for the binary indexed tree, I think this algorithm is easier to implement than the tree one.

Problem 3

1. **Possible solutions:** The easier one to implement but hard to think is the mathematical formulas. You can find the math formula of each input and solve the whole problem by a lot of if statements, since the input k is at most 7. The most commonly thought would be solving it by recursion.

2. **I solve the problem by the following logic:** Because I solve it by recursion, I first define the base case, which is the same as the sample output 1:

```
###
# #
###
```

Then I define the length of the whole picture, after that, I fill the surrounding of the center by empty spaces. Finally, recursively solve the 8 remaining directions from the center, and print the whole picture into the terminal.

3. **My solution** is better than the mathematical formula since it is easy to implement and also easy to think about.

Problem 4

1. **Possible solutions:** A possible solution is to use the binary search tree, or to be specific, the splay tree. Another approach is to use the linked-list form of array. I use the latter one, and I'll precisely explain what this term means.

2. **I solve the problem by the following logic:** This solution is linked-list based, so it must contain nodes. Every node in the linked-list contains a pointer to the next node, a size of how many elements currently stored in the node, a sum of the elements in the array in the node, and an array with size of $2\sqrt{n}+5$. Typically, the array in the node stores in average $\sqrt{n}$ elements. This is because, suppose we have n elements in total, we have s elements in each array in the node. Then when we have to iterate through the linked list to find elements, our time complexity to do the iteration is $O\left(\frac{n}{s}+s\right)$, we can optimize the inner equation when $s=\sqrt{n}$, here is why we should store $\sqrt{n}$ elements in each array in the nodes. Before the whole program runs, there are three

hyperparameters to be determined. The first one is the maxsize of the whole array, which is around 450 since it is the square root of 2×10^5, and also a integer storing the current whole arraylength. The last parameter is the number of operations of the input. When I insert element into the big array, first check whether the position is larger than the current arraylength. If yes, insert it into the end of the array, check if the array should be split. Then, use a variable to record the index of the element in the array in the particular node, and another variable to record the number of elements contained before the array that we want. Then iterate through the whole linked-list to insert the element into the correct position. The delete operation is quite similar. The split operation, when the size of the small array in each node exceeds $2\sqrt{n}$, we split it into two nodes, and also maintain the total sum of elements in each node. For the summation part, we should define two pointers to point at the beginning node and the ending node according to the index. There are three situations, if left equals to right, return the element, if left and right are in the same node, count it normally. When it is in different nodes, we should use two for loops to sum the elements in the beginning and ending nodes, and plus all the sum stored in the intermediate nodes. By doing this, you can reduce the complexity of summation when you repeatly input the operation "3".

3. **My solution** in terms of complexity is not as good as the tree version since it has complexity of $O(\sqrt{n})$, worse than $O(logn)$ in the tree structure, but it is easier to implement since you won't need to move the neighbor node recursively when you are deleting elements in the tree structure.