Report for CSC3100 Assignment 3

Problem 1: The PigCup Final

Language: Java

Observations:

1. There are only 3 letters in the given string, A, B, and O with total length smaller than 20.

Possible Solutions:

1. The key to all algorithms should be iteratively going through the input string representing the result of each game;
2. Counting all A's in the string to calculate the score of A. Then counting all B's and calculating the score of B. Compare the scores and decide who wins at last. O(n)

Algorithm:

1. Use 2 variables noting the scores of the 2 teams;
2. After each game, add 1 point to the variable representing the winning team. If tie, add no point;
3. After the whole match, compare the scores and decide the final winner.

Complexity Analysis:

n: the length of the input string

Only one for-loop is used in the function to "check the result of each single game". Thus, the time complexity is O(n).

The space complexity is O(1) because only 3 variables are used.

Advantages of the Algorithm:

1. The algorithm may be one of the fastest since there's only one for-loop used. The demand of the problem is checking all games' results at least once and the algorithm checks them only once;
2. Meanwhile, its space complexity is not higher than other possible solutions.

Problem 2: Watermelon

Language: Java

Observations:

1. The volume of a watermelon (assumed a sphere) is $V = \frac{4}{3}\pi R^3$ and the half diameter is $r = \sqrt[3]{\frac{3V}{4\pi}}$. During the calculation process, the constant $\frac{4}{3}$ and π can be cancelled because the inputs and outputs are all half diameters while the volume is intermediate, and we don't need its certain value;
2. Only integers are operated before giving the final answer, which can avoid imprecision led by calculation of floating numbers on computer.

Algorithms:

1. Iteratively dual with each group of watermelon data in a line;
2. Calculate the intermediate value of the rind ($\frac{3}{4\pi}V$) by ${r_1}^3 - (r_1 - d)^3$;
3. Calculate the thickness of rind in the bigger watermelon by $r_2 - \sqrt[3]{rind}$, where $rind = {r_1}^3 - (r_1 - d)^3$. Only in this step that calculations which may lead to imprecision is done;
4. The 2 steps above can be combined into one in codes, but I separate them for clear statement.

Complexity Analysis:

T: Number of groups of data (number of lines of data inputted)

The time complexity of the algorithm is O(T) because there's only one for-loop and only 1 arithmetic calculation (or 2) is done for each group of data.

The space complexity of the algorithm is O(1) because there only exist at most 4 variables storing different values.

Advantages of the Algorithm:

1. Low time complexity and space complexity;
2. Avoiding imprecision by calculating floating numbers and power as much as possible, leading to high precision of the result.

Problem 3: Tree Problem

Language: Java

Possible Solutions:

1. For each node, use a depth first search to calculate the distance between each black node to the current one and add them up. Time: O(n^2), Space: O(n)

2. Create a n*n 2-d array storing all node-to-node distance in each cell. Keep on calculating distances from all nodes to one until all cells are filled. After that, sum up the array according to one dimension. Time: $O(n^2)$, Space: $O(n^2)$

Observations:

1. For each node, the summation of distances between it and all other black nodes is the summation of distances to its adjacent nodes time the number of black nodes in each subtree of an adjacent node plus the total distance in the subtree to the adjacent node (root of the subtree). Thus, we can calculate ans_i recursively;
2. Assume we now have the summation of distances from all black nodes to the current node i. Now, we can calculate the same value of an adjacent node of node i (assume node j), and the edge between nodes i and j is e_{ij} (undirected). The summation of the distances from all black nodes to the node j is the distance to the node i plus number of black nodes in the i side of edge e_{ij} multiplied by e_{ij} minus the number of black nodes in the j side of edge e_{ij} multiplied by e_{ij}, i.e.
$$ans_j = ans_i + (total\_black - black[i]) * e_{ij} - black[j] * e_{ij}$$
Therefore, we don't need to do more calculations in recursion as presented in Observation 1 but only to do simple calculations with O(1) time complexity for each ans_j $(j \neq i)$;
3. However, we need to store all data regarding the number of black nodes in the subtree rooted at every node. This can be done in the process of Observation 1.

Algorithms: 2 Depth First Search Method

1. I didn't construct a real tree using self-built nodes. Instead, I used an integer array of size n (n is the very first input integer, meaning the total number of all nodes) to store all color information in binary form and used an array of size n containing n LinkedLists (in pactice, n+1 in total, but the first one at index 0 stayed useless) to store the edge information, because the edges in a tree is only (n-1) and it wastes much using 2-d array, from the ith node to all its neighbors. Elements in a LinkedList was an array of size 2, containing the id of the adjacent node and the distance of the edge between the two nodes. Besides, I created one extra integer array of size n except the *ans* to store the number of black nodes in the subtree rooted at ith node. In practice, the 2 depth first searches all started from node 1 (regard node 1 as the root);
2. According to the Observation 1, I first calculated the target value (summation of distances from all other black nodes to the ith node) using a Depth First Search (DFS) algorithm. By passing the distances from last recursive layer to the next plus the edge newly encountered, I could precisely obtain the distance of the very root node (node 1) to the current node. Then, simply added the value to *ans[1]* (actually the index is 0, which is 1-1, in the following report I will always match the id but not the index);
3. According to the Observation 2 and 3, I also noted down the number of black nodes in the array mentioned in step 1. The black size (number of black nodes in the subtree) of node 1

is the total number of black nodes, which was noted by increment by 1 each time encounter a black node in the first DFS. The black size of subtree rooted at other nodes were noted using the DFS idea. By putting the noting process after each DFS recursive call, I could dive into the very end leaf first and note the black size (whether 0 or 1 since their subtree only contains themselves). Returning to their parent nodes, I could obtain the black size of the parent nodes by summing the black size in their neighbors except their parent nodes, and plus one if the current node is black. Thus, I could recursively obtain the black size of each node and note them in the aforementioned array, which would be used in the following computation;

4. Finally, we have *ans[1]* and the array containing all black sizes (*black*, recall that *black[1]* equal to the total number of black nodes). According to the formula mentioned in Observation 2. For each *ans[j]*, we only need the value of the adjacent node i, which is *ans[i]*. Thus, we can expand the *ans* following the tree structure, no matter DFS, BFS, or other algorithms of expansion through tree. I choose DFS because the recursion was easy to realize in the current setting;
5. Also note that the parent information should also be passed in both DFS recursions to avoid infinite recursive call for each other between 2 adjacent nodes;
6. Besides, creating LinkedList array led to error warning in my VS Code IDE. It provided a quick fix method of adding *"@SuppressWarnings("unchecked")"* before the main function to avoid the error warning. I had no idea whether it would lead to more hidden errors and simply kept it;
7. Also note that all results in the *ans* array were stored in long type and mod is taken each time an addition occurs on it.

Complexity Analysis:

1. The input and construction part are of O(n) time complexity. There are n nodes whose colors need to be noted one by one. There are also n nodes which all have at least 1 neighbor and at most (n-1) neighbors (if all others connect to one node), which need n LinkedList to store. The two processes are parallel. Therefore, the total time complexity of this part is O(n). For space complexity, I stored color information of size n and all edge information of size 2*(n-1) (n-1 edges for both 2 end nodes). Thus, the space complexity is therefore O(n);
2. The main function after the input and construction parts, only consists of 2 DFS functions and the output with time O(n) since we only use one for-loop of n to print out all *ans[i]* and no extra space complexity;
3. For the first DFS function, although both recursion and for-loop are used in the function, the time complexity of it is still O(n) because its function is to go over all nodes in the tree once. Thus, the number of recursive calls sums up to n and each iteration of a for-loop call a recursive call. This is also the property of DFS algorithm. Thus, the time

complexity is O(n), and no extra space complexity is added. It’s generally the same for the second DFS algorithm, which is parallel to the first DFS algorithm;

4. Thus, the total time complexity is still O(n) and no extra space complexity.

Advantage of the Algorithm:

1. Much lower time complexity compared to ordinary methods that follows intuition by extracting formula between adjacent nodes;
2. Relatively low space complexity using LinkedList compared to 2-d array.