Report for Programming Problem 2

Team:
Student ID: 2019216747 Name: João António da Silva Melo
Student ID: 2019216809 Name: Miguel António Gabriel de Almeida Faria

1. Algorithm description

The program starts by reading the input as it's stated in the Problem. As we read a line, we make a new element on the *people* map for the person (father), with the key being the id and the value the information, like the amount paid and, if there are any, the ids associated with the people recruited (sons) by them. After reading all the people, the algorithm starts on the person with the id 0 and it will return the best solution(s) to the problem.

In each node, if it has any sons, there is a recursive call of the function *findMembersAndAmount*. At the end of each recursive call, it is returned a struct *iterations* containing:

- a bi-dimensional vector with the one/two best options from the current call; each vector is constituted by 2 integers, the first one has the number of nodes chosen, and the second has the sum of the values of these.
- an integer used to help with the process of choosing the best options for the next call.

When the function is called to a node that has no sons (leaf), the struct *iterations* is returned with only one vector containing the number 1 (since we only chose this node) and the current node's value. The *auxCase* variable is returned with the value 0 (being a special case in the *cases* function, that will be explained further in the report).

In the case of the function being called in a node that has one or more sons, the value returned by each one of them is used by the father in the function *cases*, where the value obtained will contain the best option with or without him. These values are stored in an unordered map *temp* that has two possible keys: "1" if the *auxCase* received is 1 and "2" if the *auxCase* received is 0 or 2. This information is important so that it is easier to identify the solutions that contain the current node.

The function *cases* has a really important role in this algorithm, since it makes the process of choosing the best options with or without the father less complex. It receives the current node and the sons *iterations* obtained recursively and returns a bi-dimensional vector containing the best options. There are three possible cases:

- if *auxCase* is equal to 0, it's inserted in the first position the integer 1 and the value of the current node and in the second one the vector obtained previously.
- if *auxCase* is equal to 1, it's inserted in the first position the first vector of the result obtained previously (that contains the son) and in the second position a vector adding 1 and the value of the father to the best option of the son.
- if *auxCase* is equal to 2, it's inserted in the first position a vector adding 1 and the value of the father to the first option of the son and in the second position the second vector of the result obtained previously (that contains the son).

When it reaches the last son, all the results are analyzed so we can get the best options with and without the father to return them. There are three situations for the return values of the *findMembersAndAmount* function:

- if the solution with the father has a lower number of nodes used, it's the only one that needs to be returned.
- if both have the same number of nodes, it is when *auxCase* is equal to 2 (or 0), so the option with the father needs to be returned in the first position.
- otherwise, when *auxCase* is equal to 1, the option with the father needs to be returned in the second position.

2. Data structures

- *iterations* – it's a struct used to return the values from the *findMembersAndAmount* function; it contains a bi-dimensional *vector last* for the options and an integer *auxCase* to help on *cases* function.
- *node* – it's a struct used to store the information about a person, like the members recruited ids (*vector* of integers) and the value of the money paid (*integer*).
- *people* – it's an unordered map used to store all people, that contains the number (id) of a person as key (*integer)* and the person information as value (*node*).

3. Correctness

In the submissions in Mooshak, our best result was 180/200 and the error message was Runtime Error, meaning that our approach, in a certain case(s), has an execution error. Although we did a lot of tests, the program never ended with any type of error and with the correct result (since we compared them with groups that had achieved the maximum score). Therefore, we couldn't find the error, but we suspect that it can be an unexpected result (because we had a previous error like this with the same error message).

Despite not achieving the maximum score, we still think our algorithm is correct. By having the bottom-up approach, we know that, in each node, all the options are being considered and the results obtained in each iteration contain the best solution, one with the node and the other without.

Relatively to the optimal substructure property, using our algorithm, we can take the following conclusions:

1) S contains the two optimal solutions for the node *x*

   ||

   v

2) S' contains the two optimal solutions for the father node of *x*:
   a. Both son's optimal solutions, adding the father to the first one, if it's an even level
   b. Both son's optimal solutions, adding the father to the second one, if it's an odd level

4. Algorithm Analysis

In the resolution of this problem, we used a bottom-up approach. Although we start the program at the top (the person with id 0), we use the values obtained on the lower level (sons). By starting at the top, having *n* people and knowing that each person has a maximum of 10 people recruited by them, in each recursive call, the solution obtained, in the worst case, goes through a maximum of 10 results; therefore, the time complexity of this approach is T(n) = 10*n => O(N).

When it comes to spatial complexity, our program uses a map to store the people, which allocates an integer and an element of the struct node for each person; therefore, in the worst case, the struct node has a size of 11, so each element has a size of 12. This map has *n* number of elements, with *n* being the number of people. To store the final result, we also use an *iterations* struct that allocates 2*2+1 = 5 of space, and an integer array with size 2 to store the best option. Consequently, the spatial complexity is S(n) = 12*n + 5 + 2 => O(N).

5. References

[1] https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/