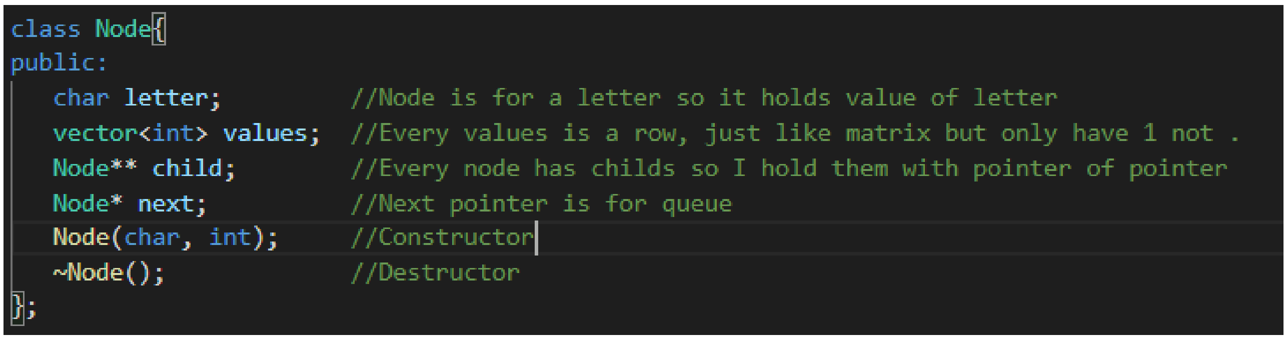
1. Report your problem formulation:

a) Describe node and assignment representations in detail?

Node representation:

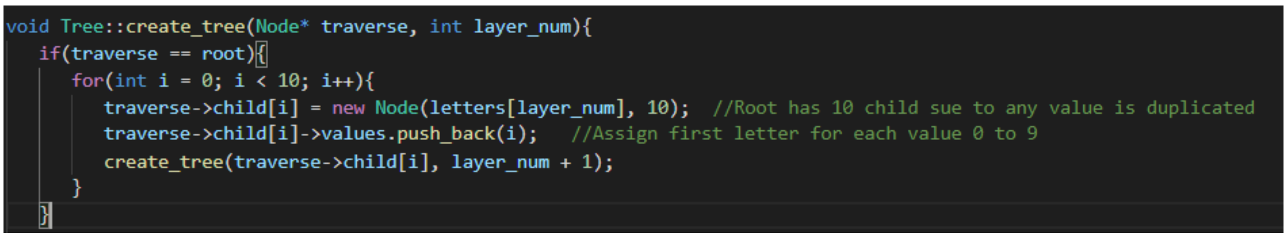


```
class Node{
public:
    char letter;          //Node is for a letter so it holds value of letter
    vector<int> values;   //Every values is a row, just like matrix but only have 1 not .
    Node** child;         //Every node has childs so I hold them with pointer of pointer
    Node* next;           //Next pointer is for queue
    Node(char, int);      //Constructor
    ~Node();              //Destructor
};
```

Assignment representations:

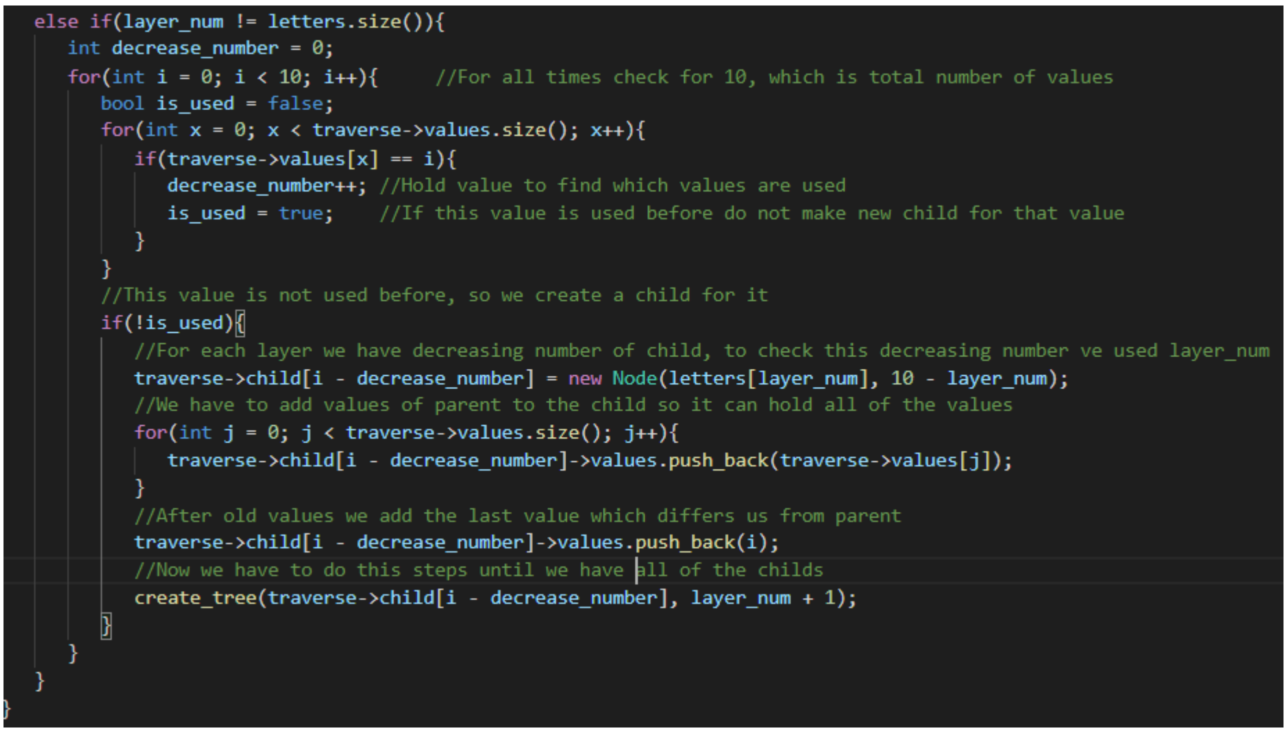
void create_tree()

At first, the partial tree must be created. To create the tree root node must be have 10 child and for each child of Node, the number of child must be decreased by one (Actually is is because of the partial tree wants too much ram and due to that SEND MORE MONEY does not work on SSH.) for each layer.



```
void Tree::create_tree(Node* traverse, int layer_num){
    if(traverse == root){
        for(int i = 0; i < 10; i++){
            traverse->child[i] = new Node(letters[layer_num], 10);  //Root has 10 child sue to any value is duplicated
            traverse->child[i]->values.push_back(i);   //Assign first letter for each value 0 to 9
            create_tree(traverse->child[i], layer_num + 1);
        }
    }
```

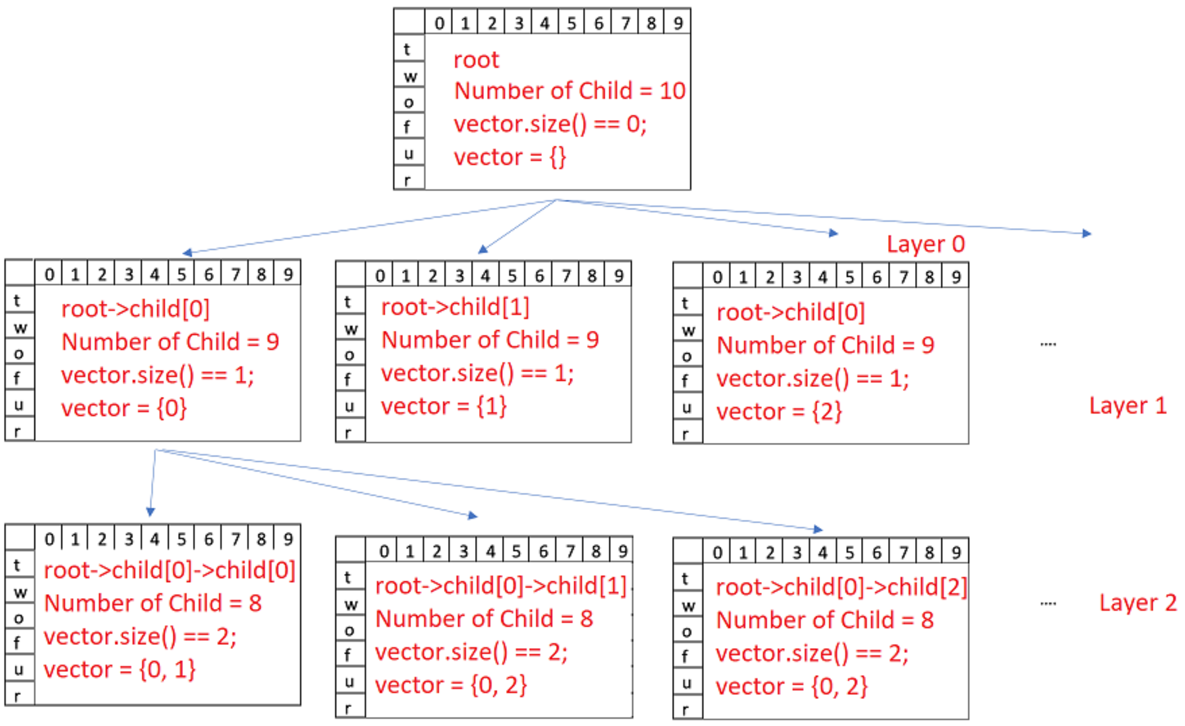
As can be seen from this code, at first we create 10 different child that holds a different value for first letter(layer 1). For example if the first letter that we read from file or terminal is T, all of the child's has different value for T such as 0, 1 to 9.



```
    else if(layer_num != letters.size()){
        int decrease_number = 0;
        for(int i = 0; i < 10; i++){     //For all times check for 10, which is total number of values
            bool is_used = false;
            for(int x = 0; x < traverse->values.size(); x++){
                if(traverse->values[x] == i){
                    decrease_number++; //Hold value to find which values are used
                    is_used = true;    //If this value is used before do not make new child for that value
                }
            }
            //This value is not used before, so we create a child for it
            if(!is_used){
                //For each layer we have decreasing number of child, to check this decreasing number ve used layer_num
                traverse->child[i - decrease_number] = new Node(letters[layer_num], 10 - layer_num);
                //We have to add values of parent to the child so it can hold all of the values
                for(int j = 0; j < traverse->values.size(); j++){
                    traverse->child[i - decrease_number]->values.push_back(traverse->values[j]);
                }
                //After old values we add the last value which differs us from parent
                traverse->child[i - decrease_number]->values.push_back(i);
                //Now we have to do this steps until we have all of the childs
                create_tree(traverse->child[i - decrease_number], layer_num + 1);
            }
        }
    }
}
```

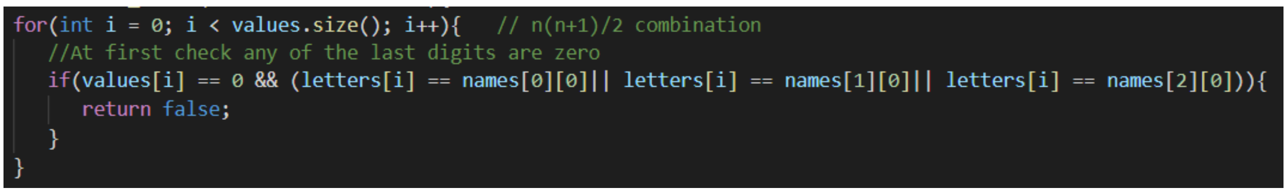
As can be seen from the rest of the create_tree() function, the function works until we have all the childs. For each node we have different number of childs as I mentioned before. And also for all of the layer depth, we add one new value so total number of the child must be equal to 10 – layer_number. And because of it we create that much child. Now the main problem is how to decide which value is assigned. To solve that we add for loop to find is that value used, if it is not we continue to add. For each child we find the value is not used, at first we push all of the values in its parent and then add its last value. With that vector size of the child is equal to vector size of parent + 1. Then owing to layer_num != letter.size(), we continue to create child. If we reach the total layer size recursion stopped itself.

And with that operations we get a tree like this



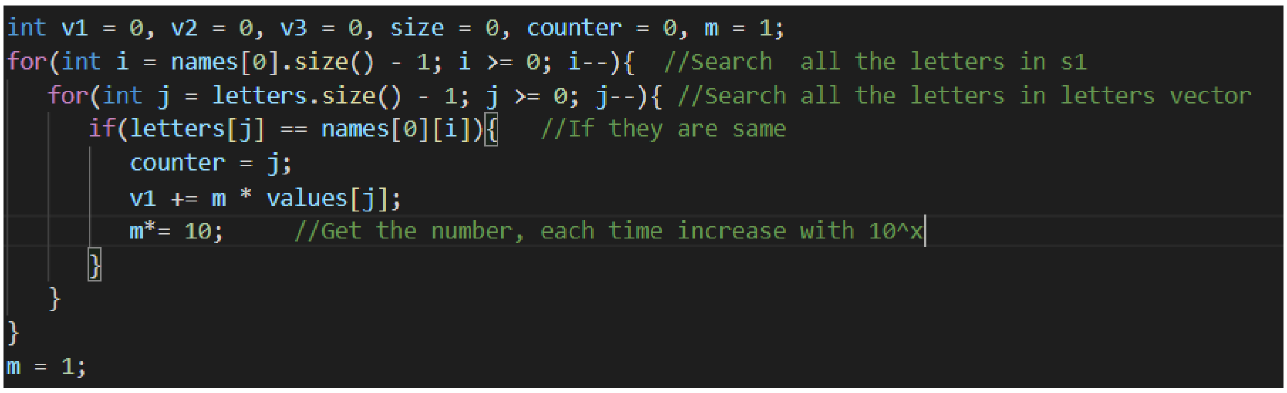
bool is_valid();

At first we check that if any of the first letters assigned by 0. For is from 0 to values.size() due to if it is bigger than values.size() we will try to reach outside of the given area and the code will give segmentation fault.



```
for(int i = 0; i < values.size(); i++){   // n(n+1)/2 combination
    //At first check any of the last digits are zero
    if(values[i] == 0 && (letters[i] == names[0][0]|| letters[i] == names[1][0]|| letters[i] == names[2][0])){
        return false;
    }
}
```

Then we have to check that is the total sum correct. We are not checking that if there is any two number which are assinged same value caues of we check it while implementing tree(SEND MORE MONEY error).



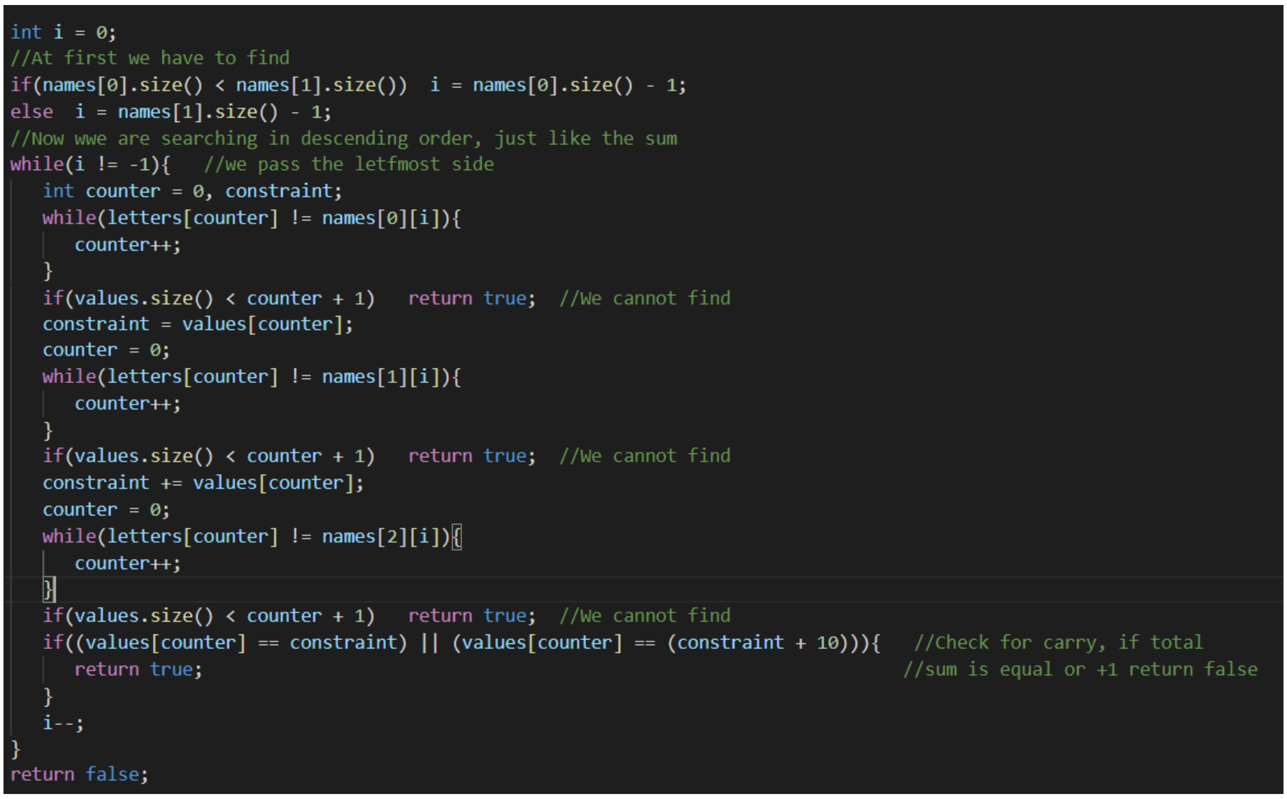
```
int v1 = 0, v2 = 0, v3 = 0, size = 0, counter = 0, m = 1;
for(int i = names[0].size() - 1; i >= 0; i--){  //Search  all the letters in s1
   for(int j = letters.size() - 1; j >= 0; j--){ //Search all the letters in letters vector
      if(letters[j] == names[0][i]){   //If they are same
         counter = j;
         v1 += m * values[j];
         m*= 10;     //Get the number, each time increase with 10^x
      }
   }
}
m = 1;
```

As can be seen in there, we get the real value of first word. Just like it, do the same operations for other two words and compare the sum; if v3 == v1 + v2 we know that the assignment on letters are valid so return true. Else return false.

bool summation_rules();

Main purpose of this function is decrease the number of visited trees. To do that we have 2 steps. First step is just like the is_valid() function, we are checking that if any of the first letters in words are equal to zero.

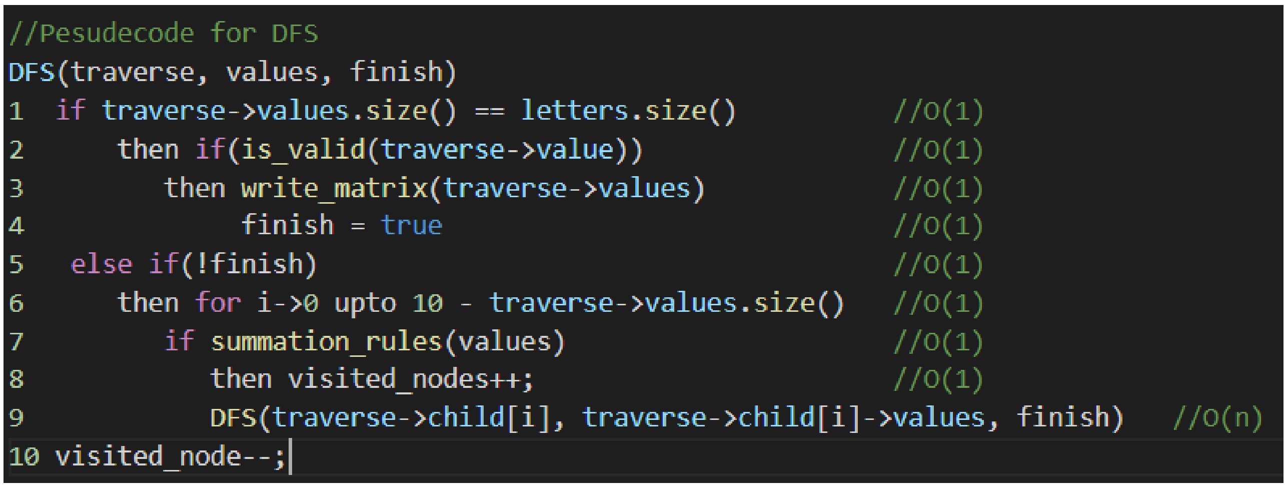
After that we have to check other constraint, carry. Just like sum two integers, we are starting the operation from right side and go th the left and that means i must be decreased until it becomes -1.



```
int i = 0;
//At first we have to find
if(names[0].size() < names[1].size())  i = names[0].size() - 1;
else  i = names[1].size() - 1;
//Now wwe are searching in descending order, just like the sum
while(i != -1){   //we pass the letfmost side
   int counter = 0, constraint;
   while(letters[counter] != names[0][i]){
      counter++;
   }
   if(values.size() < counter + 1)   return true;  //We cannot find
   constraint = values[counter];
   counter = 0;
   while(letters[counter] != names[1][i]){
      counter++;
   }
   if(values.size() < counter + 1)   return true;  //We cannot find
   constraint += values[counter];
   counter = 0;
   while(letters[counter] != names[2][i]){
      counter++;
   }
   if(values.size() < counter + 1)   return true;  //We cannot find
   if((values[counter] == constraint) || (values[counter] == (constraint + 10))){   //Check for carry, if total
      return true;                                                                  //sum is equal or +1 return false
   }
   i--;
}
return false;
```

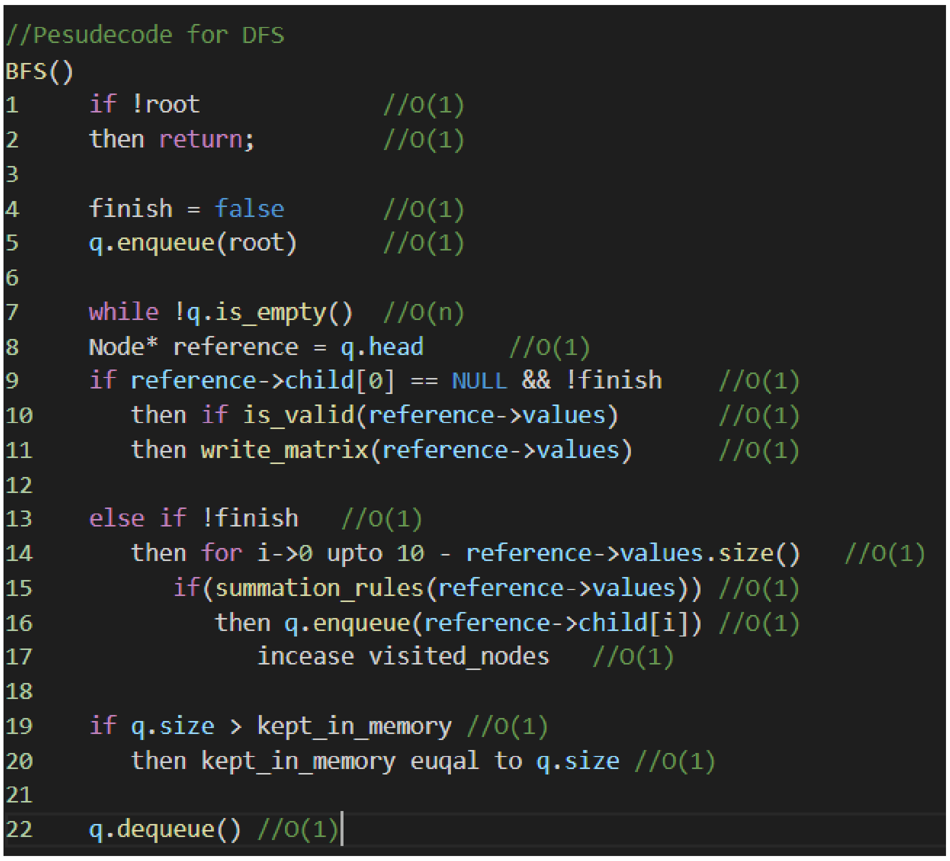
b) Write your pseudo-code. (I assume that is for DFS - BFS due to there is not any explanation)

DFS Pseodecode



```
//Pesudecode for DFS
DFS(traverse, values, finish)
 if traverse->values.size() == letters.size()          //O(1)
     then if(is_valid(traverse->value))                //O(1)
        then write_matrix(traverse->values)            //O(1)
             finish = true                             //O(1)
  else if(!finish)                                     //O(1)
     then for i->0 upto 10 - traverse->values.size()   //O(1)
        if summation_rules(values)                     //O(1)
           then visited_nodes++;                       //O(1)
           DFS(traverse->child[i], traverse->child[i]->values, finish)   //O(n)
visited_node--;
```

BFS Pseudecode



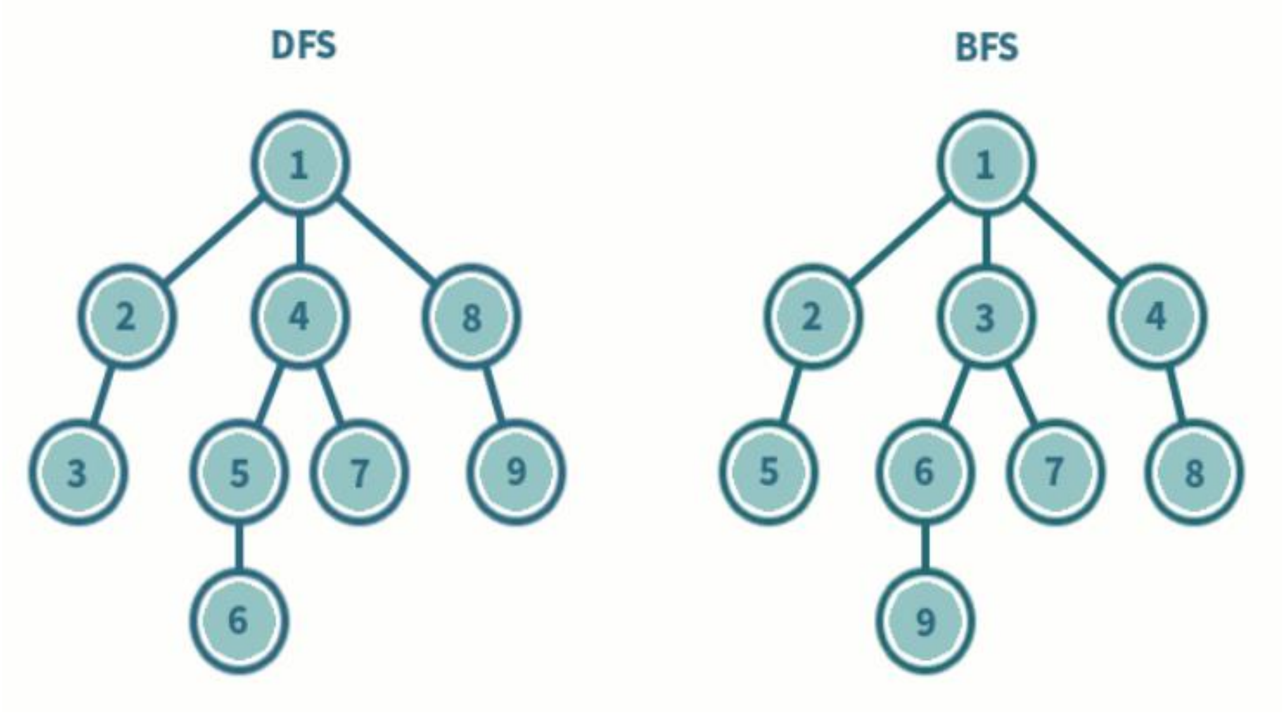
```
//Pesudecode for DFS
BFS()
    if !root              //O(1)
    then return;          //O(1)

    finish = false        //O(1)
    q.enqueue(root)       //O(1)

    while !q.is_empty()  //O(n)
    Node* reference = q.head       //O(1)
    if reference->child[0] == NULL && !finish    //O(1)
       then if is_valid(reference->values)       //O(1)
       then write_matrix(reference->values)      //O(1)

    else if !finish   //O(1)
       then for i->0 upto 10 - reference->values.size()   //O(1)
          if(summation_rules(reference->values)) //O(1)
             then q.enqueue(reference->child[i]) //O(1)
                incease visited_nodes   //O(1)

    if q.size > kept_in_memory //O(1)
       then kept_in_memory euqal to q.size //O(1)

    q.dequeue() //O(1)
```

c) Show the complexity of your algorithm on the pseudo-code.

Explanation: As we learn in lecture, time complexity of DFS and BFS are O(m + n). m is the edge number and n is the node number. In our case, except for the root node all nodes are connected by two edge, parent and child. Due to that total edge number is equal to n – 1(two edge connects two number). And with that, time complexity is equal to O(n + n - 1) = O(n) in both search algoirthms.

2) Analyze and compare the algorithm results in terms of:

a) the number of visited nodes:



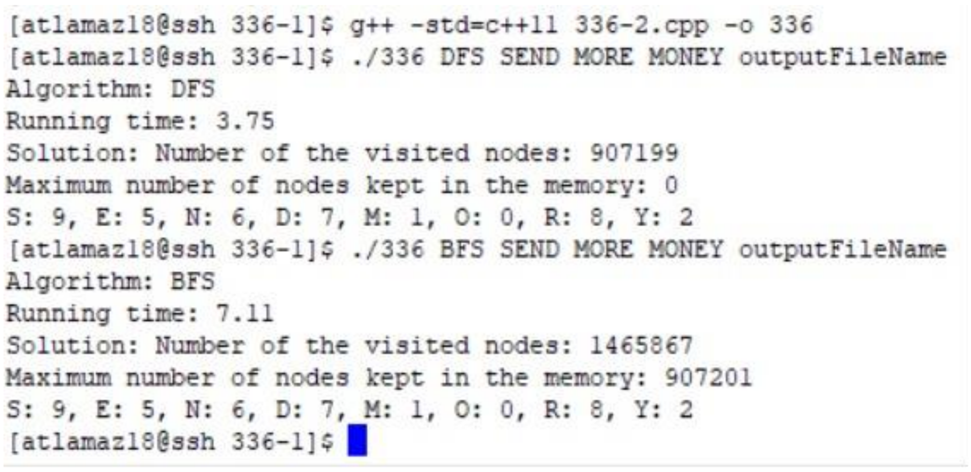
As it can be seen from the DFS-BFS example, leaf nodes searched before in DFS. In our cryptarithmetic example, all of the values are determined in the last layer which is leaf. Due to that to find a random node in tree DFS must traverse less nodes than BFS. And also it is not possible for BFS to traverse less nodes than DFS for our example. In the best scenario they must have traverse same number of node.

b) The maximum number of nodes kept in the memory:

For BFS, before passing the next layer the algorithm must add all of the values to the queue in current layer. And for our tree number of Node's are depends on the size of letters and constraints.
For example if we get TWO TWO FOUR from input we have 6 different letters such as t, w, o, f, u, r. If we does not have any constraint the maximum node kept in memory must be equal to 10*9*8*7*6 which is also equal to total number of Node's in fifth layer but with constraints it is less.

For DFS, if I used stack just like I used queue in BFS we obtain different results. But instead of this I used recursive method to write DFS. Due to that each time the traverse is not in the last layer t goes to its childs. And due to that I used difference method to calculate kept in memory number.

c) The running time: As I mentioned in q2 part a, BFS have to traverse more nodes than DFS, and because of that running time of BFS is higher than DFS. Theoritically, this condition satisfied for all the time.



```
[atlamaz18@ssh 336-1]$ g++ -std=c++11 336-2.cpp -o 336
[atlamaz18@ssh 336-1]$ ./336 DFS SEND MORE MONEY outputFileName
Algorithm: DFS
Running time: 3.75
Solution: Number of the visited nodes: 907199
Maximum number of nodes kept in the memory: 0
S: 9, E: 5, N: 6, D: 7, M: 1, O: 0, R: 8, Y: 2
[atlamaz18@ssh 336-1]$ ./336 BFS SEND MORE MONEY outputFileName
Algorithm: BFS
Running time: 7.11
Solution: Number of the visited nodes: 1465867
Maximum number of nodes kept in the memory: 907201
S: 9, E: 5, N: 6, D: 7, M: 1, O: 0, R: 8, Y: 2
[atlamaz18@ssh 336-1]$
```

Other values corrected, but time difference can be seen

3) Discuss why we should maintain a list of discovered nodes? How this affects the outcome of the algorithms?

If we have graph instead of tree, there might be some edges which return back. In that situation, we might return back and traverse some of the nodes more than once. To prevent this, we should maintain a list of discovered nodes. In our homework we used tree, not graph so I did not have a discovered list.

Burak Atlamaz
150180067