The Celebrity Problem

Another classical problem.

In a party of N people, only one person is known to everyone. Such a person may be present in the party, if yes, (s)he doesn't know anyone in the party. We can only ask questions like "does A know B? ". Find the stranger (celebrity) in minimum number of questions.

We can describe the problem input as an array of numbers/characters representing persons in the party. We also have a hypothetical function HaveAcquaintance(A, B) which returns true if A knows B, false otherwise. How can we solve the problem, try yourself first.

We measure the complexity in terms of calls made to HaveAcquaintance().

Graph:

We can model the solution using graphs. Initialize indegree and outdegree of every vertex as 0. If A knows B, draw a directed edge from A to B, increase indegree of B and outdegree of A by 1. Construct all possible edges of the graph for every possible pair [i, j]. We have N_{C_2} pairs. **If celebrity is present in the party, we will have one sink node in the graph with outdegree of zero, and indegree of N-1. We can find the sink node in O(N) time, but the overall complexity is O(N^2) as we need to construct the graph first.**

Recursion:

We can decompose the problem into combination of smaller instances. Say, if we know celebrity of N-1 persons, can we extend the solution to N? We have two possibilities, **Celebrity(N-1) know N**, or **N already knew Celebrity(N-1)**. **In the former case, N will be celebrity if N doesn't know anyone else. In the later case we need to check that Celebrity(N-1) doesn't know N.**

CASE 1: **Celebrity(N-1) know N --- N will be celebrity if N doesn't know anyone else**
CASE 2: **N already knew Celebrity(N-1) --- we need to check that Celebrity(N-1) doesn't know N**

Solve the problem of smaller instance during divide step. On the way back, we may find a celebrity from the smaller instance. **During combine stage, check whether the returned celebrity is known to everyone and he doesn't know anyone.** The recurrence of the recursive decomposition is,

T(N) = T(N-1) + O(N)

T(N) = O(N^2). You may try writing pseudo code to check your recursion skills.

Using Stack:

The graph construction takes O(N^2) time, it is similar to brute force search. In case of recursion, we reduce the problem instance by not more than one, and also combine step may examine M-1 persons (M – instance size).

We have the following observation based on elimination technique (Refer Polya's *How to Solve It* book).

- If A knows B, then A can't be celebrity. Discard A, and B may be celebrity.
- If A doesn't know B, then B can't be celebrity. Discard B, and A may be celebrity.
- Repeat above two steps until we left with only one person.
- Ensure the remained person is celebrity. (Why do we need this step?)

We can use stack to verify celebrity.
- Push all the celebrities into a stack.
- Pop off top two persons from the stack, discard one person based on return status of HaveAcquaintance(A, B).
- Push the remained person onto stack.
- Repeat step 2 and 3 until only one person remains in the stack.
- Check the remained person in stack doesn't have acquaintance with anyone else.

We will discard N elements utmost (Why?).

```
#include <iostream>
#include <list>
using namespace std;

// Max # of persons in the party
#define N 8

// Celebrities identified with numbers from 0 through size-1
int size = 4;
// Person with 2 is celebrity
bool MATRIX[N][N] = {{0, 0, 1, 0}, {0, 0, 1, 0}, {0, 0, 0, 0}, {0, 0, 1, 0}};

bool HaveAcquiantance(int a, int b) { return MATRIX[a][b]; }

int CelebrityUsingStack(int size)
{
   // Handle trivial case of size = 2

   list<int> stack; // Careful about naming
```

```
int i;
int C; // Celebrity

i = 0;
while( i < size )
{
    stack.push_back(i);
    i = i + 1;
}

int A = stack.back();
stack.pop_back();

int B = stack.back();
stack.pop_back();

while( stack.size() != 1 )
{
    if( HaveAcquiantance(A, B) )
    {
        A = stack.back();
        stack.pop_back();
    }
    else
    {
        B = stack.back();
        stack.pop_back();
    }
}

// Potential candidate?
C = stack.back();
stack.pop_back();

// Last candidate was not examined, it leads one excess comparison (optimise)
if( HaveAcquiantance(C, B) )
    C = B;

if( HaveAcquiantance(C, A) )
    C = A;

// I know these are redundant,
// we can simply check i against C
i = 0;
```

```
    while( i < size )
    {
        if( C != i )
        stack.push_back(i);
        i = i + 1;
    }

    while( !stack.empty() )
    {
        i = stack.back();
        stack.pop_back();

        // C must not know i
        if( HaveAcquiantance(C, i) )
            return -1;

        // i must know C
        if( !HaveAcquiantance(i, C) )
            return -1;
    }

    return C;
}

int main()
{
    int id = CelebrityUsingStack(size);
    id == -1 ? cout << "No celebrity" : cout << "Celebrity ID " << id;
    return 0;
}
```

Output: Celebrity ID 2

Complexity O(N).

Try the above code for successful MATRIX {{0, 0, 0, 1}, {0, 0, 0, 1}, {0, 0, 0, 1}, {0, 0, 0, 1}}.

A Note:

You may think that why do we need a new graph as we already have access to input matrix. Note that the matrix MATRIX used to help the hypothetical function HaveAcquaintance(A, B), but never accessed via usual notation MATRIX[i, j]. We have access to the input only through the function HaveAcquiantance(A, B). Matrix is just a way to code the solution. We can assume the cost of hypothetical function as O(1).

If still not clear, assume that the function HaveAcquiantance() accessing information stored in a set of linked lists arranged in levels. List node will have next and nextLevel pointers. Every level will have N nodes i.e. an N element list, next points to next node in the current level list and the nextLevel pointer in last node of every list will point to head of next level list. For example the linked list representation of above matrix looks like,

```
L0 0->0->1->0
            |
L1          0->0->1->0
                     |
L2                   0->0->1->0
                              |
L3                            0->0->1->0
```

The function HaveAcquanintance(i, j) will search in the list for j-th node in the i-th level. Out goal is to minimize calls to HaveAcquanintance function.

Exercises:

1. Write code to find celebrity. Don't use any data structures like graphs, stack, etc… you have access to N and HaveAcquaintance(int, int) only.

2. Implement the algorithm using Queues. What is your observation? Compare your solution with Finding Maximum and Minimum in an array and Tournament Tree. What are minimum number of comparisons do we need (optimal number of calls to HaveAcquaintance())?

— Venki. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.