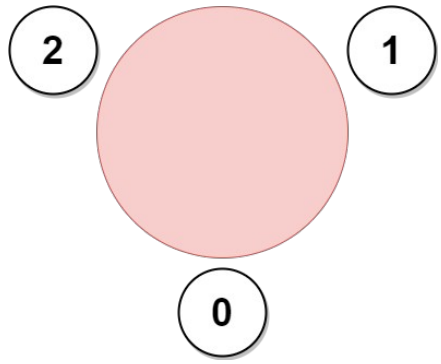
2127. Maximum Employees to Be Invited to a Meeting

A company is organizing a meeting and has a list of `n` employees, waiting to be invited. They have arranged for a large **circular** table, capable of seating **any number** of employees.

The employees are numbered from `0` to `n - 1`. Each employee has a **favorite** person and they will attend the meeting **only if** they can sit next to their favorite person at the table. The favorite person of an employee is **not** themself.

Given a **0-indexed** integer array `favorite`, where `favorite[i]` denotes the favorite person of the `i`th employee, return *the **maximum number of employees** that can be invited to the meeting*.

Example 1:

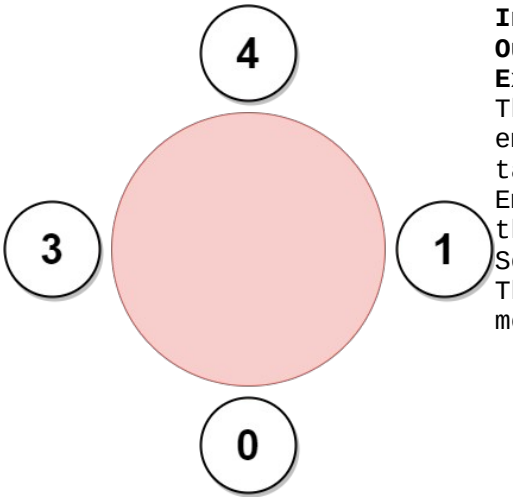



```
Input: favorite = [2,2,1,2]
Output: 3
Explanation:
The above figure shows how the company can invite employees 0, 1, and 2, and seat them at the round table.
All employees cannot be invited because employee 2 cannot sit beside employees 0, 1, and 3, simultaneously.
Note that the company can also invite employees 1, 2, and 3, and give them their desired seats.
The maximum number of employees that can be invited to the meeting is 3.
```

Example 2:

```
Input: favorite = [1,2,0]
Output: 3
Explanation:
Each employee is the favorite person of at least one other employee, and the only way the company can invite them is if they invite every employee.
The seating arrangement will be the same as that in the figure given in example 1:
- Employee 0 will sit between employees 2 and 1.
- Employee 1 will sit between employees 0 and 2.
- Employee 2 will sit between employees 1 and 0.
The maximum number of employees that can be invited to the meeting is 3.
```

Example 3:



Input: favorite = [3,0,1,4,1]
Output: 4
Explanation:
The above figure shows how the company will invite employees 0, 1, 3, and 4, and seat them at the round table.
Employee 2 cannot be invited because the two spots next to their favorite employee 1 are taken.
So the company leaves them out of the meeting.
The maximum number of employees that can be invited to the meeting is 4.

Constraints:

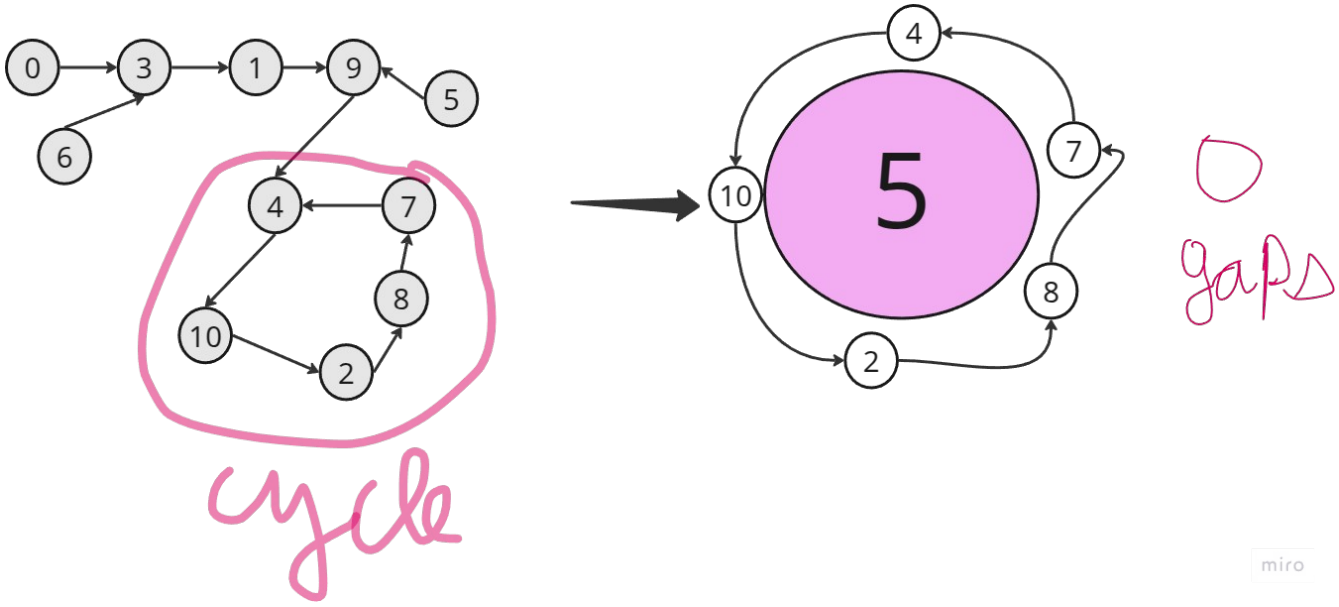
- `n == favorite.length`
- `2 <= n <=` 10^5
- `0 <= favorite[i] <= n - 1`
- `favorite[i] != i`

Observation

The problem can de reduced to a Directed Cyclic Graph.

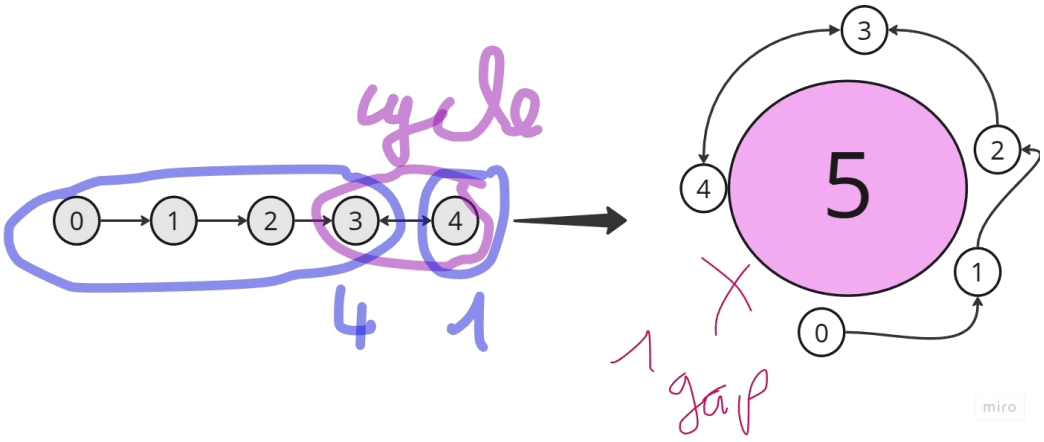
favorite:										
3	9	8	1	10	9	3	4	7	4	2
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*

Any group of **employees in a cycle can sit next to each other**.

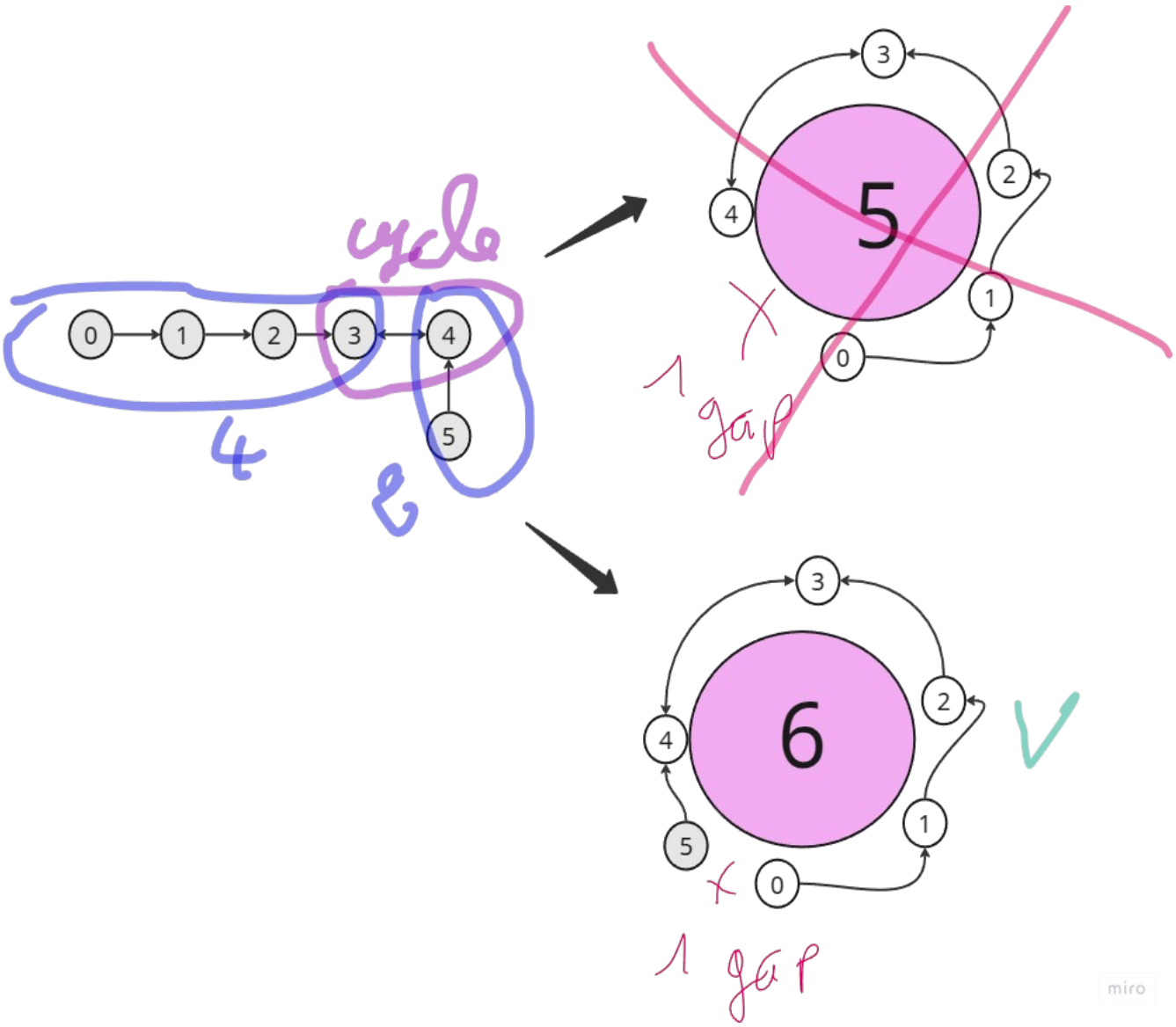


Let's take more examples to break down this problem:

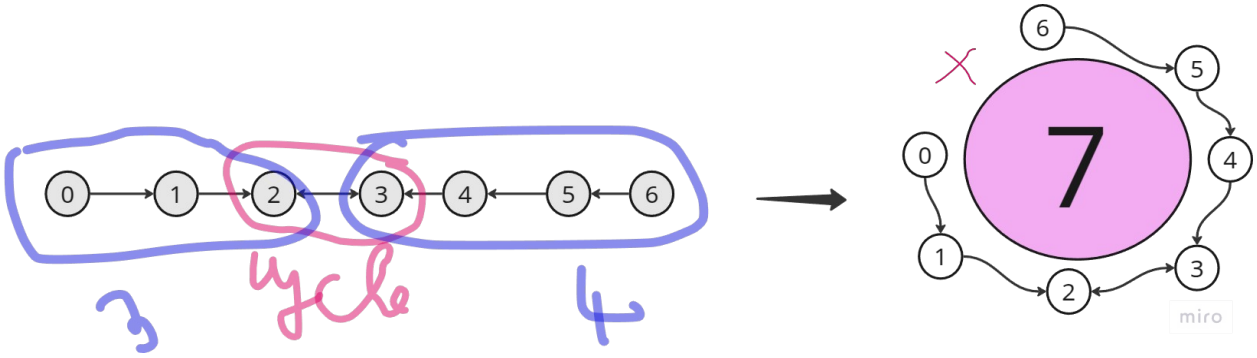
favorite:				
1	2	3	4	3
0	*1*	*2*	*3*	*4*



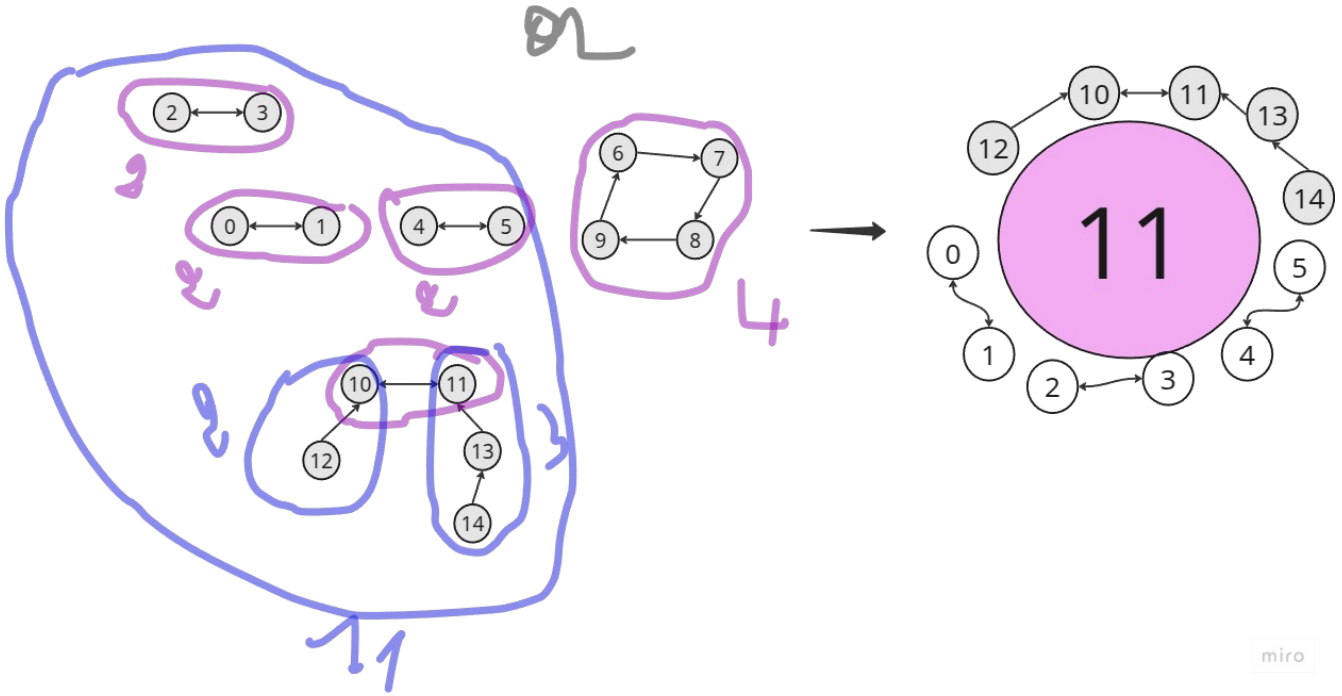
favorite:					
1	2	3	4	3	4
0	*1*	*2*	*3*	*4*	*5*



favorite:					
1	2	3	4	3	4
0	*1*	*2*	*3*	*4*	*5*

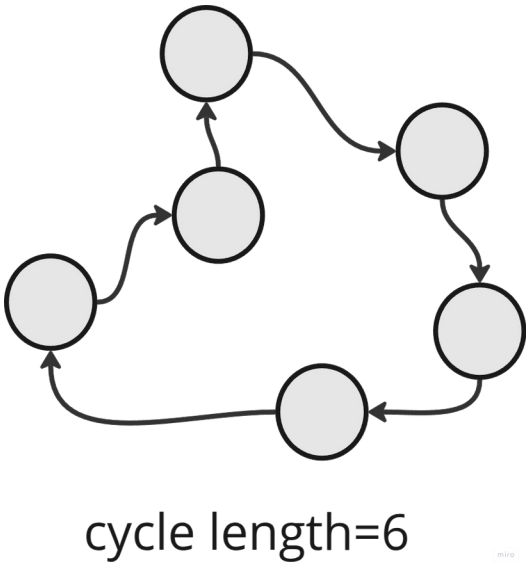


favorite:														
1	0	3	2	5	4	7	8	9	6	11	10	10	11	13
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*

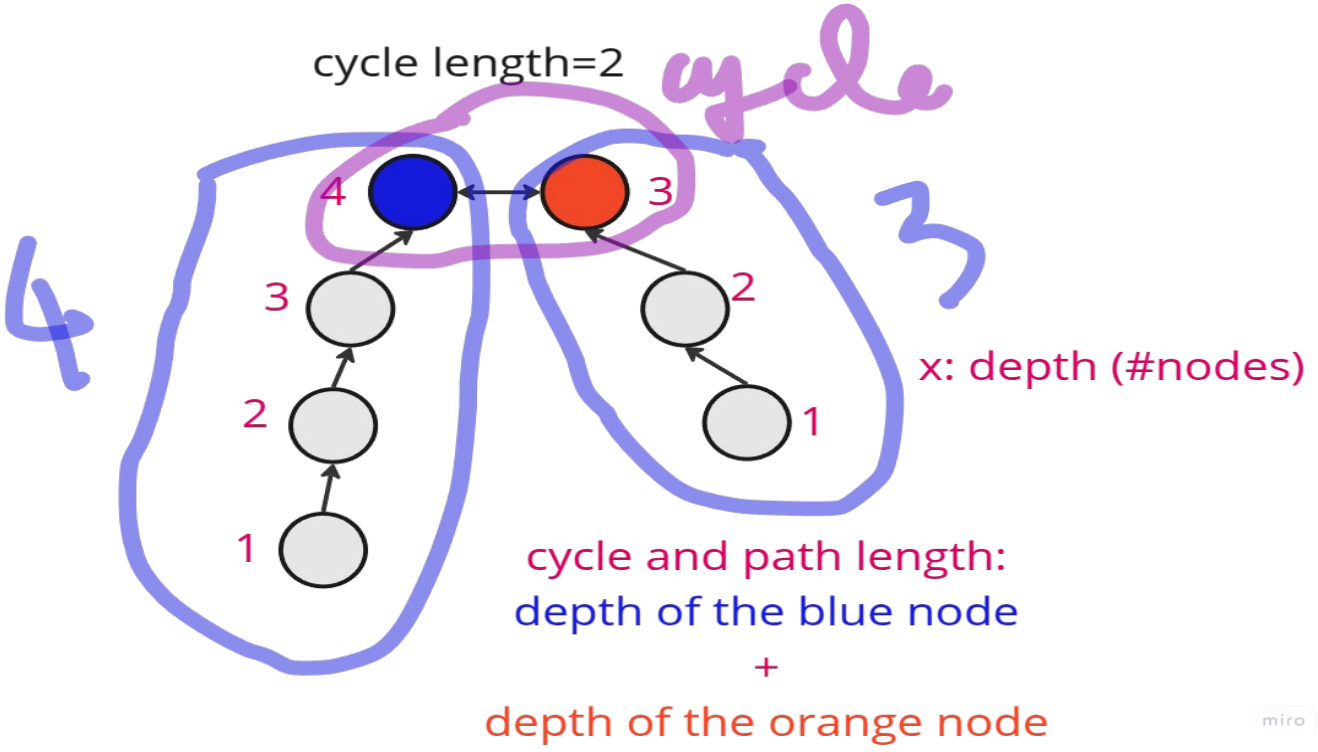


So, determine the cycles is the main task to do in order to be able to solve this problem. After that, we need to compute the length of each cycle. **We mean by length, here, the number of nodes**.

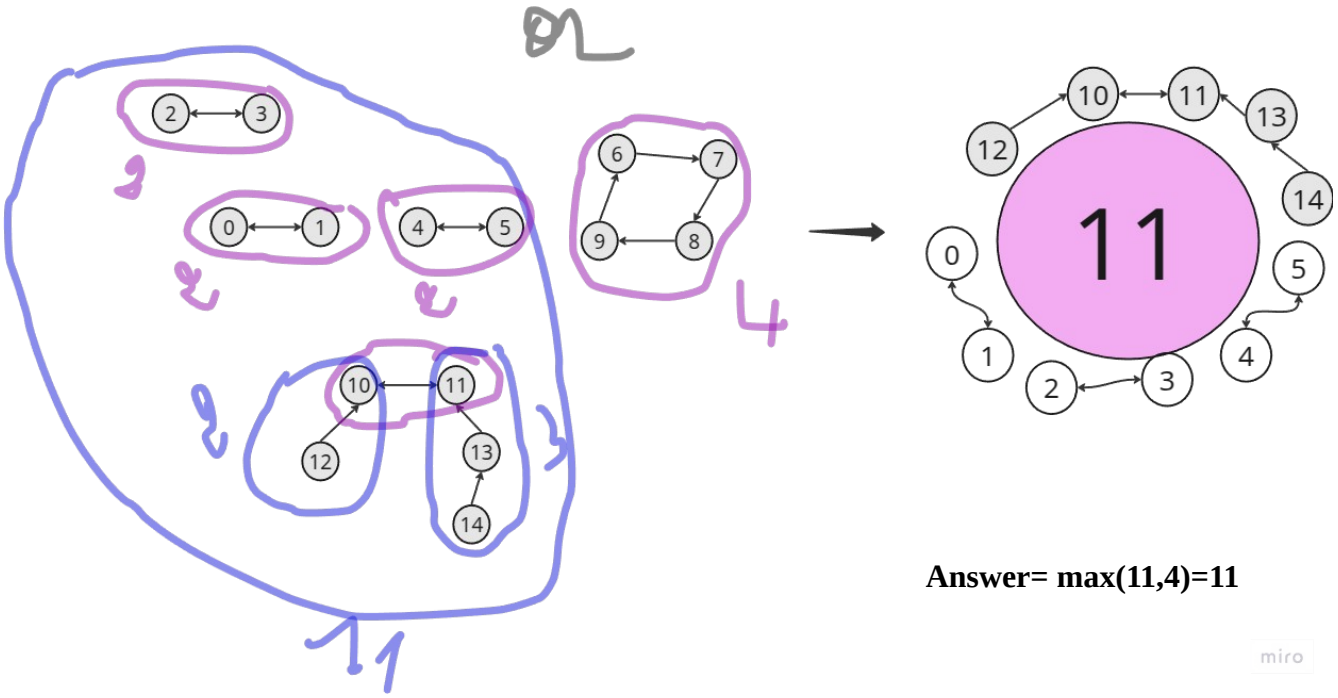
If cycle length is greater than 2 , It could be our answer:



If cycle length is equal to 2 , we need to take all the node in the whole path, so we should compute the depth(in this context, depth is the #nodes) of both nodes in the cycle:



But, we should be careful, the longest cycle is not always the correct answer:



So, our solution consists of three parts:

Cycle detection

Eliminating non-cycle nodes.

Find the longest cycle

Find the longest cycle of all cycles which their length is greater than 2 . Also, determine the longest path that belong to a cycle of length equal to 2 .

Final answer

$$answer = max(longest\ cycle\ of\ length\ greater\ than\ 2, longest\ path\ belongs\ to\ cycle\ equal\ to\ 2)$$

In fact, identifying and processing these cycles is key to finding the solution, the most intuitive algorithm to eliminate non-cycle nodes is ***topological sort***.

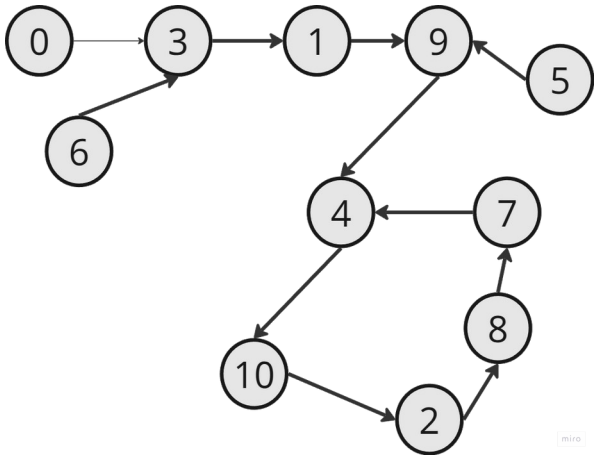
Topological sort is an algorithm traditionally **used in DAGs** (Directed Acyclic Graphs), it allows us to process nodes one by one, ensuring that we handle dependencies.

However, in this context, we don't have a pure DAG because of the cycles. But we can still use topological sorting to help with eliminating non-cycle nodes and focusing on cycles that we need to handle more carefully.

Topological sort: dry run

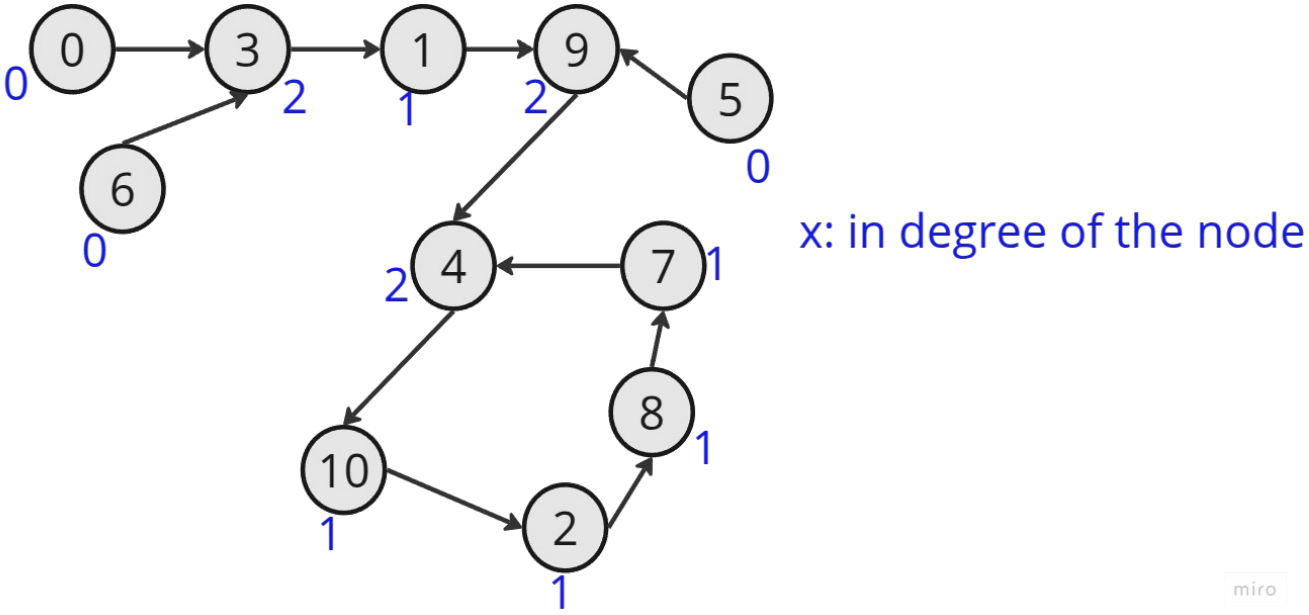
[3,9,8,1,10,9,3,4,7,4,2]

favorite:										
3	9	8	1	10	9	3	4	7	4	2
0	1	2	3	4	5	6	7	8	9	10



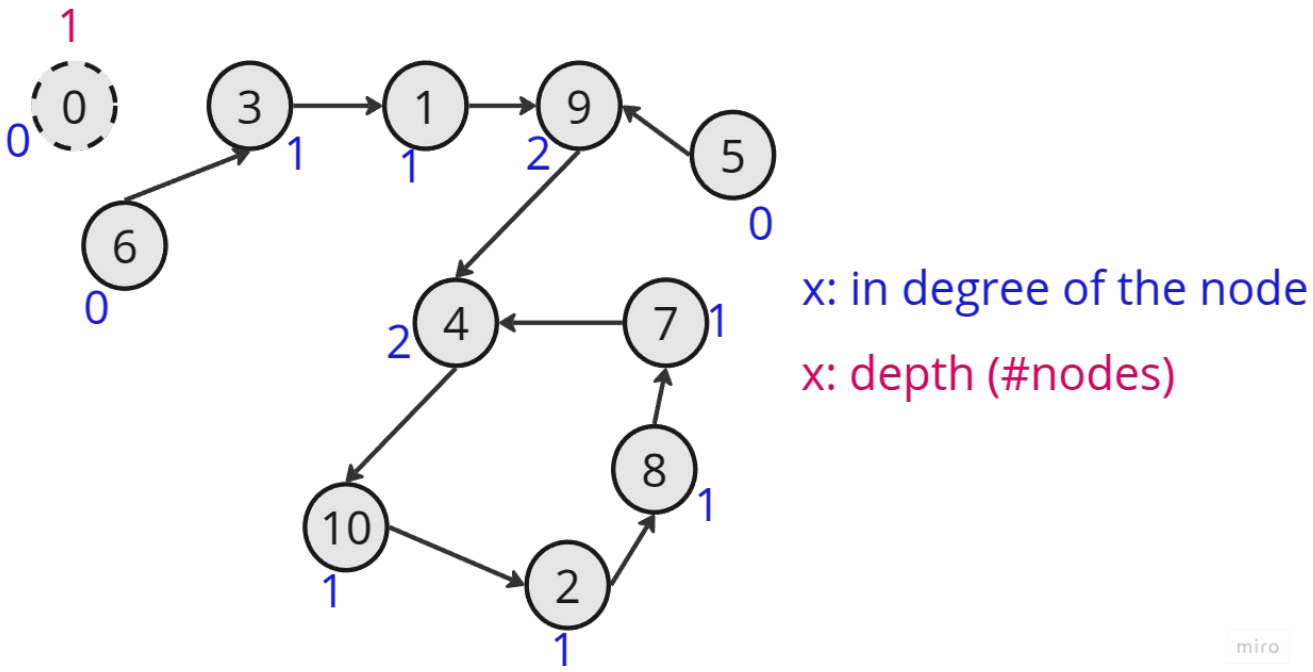
Cycle detection

- Compute the indegree of each node:

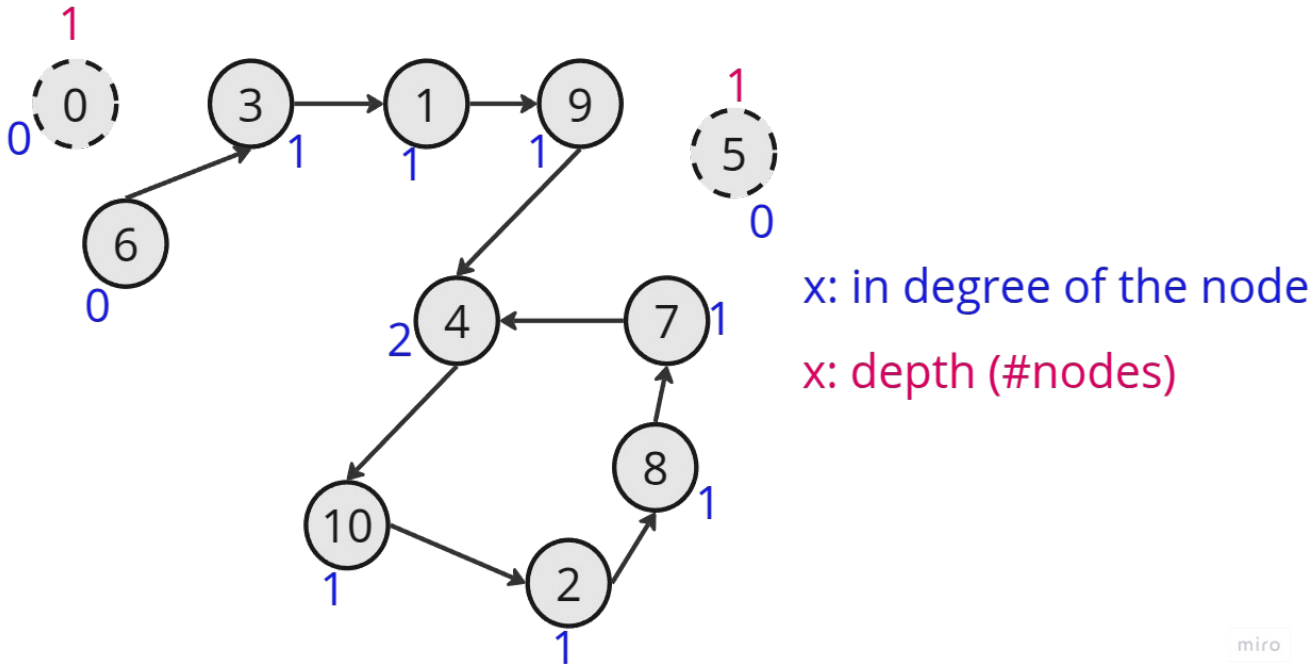


- Eliminate the non-cycle nodes and computing nodes' depth

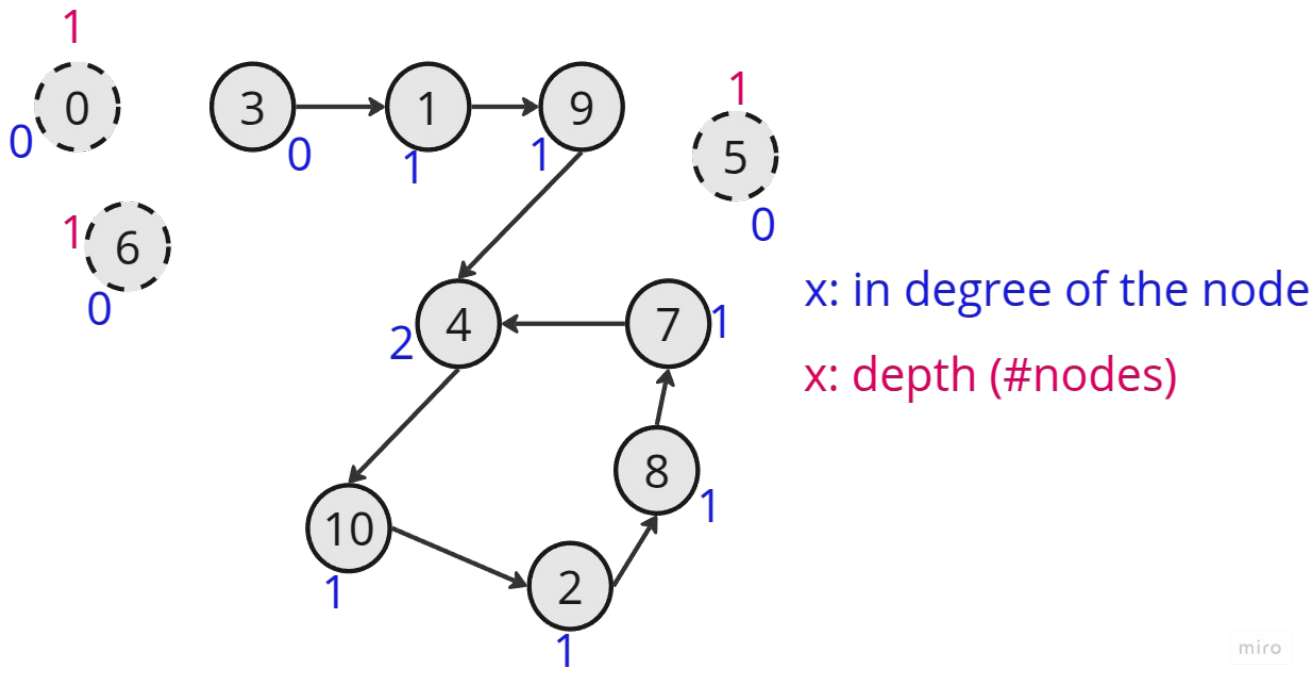
Process node 0:



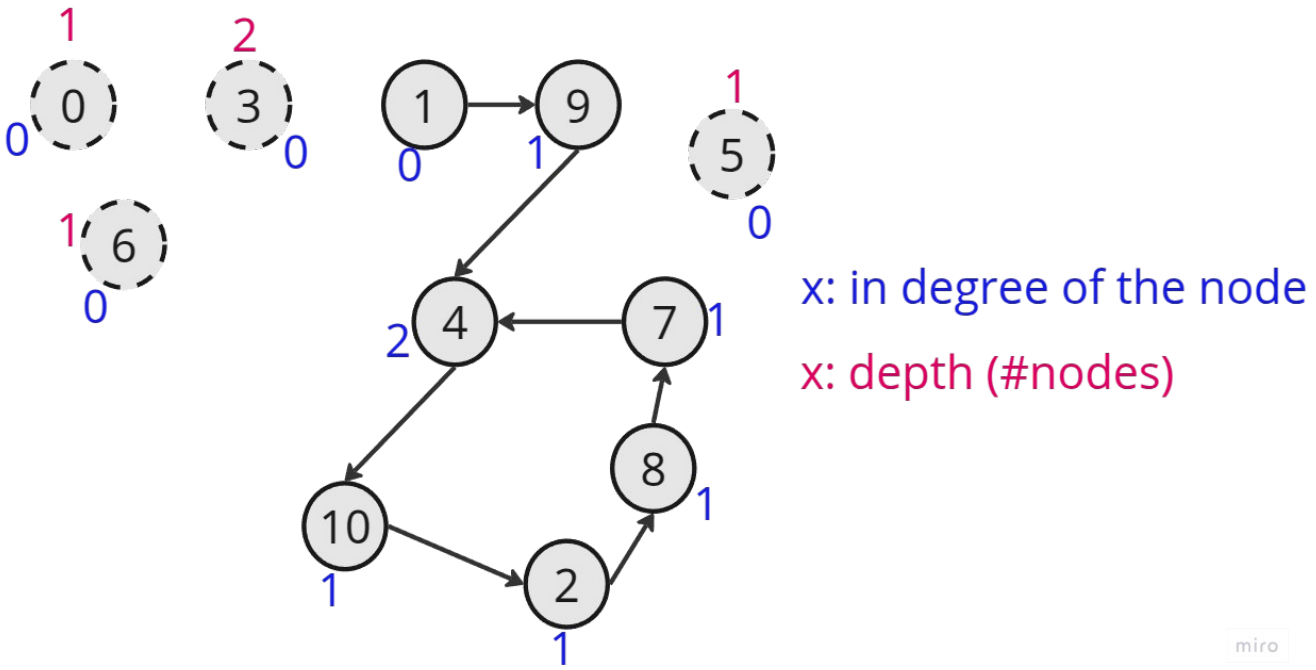
Process node 5:



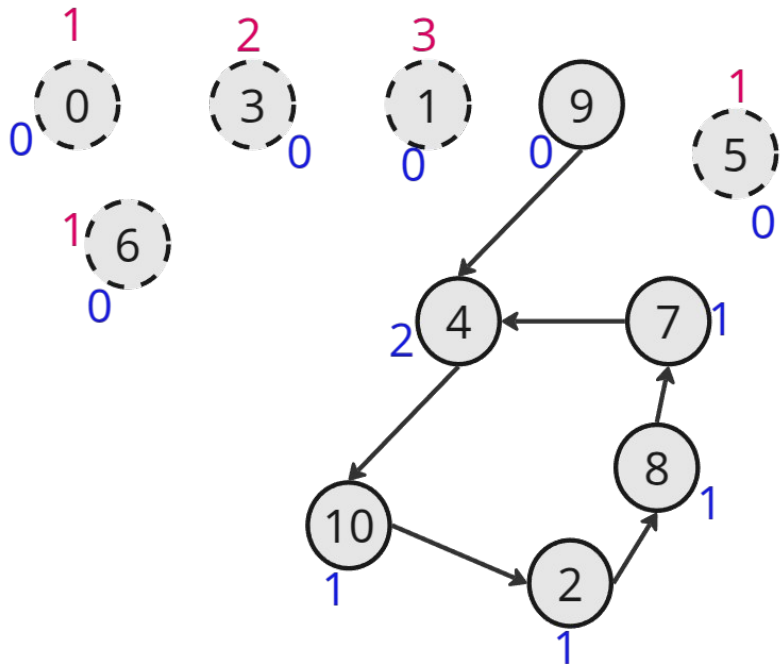
Process node 6:



Process node 3:

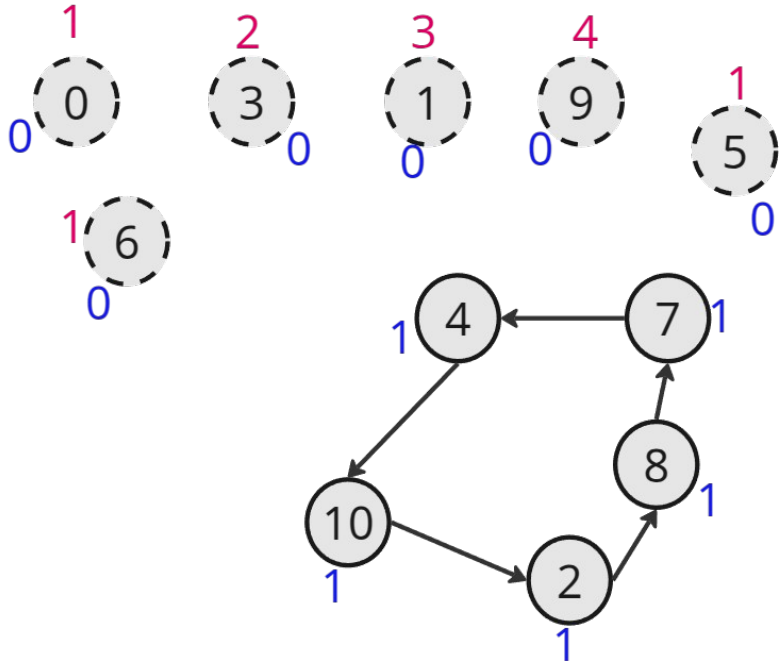


Process node 1:



Process node 9:

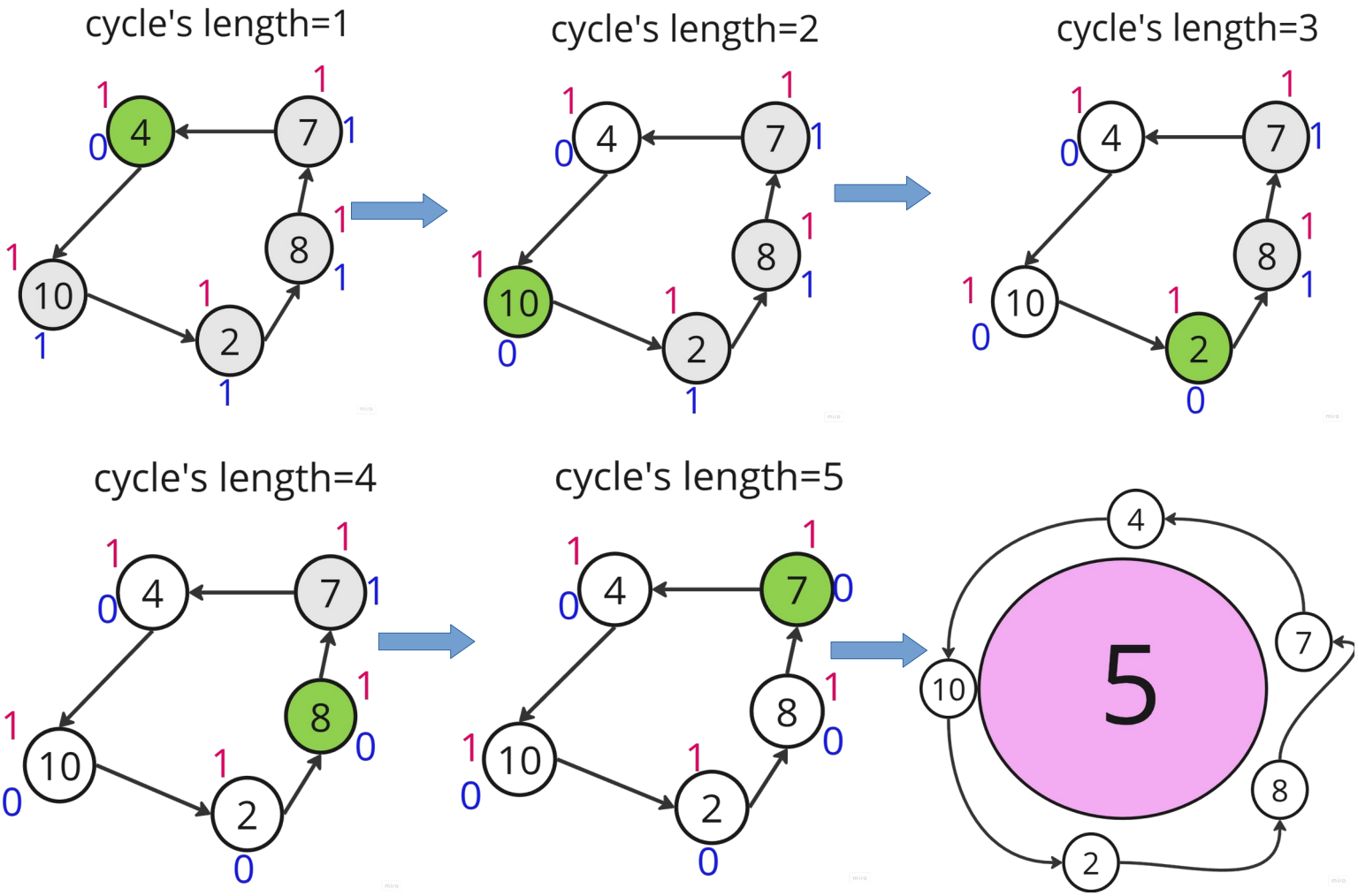
We have two paths to reach node 9 : $(0 \to 3 \to 1 \to 9)$ and $(5 \to 9)$, its depth is the maximum of depth from 0 to 9 and from 5 to 9 .



Once cycles are founded. We compute their lengths.

Find the longest cycle

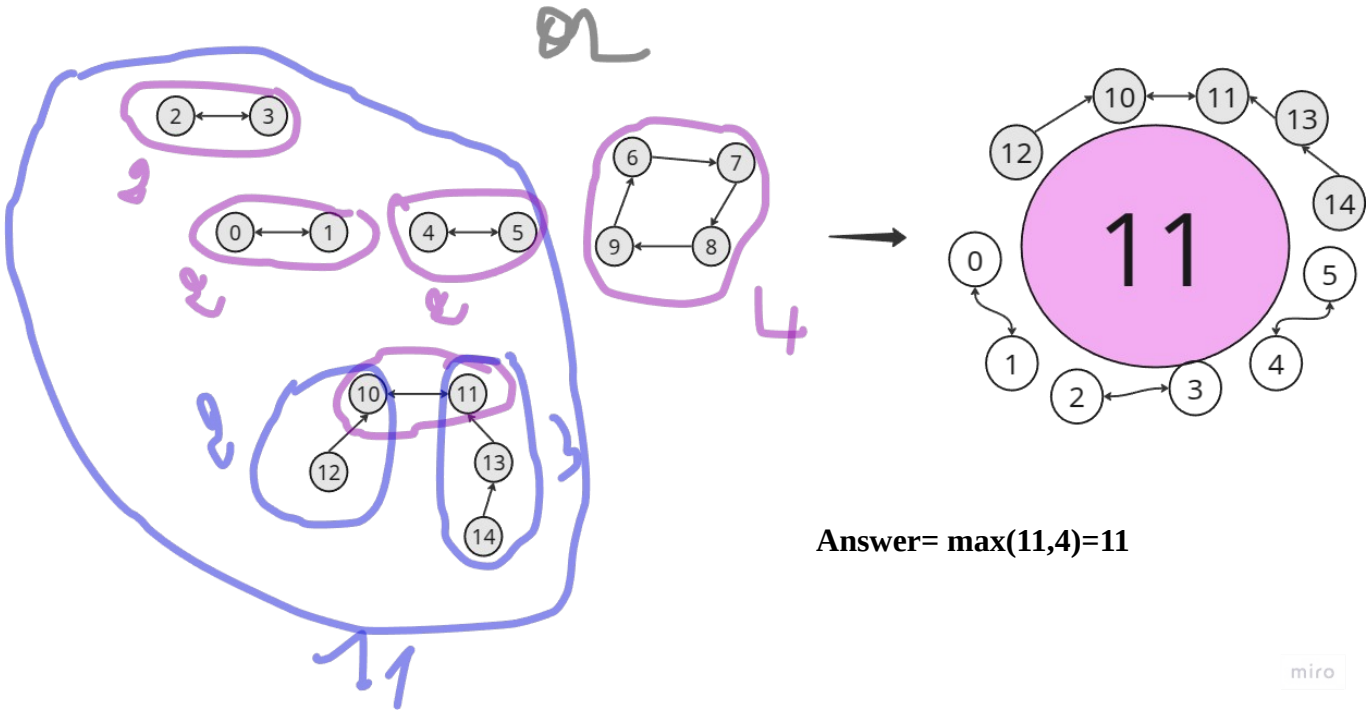
Starting from any node, while iterating over the cycle, increment the cycle length, and attribute 0 as in degree, to indicate to the algorithm to not revisit that node again.



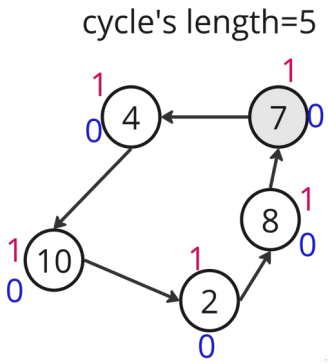
Final answer

As mentioned before, he longest cycle is not always the correct answer:

$$answer = max(longest\ cycle\ of\ length\ greater\ than\ 2, longest\ path\ belongs\ to\ a\ cycle\ equal\ to\ 2)$$



But, in our example we don't have a cycle of length 2 :



so, $ans = max(5, 0) = 5$

2127. Maximum Employees to Be Invited to a Meeting

```
typedef std::vector<int> vi;
/*
   Topological sort
   Time complexity: O(6n)=O(n)
   Space complexity: O(3n)
*/
class Solution {
   public:
      // Cycle detection
      void find_cycles(vi& favorite,vi& indegree,vi& depth,int n){
         // Compute the indegree of each node
         for(int node=0;node<n;++node) indegree[favorite[node]]++;

         // Put all degree with in degree==0 in a FIFO queue
         std::queue<int> q;
         for(int node=0;node<n;++node){
            if(indegree[node]==0) q.push(node);
         }

         // Eliminate non-cycle node and compute the depth of each node
         while(!q.empty()){
            int cur_node=q.front();
            q.pop();
            int next_node=favorite[cur_node];
            depth[next_node]=std::max(depth[next_node],depth[cur_node]+1);
            if(--indegree[next_node]==0) q.push(next_node);
         }
      }
```

```
    // Longest length and final answer
    int get_longest_length_and_final_answer(vi& favorite,vi& indegree,vi& depth,int n){
        int cycle_gt_2=0,cycle_eq_2_and_path=0;
        for(int node=0;node<n;++node){
            // Just focus on node with non zero in degree
            if(indegree[node]==0) continue;

            // First node in the current cycle with non zero in degree
            int cur=node;
            int cycle_len=0; // Length of current cycle
            while(indegree[cur]!=0){
                indegree[cur]=0; // Mark current node as visited
                cycle_len++;
                cur=favorite[cur]; // Go to next node
            }

            // If the current cycle length is greater than 2
            // than, take the maximum length of all cycles which
            //their length greater than 2
            if(cycle_len>2) cycle_gt_2=std::max(cycle_gt_2,cycle_len);
            // If the current cycle length is equal to 2
            // take the the length of two paths (depth[node]+depth[favorite[node]])
            // Cumulate all the lengths.
            else cycle_eq_2_and_path+=depth[node]+depth[favorite[node]];
        }

        // Final answer
        return std::max(cycle_gt_2,cycle_eq_2_and_path);
    }

    int maximumInvitations(vi& favorite){
        int n=favorite.size();
        vi indegree(n,0),depth(n,1);
        find_cycles(favorite,indegree,depth,n);
        return get_longest_length_and_final_answer(favorite,indegree,depth,n);
    }
};
```