<https://leetcode.com/problems/array-nesting/description/>

You are given an integer array nums of length n where nums is a permutation of the numbers in the range [0, n - 1].

You should build a set s[k] = {nums[k], nums[nums[k]], nums[nums[nums[k]]], ... } subjected to the following rule:

The first element in s[k] starts with the selection of the element nums[k] of index = k.

The next element in s[k] should be nums[nums[k]], and then nums[nums[nums[k]]], and so on.

We stop adding right before a duplicate element occurs in s[k].

Return the longest length of a set s[k].

**Example 1:**

Input: nums = [5,4,0,3,1,6,2]

Output: 4

Explanation:

nums[0] = 5, nums[1] = 4, nums[2] = 0, nums[3] = 3, nums[4] = 1, nums[5] = 6, nums[6] = 2.

One of the longest sets s[k]:

s[0] = {nums[0], nums[5], nums[6], nums[2]} = {5, 6, 2, 0}

**Example 2:**

Input: nums = [0,1,2]

Output: 1

**Constraints:**

1 <= nums.length <= 10^5

0 <= nums[i] < nums.length

All the values of nums are **unique**.

**Attempt 1: 2023-09-14**

**Solution 1: Directed Graph cycle detection (30 min)**

**Style 1: No 'recStack'**

class Solution {

    public int arrayNesting(int[] nums) {

        int max = 0;

        int n = nums.length;

        boolean[] visited = new boolean[n];

        // Why we can skip already visited element in the array

        // to start next round circle check ?

        // If a number is visited before, then the set that starts

        // at this number must be smaller then previous max

        for(int i = 0; i < n; i++) {

            if(!visited[i]) {

                max = Math.max(max, helper(nums, i, visited, 0));

            }

        }

        return max;

    }

    // Why we can find circle for nums ?

    // You are given an integer array nums of length n where

    // nums is a permutation of the numbers in the range [0, n - 1].

    // every number can be mapped to its index without out of range.

    // Actually, what we have is permutation of numbers, where we

    // need to find the longest loop. What we need to do is to

    // look at our nums as graph, imagine we have

    // nums = [5, 4, 0, 3, 1, 6, 2], then we have connections:

    // 0 -> 5, 1 -> 4, 2 -> 0, 3 -> 3, 4 -> 1, 5 -> 6, 6 -> 2,

    // which in fact can be looked as several disjoint loops

    // 0 -> 5 -> 6 -> 2 -> 0, 1 -> 4 -> 1, 3 -> 3.

    // So, we start from value 0 and start to traverse our graph.

    // We will keep already visited nodes in visited set, so if

    // we already traversed some loop, we never do it again: it

    // is very similar what we do in classical dfs algorithm when

    // we want to find islands.

    private int helper(int[] nums, int index, boolean[] visited, int circleLength) {

        if(visited[index]) {

            return circleLength;

        }

        visited[index] = true;

        return helper(nums, nums[index], visited, circleLength + 1);

    }

}

Time Complexity: O(N), where N <= 10^5 is number of elements in nums array.

Space Complexity: O(N)

**Style 2: Standard Directed Graph cycle detection with 'recStack'**

class Solution {

    public int arrayNesting(int[] nums) {

        int max = 0;

        int n = nums.length;

        boolean[] visited = new boolean[n];

        boolean[] recStack = new boolean[n];

        // Why we can skip already visited element in the array

        // to start next round circle check ?

        // If a number is visited before, then the set that starts

        // at this number must be smaller than previous max

        for(int i = 0; i < n; i++) {

            if(!visited[i]) {

                max = Math.max(max, helper(nums, i, visited, recStack, 0));

            }

        }

        return max;

    }

    // Why we can find circle for nums ?

    // You are given an integer array nums of length n where

    // nums is a permutation of the numbers in the range [0, n - 1].

    // every number can be mapped to its index without out of range.

    // Actually, what we have is permutation of numbers, where we

    // need to find the longest loop. What we need to do is to

    // look at our nums as graph, imagine we have

    // nums = [5, 4, 0, 3, 1, 6, 2], then we have connections:

    // 0 -> 5, 1 -> 4, 2 -> 0, 3 -> 3, 4 -> 1, 5 -> 6, 6 -> 2,

    // which in fact can be looked as several disjoint loops

    // 0 -> 5 -> 6 -> 2 -> 0, 1 -> 4 -> 1, 3 -> 3.

    // So, we start from value 0 and start to traverse our graph.

    // We will keep already visited nodes in visited set, so if

    // we already traversed some loop, we never do it again: it

    // is very similar what we do in classical dfs algorithm when

    // we want to find islands.

    private int helper(int[] nums, int index, boolean[] visited, boolean[] recStack, int circleLength) {

        if(recStack[index]) {

            return circleLength;

        }

        if(visited[index]) {

            return 0;

        }

        visited[index] = true;

        recStack[index] = true;

        int result = helper(nums, nums[index], visited, recStack, circleLength + 1);

        recStack[index] = false;

        return result;

    }

}

Time Complexity: O(N), where N <= 10^5 is number of elements in nums array.

Space Complexity: O(N)

**A more regular style from Deepseek**

**No need passing "cycleLength" as parameter in recursion call.**

class Solution {

    public int arrayNesting(int[] nums) {

        int n = nums.length;

        boolean[] visited = new boolean[n]; // Track visited nodes

        boolean[] recStack = new boolean[n]; // Track nodes in the current DFS path

        int maxLength = 0; // Store the length of the longest cycle

        // Iterate through each node

        for (int i = 0; i < n; i++) {

            if (!visited[i]) {

                int currentLength = dfs(i, nums, visited, recStack);

                maxLength = Math.max(maxLength, currentLength);

            }

        }

        return maxLength;

    }

    private int dfs(int index, int[] nums, boolean[] visited, boolean[] recStack) {

        // If the current node is already in the recursion stack, a cycle is detected

        if (recStack[index]) {

            return 0; // Cycle detected, but we are interested in the length of the cycle

        }

        // If the node is already visited, skip it

        if (visited[index]) {

            return 0;

        }

        // Mark the current node as visited and part of the recursion stack

        visited[index] = true;

        recStack[index] = true;

        // Traverse to the next node in the cycle

        int next = nums[index];

        int cycleLength = 1 + dfs(next, nums, visited, recStack);

        // Backtrack: Remove the current node from the recursion stack

        recStack[index] = false;

        return cycleLength;

    }

}

Time Complexity: O(N), where N <= 10^5 is number of elements in nums array.

Space Complexity: O(N)

**Solution 2: Directed Graph cycle detection but save space to O(1) by set visited element to -1 (10 min)**

class Solution {

    public int arrayNesting(int[] nums) {

        int max = 0;

        int n = nums.length;

        for(int i = 0; i < n; i++) {

            if(nums[i] != -1) {

                max = Math.max(max, helper(nums, i, 0));

            }

        }

        return max;

    }

    private int helper(int[] nums, int index, int circleLength) {

        if(nums[index] == -1) {

            return circleLength;

        }

        int tmp = nums[index];

        nums[index] = -1;

        return helper(nums, tmp, circleLength + 1);

    }

}

Time Complexity: O(N), where N <= 10^5 is number of elements in nums array.

Space Complexity: O(1)

**Refer to standard Directed Graph cycle detection**

**Refer to**

<https://www.geeksforgeeks.org/detect-cycle-in-a-graph/>

**Approach:**

The problem can be solved based on the following idea:

*To find cycle in a directed graph we can use the Depth First Traversal (DFS) technique. It is based on the idea that there is a cycle in a graph* ***only if there is a back edge*** *[i.e., a node points to one of its ancestors] present in the graph.*

***To detect a back edge, we need to keep track of the nodes visited till now and the nodes that are in the current recursion stack [i.e., the current path that we are visiting]. If during recursion, we reach a node that is already in the recursion stack, there is a cycle present in the graph.***

**Note:** If the graph is disconnected then get the DFS forest and check for a cycle in individual trees by checking back edges.

Follow the below steps to Implement the idea:

Create a recursive **dfs** function that has the following parameters –**current vertex, visited array,** and **recursion stack**.

Mark the current node as visited and also mark the index in the recursion stack.

Iterate a loop for all the vertices and for each vertex, call the recursive function if it is not yet visited (This step is done to make sure that if there is a forest of graphs, we are checking each forest):

In each recursion call, Find all the adjacent vertices of the current vertex which are not visited:

If an adjacent vertex is already marked in the recursion stack then return **true**.

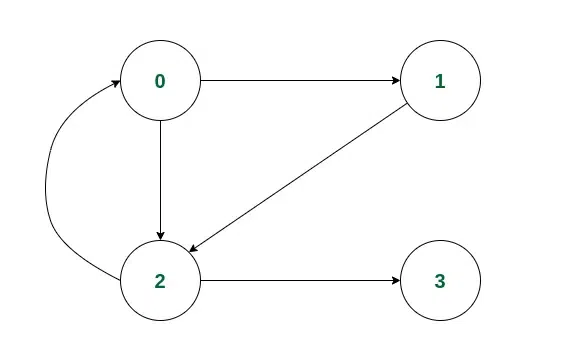
Otherwise, call the recursive function for that adjacent vertex.

**While returning from the recursion call, unmark the current node from the recursion stack, to represent that the current node is no longer a part of the path being traced.**

If any of the functions returns **true**, stop the future function calls and return **true** as the answer.

**Illustration:**

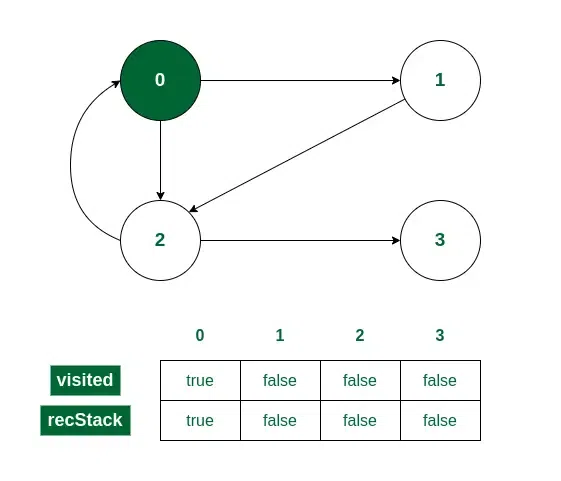
Consider the following graph:



*Example of a Directed Graph*

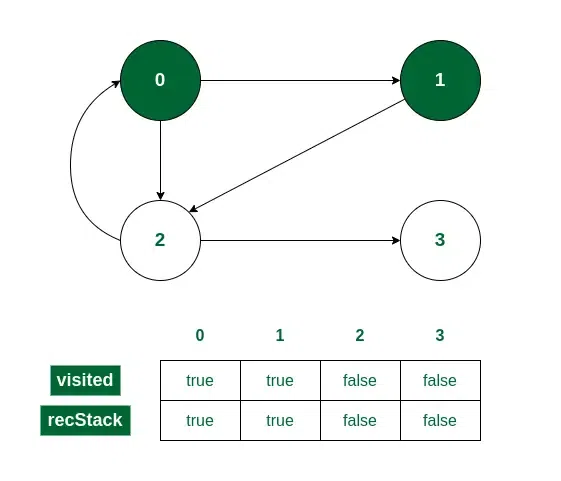
*Consider we start the iteration from vertex* ***0****.*

*Initially,* ***0*** *will be marked in both the* ***visited[]*** *and* ***recStack[]*** *array as it is a part of the current path.*



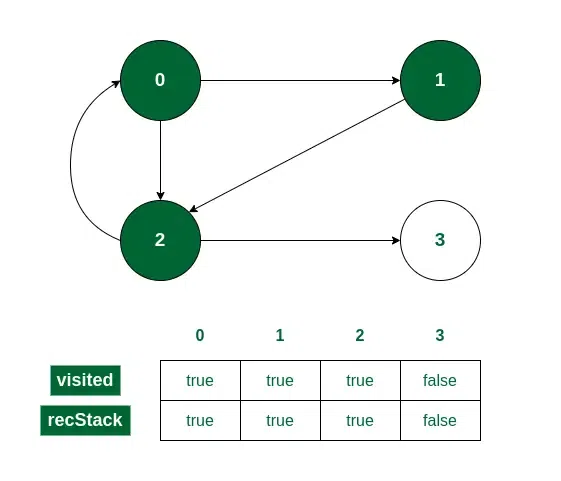
*Vertex 0 is visited*

*Now* ***0*** *has two adjacent vertices* ***1*** *and* ***2****. Let us consider traversal to the vertex 1. So 1 will be marked in both* ***visited[]*** *and* ***recStack[]****.*



*Vertex 1 is visited*

*Vertex* ***1*** *has only one adjacent vertex. Call the recursive function for* ***2*** *and mark it in* ***visited[]*** *and* ***recStack[]****.*

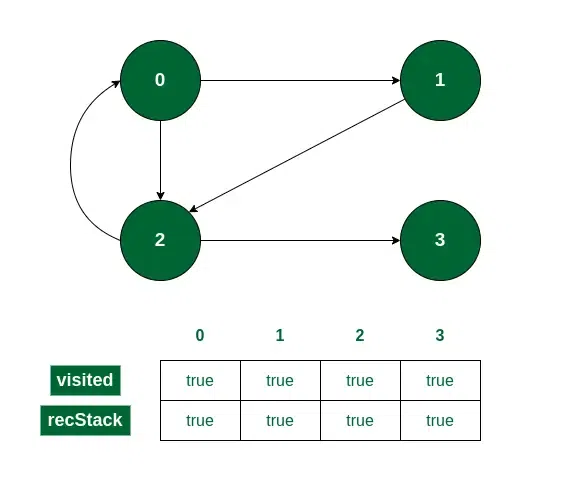


*Vertex 2 is visited*

*Vertex* ***2*** *also has two adjacent vertices.*

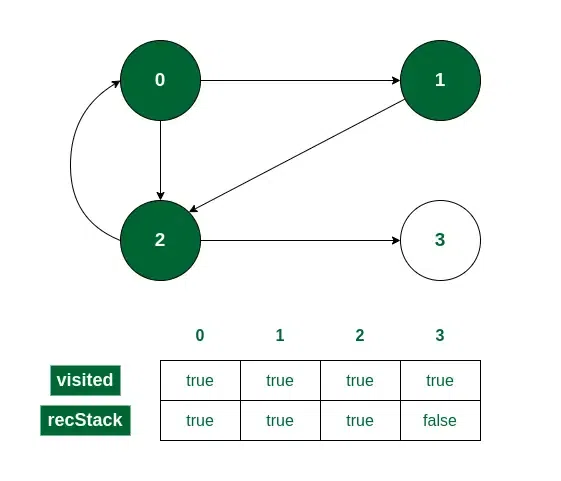
*Vertex* ***0*** *is visited and already marked in the* ***recStack[]****. So if 0 is checked first, we will get the answer that there is a cycle present.*

*On the other hand, if vertex 3 is checked first, then 3 will be marked in* ***visited[]****and* ***recStack[]****.*



*Vertex 3 is visited*

*While returning from the recursion call for* ***3****, it will be unmarked from* ***recStack[]*** *as it is now not a part of the path currently being traced.*



*Vertex 3 is unmarked from recStack[]*

*Now we have only one option to check, vertex 0, which is already marked in recStack[].*

*So, we can conclude that a cycle exists. We can also find the cycle if we have traversed to vertex 2 from 0 itself in this same way.*

4. Detect Cycle in a Directed Graph using DFS(Backtracking)

<https://www.geeksforgeeks.org/detect-cycle-in-a-graph/>

class Graph {

    private final int V;

    private final List < List < Integer >> adj;

    public Graph(int V) {

        this.V = V;

        adj = new ArrayList < > (V);

        for (int i = 0; i < V; i++)

            adj.add(new LinkedList < > ());

    }

    // This function is a variation of DFSUtil() in

    // https://www.geeksforgeeks.org/archives/18212

    private boolean isCyclicUtil(int i, boolean[] visited, boolean[] recStack) {

        // Mark the current node as visited and

        // part of recursion stack

        if (recStack[i])

            return true;

        if (visited[i])

            return false;

        visited[i] = true;

        recStack[i] = true;

        List < Integer > children = adj.get(i);

        for (Integer c: children)

            if (isCyclicUtil(c, visited, recStack))

                return true;

        recStack[i] = false;

        return false;

    }

    private void addEdge(int source, int dest) {

        adj.get(source).add(dest);

    }

    // Returns true if the graph contains a

    // cycle, else false.

    // This function is a variation of DFS() in

    // https://www.geeksforgeeks.org/archives/18212

    private boolean isCyclic() {

        // Mark all the vertices as not visited and

        // not part of recursion stack

        boolean[] visited = new boolean[V];

        boolean[] recStack = new boolean[V];

        // Call the recursive helper function to

        // detect cycle in different DFS trees

        for (int i = 0; i < V; i++)

            if (isCyclicUtil(i, visited, recStack))

                return true;

        return false;

    }

    // Driver code

    public static void main(String[] args) {

        Graph graph = new Graph(4);

        graph.addEdge(0, 1);

        graph.addEdge(0, 2);

        graph.addEdge(1, 2);

        graph.addEdge(2, 0);

        graph.addEdge(2, 3);

        graph.addEdge(3, 3);

        if (graph.isCyclic())

            System.out.println("Graph contains cycle");

        else

            System.out.println("Graph doesn't " + "contain cycle");

    }

}

The DFS implement examples on below problems:  CourseSchedule.java

**Why we don't have to use 'recStack' to detect the cycle happening in L565.Array Nesting even its a Directed Graph cycle detection problem ?**

**Refer to**

<https://leetcode.com/problems/array-nesting/solutions/102472/this-is-actually-dfs/>

<https://stackoverflow.com/questions/19113189/detecting-cycles-in-a-graph-using-dfs-2-different-approaches-and-whats-the-dif>

**First, in directed graph even visited node doesn't mean a cycle exist**

**In a directed graph 2 different paths to the same vertex don't make a cycle.**

**For example: A --> B, B --> C, A --> C - don't make a cycle whereas in undirected ones: A -- B, B -- C, A -- C (or we can think as C -- A as its undirected) does.**

**in directed graph, even node C been visited, doesn't mean a cycle happen**

**Find a cycle in undirected graphs**

An undirected graph has a cycle if and only if a depth-first search (DFS) finds an edge that points to an already-visited vertex (a back edge). In undirected graph there exists a cycle only if there is a back edge excluding the parent of the edge from where the back edge is found. Else every undirected graph has a cycle by default if we don't exclude the parent edge when finding a back edge.

**Find a cycle in directed graphs**

In addition to visited vertices we need to keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree.

And why for L565.Array Nesting even its a directed graph problem (when mapping index and nums[index], the direction is build as index -> nums[index], not mutual way), we can omit the usage of 'recStack' and only use 'visited' to detect a cycle ?

**This is regarding the special input of L565:**

You are given an integer array nums of length n where nums is a permutation of the numbers in the range [0, n - 1].

Since the given input as above, we are surely there will be a cycle happening, since the index -> nums[index] relation cannot stretch out of the range [0, n - 1], hence for any number locate on index [0, n - 1] must have a chance to collide with its value that also between [0, n - 1], e.g nums = [5, 4, 0, 3, 1, 6, 2], then we have connections: 0 -> 5, 1 -> 4, 2 -> 0, 3 -> 3, 4 -> 1, 5 -> 6, 6 -> 2, which in fact can be looked as several disjoint cycles: 0 -> 5 -> 6 -> 2 -> 0, 1 -> 4 -> 1, 3 -> 3.

And since there is surely a cycle will happen based on definition of L565 input, we can use 'visited' boolean array only to detect the cycle by omitting the 'recStack' boolean array since 'recStack' always working as an auxiliary tracking array to determine a node will cause a cycle happening or not based on current path regardless of its already a visited node, in another word, even a node been visited before, in a directed graph we cannot say there must be a cycle happening when we encounter this node again. But for this question L565, since the special definition of input array, we don't need to worry the cycle not there, if encounter a visited number again, then that's a cycle happening.

And one more important tip for why we can omit the already visited number in array if we want to scan the forest (DFS on Graph have to check all potential start tree root, and all numbers in the given array are potential start tree root here) ? **Its beause if a number is visited before, then the path length that starts at this number now must be smaller then previous max.**

**Refer to**

<https://leetcode.com/problems/array-nesting/solutions/1438159/python-loop-detection-explained/>

Actually, what we have is permutation of numbers, where we need to find the longest loop. What we need to do is to look at our nums as graph, imagine we have

nums = [5, 4, 0, 3, 1, 6, 2], then we have connections: 0 -> 5, 1 -> 4, 2 -> 0, 3 -> 3, 4 -> 1, 5 -> 6, 6 -> 2, which in fact can be looked as several disjoint loops

0 -> 5 -> 6 -> 2 -> 0, 1 -> 4 -> 1, 3 -> 3.

So, we start from value 0 and start to traverse our graph. We will keep already visited nodes in visited set, so if we already traversed some loop, we never do it again: it is very similar what we do in classical dfs algorithm when we want to find islands.

**Complexity**

Time and space complexity is O(n), because we never visit node twice.

**Code**

class Solution:

    def arrayNesting(self, nums):

        n = len(nums)

        visited = [0]\*n

        for i in range(n):

            start, depth = i, 1

            while not visited[start]:

                visited[start] = depth

                start = nums[start]

                depth += 1

        return max(visited)

**Remark**

Can we do better? Not in time: we need to look at our data at least once, what about space? We can also rewrite original array with -1 with O(n) time and O(1) space. Finally, if we are not allowed to modify data, we can use loop detection technique with slow and fast iterators with O(n) time and O(1) memory.

**Refer to**

<https://leetcode.com/problems/array-nesting/solutions/1438313/c-python-find-the-longest-length-between-cycles-with-picture-clean-concise/>

**Version 1: Straightforward - O(N) Space**

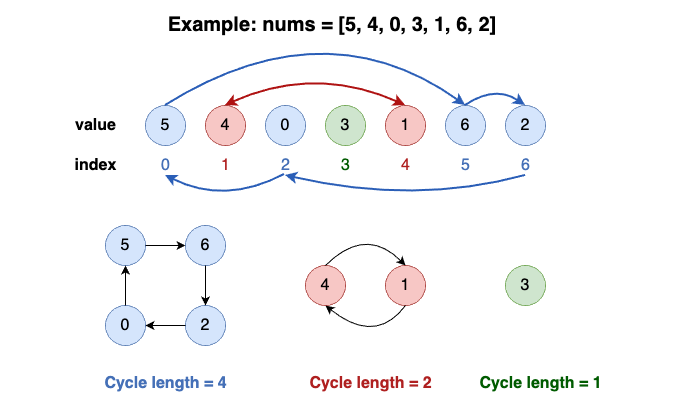
Elements in the same set will form a cycle.

We just traverse elements x in nums:

If x is not visited then we dfs(x) to find elements in the same cycle with node x.

Update the ans if the current cycle has length greater than ans.

Check the following picture for more clearly.



class Solution {

public:

    int arrayNesting(vector<int>& nums) {

        int n = nums.size(), ans = 0;

        vector<bool> visited(n, false);

        for (int x : nums) {

            int cnt = 0;

            while (!visited[x]) {

                cnt += 1;

                visited[x] = true;

                x = nums[x];

            }

            ans = max(ans, cnt);

        }

        return ans;

    }

};

**Complexity**

Time: O(N), where N <= 10^5 is number of elements in nums array.

Space: O(N)

**Version 2: In space modification - O(1) Space**

We can achieve O(1) in space by modify nums array directly, mark nums[i] = -1 with the meaning visited[i] = True.

class Solution {

public:

    int arrayNesting(vector<int>& nums) {

        int ans = 0;

        for (int x : nums) {

            if (x == -1) continue;

            int cnt = 0;

            while (nums[x] != -1) {

                cnt += 1;

                int prev = x;

                x = nums[x];

                nums[prev] = -1; // same with visited[prev] = True

            }

            ans = max(ans, cnt);

        }

        return ans;

    }

};

**Complexity**

Time: O(N), where N <= 10^5 is number of elements in nums array.

Space: O(1)

**Refer to**

<https://grandyang.com/leetcode/565/>

这道题让我们找嵌套数组的最大个数，给的数组总共有n个数字，范围均在 [0, n-1] 之间，题目中也把嵌套数组的生成解释的很清楚了，其实就是值变成坐标，得到的数值再变坐标。那么实际上当循环出现的时候，嵌套数组的长度也不能再增加了，而出现的这个相同的数一定是嵌套数组的首元素，博主刚开始没有想清楚这一点，以为出现重复数字的地方可能是嵌套数组中间的某个位置，于是用个 set 将生成的嵌套数组存入，然后每次查找新生成的数组是否已经存在。而且还以原数组中每个数字当作嵌套数组的起始数字都算一遍，结果当然是 TLE 了。其实对于遍历过的数字，我们不用再将其当作开头来计算了，而是只对于未遍历过的数字当作嵌套数组的开头数字，不过在进行嵌套运算的时候，并不考虑中间的数字是否已经访问过，而是只要找到和起始位置相同的数字位置，然后更新结果 res，参见代码如下：

class Solution {

    public:

    int arrayNesting(vector<int>& nums) {

        int n = nums.size(), res = INT\_MIN;

        vector<bool> visited(n, false);

        for (int i = 0; i < nums.size(); ++i) {

            if (visited[nums[i]]) continue;

            res = max(res, helper(nums, i, visited));

        }

        return res;

    }

    int helper(vector<int>& nums, int start, vector<bool>& visited) {

        int i = start, cnt = 0;

        while (cnt == 0 || i != start) {

            visited[i] = true;

            i = nums[i];

            ++cnt;

        }

        return cnt;

    }

};

**Refer to**

<https://leetcode.com/problems/array-nesting/editorial/>

**Approach #3 Without Using Extra Space [Accepted]**

**Algorithm**

In the last approach, the visited array is used just to keep a track of the elements of the array which have already been visited. Instead of making use of a separate array to keep track of the same, we can mark the visited elements in the original array nums itself. Since, the range of the elements can only be between 1 to 20,000, we can put a very large integer value Integer.MAX\_VALUE at the position which has been visited. The rest process of traversals remains the same as in the last approach.

public class Solution {

    public int arrayNesting(int[] nums) {

        int res = 0;

        for (int i = 0; i < nums.length; i++) {

            if (nums[i] != Integer.MAX\_VALUE) {

                int start = nums[i], count = 0;

                while (nums[start] != Integer.MAX\_VALUE) {

                    int temp = start;

                    start = nums[start];

                    count++;

                    nums[temp] = Integer.MAX\_VALUE;

                }

                res = Math.max(res, count);

            }

        }

        return res;

    }

}

**Complexity Analysis**

Time complexity : O(n). Every element of the nums array will be considered at most once.

Space complexity : *O*(1). Constant Space is used.

**Refer to Deepseek**

**Instead of using both 'recStack' and 'visited' array, this solution omit the use of 'recStack' array, why it works ?**

In the solution provided earlier, the recStack array is omitted, and only the visited array is used. Despite this, the solution still works for detecting cycles in the context of LeetCode 565. Array Nesting. Let me explain why this works and why the recStack array is not necessary for this specific problem.

**Why the recStack Array is Not Needed**

**1. Nature of the Problem:**

**The problem involves a permutation array, where each index i points to exactly one other index nums[i].**

**This means the graph formed by the array is a collection of disjoint cycles (each node has exactly one outgoing edge, and no two cycles overlap).**

**2. Key Insight:**

**Once a node is visited, it cannot be part of another cycle because each node has exactly one outgoing edge.**

**Therefore, if you revisit a node, it must be part of the same cycle you are currently exploring.**

**3. Cycle Detection Without recStack:**

**The visited array alone is sufficient because:**

**If you encounter a node that is already visited, it means you have completed a cycle.**

**You don't need to track the current path (which is what recStack does) because the problem guarantees that each node belongs to exactly one cycle.**

**Why the recStack Array is Necessary in General Directed Graphs**

**In general directed graphs, nodes can have multiple outgoing edges**, and cycles can overlap or share nodes. For example:

A node might be part of multiple cycles.

A node might be revisited through a different path, which does not necessarily imply a cycle.

In such cases:

**The visited array alone cannot distinguish between:**

**Revisiting a node in the same path (which indicates a cycle).**

**Revisiting a node in a different path (which does not indicate a cycle).**

**The recStack array is used to track the current traversal path and detect cycles correctly.**

**Why the recStack Array is Not Needed for Array Nesting**

In the Array Nesting problem:

Each node has exactly one outgoing edge.

The graph is a collection of disjoint cycles.

Once a node is visited, it cannot be part of another cycle.

Therefore, revisiting a node always means you have completed a cycle in the current path.

Thus, the visited array alone is sufficient to detect cycles and calculate their lengths.

**Comparison of the Two Approaches**

|  |  |  |
| --- | --- | --- |
| **Aspect** | With recStack | Without recStack |
| **Problem Type** | General directed graphs | Permutation arrays (disjoint cycles) |
| **Cycle Detection** | Requires recStack to track current path | visited array suffices |
| **Revisiting Nodes** | May revisit nodes in different paths | Revisiting nodes always means a cycle |
| **Space Complexity** | O(n) for visited + O(n) for recStack | O(n) for visited |
| **Applicability** | Works for any directed graph | Works only for permutation-like structures |

**Why the Earlier Solution Works**

The earlier solution works because:

Disjoint Cycles:

The graph formed by the array is a collection of disjoint cycles.

Each node belongs to exactly one cycle.

No Overlapping Paths:

Once a node is visited, it cannot be part of another cycle.

Revisiting a node always means you have completed a cycle.

Efficient Cycle Length Calculation:

The length of the cycle is calculated by traversing the path until a visited node is encountered.

**When to Use recStack**

Use recStack when dealing with general directed graphs where:

Nodes can have multiple outgoing edges.

Cycles can overlap or share nodes.

Revisiting a node does not always imply a cycle.

**Conclusion**

The earlier solution works without recStack because of the specific structure of the problem (permutation array with disjoint cycles). However, for general directed graphs, the recStack array is necessary to correctly detect cycles. The choice of approach depends on the problem's constraints and the nature of the graph.

**Refer to**

[Directed and Undirected Graph Cycle Detection in DFS and BFS](note://8D51E6EB87834766A8F4688C191B70AC)

[L341.Flatten Nested List Iterator (Ref.L565)](note://WEB5b40195fe3836529b9d98cd492a2679b)