**July 2020 Challenge**

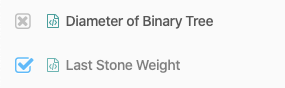
Introduction

This Challenge is beginner-friendly and available to both Premium and non-Premium users. It consists of 31 daily problems over the month of July. A problem is added here each day, and you have 24 hours to make a valid submission for it in order to be eligible for rewards.

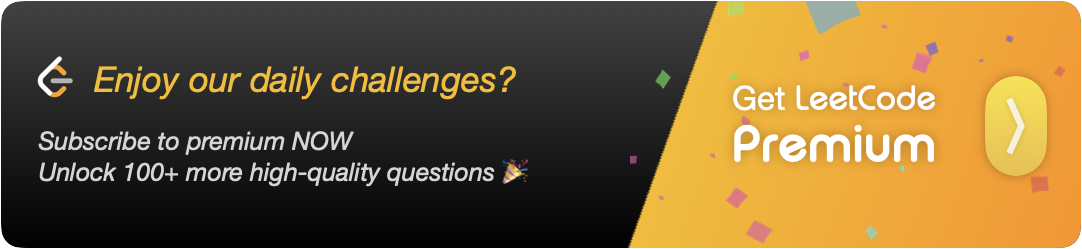
Rules

In order to be eligible for prizes, you must comply with all the Challenge rules.

1. **Valid submission:**only problems where a valid submission is made will be eligible for rewards. This means the submission was:
   1. within the **valid time period** for that problem; this is before 23:59 PM PT on the day it appeared.
   2. made **within this Explore Card**. If you have completed the question previously, you will need to re-submit an answer within this Explore Card.
   3. given the judgement **"Accepted"** when submitted. Please note that if your submission receives a different judgement, including, but not limited to, Wrong Answer or Time Limit Exceeded, you are allowed to resubmit, as long as it's still within the valid time period.

After making a valid submission for a problem, a **blue checkmark** will appear next to that problem in the left bar. A grey cross means that a valid submission was not made on time for that problem, and so, unfortunately, you're no longer eligible for prizes for that problem.  


1. **Late or invalid submissions:**will not be accepted under **any** circumstances. It is ***your responsibility*** to make a valid submission for a problem within its valid time period, and to confirm that the blue checkmark has appeared.
2. **Rewards:**only valid submissions will be eligible for rewards. For more information about the rewards, [check out the Discuss post](https://leetcode.com/discuss/general-discussion/655704).
3. **Disclaimer:** LeetCode reserves the right, in our sole discretion, to disqualify any entries where we believe a user undermines the fairness of this July's LeetCoding Challenge event, which includes, but is not limited to, copying and pasting solutions from other places directly into your submission.

[](https://leetcode.com/subscribe/?ref=ex_dc)

Week 1: July 1st - July 7th

~~+10~~

**Arranging Coins**

~~+10~~

**Binary Tree Level Order Traversal II**

~~+10~~

 Prison Cells After N Days

~~+10~~

**Ugly Number II**

~~+10~~

 Hamming Distance

~~+10~~

 Plus One

~~+10~~

**Island Perimeter**

Week 2: July 8th - July 14th

~~+10~~

 3Sum

~~+10~~

**Maximum Width of Binary Tree**

~~+10~~

 Flatten a Multilevel Doubly Linked List

~~+10~~

 Subsets

~~+10~~

 Reverse Bits

~~+10~~

 Same Tree

~~+10~~

**Angle Between Hands of a Clock**

Week 3: July 15th - July 21st

~~+10~~

 Reverse Words in a String

~~+10~~

 Pow(x, n)

~~+10~~

 Top K Frequent Elements

~~+10~~

 Course Schedule II

~~+10~~

 Add Binary

~~+10~~

 Remove Linked List Elements

~~+10~~

 Word Search

Week 4: July 22nd - July 28th

~~+10~~

 Binary Tree Zigzag Level Order Traversal

~~+10~~

**Single Number III**

~~+10~~

**All Paths From Source to Target**

~~+10~~

 Find Minimum in Rotated Sorted Array II

~~+10~~

 Add Digits

~~+10~~

 Construct Binary Tree from Inorder and Postorder Traversal

~~+10~~

 Task Scheduler

Week 5: July 29th - July 31st

~~+10~~

 Best Time to Buy and Sell Stock with Cooldown

~~+10~~

 Word Break II

~~+10~~

 Climbing Stairs

**Arranging Coins**

You have a total of *n* coins that you want to form in a staircase shape, where every *k*-th row must have exactly *k* coins.

Given *n*, find the total number of **full** staircase rows that can be formed.

*n* is a non-negative integer and fits within the range of a 32-bit signed integer.

**Example 1:**

n = 5

The coins can form the following rows:

¤

¤ ¤

¤ ¤

Because the 3rd row is incomplete, we return 2.

**Example 2:**

n = 8

The coins can form the following rows:

¤

¤ ¤

¤ ¤ ¤

¤ ¤

Because the 4th row is incomplete, we return 3.

## Solution

#### **Approach 1: Binary Search**

This question is easy in a sense that one could run an **exhaustive iteration** to obtain the result. That could work, except that it would run out of time when the input becomes too large. So let us take a step back to look at the problem, before rushing to the implementation.

Assume that the answer is k*k*, i.e. we've managed to complete k*k* rows of coins. These completed rows contain in total 1 + 2 + ... + k = k (k + 1)/2 ​ coins.

We could now reformulate the problem as follows:

Find the maximum k*k* such that k (k + 1)/2} <= *N*.

The problem seems to be one of those **search** problems. And instead of naive iteration, one could resort to another more efficient algorithm called [**binary search**](https://en.wikipedia.org/wiki/Binary_search_algorithm), as we can find in another similar problem called [search insert position](https://leetcode.com/articles/search-insert-position/).

**Implementation**

|  |
| --- |
| class Solution {  public int arrangeCoins(int n) {  long left = 0, right = n;  long k, curr;  while (left <= right) {  k = left + (right - left) / 2;  curr = k \* (k + 1) / 2;  if (curr == n) return (int)k;  if (n < curr) {  right = k - 1;  } else {  left = k + 1;  }  }  return (int)right;  }  } |

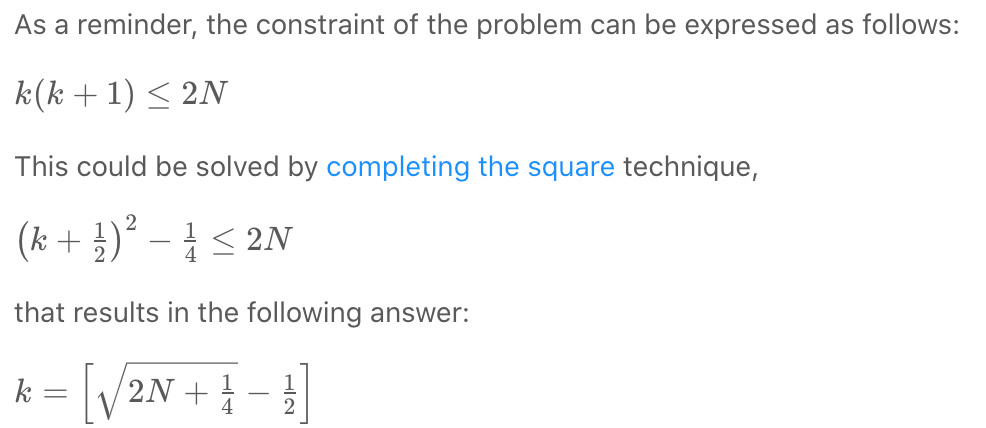
**Complexity Analysis**

* Time complexity : \mathcal{O}(\log N)O(log*N*).
* Space complexity : \mathcal{O}(1)O(1).

#### **Approach 2: Math**

If we look deeper into the formula of the problem, we could actually solve it with the help of mathematics, without using any iteration.

As a reminder, the constraint of the problem can be expressed as follows:



This could be solved by [completing the square](https://en.wikipedia.org/wiki/Completing_the_square) technique,

**Implementation**

|  |
| --- |
| class Solution {  public int arrangeCoins(int n) {  return (int)(Math.sqrt(2 \* (long)n + 0.25) - 0.5);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(1)O(1).
* Space complexity : \mathcal{O}(1)O(1).

**Binary Tree Level Order Traversal II**

Given the root of a binary tree, return the bottom-up level order traversal of its nodes' values. (i.e., from left to right, level by level from leaf to root).

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[15,7],[9,20],[3]]

**Example 2:**

**Input:** root = [1]

**Output:** [[1]]

**Example 3:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 2000].
* -1000 <= Node.val <= 1000

## Solution

#### **How to traverse the tree**

There are two general strategies to traverse a tree:

* Depth First Search (DFS)

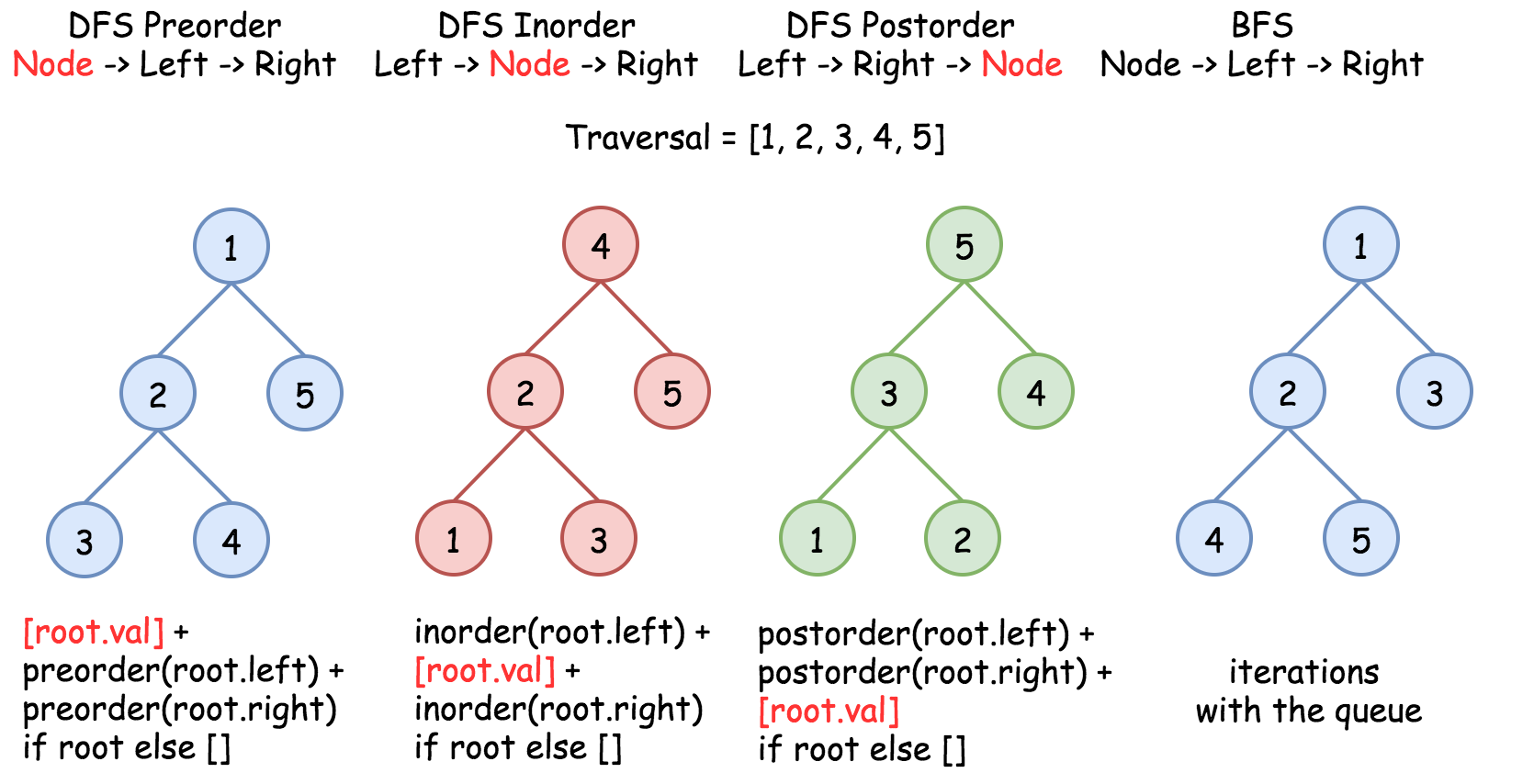
In this strategy, we adopt the depth as the priority, so that one would start from a root and reach all the way down to a certain leaf, and then back to root to reach another branch.

The DFS strategy can further be distinguished as preorder, inorder, and postorder depending on the relative order among the root node, left node, and right node.

* Breadth First Search (BFS)

We scan through the tree level by level, following the order of height, from top to bottom. The nodes on a higher level would be visited before the ones on lower levels.

In the following figure the nodes are enumerated in the order you visit them, please follow 1-2-3-4-5 to compare different strategies.



Here the problem is to implement split-level BFS traversal : [[4, 5], [2, 3], [1]]. That means we could use one of the Node->Left->Right techniques: BFS or DFS Preorder.

We already discussed [three different ways](https://leetcode.com/articles/binary-tree-right-side-view/) to implement iterative BFS traversal with the queue, and compared [iterative BFS vs. iterative DFS](https://leetcode.com/problems/deepest-leaves-sum/solution/). Let's use this article to discuss the two most simple and fast techniques:

* Recursive DFS.
* Iterative BFS with two queues.

Note, that both approaches are root-to-bottom traversals, and we're asked to provide bottom-up output. To achieve that, the final result should be reversed.

#### **Approach 1: Recursion: DFS Preorder Traversal**

**Intuition**

The first step is to ensure that the tree is not empty. The second step is to implement the recursive function helper(node, level), which takes the current node and its level as the arguments.

**Algorithm for the Recursive Function**

Here is its implementation:

* Initialize the output list levels. The length of this list determines which level is currently updated. You should compare this level len(levels) with a node level level, to ensure that you add the node on the correct level. If you're still on the previous level - add the new level by adding a new list into levels.
* Append the node value to the last level in levels.
* Process recursively child nodes if they are not None: helper(node.left / node.right, level + 1).

**Implementation**



|  |
| --- |
| class Solution {  List<List<Integer>> levels = new ArrayList<List<Integer>>();  public void helper(TreeNode node, int level) {  // start the current level  if (levels.size() == level)  levels.add(new ArrayList<Integer>());  // append the current node value  levels.get(level).add(node.val);  // process child nodes for the next level  if (node.left != null)  helper(node.left, level + 1);  if (node.right != null)  helper(node.right, level + 1);  }    public List<List<Integer>> levelOrderBottom(TreeNode root) {  if (root == null) return levels;  helper(root, 0);  Collections.reverse(levels);  return levels;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*) since each node is processed exactly once.
* Space complexity: \mathcal{O}(N)O(*N*) to keep the output structure which contains N*N* node values.

#### **Approach 2: Iteration: BFS Traversal**

**Algorithm**

The recursion above could be rewritten in the iteration form.

Let's keep each tree level in the queue structure, which typically orders elements in a FIFO (first-in-first-out) manner. In Java one could use [ArrayDeque implementation of the Queue interface](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html). In Python using [Queue structure](https://docs.python.org/3/library/queue.html) would be an overkill since it's designed for a safe exchange between multiple threads and hence requires locking which leads to a performance downgrade. In Python the queue implementation with a fast atomic append() and popleft() is [deque](https://docs.python.org/3/library/collections.html#collections.deque).

**Algorithm**

* Initialize two queues: one for the current level, and one for the next. Add root into nextLevel queue.
* While nextLevel queue is not empty:
  + Initialize the current level currLevel = nextLevel, and empty the next level nextLevel.
  + Iterate over the current level queue:
    - Append the node value to the last level in levels.
    - Add first left and then right child node into nextLevel queue.
* Return reversed levels.

**Implementation**

|  |
| --- |
| class Solution {  public List<List<Integer>> levelOrderBottom(TreeNode root) {  List<List<Integer>> levels = new ArrayList<List<Integer>>();  if (root == null) return levels;    ArrayDeque<TreeNode> nextLevel = new ArrayDeque() {{ offer(root); }};  ArrayDeque<TreeNode> currLevel = new ArrayDeque();    while (!nextLevel.isEmpty()) {  currLevel = nextLevel.clone();  nextLevel.clear();  levels.add(new ArrayList<Integer>());    for (TreeNode node : currLevel) {  // append the current node value  levels.get(levels.size() - 1).add(node.val);  // process child nodes for the next level  if (node.left != null)  nextLevel.offer(node.left);  if (node.right != null)  nextLevel.offer(node.right);  }  }    Collections.reverse(levels);  return levels;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*) since each node is processed exactly once.
* Space complexity: \mathcal{O}(N)O(*N*) to keep the output structure which contains N*N* node values.

**Ugly Number II**

Write a program to find the n-th ugly number.

Ugly numbers are**positive numbers** whose prime factors only include 2, 3, 5.

**Example:**

**Input:** n = 10

**Output:** 12

**Explanation:** 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers.

**Note:**

1. 1 is typically treated as an ugly number.
2. n **does not exceed 1690**.

   Hide Hint #1

The naive approach is to call isUgly for every number until you reach the nth one. Most numbers are *not* ugly. Try to focus your effort on generating only the ugly ones.

   Hide Hint #2

An ugly number must be multiplied by either 2, 3, or 5 from a smaller ugly number.

   Hide Hint #3

The key is how to maintain the order of the ugly numbers. Try a similar approach of merging from three sorted lists: L1, L2, and L3.

   Hide Hint #4

Assume you have Uk, the kth ugly number. Then Uk+1 must be Min(L1 \* 2, L2 \* 3, L3 \* 5).

## Solution

#### **Two levels of optimisation**

Let's imagine that the problem is solved somehow for the number n and we've put the solution directly in nthUglyNumber method of the Solution class.

Now let's check the context: there are 596 test cases, for the most of them n is larger than 50, and n is known to be smaller than 1691.

Hence instead of computing 596 \times 50 = 29800596×50=29800 ugly numbers in total, one could precompute all 1690 numbers, and significantly speed up the submission.

How to precompute? Use another class Ugly with all computations in the constructor and then declare Ugly instance as a static variable of Solution class.

Now let's consider two different approaches to perform the preliminary computations.

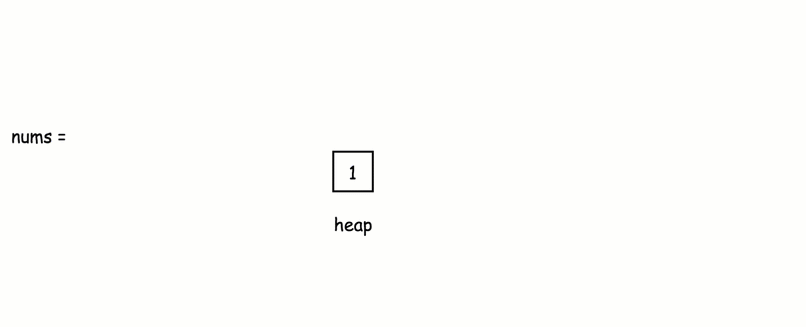
#### **Approach 1: Heap**

**Intuition**

Let's start from the heap which contains just one number: 1.  
To compute next ugly numbers, pop 1 from the heap and push instead three numbers: 1 \times 21×2, 1 \times 31×3, and 1 \times 51×5.

Now the smallest number in the heap is 2. To compute next ugly numbers, pop 2 from the heap and push instead three numbers: 2 \times 22×2, 2 \times 32×3, and 2 \times 52×5.

One could continue like this to compute first 1690 ugly numbers. At each step, pop the smallest ugly number k from the heap, and push instead three ugly numbers: k \times 2*k*×2, k \times 3*k*×3, and k \times 5*k*×5.



**Algorithm**

* Precompute 1690 ugly numbers:
  + Initiate array of precomputed ugly numbers nums, heap heap and hashset seen to track all elements already pushed in the heap in order to avoid duplicates.
  + Make a loop of 1690 steps. At each step:
    - Pop the smallest element k out of heap and add it into the array of precomputed ugly numbers.
    - Push 2k, 3k and 5k in the heap if they are not yet in the hashset. Update the hashset of seen ugly numbers as well.
* Retrieve needed ugly number from the array of precomputed numbers.

**Implementation**

|  |
| --- |
| class Ugly {  public int[] nums = new int[1690];  Ugly() {  HashSet<Long> seen = new HashSet();  PriorityQueue<Long> heap = new PriorityQueue<Long>();  seen.add(1L);  heap.add(1L);  long currUgly, newUgly;  int[] primes = new int[]{2, 3, 5};  for(int i = 0; i < 1690; ++i) {  currUgly = heap.poll();  nums[i] = (int)currUgly;  for(int j : primes) {  newUgly = currUgly \* j;  if (!seen.contains(newUgly)) {  seen.add(newUgly);  heap.add(newUgly);  }  }  }  }  }  class Solution {  public static Ugly u = new Ugly();  public int nthUglyNumber(int n) {  return u.nums[n - 1];  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(1)O(1) to retrieve preliminary computed ugly number, and more than 12 \times 10^612×106 operations for preliminary computations. Let's estimate the number of operations needed for the preliminary computations. For loop here has 1690 steps, and each step performs 1 pop, not more than 3 pushes and 3 contains / in operations for the hashset. Pop and push have logarithmic time complexity and hence much cheaper than the linear search, so let's estimate only the last term. This arithmetic progression is easy to estimate:

1 + 2 + 3 + ... + 1690 \times 3 = \frac{(1 + 1690 \times 3) \times 1690 \times 3}{2} > 4.5 \times 1690^21+2+3+...+1690×3=2(1+1690×3)×1690×3​>4.5×16902

* Space complexity : constant space to keep an array of 1690 ugly numbers, the heap of not more than 1690 \times 21690×2 elements and the hashset of not more than 1690 \times 31690×3 elements.

#### **Approach 2: Dynamic Programming**

**Intuition**

Preliminary computations in Approach 1 are quite heavy, and could be optimised with dynamic programming.

Let's start from the array of ugly numbers which contains just one number - 1. Let's use three pointers i\_2*i*2​, i\_3*i*3​ and i\_5*i*5​, to mark the last ugly number which was multiplied by 2, 3 and 5, correspondingly.

The algorithm is straightforward: choose the smallest ugly number among 2 \times \textrm{nums}[i\_2]2×nums[*i*2​], 3 \times \textrm{nums}[i\_3]3×nums[*i*3​], and 5 \times \textrm{nums}[i\_5]5×nums[*i*5​] and add it into the array. Move the corresponding pointer by one step. Repeat till you'll have 1690 ugly numbers.



**Algorithm**

* Precompute 1690 ugly numbers:
  + Initiate array of precomputed ugly numbers nums and three pointers i2, i3 and i5 to track the index of the last ugly number used to produce the next ones.
  + Make a loop of 1690 steps. At each step:
    - Choose the smallest number among nums[i2] \* 2, nums[i3] \* 3, and nums[i5] \* 5 and add it into nums.
    - Move by one the pointer which corresponds to the "ancestor" of the added number.
* Retrieve needed ugly number from the array of precomputed numbers.

**Implementation**

|  |
| --- |
| class Ugly {  public int[] nums = new int[1690];  Ugly() {  nums[0] = 1;  int ugly, i2 = 0, i3 = 0, i5 = 0;  for(int i = 1; i < 1690; ++i) {  ugly = Math.min(Math.min(nums[i2] \* 2, nums[i3] \* 3), nums[i5] \* 5);  nums[i] = ugly;  if (ugly == nums[i2] \* 2) ++i2;  if (ugly == nums[i3] \* 3) ++i3;  if (ugly == nums[i5] \* 5) ++i5;  }  }  }  class Solution {  public static Ugly u = new Ugly();  public int nthUglyNumber(int n) {  return u.nums[n - 1];  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(1)O(1) to retrieve preliminary computed ugly number, and about 1690 \times 5 = 84501690×5=8450 operations for preliminary computations.
* Space complexity : constant space to keep an array of 1690 ugly numbers.

**Island Perimeter**

**Solution**

You are given row x col grid representing a map where grid[i][j] = 1 represents land and grid[i][j] = 0 represents water.

Grid cells are connected **horizontally/vertically** (not diagonally). The grid is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells).

The island doesn't have "lakes", meaning the water inside isn't connected to the water around the island. One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

**Example 1:**



**Input:** grid = [[0,1,0,0],[1,1,1,0],[0,1,0,0],[1,1,0,0]]

**Output:** 16

**Explanation:** The perimeter is the 16 yellow stripes in the image above.

**Example 2:**

**Input:** grid = [[1]]

**Output:** 4

**Example 3:**

**Input:** grid = [[1,0]]

**Output:** 4

**Constraints:**

* row == grid.length
* col == grid[i].length
* 1 <= row, col <= 100
* grid[i][j] is 0 or 1.

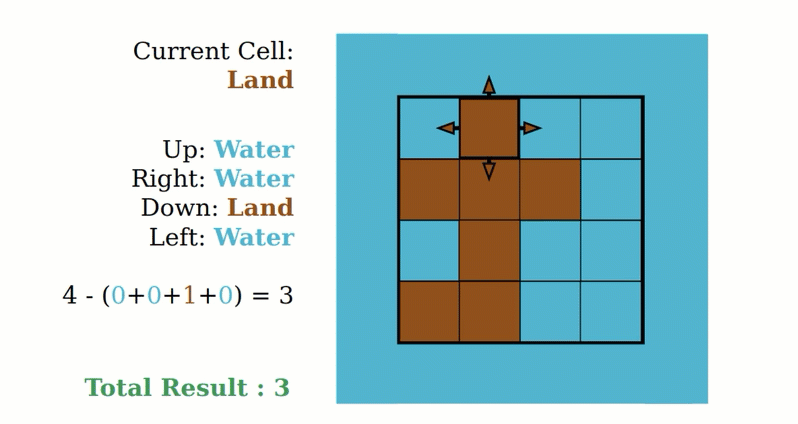
## Solution

#### **Approach 1: Simple Counting**

**Intuition**

Go through every cell on the grid and whenever you are at cell 1 (land cell), look for surrounding (UP, RIGHT, DOWN, LEFT) cells. A land cell without any surrounding land cell will have a perimeter of 4. Subtract 1 for each surrounding land cell.

When you are at cell 0 (water cell), you don't need to do anything. Just proceed to another cell.



|  |
| --- |
| public class Solution {  public int islandPerimeter(int[][] grid) {    int rows = grid.length;  int cols = grid[0].length;    int up, down, left, right;  int result = 0;    for (int r = 0; r < rows; r++) {  for (int c = 0; c < cols; c++) {  if (grid[r][c] == 1) {  if (r == 0) { up = 0; }  else { up = grid[r-1][c]; }    if (c == 0) { left = 0; }  else { left = grid[r][c-1]; }    if (r == rows-1) { down = 0; }  else { down = grid[r+1][c]; }    if (c == cols-1) { right = 0; }  else { right = grid[r][c+1]; }    result += 4-(up+left+right+down);  }  }  }  return result;  }  } |

**Complexity Analysis**

* Time complexity : O(mn)*O*(*mn*) where m*m* is the number of rows of the grid and n*n* is the number of columns of the grid. Since two for loops go through all the cells on the grid, for a two-dimensional grid of size m\times n*m*×*n*, the algorithm would have to check mn*mn* cells.
* Space complexity : O(1)*O*(1). Only the result variable is updated and there is no other space requirement.

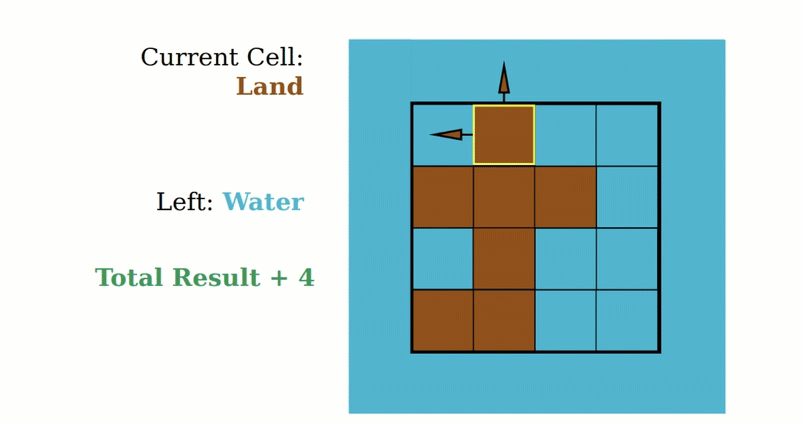
#### **Approach 2: Better Counting**

Approach 2 has the same time and space complexity as Approach 1. Even though they have the same time and space complexities, Approach 2 is slightly more efficient than the Approach 1. Rather than checking 4 surrounding neighbors, we only need to check two neighbors (*LEFT* and *UP*) in Approach 2.

**Intuition**

Since we are traversing the grid from left to right, and from top to bottom, for each land cell we are currently at, we only need to check whether the LEFT and UP cells are land cells with a slight modification on previous approach.

* As you go through each cell on the grid, treat all the land cells as having a perimeter of 4 and add that up to the accumulated result.
* If that land cell has a neighboring land cell, remove 2 sides (one from each land cell) which will be touching between these two cells.
  + If your current land cell has a UP land cell, subtract 2 from your accumulated result.
  + If your current land cell has a LEFT land cell, subtract 2 from your accumulated result.



|  |
| --- |
| class Solution {  public int islandPerimeter(int[][] grid) {  int rows = grid.length;  int cols = grid[0].length;    int result = 0;  for (int r = 0; r < rows; r++) {  for (int c = 0; c < cols; c++) {  if (grid[r][c] == 1) {  result += 4;    if (r > 0 && grid[r-1][c] == 1) {  result -= 2;  }    if (c > 0 && grid[r][c-1] == 1) {  result -= 2;  }  }  }  }    return result;  }  } |

**Complexity Analysis**

* Time complexity : O(mn)*O*(*mn*) where m*m* is the number of rows of the grid and n*n* is the number of columns of the grid. Since two for loops go through all the cells on the grid, for a two-dimensional grid of size m\times n*m*×*n*, the algorithm would have to check mn*mn* cells.
* Space complexity : O(1)*O*(1). Only the result variable is updated and there is no other space requirement.

**Maximum Width of Binary Tree**

Given a binary tree, write a function to get the maximum width of the given tree. The maximum width of a tree is the maximum width among all levels.

The width of one level is defined as the length between the end-nodes (the leftmost and right most non-null nodes in the level, where the null nodes between the end-nodes are also counted into the length calculation.

It is **guaranteed** that the answer will in the range of 32-bit signed integer.

**Example 1:**

**Input:**

1

/ \

3 2

/ \ \

5 3 9

**Output:** 4

**Explanation:** The maximum width existing in the third level with the length 4 (5,3,null,9).

**Example 2:**

**Input:**

1

/

3

/ \

5 3

**Output:** 2

**Explanation:** The maximum width existing in the third level with the length 2 (5,3).

**Example 3:**

**Input:**

1

/ \

3 2

/

5

**Output:** 2

**Explanation:** The maximum width existing in the second level with the length 2 (3,2).

**Example 4:**

**Input:**

1

/ \

3 2

/ \

5 9

/ \

6 7

**Output:** 8

**Explanation:**The maximum width existing in the fourth level with the length 8 (6,null,null,null,null,null,null,7).

**Constraints:**

* The given binary tree will have between 1 and 3000 nodes.

## Solution

#### **Overview**

The problem defines the concept of **width** for the binary tree. In essence, it is about binary tree traversal, since we need to traverse the tree in order to measure its width.

As one would probably know, the common strategies to traverse a binary tree are Breadth-First Search (a.k.a. BFS) and Depth-First Search (a.k.a. DFS). Furthermore, the DFS strategy can be distinguished as preorder DFS, inorder DFS and postorder DFS, depending on the relative order of visit among the node itself and its child nodes.

If one is not familiar with the concepts of BFS and DFS, we have an Explore card called [Queue & Stack](https://leetcode.com/explore/learn/card/queue-stack/) where we cover the [BFS traversal](https://leetcode.com/explore/learn/card/queue-stack/231/practical-application-queue/) as well as the [DFS traversal](https://leetcode.com/explore/learn/card/queue-stack/232/practical-application-stack/). Hence, in this article, we won't repeat ourselves on these concepts.

**Intuition**

The key to solve the problem though lie on how we **index** the nodes that are on the same level.

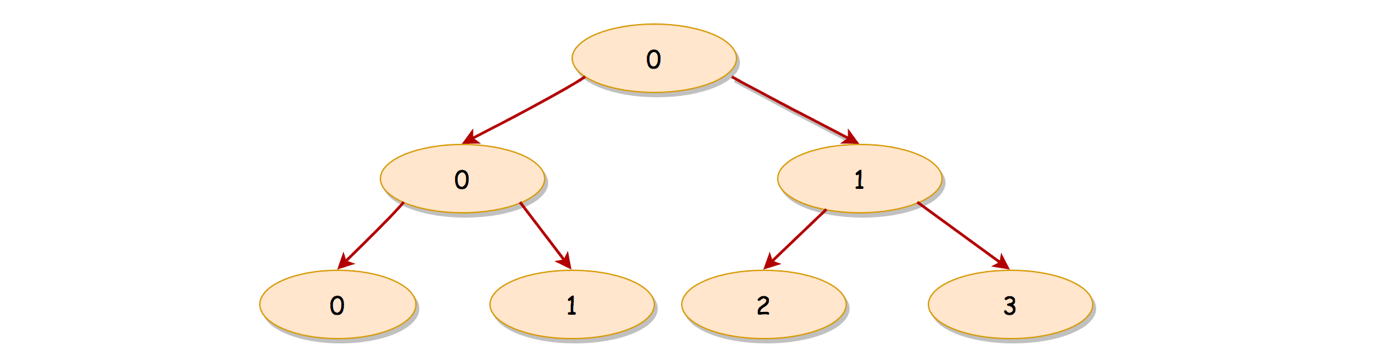
Suppose that the indices for the first and the last nodes of one particular level are C\_1*C*1​ and C\_n*Cn*​ respectively, we could then calculate the width of this level as C\_n - C\_i + 1*Cn*​−*Ci*​+1.

Now, let us try to come up with a schema to index the nodes, so that the problem can be solved easily with the above formula.

As we know, for a full binary tree, the number of nodes double at each level, since each parent node has two child nodes. Naturally, the range of our node index would double as well.

If the index of a parent node is C\_i*Ci*​, accordingly we can define the index of its **left** child node as 2\cdot C\_i2⋅*Ci*​ and the index of its **right** child node as 2 \cdot C\_i + 12⋅*Ci*​+1.

In the following graph, we show an example of how the index works for a full binary tree, where on each node we label its index rather than its value.



With the above indexing schema, we manage to assign a unique index for each node on the same level, and in addition there is no gap among all the indices if it is a full binary tree.

For a non-full binary tree, the relationship between the indices of a parent and its child node still holds.

Now that we have an indexing schema, all we need to do is to assign an index for each node in the tree. Once it is done, we can calculate the width for each level, and finally we could return the maximal value among them as the solution.

Voila. This is the key insight to solve the problem. With this hint, we believe that one could definitely come up with some solutions.

As a spoiler alert, we will cover how to implement different solutions with BFS and DFS traversal strategies in the remaining sections.

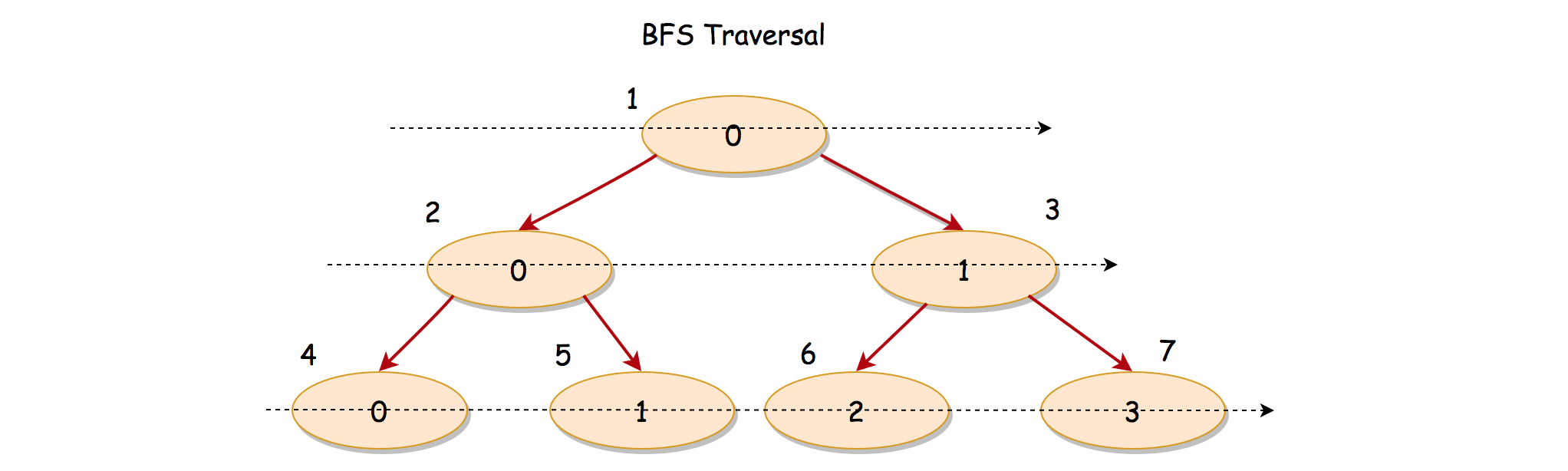
#### **Approach 1: BFS Traversal**

**Intuition**

Naturally, one might resort to the BFS traversal. After all, the width is measured among the nodes on the same level. So let us get down to the BFS traversal first.

There are several ways to implement the BFS traversal. Almost all of them share a common point, i.e. using the queue data structure to maintain the order of visits.

In brief, we push the nodes into the queue level by level. As a result, the priorities of visiting would roll out from top to down and from left to right, due to the FIFO (First-In First-Out) principle of the queue data structure, i.e. the element that enters the queue first would exit first as well.



In the above graph, we show an example of BFS traversal on a full binary tree where we indicate the global order of visiting along with each node.

**Algorithm**

Here are a few steps to implement a solution with the BFS traversal.

* First of all, we create a queue data structure, which would be used to hold elements of tuple as (node, col\_index), where the node is the tree node and the col\_index is the corresponding index that is assigned to the node based on our indexing schema. Also, we define a global variable called max\_width which holds the maximal width that we've seen so far.
* Then we append the root node along with its index 0, to kick off the BFS traversal.
* The BFS traversal is basically an iteration over the elements of queue. We visit the nodes level by level until there are no more elements in the queue.
  + At the end of each level, we use the indices of the first and the last elements on the same level, in order to obtain the width of the level.
* At the end of BFS traversal, we then return the maximal width that we've seen over all levels.

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Solution {  public int widthOfBinaryTree(TreeNode root) {  if (root == null)  return 0;  // queue of elements [(node, col\_index)]  LinkedList<Pair<TreeNode, Integer>> queue = new LinkedList<>();  Integer maxWidth = 0;  queue.addLast(new Pair<>(root, 0));  while (queue.size() > 0) {  Pair<TreeNode, Integer> head = queue.getFirst();  // iterate through the current level  Integer currLevelSize = queue.size();  Pair<TreeNode, Integer> elem = null;  for (int i = 0; i < currLevelSize; ++i) {  elem = queue.removeFirst();  TreeNode node = elem.getKey();  if (node.left != null)  queue.addLast(new Pair<>(node.left, 2 \* elem.getValue()));  if (node.right != null)  queue.addLast(new Pair<>(node.right, 2 \* elem.getValue() + 1));  }  // calculate the length of the current level,  // by comparing the first and last col\_index.  maxWidth = Math.max(maxWidth, elem.getValue() - head.getValue() + 1);  }  return maxWidth;  }  } |

**Note:** in the above implementation, we use the size of the queue as a delimiter to determine the boundary between each levels.

One could also use a specific dummy element as a marker to separate nodes of different levels in the queue.

**Complexity Analysis**

Let N*N* be the total number of nodes in the input tree.

* Time Complexity: \mathcal{O}(N)O(*N*)
  + We visit each node once and only once. And at each visit, it takes a constant time to process.
* Space Complexity: \mathcal{O}(N)O(*N*)
  + We used a queue to maintain the nodes along with its indices, which is the main memory consumption of the algorithm.
  + Due to the nature of BFS, at any given moment, the queue holds no more than two levels of nodes. In the worst case, a level in a full binary tree contains at most half of the total nodes (i.e. \frac{N}{2}2*N*​), i.e. this is also the level where the leaf nodes reside.
  + Hence, the overall space complexity of the algorithm is \mathcal{O}(N)O(*N*).

#### **Approach 2: DFS Traversal**

**Intuition**

Although it is definitely more intuitive to implement a solution with BFS traversal, it is not impossible to do it with DFS.

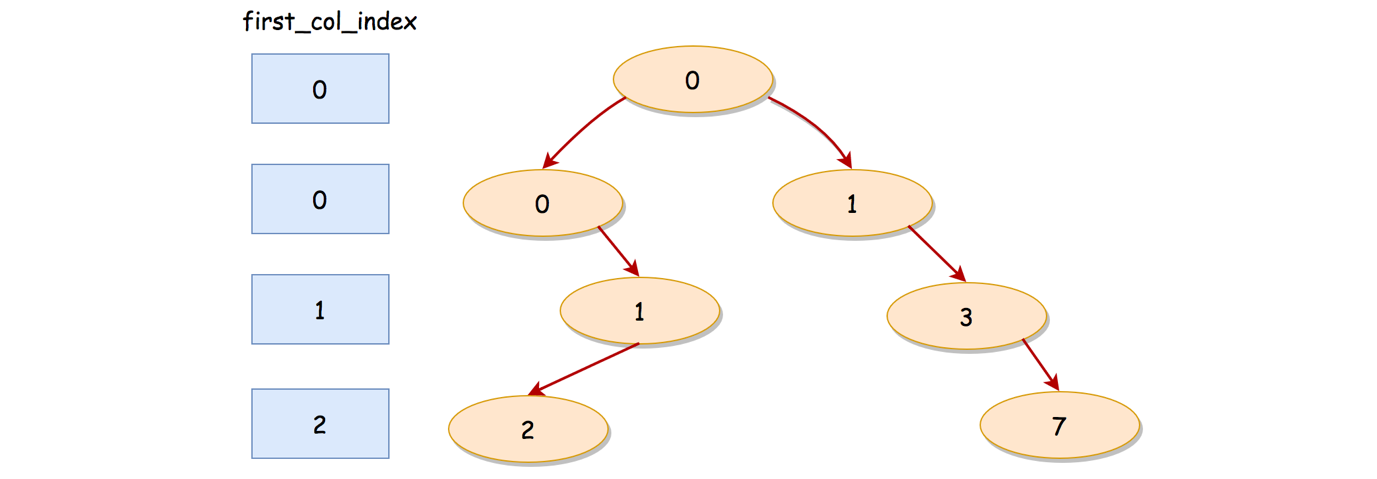
It might sound twisted, but we don't need to visit the nodes strictly in the order of BFS. All we need is to compare the indices between the first and the last elements of the same level.

We could build a table that records the indices of nodes grouped by level. Then we could scan the indices level by level to obtain the **maximal** difference among them, which is also the width of the level.

With the above idea, as we can see, any traversal will do, including the BFS and DFS.

Better yet, we don't need to keep the indices of the entire level, but the first and the last index.

We could use the table to keep **only** the index of the first element for each level, i.e. depth -> first\_col\_index, which we illustrate in the following graph.



Along with the traversal, we could compare the index of every node with the corresponding first index of its level (i.e. first\_col\_index).

Rather than keeping all the indices in the table, we save time and space by keeping only the index of the first element per level.

**Algorithm**

The tricky part is how we can obtain the index for the first element of each level.

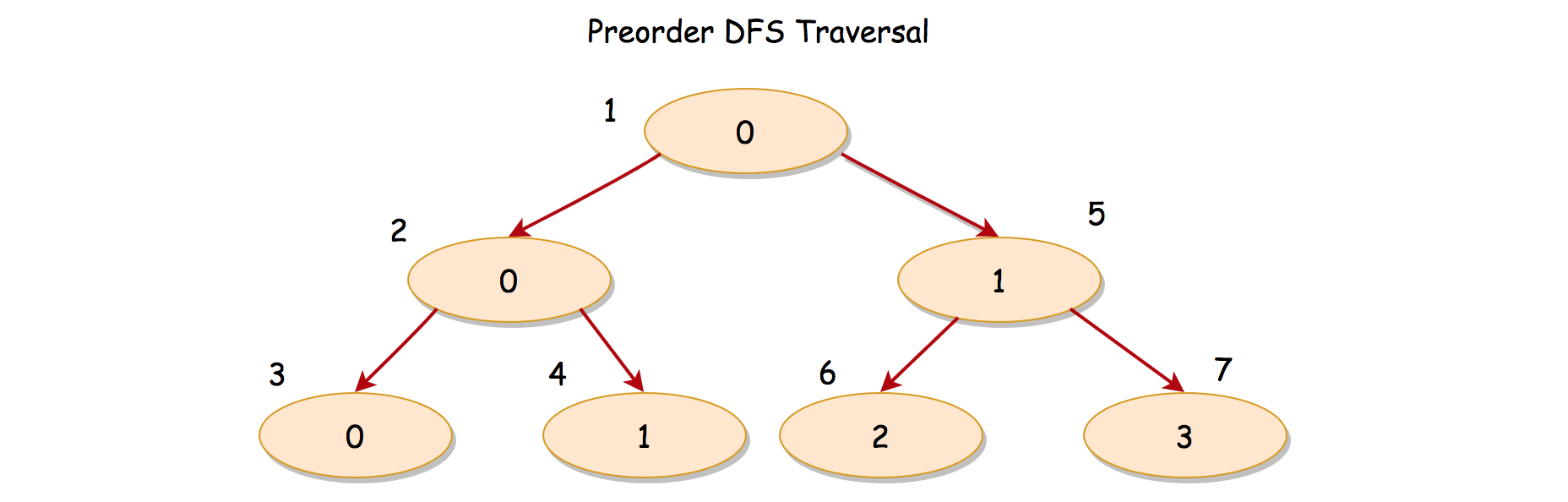
As we discussed before, we use a table with depth of the node as the key and the index of the first element for that depth (level) as the value.

If we can make sure that we visit the first element of a level before the rest of elements on that level, we then can easily populate the table along with the traversal.

In fact, a DFS traversal can assure the above priority that we desire. Even better, it could be either preorder, inorder or postorder DFS traversal, as long as we **prioritize** the visit of the left child node over the right child node.

Although in principle DFS prioritizes depth over breadth, it could also ensure the level-wise priority. By visiting the left node before the right child node in DFS traversal, we can ensure that the nodes that lean more to the left got visited earlier.

We showcase a **preorder DFS** traversal, with an example in the following graph:



We label each node with a number that indicates the global order of visit. As one can see, the nodes at the same level do get visited from left to right. For instance, on the second level, the first node would be visited at the step 2, while the next node at the same level would be visited at the step 5.

We give some sample implementations of DFS in the following.

|  |
| --- |
| class Solution {  private Integer maxWidth = 0;  private HashMap<Integer, Integer> firstColIndexTable;  protected void DFS(TreeNode node, Integer depth, Integer colIndex) {  if (node == null)  return;  // initialize the value, for the first seen colIndex per level  if (!firstColIndexTable.containsKey(depth)) {  firstColIndexTable.put(depth, colIndex);  }  Integer firstColIndex = firstColIndexTable.get(depth);  maxWidth = Math.max(this.maxWidth, colIndex - firstColIndex + 1);  // Preorder DFS. Note: it is important to put the priority on the left child  DFS(node.left, depth + 1, 2 \* colIndex);  DFS(node.right, depth + 1, 2 \* colIndex + 1);  }  public int widthOfBinaryTree(TreeNode root) {  // table contains the first col\_index for each level  this.firstColIndexTable = new HashMap<Integer, Integer>();  // start from depth = 0, and colIndex = 0  DFS(root, 0, 0);  return this.maxWidth;  }  } |

**Complexity Analysis**

Let N*N* be the total number of nodes in the input tree.

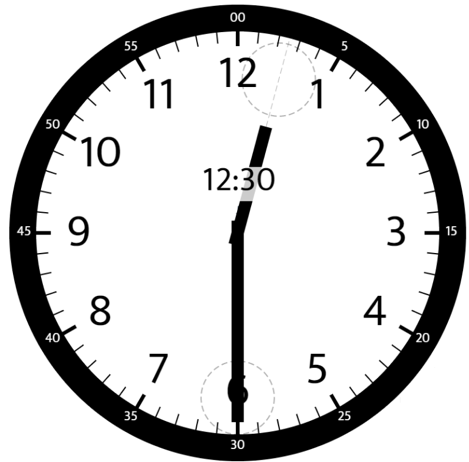
* Time Complexity: \mathcal{O}(N)O(*N*).
  + Similar to the BFS traversal, we visit each node once and only once in DFS traversal. And each visit takes a constant time to process as well.
* Space Complexity: \mathcal{O}(N)O(*N*)
  + Unlike the BFS traversal, we used an additional table to keep the index for the first element per level. In the worst case where the tree is extremely skewed, there could be as many levels as the number of nodes. As a result, the space complexity of the table would be \mathcal{O}(N)O(*N*).
  + Since we implement DFS traversal with recursion which would incur some additional memory consumption in the function call stack, we need to take this into account for the space complexity.
  + The consumption of function stack is proportional to the depth of recursion. Again, in the same worst case above, where the tree is extremely skewed, the depth of the recursion would be equal to the number of nodes in the tree. Therefore, the space complexity of the function stack would be \mathcal{O}(N)O(*N*).
  + To sum up, the overall space complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N)O(*N*)+O(*N*)=O(*N*).

**Angle Between Hands of a Clock**

**Solution**

Given two numbers, hour and minutes. Return the smaller angle (in degrees) formed between the hour and the minute hand.

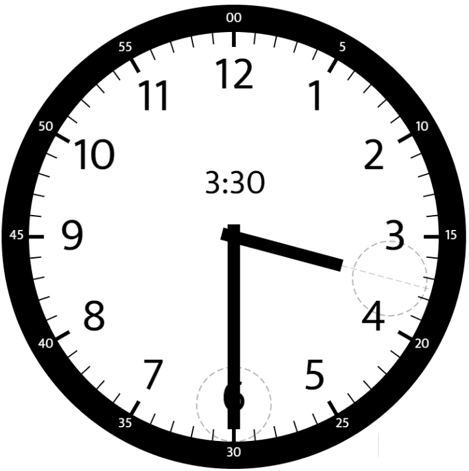
**Example 1:**



**Input:** hour = 12, minutes = 30

**Output:** 165

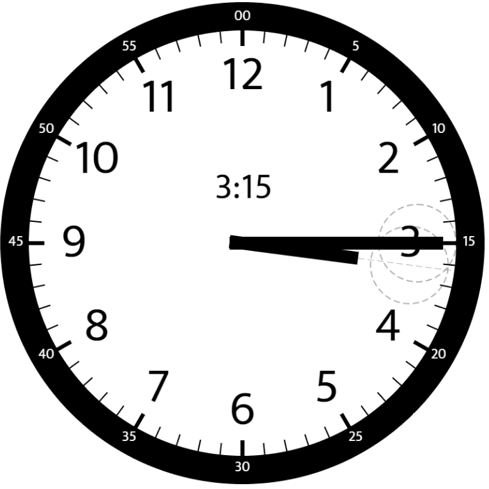
**Example 2:**



**Input:** hour = 3, minutes = 30

**Output:** 75

**Example 3:**

****

**Input:** hour = 3, minutes = 15

**Output:** 7.5

**Example 4:**

**Input:** hour = 4, minutes = 50

**Output:** 155

**Example 5:**

**Input:** hour = 12, minutes = 0

**Output:** 0

**Constraints:**

* 1 <= hour <= 12
* 0 <= minutes <= 59
* Answers within 10^-5 of the actual value will be accepted as correct.

Hide Hint #1

The tricky part is determining how the minute hand affects the position of the hour hand.

   Hide Hint #2

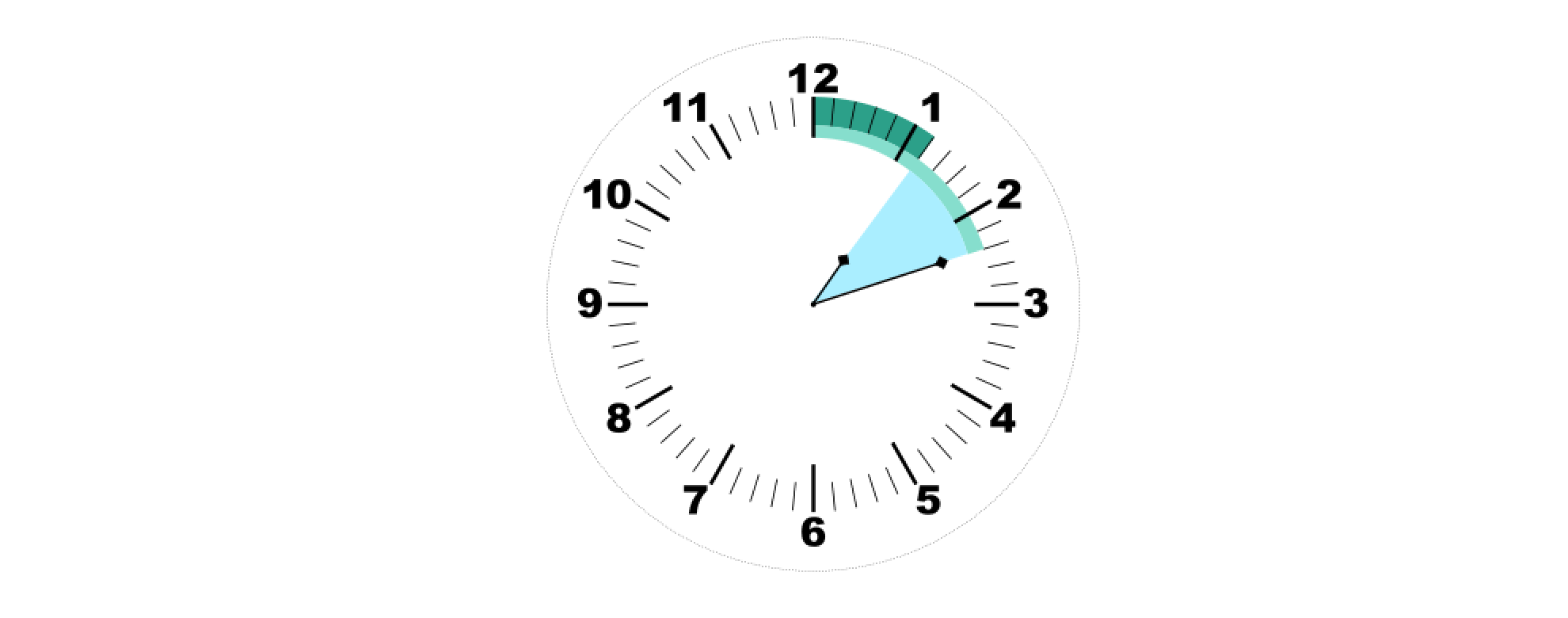
Calculate the angles separately then find the difference.

## Solution

#### **Approach 1: Math**

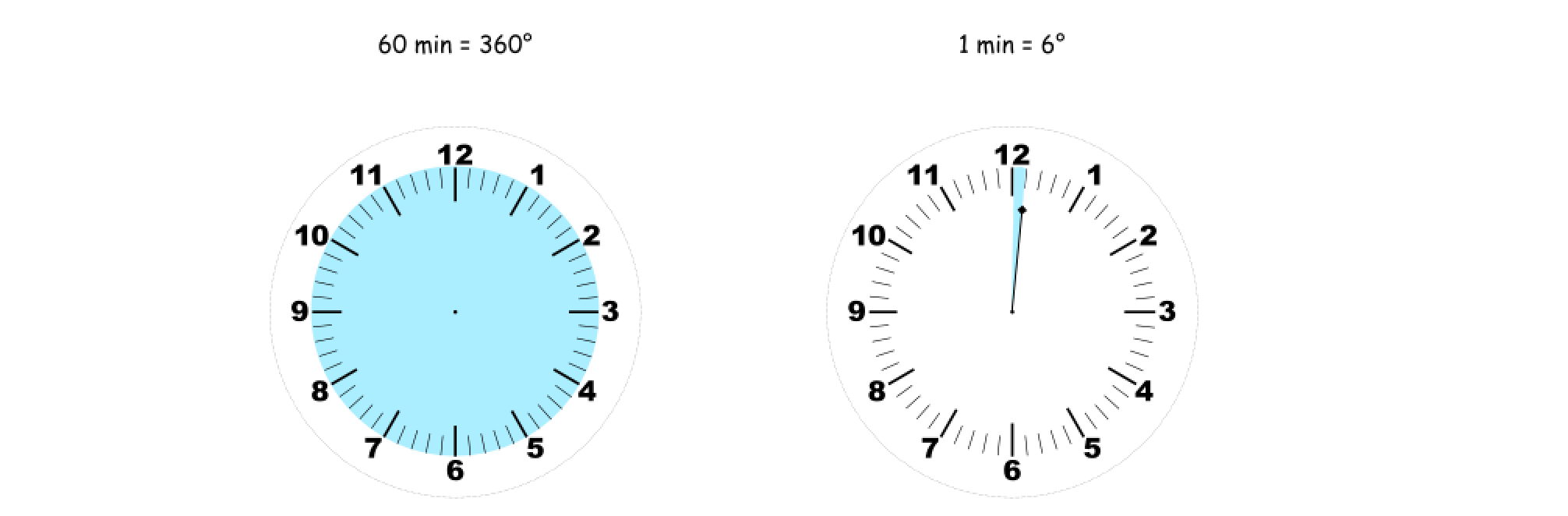
**Intuition**

The idea is to calculate separately the angles between 0-minutes vertical line and each hand. The answer is the difference between these two angles.

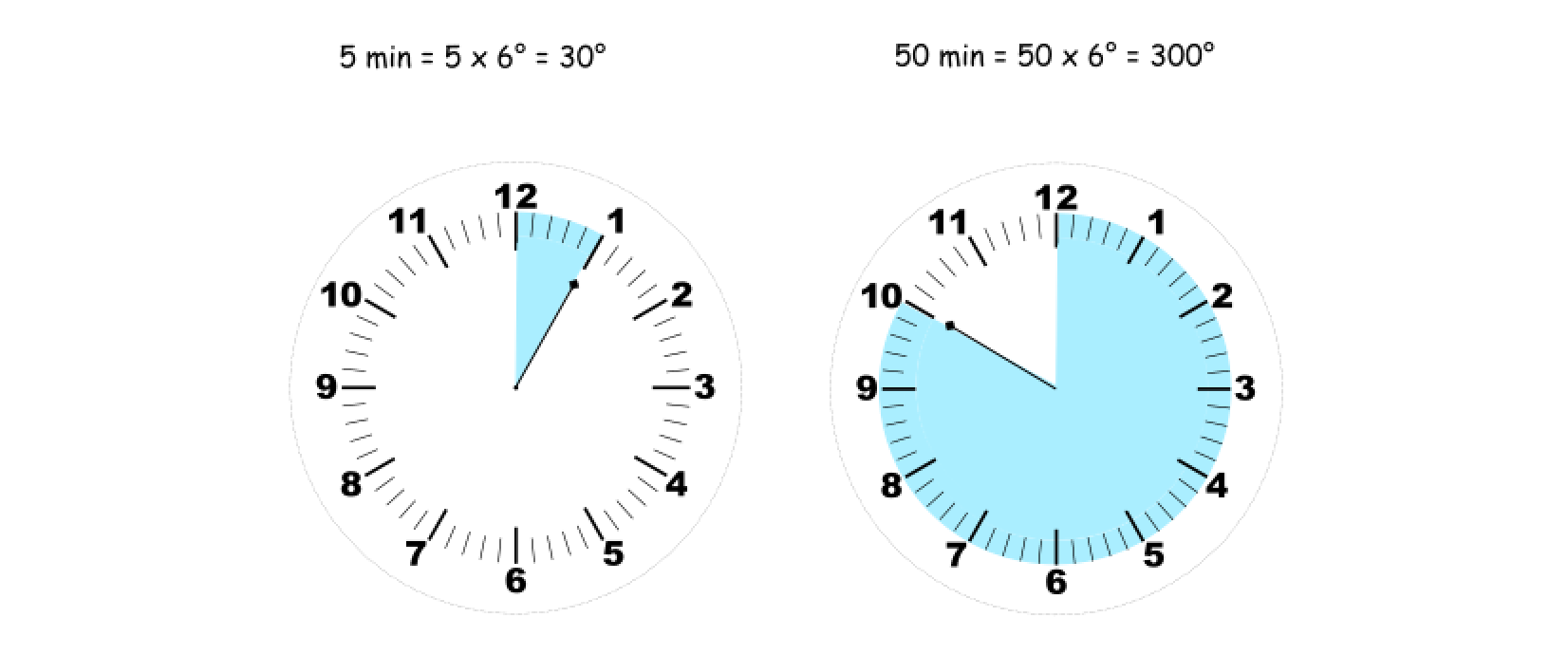


**Minute Hand Angle**

Let's start from the minute hand. The whole circle is equal to 360°360° or 60 minutes, i.e. minute hand moves 1 \text{ min} = 360° / 60 = 6°1 min=360°/60=6° degree at each minute.

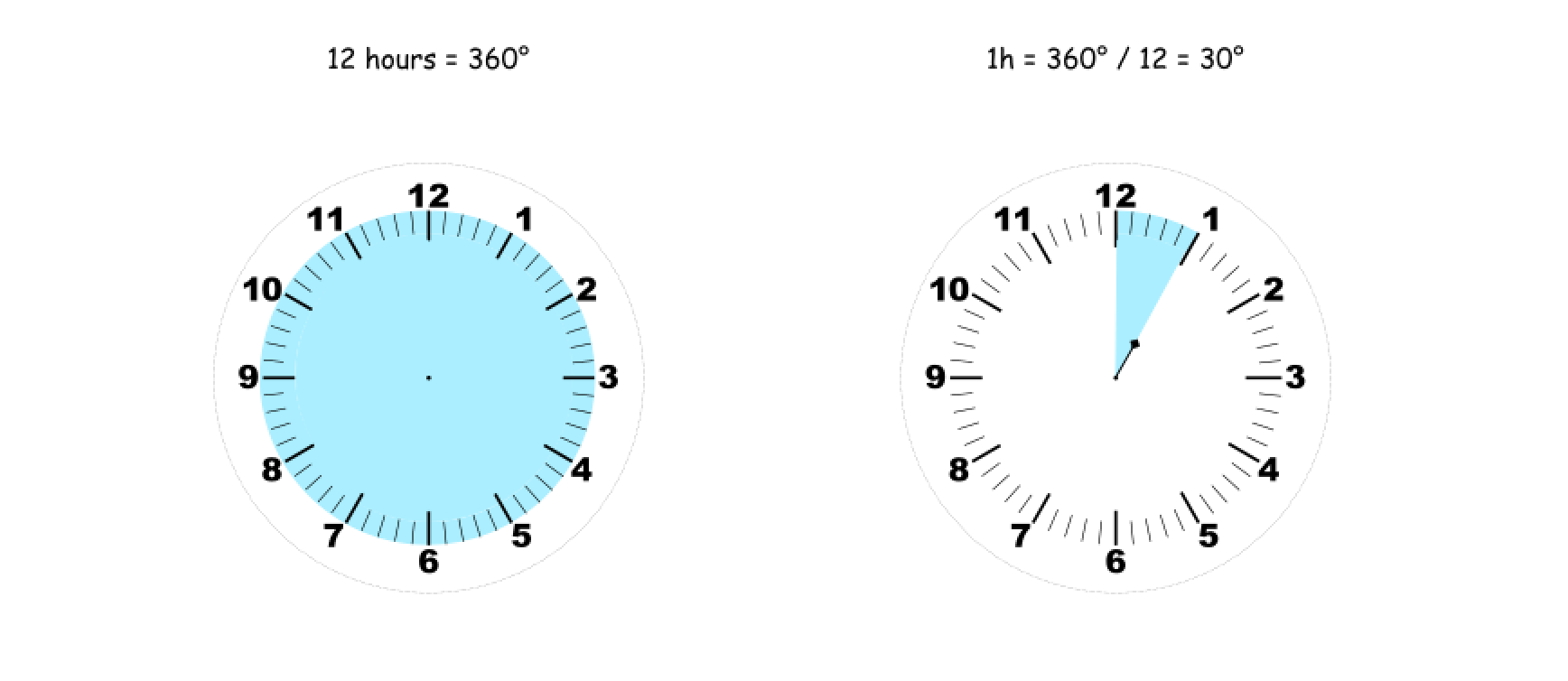


Now one could easily find an angle between 0-minutes vertical line and a minute hand: \text{minutes\\_angle} = \text{minutes} \times 6°minutes\_angle=minutes×6°.

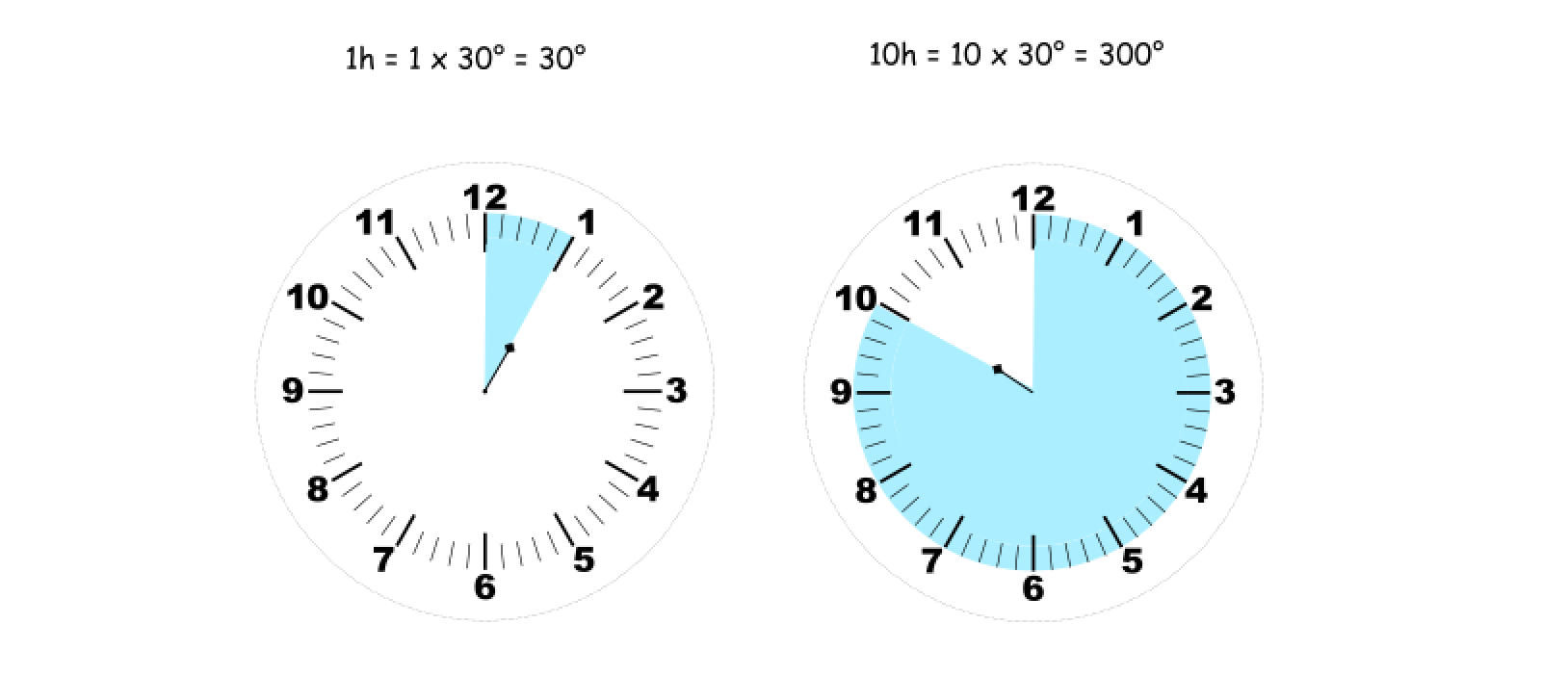


**Hour Hand Angle**

Similarly with the minute hand angle, the whole circle is equal to 360°360° or 12 hours, hence for each hour, the hour hand moves 1 \text{h} = 360° / 12 = 30°1h=360°/12=30° degree.



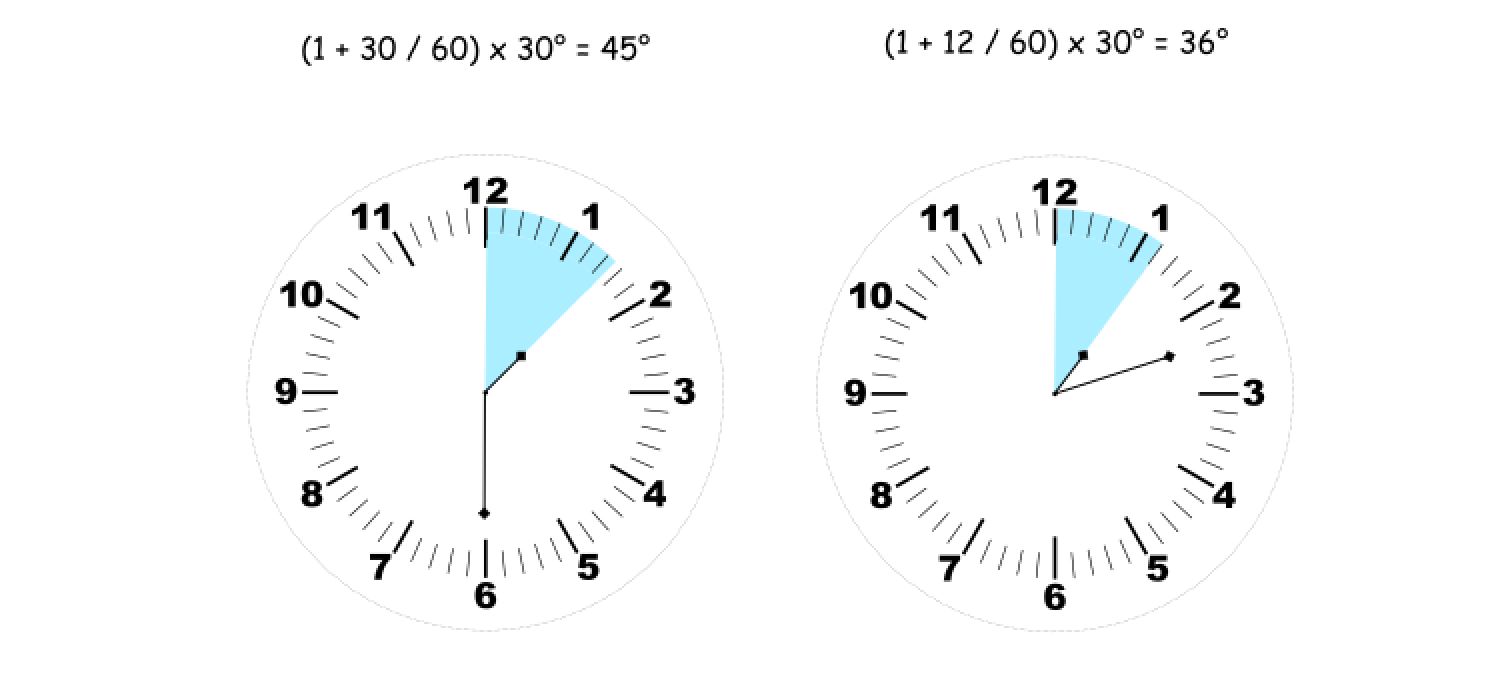
Now for the "minutes = 0" case one could easily find an angle between 12-hour vertical line and an hour hand: \text{hour\\_angle} = \text{hour} \times 30°hour\_angle=hour×30°.



Note that for 12-hour the actual angle is zero, therefore the expression has to be corrected \text{hour\\_angle} = (\text{hour mod } 12) \times 30°hour\_angle=(hour mod 12)×30°.

In a more general case where "minutes > 0", one has to take into account an additional movement of hour hand: it doesn't jump between the integer values but follows the movement of minute hand as well

\text{hour\\_angle} = \left(\text{hour mod } 12 + \text{minutes} / 60 \right)\times 30°hour\_angle=(hour mod 12+minutes/60)×30°



**Algorithm**

* Initialize the constants: one\_min\_angle = 6, one\_hour\_angle = 30.
* The angle between minute hand and 0-minutes vertical line is minutes\_angle = one\_min\_angle \* minutes.
* The angle between hour hand and 12-hour vertical line is hour\_angle = (hour % 12 + minutes / 60) \* one\_hour\_angle.
* Find the difference: diff = abs(hour\_angle - minutes\_angle).
* Return the smallest angle: min(diff, 360 - diff).

**Implementation**

|  |
| --- |
| class Solution {  public double angleClock(int hour, int minutes) {  int oneMinAngle = 6;  int oneHourAngle = 30;  double minutesAngle = oneMinAngle \* minutes;  double hourAngle = (hour % 12 + minutes / 60.0) \* oneHourAngle;  double diff = Math.abs(hourAngle - minutesAngle);  return Math.min(diff, 360 - diff);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(1)O(1).
* Space complexity : \mathcal{O}(1)O(1).

**Single Number III**

Given an integer array nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once. You can return the answer in **any order**.

**Follow up:**Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

**Example 1:**

**Input:** nums = [1,2,1,3,2,5]

**Output:** [3,5]

**Explanation:**  [5, 3] is also a valid answer.

**Example 2:**

**Input:** nums = [-1,0]

**Output:** [-1,0]

**Example 3:**

**Input:** nums = [0,1]

**Output:** [1,0]

**Constraints:**

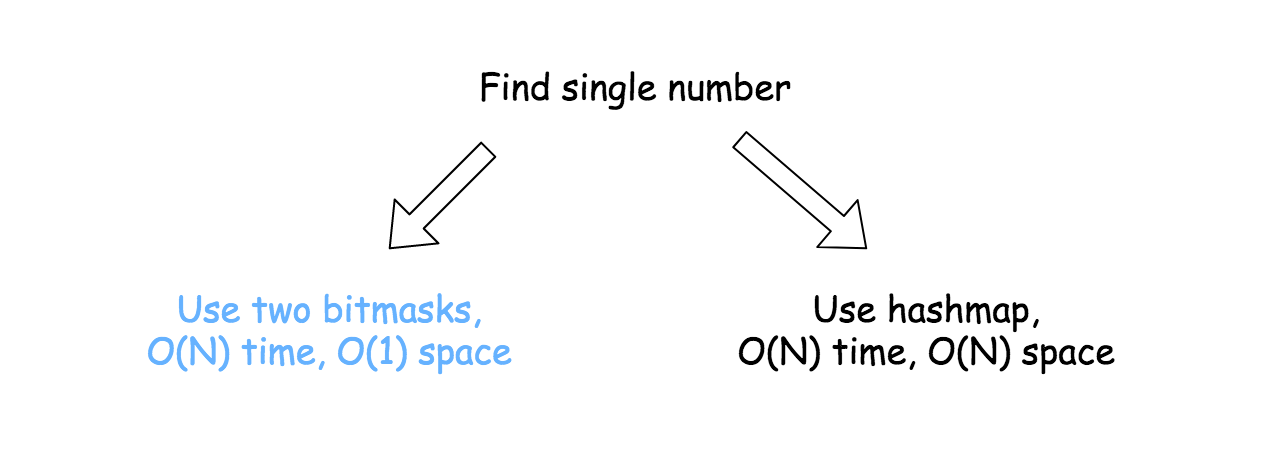
* 2 <= nums.length <= 3 \* 104
* -231 <= nums[i] <= 231 - 1
* Each integer in nums will appear twice, only two integers will appear once.

## Solution

#### **Overview**

The problem could be solved in \mathcal{O}(N)O(*N*) time and \mathcal{O}(N)O(*N*) space by using hashmap.

To solve the problem in a constant space is a bit tricky but could be done with the help of two bitmasks.



#### **Approach 1: Hashmap**

Build a hashmap : element -> its frequency. Return only the elements with the frequency equal to 1.

**Implementation**

|  |
| --- |
| class Solution {  public int[] singleNumber(int[] nums) {  Map<Integer, Integer> hashmap = new HashMap();  for (int n : nums)  hashmap.put(n, hashmap.getOrDefault(n, 0) + 1);  int[] output = new int[2];  int idx = 0;  for (Map.Entry<Integer, Integer> item : hashmap.entrySet())  if (item.getValue() == 1) output[idx++] = item.getKey();  return output;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) to iterate over the input array.
* Space complexity : \mathcal{O}(N)O(*N*) to keep the hashmap of N*N* elements.

#### **Approach 2: Two bitmasks**

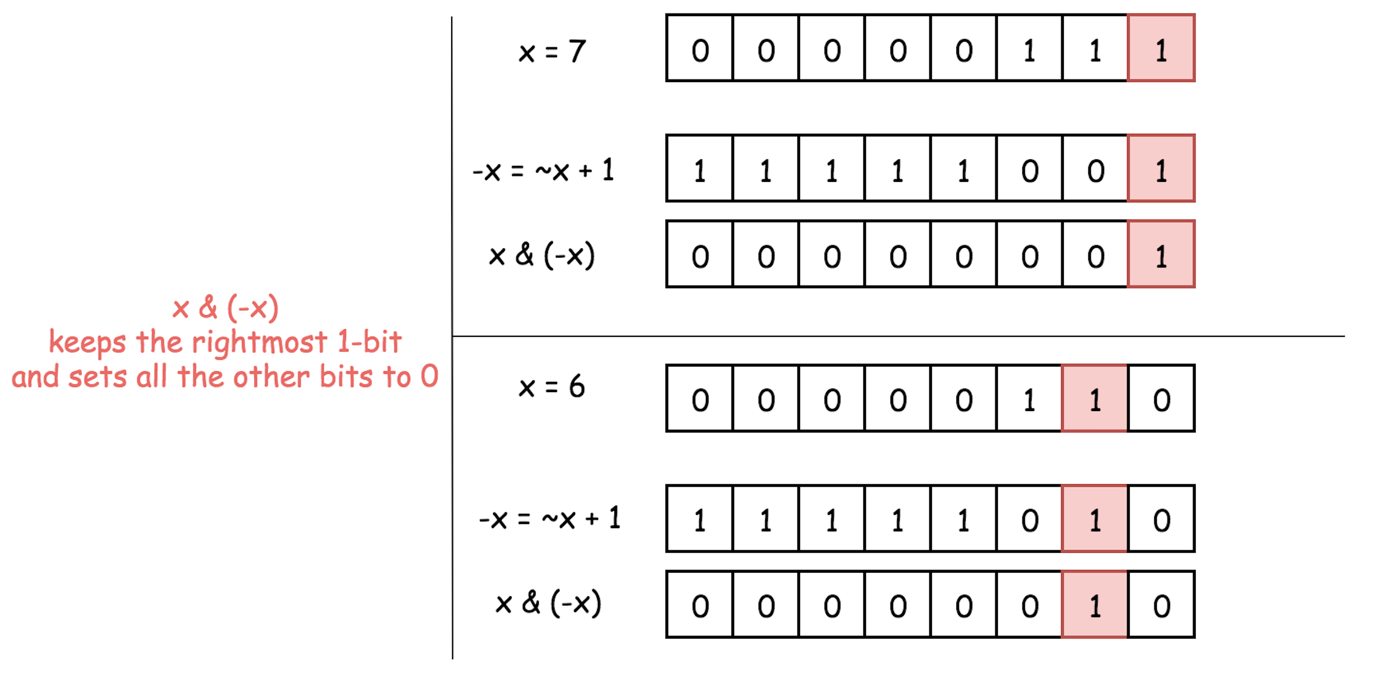
**Prerequisites**

This article will use two bitwise tricks, discussed in details last week :

* If one builds an array bitmask with the help of XOR operator, following bitmask ^= x strategy, the bitmask would keep only the bits which appear odd number of times. That was discussed in details in the article [Single Number II](https://leetcode.com/articles/single-number-ii/).



* x & (-x) is a way to isolate the rightmost 1-bit, i.e. to keep the rightmost 1-bit and to set all the others bits to zero. Please refer to the article [Power of Two](https://leetcode.com/articles/power-of-two/) for the detailed explanation.

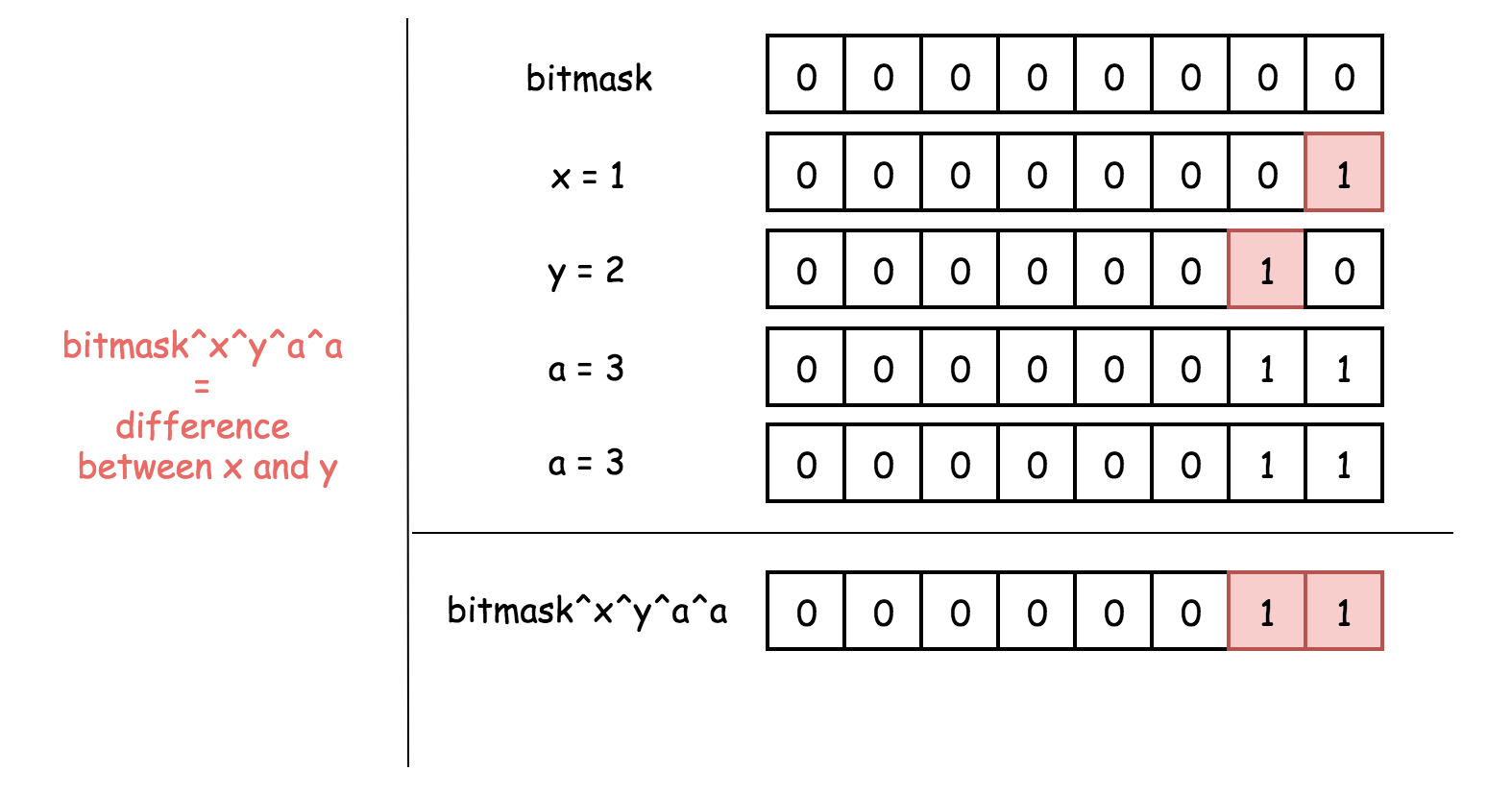


**Intuition**

An interview tip. Imagine, you have a problem to indentify an array element (or elements), which appears exactly given number of times. Probably, the key is to build first an array bitmask using XOR operator. Examples: [In-Place Swap](https://leetcode.com/problems/single-number-iii/solution/leetcode.com/articles/single-number-ii/356460/Single-Number-II/324042), [Single Number](https://leetcode.com/articles/single-number/), [Single Number II](https://leetcode.com/problems/single-number-iii/solution/leetcode.com/articles/single-number-ii/356460/Single-Number-II/324042).

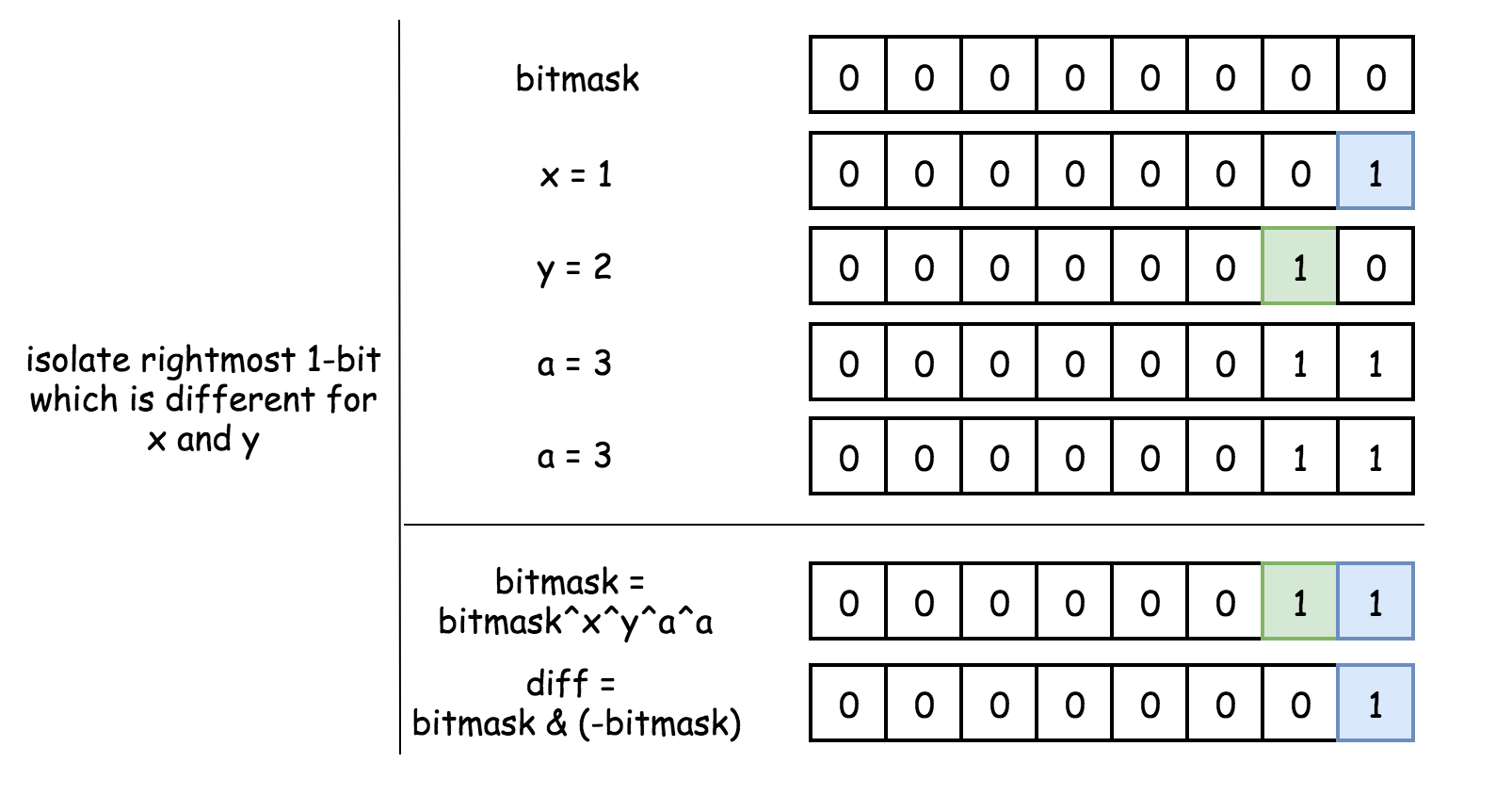
So let's create an array bitmask : bitmask ^= x. This bitmask will not keep any number which appears twice because XOR of two equal bits results in a zero bit a^a = 0.

Instead, the bitmask would keep only the difference between two numbers (let's call them x and y) which appear just once. The difference here it's the bits which are different for x and y.



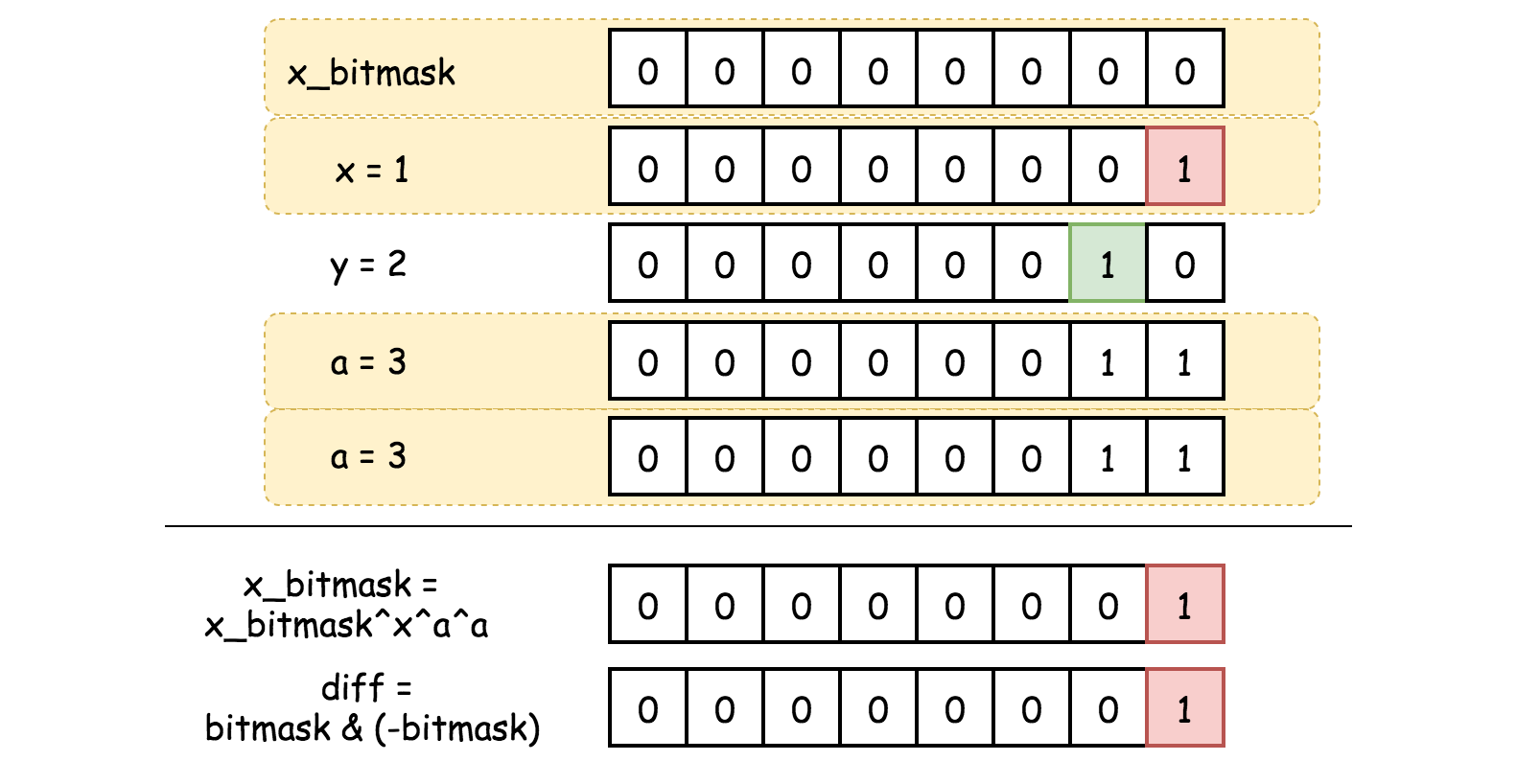
Could we extract x and y directly from this bitmask? No. Though we could use this bitmask as a marker to separate x and y.

Let's do bitmask & (-bitmask) to isolate the rightmost 1-bit, which is different between x and y. Let's say this is 1-bit for x, and 0-bit for y.



Now let's use XOR as before, but for the new bitmask x\_bitmask, which will contain only the numbers which have 1-bit in the position of bitmask & (-bitmask). This way, this new bitmask will contain only number x x\_bitmask = x, because of two reasons:

* y has 0-bit in the position bitmask & (-bitmask) and hence will not enter this new bitmask.
* All numbers but x will not be visible in this new bitmask because they appear two times.



Voila, x is identified. Now to identify y is simple: y = bitmask^x.

**Implementation**

|  |
| --- |
| class Solution {  public int[] singleNumber(int[] nums) {  // difference between two numbers (x and y) which were seen only once  int bitmask = 0;  for (int num : nums) bitmask ^= num;  // rightmost 1-bit diff between x and y  int diff = bitmask & (-bitmask);  int x = 0;  // bitmask which will contain only x  for (int num : nums) if ((num & diff) != 0) x ^= num;  return new int[]{x, bitmask^x};  }  } |

**Complexity Analysis**

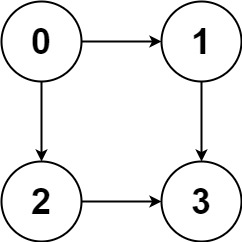
* Time complexity : \mathcal{O}(N)O(*N*) to iterate over the input array.
* Space complexity : \mathcal{O}(1)O(1), it's a constant space solution.

**All Paths From Source to Target**

Given a directed acyclic graph (**DAG**) of n nodes labeled from 0 to n - 1, find all possible paths from node 0 to node n - 1, and return them in any order.

The graph is given as follows: graph[i] is a list of all nodes you can visit from node i (i.e., there is a directed edge from node i to node graph[i][j]).

**Example 1:**

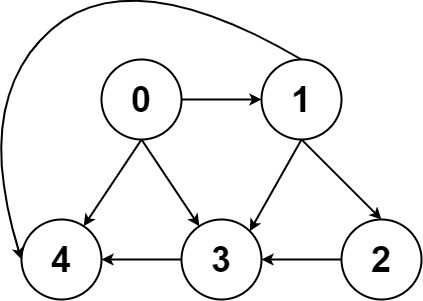


**Input:** graph = [[1,2],[3],[3],[]]

**Output:** [[0,1,3],[0,2,3]]

**Explanation:** There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.

**Example 2:**



**Input:** graph = [[4,3,1],[3,2,4],[3],[4],[]]

**Output:** [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]

**Example 3:**

**Input:** graph = [[1],[]]

**Output:** [[0,1]]

**Example 4:**

**Input:** graph = [[1,2,3],[2],[3],[]]

**Output:** [[0,1,2,3],[0,2,3],[0,3]]

**Example 5:**

**Input:** graph = [[1,3],[2],[3],[]]

**Output:** [[0,1,2,3],[0,3]]

**Constraints:**

* n == graph.length
* 2 <= n <= 15
* 0 <= graph[i][j] < n
* graph[i][j] != i (i.e., there will be no self-loops).
* The input graph is **guaranteed** to be a **DAG**.

## Solution

#### **Approach 1: Backtracking**

**Overview**

If a hint is ever given on the problem description, that would be ***backtracking***.

Indeed, since the problem concerns about the path exploration in a graph data structure, it is a perfect scenario to apply the backtracking algorithm.

As a reminder, [backtracking](https://en.wikipedia.org/wiki/Backtracking) is a general algorithm that incrementally builds candidates to the solutions, and abandons a candidate ("backtrack") as soon as it determines that the candidate cannot possibly lead to a valid solution.

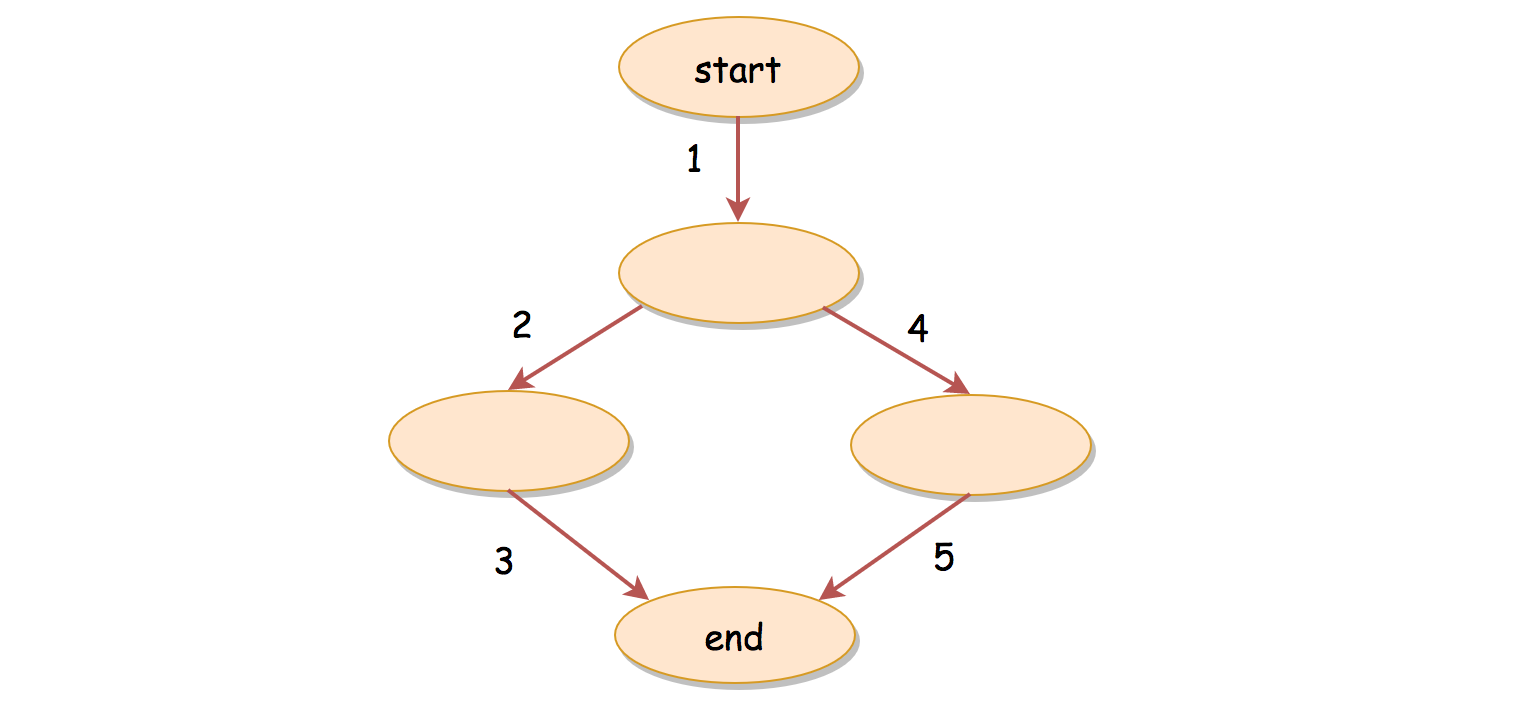
For more details about how to implement a backtracking algorithm, one can refer to our [Explore card](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/).

**Intuition**

Specifically, for this problem, we could assume ourselves as an agent in a game, we can explore the graph one step at a time.

At any given node, we try out each neighbor node recursively until we reach the target or there is no more node to hop on. By trying out, we mark the choice before moving on, and later on we reverse the choice (i.e. backtrack) and start another exploration.

To better demonstrate the above idea, we illustrate how an agent would explore the graph with the backtracking strategy, in the following image where we mark the order that each edge is visited.



**Algorithm**

The above idea might remind one of the Depth-First Search (**DFS**) traversal algorithm. Indeed, often the backtracking algorithm assumes the form of DFS, but with the additional step of backtracking.

And for the DFS traversal, we often adopt the **recursion** as its main form of implementation. With recursion, we could implement a backtracking algorithm in a rather intuitive and concise way. We break it down into the following steps:

* Essentially, we want to implement a recursive function called backtrack(currNode, path) which continues the exploration, given the current node and the path traversed so far.
  + Within the recursive function, we first define its base case, i.e. the moment we should terminate the recursion. Obviously, we should stop the exploration when we encounter our target node. So the condition of the base case is currNode == target.
  + As the body of our recursive function, we should enumerate through all the neighbor nodes of the current node.
  + For each iteration, we first mark the choice by appending the neighbor node to the path. Then we recursively invoke our backtrack() function to explore deeper. At the end of the iteration, we should reverse the choice by popping out the neighbor node from the path, so that we could start all over for the next neighbor node.
* Once we define our backtrack() function, it suffices to add the initial node (i.e. node with index 0) to the path, to kick off our backtracking exploration.

|  |
| --- |
| class Solution {  private int target;  private int[][] graph;  private List<List<Integer>> results;  protected void backtrack(int currNode, LinkedList<Integer> path) {  if (currNode == this.target) {  // Note: one should make a deep copy of the path  this.results.add(new ArrayList<Integer>(path));  return;  }  // explore the neighbor nodes one after another.  for (int nextNode : this.graph[currNode]) {  // mark the choice, before backtracking.  path.addLast(nextNode);  this.backtrack(nextNode, path);  // remove the previous choice, to try the next choice  path.removeLast();  }  }  public List<List<Integer>> allPathsSourceTarget(int[][] graph) {  this.target = graph.length - 1;  this.graph = graph;  this.results = new ArrayList<List<Integer>>();  // adopt the LinkedList for fast access to the tail element.  LinkedList<Integer> path = new LinkedList<Integer>();  path.addLast(0);  // kick of the backtracking, starting from the source (node 0)  this.backtrack(0, path);  return this.results;  }  } |

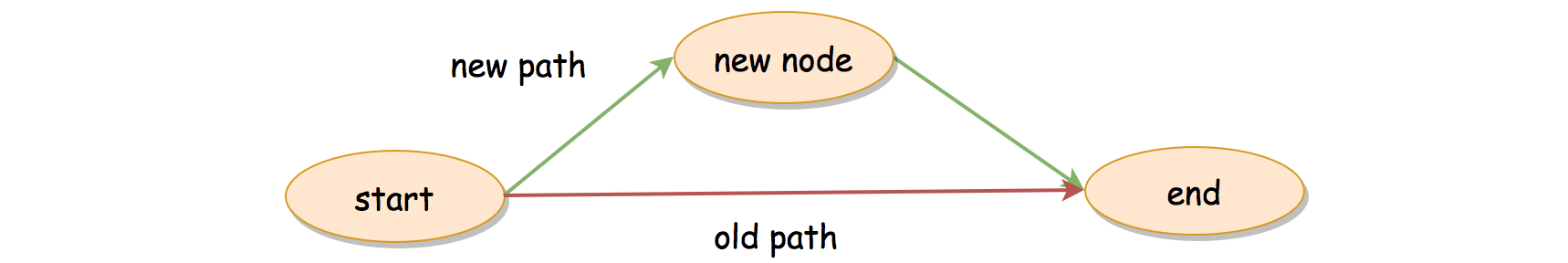
**Complexity Analysis**

Let N*N* be the number of nodes in the graph.

First of all, let us estimate how many paths there are at maximum to travel from the Node 0 to the Node N-1 for a graph with N*N* nodes.

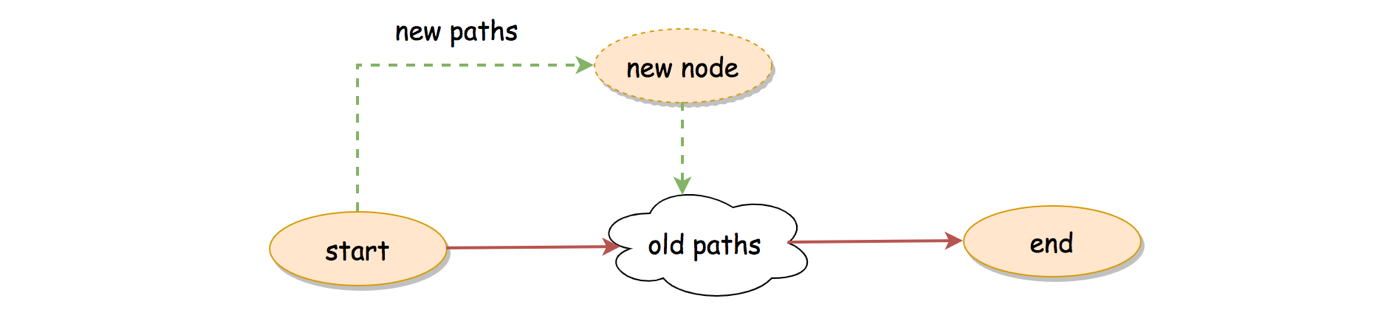
Let us start from a graph with only two nodes. As one can imagine, there is only one single path to connect the only two nodes in the graph.

Now, let us add a new node into the previous two-nodes graph, we now have two paths, one from the previous path, the other one is bridged by the newly-added node.



If we continue to add nodes to the graph, one insight is that every time we add a new node into the graph, the number of paths would **double**.

With the newly-added node, new paths could be created by preceding all previous paths with the newly-added node, as illustrated in the following graph.



As a result, for a graph with N*N* nodes, at maximum, there could be \sum\_{i=0}^{N-2}{2^i} = 2^{N-1} - 1∑*i*=0*N*−2​2*i*=2*N*−1−1 number of paths between the starting and the ending nodes.

* Time Complexity: \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*)
  + As we calculate shortly before, there could be at most 2^{N-1} - 12*N*−1−1 possible paths in the graph.
  + For each path, there could be at most N-2*N*−2 intermediate nodes, i.e. it takes \mathcal{O}(N)O(*N*) time to build a path.
  + To sum up, a **loose** upper-bound on the time complexity of the algorithm would be (2^{N-1} - 1) \cdot \mathcal{O}(N) = \mathcal{O}(2^N \cdot N)(2*N*−1−1)⋅O(*N*)=O(2*N*⋅*N*), where we consider it takes \mathcal{O}(N)O(*N*) efforts to build each path.
  + It is a loose uppper bound, since we could have overlapping among the paths, therefore the efforts to build certain paths could benefit others.
* Space Complexity: \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*)
  + Similarly, since at most we could have 2^{N-1}-12*N*−1−1 paths as the results and each path can contain up to N*N* nodes, the space we need to store the results would be \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*).
  + Since we also applied recursion in the algorithm, the recursion could incur additional memory consumption in the function call stack. The stack can grow up to N*N* consecutive calls. Meanwhile, along with the recursive call, we also keep the state of the current path, which could take another \mathcal{O}(N)O(*N*) space. Therefore, in total, the recursion would require additional \mathcal{O}(N)O(*N*) space.
  + To sum up, the space complexity of the algorithm is \mathcal{O}(2^N \cdot N) + \mathcal{O}(N) = \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*)+O(*N*)=O(2*N*⋅*N*).

#### **Approach 2: Top-Down Dynamic Programming**

**Intuition**

The backtracking approach applies the paradigm of divide-and-conquer, which breaks the problem down to smaller steps. As one knows, there is another algorithm called ***Dynamic Programming*** (DP), which also embodies the idea of divide-and-conquer.

As it turns out, we could also apply the DP algorithm to this problem, although it is less optimal than the backtracking approach as one will see later.

More specifically, we adopt the **Top-Down** DP approach, where we take a laissez-faire strategy assuming that the target function would work out on its own.

Given a node currNode, our target function is allPathsToTarget(currNode), which returns all the paths from the current node to the target node.

The target function could be calculated by iterating through the neighbor nodes of the current node, which we summarize with the following recursive formula:

\forall \text{nextNode} \in \text{neighbors}(\text{currNode}), \\ \\ \text{allPathsToTarget}(\text{currNode}) = \{ \text{currNode} + \text{allPathsToTarget}(\text{nextNode}) \}∀nextNode∈neighbors(currNode),allPathsToTarget(currNode)={currNode+allPathsToTarget(nextNode)}

The above formula can be read intuitively as: "the paths from the current node to the target node consist of all the paths starting from each neighbor of the current node."

**Algorithm**

Based on the above formula, we could implement a DP algorithm.

* First of all, we define our target function allPathsToTarget(node).
  + Naturally our target function is a recursive function, whose base case is when the given node is the target node.
  + Otherwise, we iterate through its neighbor nodes, and we invoke our target function with each neighbor node, i.e. allPathsToTarget(neighbor)
  + With the returned results from the target function, we then prepend the current node to the downstream paths, in order to build the final paths.
* With the above defined target function, we simply invoke it with the desired starting node, i.e. node 0.

Note that, there is an important detail that we left out in the above step. In order for the algorithm to be fully-qualified as a DP algorithm, we should **reuse** the intermediate results, rather than re-calculating them at each occasion.

Specially, we should **cache** the results returned from the target function allPathsToTarget(node), since we would encounter a node multiple times if there is an overlapping between paths. Therefore, once we know the paths from a given node to the target node, we should keep it in the cache for reuse. This technique is also known as **memoization**.

|  |
| --- |
| class Solution {  private int target;  private int[][] graph;  private HashMap<Integer, List<List<Integer>>> memo;  protected List<List<Integer>> allPathsToTarget(int currNode) {  // memoization. check the result in the cache first  if (memo.containsKey(currNode))  return memo.get(currNode);  List<List<Integer>> results = new ArrayList<>();  // base case  if (currNode == this.target) {  ArrayList<Integer> path = new ArrayList<>();  path.add(target);  results.add(path);  return results;  }  // iterate through the paths starting from each neighbor.  for (int nextNode : this.graph[currNode]) {  for (List<Integer> path : allPathsToTarget(nextNode)) {  ArrayList<Integer> newPath = new ArrayList<>();  newPath.add(currNode);  newPath.addAll(path);  results.add(newPath);  }  }  memo.put(currNode, results);  return results;  }  public List<List<Integer>> allPathsSourceTarget(int[][] graph) {  this.target = graph.length - 1;  this.graph = graph;  this.memo = new HashMap<>();  return this.allPathsToTarget(0);  }  } |

**Complexity Analysis**

Let N*N* be the number of nodes in the graph. As we estimated before, there could be at most 2^{N-1}-12*N*−1−1 number of paths.

* Time Complexity: \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*).
  + To estimate the overall time complexity, let us start from the last step when we prepend the starting node to each of the paths returned from the target function. Since we have to copy each path in order to create new paths, it would take up to N*N* steps for each final path. Therefore, for this last step, it could take us \mathcal{O}(2^{N-1} \cdot N)O(2*N*−1⋅*N*) time.
  + Right before the last step, when the maximal length of the path is N-1*N*−1, we should have 2^{N-2}2*N*−2 number of paths at this moment.
  + Deducing from the above two steps, again a **loose** upper-bound of the time complexity would be \mathcal{O}(\sum\_{i=1}^{N}{2^{i-1}\cdot i}) = \mathcal{O}(2^N \cdot N)O(∑*i*=1*N*​2*i*−1⋅*i*)=O(2*N*⋅*N*)
  + The two approach might have the same asymptotic time complexity. However, in practice the DP approach is slower than the backtracking approach, since we copy the intermediate paths over and over.
  + Note that, the performance would be degraded further, if we did not adopt the memoization technique here.
* Space Complexity: \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*)
  + Similarly, since at most we could have 2^{N-1}-12*N*−1−1 paths as the results and each path can contain up to N*N* nodes, the space we need to store the results would be \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*).
  + Since we also applied recursion in the algorithm, it could incur additional memory consumption in the function call stack. The stack can grow up to N*N* consecutive calls. Therefore, the recursion would require additional \mathcal{O}(N)O(*N*) space.
  + To sum up, the space complexity of the algorithm is \mathcal{O}(2^N \cdot N) + \mathcal{O}(N) = \mathcal{O}(2^N \cdot N)O(2*N*⋅*N*)+O(*N*)=O(2*N*⋅*N*).