**December 2020 Challenge**

Week 1: December 1st - December 7th

~~+35~~

Shortest Word Distance

~~+10~~

Maximum Depth of Binary Tree

~~+10~~

**Linked List Random Node**

~~+10~~

**Increasing Order Search Tree**

~~+10~~

**The kth Factor of n**

~~+10~~

 Can Place Flowers

~~+10~~

Populating Next Right Pointers in Each Node II

~~+10~~

**Spiral Matrix II**

Week 2: December 8th - December 14th

~~+35~~

Missing Ranges

~~+10~~

**Pairs of Songs With Total Durations Divisible by 60**

~~+10~~

Binary Search Tree Iterator

~~+10~~

Valid Mountain Array

~~+10~~

**Remove Duplicates from Sorted Array II**

~~+10~~

**Smallest Subtree with all the Deepest Nodes**

~~+10~~

Burst Balloons

~~+10~~

Palindrome Partitioning

Week 3: December 15th - December 21st

~~+35~~

 **Plus One Linked List**

~~+10~~

Squares of a Sorted Array

~~+10~~

Validate Binary Search Tree

~~+10~~

4Sum II

~~+10~~

Increasing Triplet Subsequence

~~+10~~

**Cherry Pickup II**

~~+10~~

**Decoded String at Index**

~~+10~~

**Smallest Range II**

Week 4: December 22nd - December 28th

~~+35~~

**Find Nearest Right Node in Binary Tree**

~~+10~~

Balanced Binary Tree

~~+10~~

**Next Greater Element III**

~~+10~~

Swap Nodes in Pairs

~~+10~~

 Diagonal Traverse

~~+10~~

Decode Ways

~~+10~~

**Jump Game IV**

~~+10~~

**Reach a Number**

Week 5: December 29th - December 31st

~~+35~~

**Longest Substring with At Most K Distinct Characters**

~~+10~~

**Pseudo-Palindromic Paths in a Binary Tree**

~~+10~~

Game of Life

~~+10~~

Largest Rectangle in Histogram

**Linked List Random Node**

Given a singly linked list, return a random node's value from the linked list. Each node must have the **same probability** of being chosen.

**Example 1:**

**Input**

["Solution", "getRandom", "getRandom", "getRandom", "getRandom", "getRandom"]

[[[1, 2, 3]], [], [], [], [], []]

**Output**

[null, 1, 3, 2, 2, 3]

**Explanation**

Solution solution = new Solution([1, 2, 3]);

solution.getRandom(); // return 1

solution.getRandom(); // return 3

solution.getRandom(); // return 2

solution.getRandom(); // return 2

solution.getRandom(); // return 3

// getRandom() should return either 1, 2, or 3 randomly. Each element should have equal probability of returning.

**Constraints:**

* The number of nodes in the linked list will be in the range [1, 104]
* -104 <= Node.val <= 104
* At most 104 calls will be made to getRandom.

**Follow up:**

* What if the linked list is extremely large and its length is unknown to you?
* Could you solve this efficiently without using extra space?

Solution

Overview

The solution for this problem could be as simple as it sounds, *i.e.* sampling from a linked list. However, with the constraint raised by the follow-up question, it becomes more interesting.

As a spoiler alert, in this article we will present an algorithm called [Reservoir sampling](https://en.wikipedia.org/wiki/Reservoir_sampling) which is a family of randomized algorithms for sampling from a population of ***unknown size***.

Approach 1: Fixed-Range Sampling

**Intuition**

First of all, let us talk about the elephant in the room. Yes, the problem could be as simple as choosing a random sample from a list, which in our case happens to be a ***linked list***.

If we are given an array or a linked list with a **known size**, then it would be a no brainer to solve the problem.

One of the most intuitive ideas would be that we **convert** the linked list into an array. With the array, we would know its size and moreover we could have an instant access to its elements.

**Algorithm**

We are asked to implement two interfaces in the object, namely the init(head) and getRandom() functions.

The init(head) function would be invoked once when we construct the object. Within which, intuitively we could convert the given linked list into an array for later reuse.

Concerning the getRandom() function, with the linked list converted into an array, we could simply sample from this array.

|  |
| --- |
| class Solution {  private ArrayList<Integer> range = new ArrayList<>();  /\*\* @param head The linked list's head.  Note that the head is guaranteed to be not null, so it contains at least one node. \*/  public Solution(ListNode head) {  while (head != null) {  this.range.add(head.val);  head = head.next;  }  }  /\*\* Returns a random node's value. \*/  public int getRandom() {  int pick = (int)(Math.random() \* this.range.size());  return this.range.get(pick);  }  }  /\*\*  \* Definition for singly-linked list.  \* public class ListNode {  \* int val;  \* ListNode next;  \* ListNode() {}  \* ListNode(int val) { this.val = val; }  \* ListNode(int val, ListNode next) { this.val = val; this.next = next; }  \* }  \*/ |

The above solution is simple, which happens to be fast as well. But it comes with two **caveats**:

* It requires some space to keep the pool of elements for sampling, which does not meet the constraint asked in the follow-up question, i.e. a solution of constant space.
* It cannot cope with the scenario that we have a list with **ever-growing elements**, i.e. we don't have unlimited memory to hold all the elements. For example, we have a stream of numbers, and we would like to pick a random number at any given moment. With the above naive solution, we would have to keep all the numbers in the memory, which is not scalable.

We will address the above caveats in the next approach.

**Complexity Analysis**

* Time Complexity:
  + For the init(head) function, its time complexity is \mathcal{O}(N)O(*N*) where N*N* is the number of elements in the linked list.
  + For the getRandom() function, its time complexity is \mathcal{O}(1)O(1). The reason is two-fold: it takes a constant time to generate a random number and the access to any element in the array is of constant time as well.
* Space Complexity: \mathcal{O}(N)O(*N*)
  + The overall solution requires \mathcal{O}(N)O(*N*) space complexity, since we make a copy of elements from the input list.

#### Approach 2: Reservoir Sampling

**Intuition**

In order to do random sampling over a population of **unknown size** with **constant space**, the answer is [reservoir sampling](https://en.wikipedia.org/wiki/Reservoir_sampling). As one will see later, it is an elegant algorithm that can address the two caveats of the previous approach.

The reservoir sampling algorithm is intended to sample k elements from an population of unknown size. In our case, the k happens to be one.

Furthermore, the reservoir sampling is a **family** of algorithms which includes several variants over the time. Here we present a simple albeit slow one, also known as ***Algorithm R*** by [Alan Waterman](https://en.wikipedia.org/wiki/Reservoir_sampling#cite_note-vitter-1).

# S has items to sample, R will contain the result

def ReservoirSample(S[1..n], R[1..k])

# fill the reservoir array

for i := 1 to k

R[i] := S[i]

# replace elements with gradually decreasing probability

for i := k+1 to n

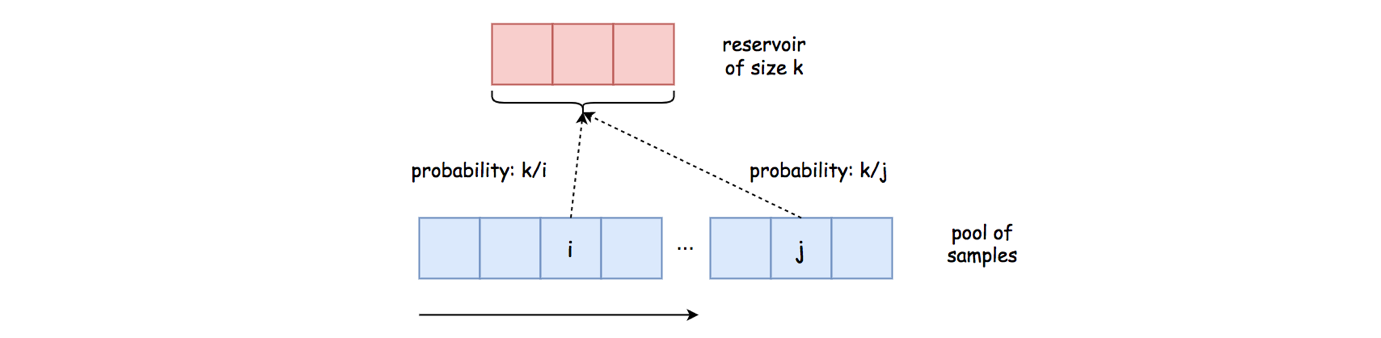
# randomInteger(a, b) generates a uniform integer

# from the inclusive range {a, ..., b} \*)

j := randomInteger(1, i)

if j <= k

R[j] := S[i]



We summarize the main idea of the algorithm as follows:

* Initially, we fill up an array of reservoir R[] with the heading elements from the pool of samples S[]. At the end of the algorithm, the reservoir will contain the final elements we sample from the pool.
* We then iterate through the rest of elements in the pool. For each element, we need to decide if we want to include it in the reservoir or not. If so, we will replace an existing element in reservoir with the current element.

One important question that one might have is that how we can ensure that each element has an **equal probability** to be chosen.

Given the above algorithm, it is guaranteed that at any moment, for each element scanned so far, it has an equal chance to be selected into the reservoir.

Here we provide a simple proof.

* Suppose that we have an element at the index of i (and i > k), when we reach the element, the chance that it will be selected into the reservoir would be \frac{k}{i}*ik*​, as we can see from the algorithm.
* Later on, there is a chance that any chosen element in the reservoir might be **replaced** with the subsequent element. More specifically, when we reach the element j (j > i), there would be a chance of \frac{1}{j}*j*1​ for any specific element in the reservoir to be replaced. Because for any specific position in the reservoir, there is \frac{1}{j}*j*1​ chance that it might be chosen by the random number generator. On the other hand, there would be \frac{j-1}{j}*jj*−1​ probability for any specific element in the reservoir to stay in the reservoir at that particular moment of sampling.
* To sum up, in order for any element in the pool to be chosen in the final reservoir, a series of **independent events** need to occur, namely:
  + Firstly, the element needs to be chosen in the reservoir when we reach the element.
  + Secondly, in the following sampling, the element should remain in the reservoir, i.e. not to be replaced.
* Therefore, for a sequence of length n, the chance that any element ends up in the final reservoir could be represented in the following formula:

\frac{k}{i} \cdot \frac{i}{i+1} \cdot \frac{i+1}{i+2} ... \frac{n-1}{n} = \frac{k}{n}*ik*​⋅*i*+1*i*​⋅*i*+2*i*+1​...*nn*−1​=*nk*​

**Algorithm**

Given the intuition above, we can now put it into implementation as follows:

* In the init() function, we simply keep the head of the linked list, rather than converting it into array.
* In the getRandom() function, we then do a reservoir sampling starting from the head of the linked list. More specifically, we scan the element one by one and decide whether we should put it into the reservoir (which in our case case happens to be of size one).

|  |
| --- |
| class Solution {  private ListNode head;  /\*\* @param head The linked list's head.  Note that the head is guaranteed to be not null, so it contains at least one node. \*/  public Solution(ListNode head) {  this.head = head;  }  /\*\* Returns a random node's value. \*/  public int getRandom() {  int scope = 1, chosenValue = 0;  ListNode curr = this.head;  while (curr != null) {  // decide whether to include the element in reservoir  if (Math.random() < 1.0 / scope)  chosenValue = curr.val;  // move on to the next node  scope += 1;  curr = curr.next;  }  return chosenValue;  }  }  /\*\*  \* Definition for singly-linked list.  \* public class ListNode {  \* int val;  \* ListNode next;  \* ListNode() {}  \* ListNode(int val) { this.val = val; }  \* ListNode(int val, ListNode next) { this.val = val; this.next = next; }  \* }  \*/ |

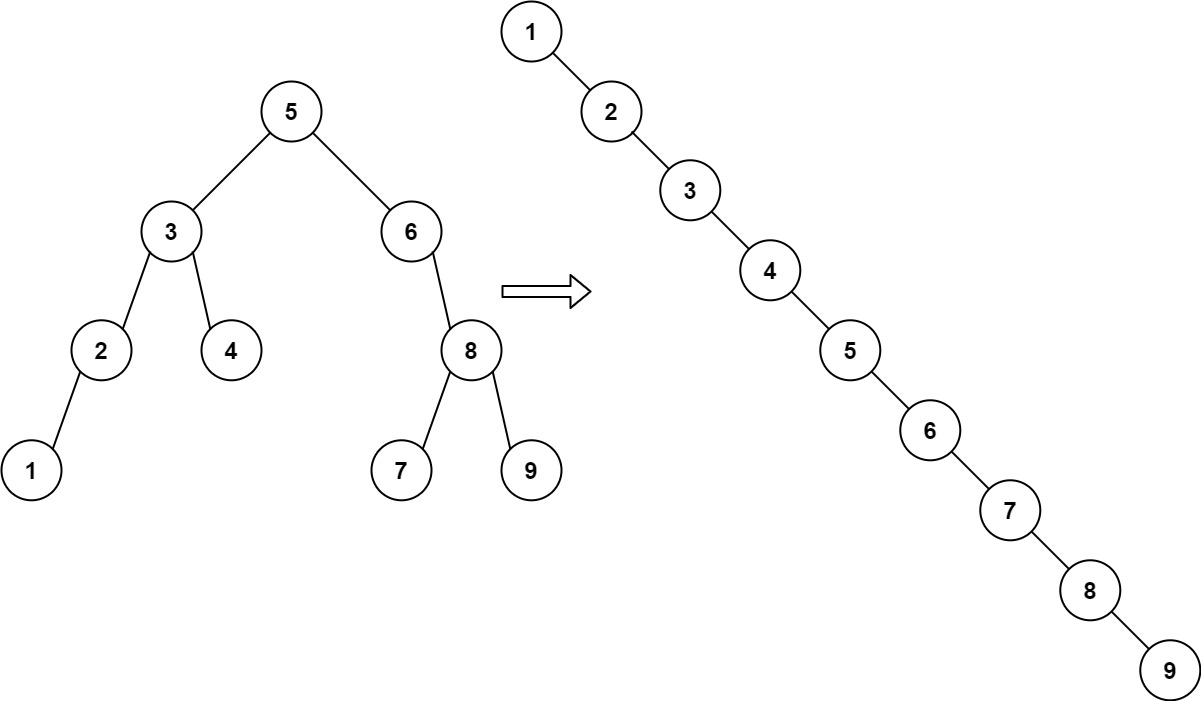
**Complexity Analysis**

* Time Complexity:
  + For the init(head) function, its time complexity is \mathcal{O}(1)O(1).
  + For the getRandom() function, its time complexity is \mathcal{O}(N)O(*N*) where N*N* is the number of elements in the input list.
* Space Complexity: \mathcal{O}(1)O(1)
  + The overall solution requires \mathcal{O}(1)O(1) space complexity, since the variables we used in the algorithm are of constant size, regardless the input.

**Increasing Order Search Tree**

Given the root of a binary search tree, rearrange the tree in **in-order** so that the leftmost node in the tree is now the root of the tree, and every node has no left child and only one right child.

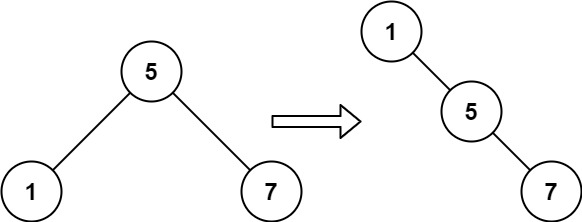
**Example 1:**



**Input:** root = [5,3,6,2,4,null,8,1,null,null,null,7,9]

**Output:** [1,null,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]

**Example 2:**



**Input:** root = [5,1,7]

**Output:** [1,null,5,null,7]

**Constraints:**

* The number of nodes in the given tree will be in the range [1, 100].
* 0 <= Node.val <= 1000

## Solution

#### Approach 1: In-Order Traversal

**Intuition**

The definition of a binary search tree is that for every node, all the values of the left branch are less than the value at the root, and all the values of the right branch are greater than the value at the root.

Because of this, an in-order traversal of the nodes will yield all the values in increasing order.

**Algorithm**

Once we have traversed all the nodes in increasing order, we can construct new nodes using those values to form the answer.

|  |
| --- |
| class Solution {  public TreeNode increasingBST(TreeNode root) {  List<Integer> vals = new ArrayList();  inorder(root, vals);  TreeNode ans = new TreeNode(0), cur = ans;  for (int v: vals) {  cur.right = new TreeNode(v);  cur = cur.right;  }  return ans.right;  }  public void inorder(TreeNode node, List<Integer> vals) {  if (node == null) return;  inorder(node.left, vals);  vals.add(node.val);  inorder(node.right, vals);  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of nodes in the given tree.
* Space Complexity: O(N)*O*(*N*).

#### Approach 2: Traversal with Relinking

**Intuition and Algorithm**

We can perform the same in-order traversal as in Approach 1. During the traversal, we'll construct the answer on the fly, reusing the nodes of the given tree by cutting their left child and adjoining them to the answer.

|  |
| --- |
| class Solution {  TreeNode cur;  public TreeNode increasingBST(TreeNode root) {  TreeNode ans = new TreeNode(0);  cur = ans;  inorder(root);  return ans.right;  }  public void inorder(TreeNode node) {  if (node == null) return;  inorder(node.left);  node.left = null;  cur.right = node;  cur = node;  inorder(node.right);  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of nodes in the given tree.
* Space Complexity: O(H)*O*(*H*) in additional space complexity, where H*H* is the height of the given tree, and the size of the implicit call stack in our in-order traversal.

**The kth Factor of n**

Given two positive integers n and k.

A factor of an integer n is defined as an integer i where n % i == 0.

Consider a list of all factors of n sorted in **ascending order**, return the kth factor in this list or return **-1** if n has less than k factors.

**Example 1:**

**Input:** n = 12, k = 3

**Output:** 3

**Explanation:** Factors list is [1, 2, 3, 4, 6, 12], the 3rd factor is 3.

**Example 2:**

**Input:** n = 7, k = 2

**Output:** 7

**Explanation:** Factors list is [1, 7], the 2nd factor is 7.

**Example 3:**

**Input:** n = 4, k = 4

**Output:** -1

**Explanation:** Factors list is [1, 2, 4], there is only 3 factors. We should return -1.

**Example 4:**

**Input:** n = 1, k = 1

**Output:** 1

**Explanation:** Factors list is [1], the 1st factor is 1.

**Example 5:**

**Input:** n = 1000, k = 3

**Output:** 4

**Explanation:** Factors list is [1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, 500, 1000].

**Constraints:**

* 1 <= k <= n <= 1000

   Hide Hint #1

The factors of n will be always in the range [1, n].

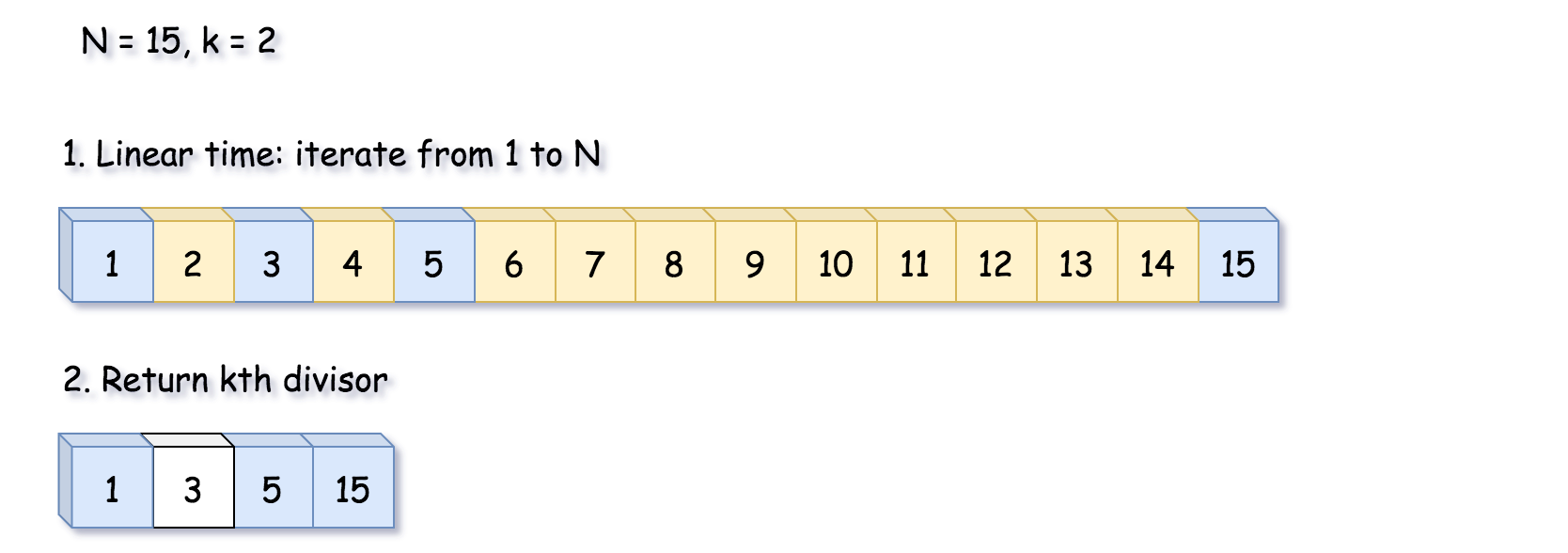
   Hide Hint #2

Keep a list of all factors sorted. Loop i from 1 to n and add i if n % i == 0. Return the kth factor if it exist in this list.

#### Overview

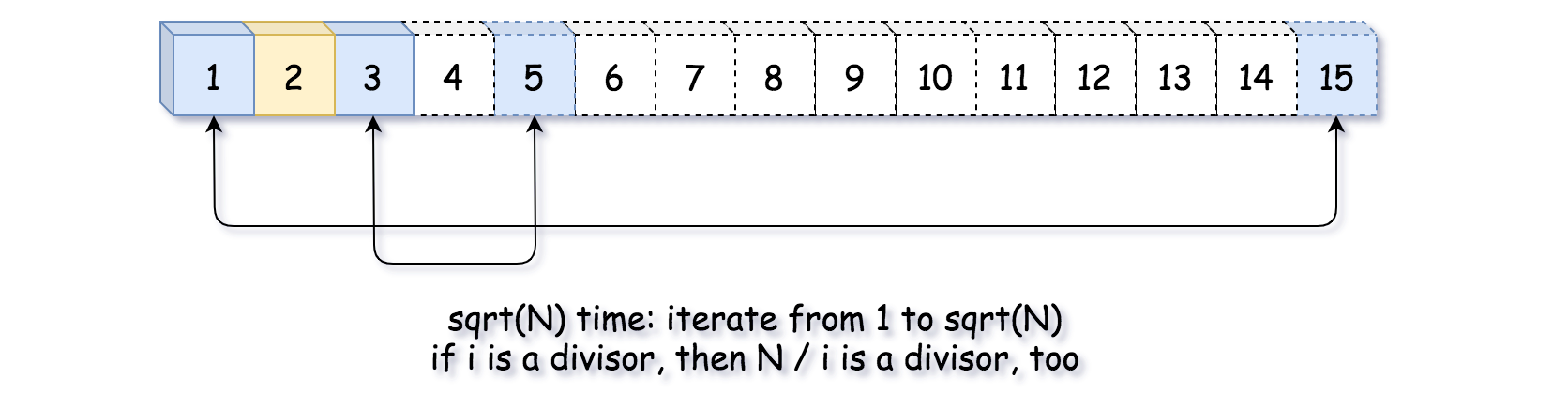
In this article, we consider three solutions.

Approach 1: Brute Force, \mathcal{O}(N)O(*N*). One could iterate from 11 to N*N*, figure out all divisors in a linear time, and then return the kth one.

 Fig. 1. Approach 1: Brute Force, *\mathcal{O}(N)O(N)*. The idea is to iterate from 1 to N to figure out all divisors.

The implementation of the next two approaches is based on the same idea.

The divisors are paired, i.e., if i*i* is a divisor of N*N*, then N / i*N*/*i* is a divisor of N*N*, too. That means it's enough to iterate up to \sqrt{N}*N*​.

 Fig. 2. Iterate from 1 to *\sqrt{N}N​* is enough.

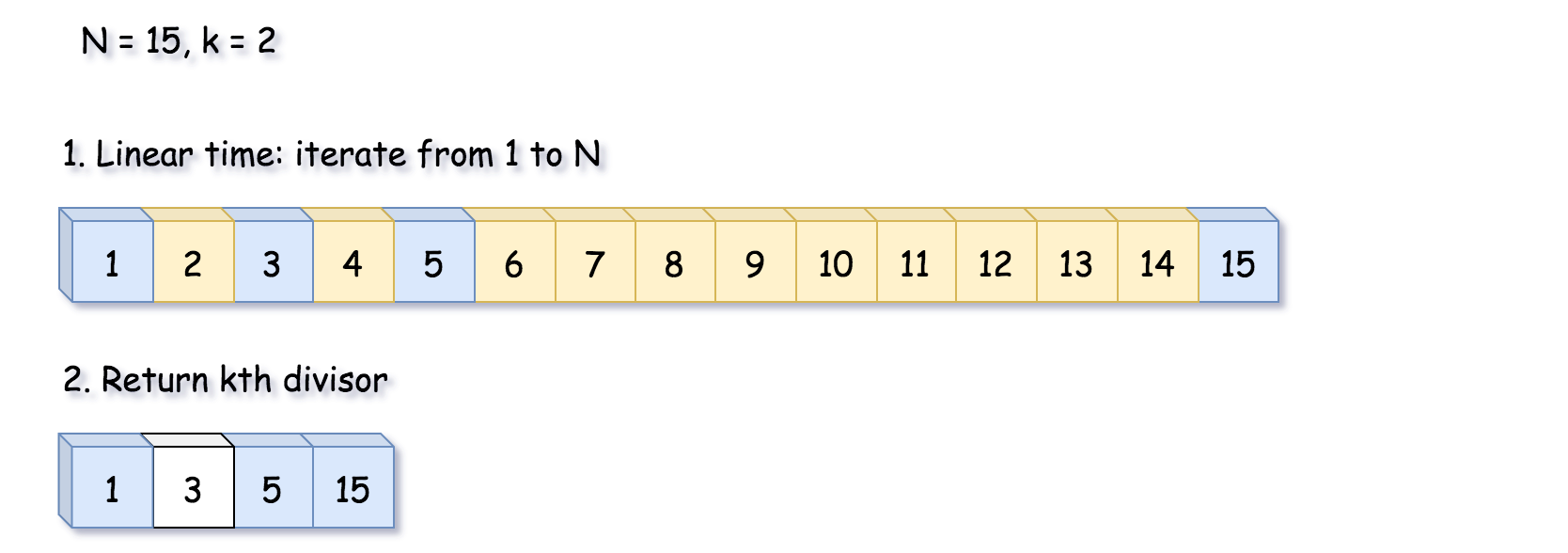
Approach 2: Heap, \mathcal{O}(\sqrt{N} \log k)O(*N*​log*k*). The idea is to iterate from 1 to \sqrt{N}*N*​, and push each divisor and its pair into max heap of size k*k*.

 Fig 3. Approach 2. Heap: Iterate from 1 to *\sqrt{N}N​* is enough, and push each divisor and its pair into max heap.

Approach 3: Math, \mathcal{O}(\sqrt{N})O(*N*​). As in Approach 2, let's iterate up to \sqrt{N}*N*​, and decrease k*k* by one at each step. If k*k* drops down to zero during the iterations - the kth divisor is here. Otherwise, the kth divisor is the paired one and could be found as N / divisors[len(divisors) - k].

 Fig. 4. Approach 3. Math: Iterate from 1 to *\sqrt{N}N​* is enough.

#### Approach 1: Brute Force, \mathcal{O}(N)O(*N*) time

 Fig. 5. Approach 1: Brute Force.

**Algorithm**

* Iterate by x*x* from 11 to N / 2*N*/2:
  + If x*x* is a divisor of N*N*, decrease k*k* by one. Return x*x* if k == 0*k*==0.
* Return N*N* if k == 1*k*==1 and -1−1 otherwise.

**Implementation**

|  |
| --- |
| class Solution {  public int kthFactor(int n, int k) {  for (int x = 1; x < n / 2 + 1; ++x) {  if (n % x == 0) {  --k;  if (k == 0) {  return x;  }  }  }    return k == 1 ? n : -1;  }  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(N)O(*N*) to iterate from 11 to N*N*.
* Space Complexity: \mathcal{O}(1)O(1), since we don't allocate any additional data structures.

#### Approach 2: Heap, \mathcal{O}(\sqrt{N} \times \log{k})O(*N*​×log*k*) time

 Fig 6. Approach 2: Heap.

**Algorithm**

* Initialize max heap. Use PriorityQueue in Java and heap in Python. heap is a min-heap. Hence, to implement max heap, change the sign of divisor before pushing it into the heap.
* Iterate by x*x* from 11 to \sqrt{N}*N*​:
  + If x*x* is a divisor of N*N*, push x*x* and its pair divisor n / x*n*/*x* into the heap of size k*k*.
* Return the head of the heap if its size is equal to k*k* and -1−1 otherwise.

**Implementation**

|  |
| --- |
| class Solution {  // max heap -> to keep max element always on top  Queue<Integer> heap = new PriorityQueue<>((o1, o2) -> o2 - o1);    // push into heap  // by limiting size of heap to k  public void heappushK(int x, int k) {  heap.add(x);  if (heap.size() > k) {  heap.remove();  }  }    public int kthFactor(int n, int k) {  int sqrtN = (int) Math.sqrt(n);  for (int x = 1; x < sqrtN + 1; ++x) {  if (n % x == 0) {  heappushK(x, k);  if (x != n / x) {  heappushK(n / x, k);  }  }  }    return k == heap.size() ? heap.poll() : -1;  }  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(\sqrt{N} \times \log k)O(*N*​×log*k*).
* Space Complexity: \mathcal{O}(\min(k, \sqrt{N}))O(min(*k*,*N*​)) to keep the heap of size k*k*.

#### Approach 3: Math, \mathcal{O}(\sqrt{N})O(*N*​) time

 Fig. 7. Approach 3: Math.

**Algorithm**

* Initialize a list divisors to store the divisors.
* Iterate by x*x* from 11 to \sqrt{N}*N*​:
  + If x*x* is a divisor of N*N*, decrease k*k* by one. Return x*x* if k == 0*k*==0.
* We're here because the kth divisor is not yet found. Although divisors already contains all "independent" divisors. All other divisors are "paired" ones, i.e, the kth divisor could be computed as N / divisors[len(divisors) - k].

But before that, we need a small correction for the case when N*N* is a perfect square. In that case, the divisor list contains a duplicate because \sqrt{N}*N*​ appears two times. To skip it, we have to increase k*k* by one.

* Return N / divisors[len(divisors) - k] if k <= len(divisors) and -1 otherwise.

**Implementation**

|  |
| --- |
| class Solution {  public int kthFactor(int n, int k) {  List<Integer> divisors = new ArrayList();  int sqrtN = (int) Math.sqrt(n);  for (int x = 1; x < sqrtN + 1; ++x) {  if (n % x == 0) {  --k;  divisors.add(x);  if (k == 0) {  return x;  }  }  }    // If n is a perfect square  // we have to skip the duplicate  // in the divisor list  if (sqrtN \* sqrtN == n) {  ++k;  }    int nDiv = divisors.size();  return (k <= nDiv) ? n / divisors.get(nDiv - k) : -1;  }  } |

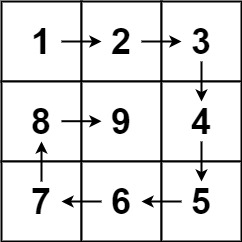
**Complexity Analysis**

* Time Complexity: \mathcal{O}(\sqrt{N})O(*N*​).
* Space Complexity: \mathcal{O}(\min(k, \sqrt{N}))O(min(*k*,*N*​)) to store the list of divisors.

**Spiral Matrix II**

Given a positive integer n, generate an n x n matrix filled with elements from 1 to n2 in spiral order.

**Example 1:**



**Input:** n = 3

**Output:** [[1,2,3],[8,9,4],[7,6,5]]

**Example 2:**

**Input:** n = 1

**Output:** [[1]]

**Constraints:**

* 1 <= n <= 20

## Solution

#### Overview

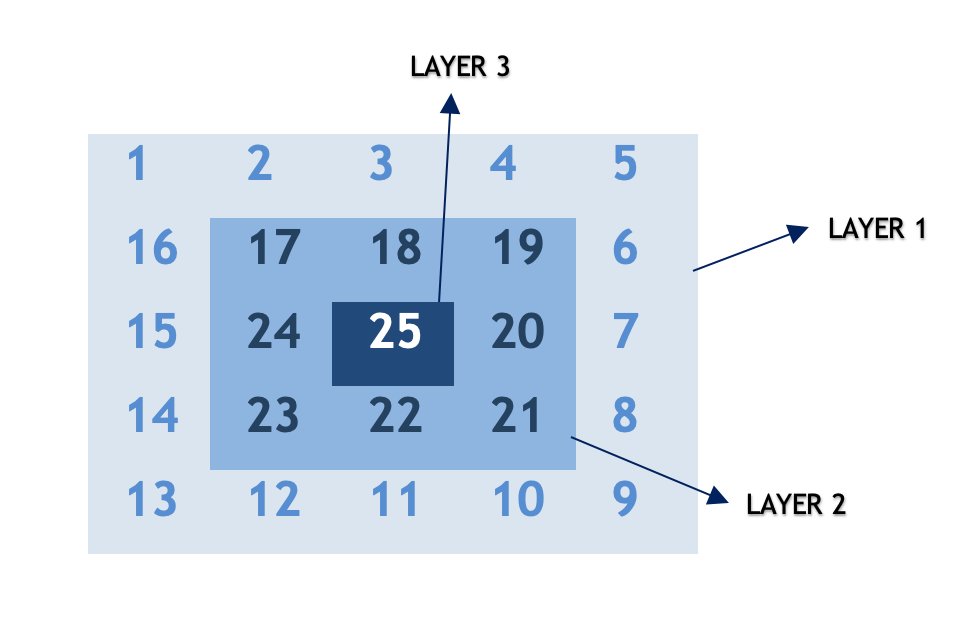
There are various problems in spiral matrix series with some variations like [Spiral Matrix](https://leetcode.com/problems/spiral-matrix/) and [Spiral Matrix III](https://leetcode.com/problems/spiral-matrix-iii/).

In order to solve such questions, the core idea is to decode the underlying pattern. This can be done by simulating the pattern and finding a generic representation that would work for any given n*n*. Let's discuss a few approaches.

#### Approach 1: Traverse Layer by Layer in Spiral Form

**Intuition**

If we try to build a pattern for a given n*n*, we observe that the pattern repeats after completing one circular traversal around the matrix. Let's call this one circular traversal as layer. We start traversing from the outer layer and move towards inner layers on every iteration.



**Algorithm**

Let's devise an algorithm for the spiral traversal:

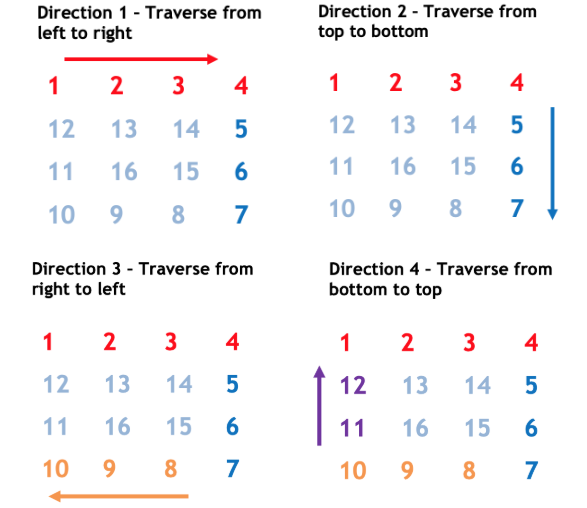
* We can observe that, for any given n*n*, the total number of layers is given by : \lfloor \frac{n+1}{2} \rfloor⌊2*n*+1​⌋ This works for both even and odd n*n*.

Example

For n = 3*n*=3, layers = 2*layers*=2

For n = 6*n*=6, total layers = 3*layers*=3

* Also, for each layer, we traverse in at most 4 directions :



In every direction, either row or column remains constant and other parameter changes (increments/decrements).

Direction 1: From top left corner to top right corner.

The row remains constant as \text{layer}layer and column increments from \text{layer}layer to n-\text{layer}-1*n*−layer−1

Direction 2: From top right corner to the bottom right corner.

The column remains constant as n-layer-1*n*−*layer*−1 and row increments from \text{layer}+1layer+1 to n-\text{layer}*n*−layer.

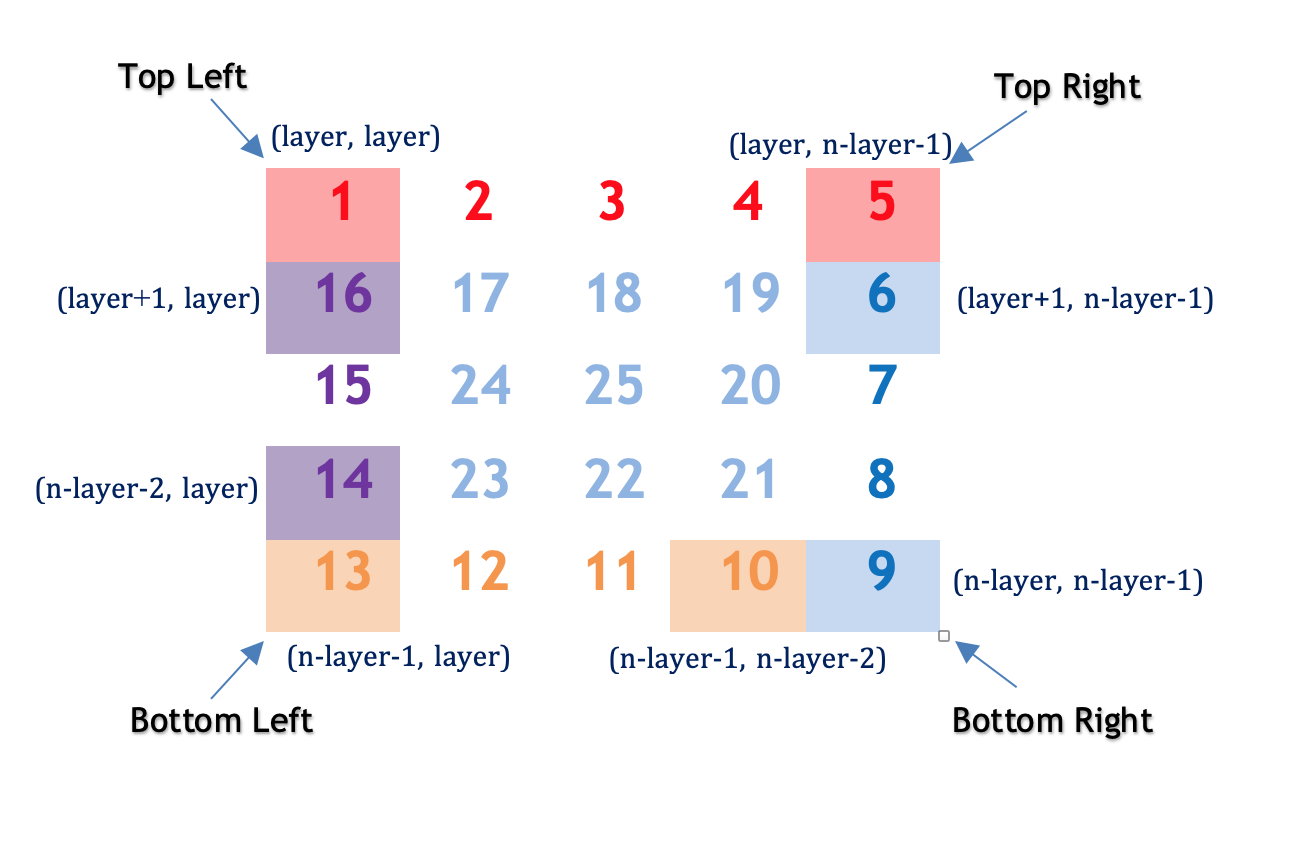
Direction 3: From bottom right corner to bottom left corner.

The row remains constant as n-\text{layer}-1*n*−layer−1 and column decrements from n-\text{layer}-2*n*−layer−2 to \text{layer}layer.

Direction 4: From bottom right corner to top left corner.

The column remains constant as \text{layer}layer and column decrements from n-\text{layer}-2*n*−layer−2 to \text{layer}+1layer+1.

This process repeats (n+1)/2(*n*+1)/2 times until all layers are traversed.



|  |
| --- |
| class Solution {  public int[][] generateMatrix(int n) {  int[][] result = new int[n][n];  int cnt = 1;  for (int layer = 0; layer < (n + 1) / 2; layer++) {  // direction 1 - traverse from left to right  for (int ptr = layer; ptr < n - layer; ptr++) {  result[layer][ptr] = cnt++;  }  // direction 2 - traverse from top to bottom  for (int ptr = layer + 1; ptr < n - layer; ptr++) {  result[ptr][n - layer - 1] = cnt++;  }  // direction 3 - traverse from right to left  for (int ptr = layer + 1; ptr < n - layer; ptr++) {  result[n - layer - 1][n - ptr - 1] = cnt++;  }  // direction 4 - traverse from bottom to top  for (int ptr = layer + 1; ptr < n - layer - 1; ptr++) {  result[n - ptr - 1][layer] = cnt++;  }  }  return result;  }  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n^2)O(*n*2). Here, n*n* is given input and we are iterating over n\cdot n*n*⋅*n* matrix in spiral form.
* Space Complexity: \mathcal{O}(1)O(1) We use constant extra space for storing cnt*cnt*.

#### Approach 2: Optimized spiral traversal

**Intuition**

Our main aim is to walk in a spiral form and fill the array in a particular pattern. In the previous approach, we used a separate loop for each direction. Here, we discuss another optimized to achieve the same result.

**Algorithm**

* We have to walk in 4 directions forming a layer. We use an array dir*dir* that stores the changes in x*x* and y*y* co-ordinates in each direction.

Example

In left to right walk ( direction #1 ), x*x* co-ordinates remains same and y*y* increments (x = 0*x*=0, y = 1*y*=1).

In right to left walk ( direction #3 ), x*x* remains same and y*y* decrements (x = 0*x*=0, y = -1*y*=−1).

Using this intuition, we pre-define an array dir*dir* having x*x* and y*y* co-ordinate changes for each direction. There are a total of 4 directions as discussed in the previous approach.

* The \text{row}row and col*col* variables represent the current x*x* and y*y* co-ordinates respectively. It updates based on the direction in which we are moving.

How do we know when we have to change the direction?

When we find the next row or column in a particular direction has a non-zero value, we are sure it is already traversed and we change the direction.

Let d*d* be the current direction index. We go to next direction in array dir*dir* using (d+ 1) \% 4(*d*+1)%4. Using this we could go back to direction 1 after completing one circular traversal from direction 1 to direction 4 .

It must be noted that we use floorMod in Java instead of modulo \%% to handle mod of negative numbers. This is required because row and column values might go negative and using \%% won't give desired results in such cases.

|  |
| --- |
| class Solution {  public int[][] generateMatrix(int n) {  int[][] result = new int[n][n];  int cnt = 1;  int dir[][] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};  int d = 0;  int row = 0;  int col = 0;  while (cnt <= n \* n) {  result[row][col] = cnt++;  int r = Math.floorMod(row + dir[d][0], n);  int c = Math.floorMod(col + dir[d][1], n);  // change direction if next cell is non zero  if (result[r][c] != 0) d = (d + 1) % 4;  row += dir[d][0];  col += dir[d][1];  }  return result;  }  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n^2)O(*n*2). Here, n*n* is given input and we are iterating over n\cdot n*n*⋅*n* matrix in spiral form.
* Space Complexity: \mathcal{O}(1)O(1) We use constant extra space for storing cnt*cnt*.

**Pairs of Songs With Total Durations Divisible by 60**

You are given a list of songs where the ith song has a duration of time[i] seconds.

Return the number of pairs of songs for which their total duration in seconds is divisible by 60. Formally, we want the number of indices i, j such that i < j with (time[i] + time[j]) % 60 == 0.

**Example 1:**

**Input:** time = [30,20,150,100,40]

**Output:** 3

**Explanation:** Three pairs have a total duration divisible by 60:

(time[0] = 30, time[2] = 150): total duration 180

(time[1] = 20, time[3] = 100): total duration 120

(time[1] = 20, time[4] = 40): total duration 60

**Example 2:**

**Input:** time = [60,60,60]

**Output:** 3

**Explanation:** All three pairs have a total duration of 120, which is divisible by 60.

**Constraints:**

* 1 <= time.length <= 6 \* 104
* 1 <= time[i] <= 500

   Hide Hint #1

We only need to consider each song length modulo 60.

   Hide Hint #2

We can count the number of songs with (length % 60) equal to r, and store that in an array of size 60.

## Solution

#### Approach 1: Brute Force

One of the most straightforward approaches would be iterating through the entire array using a nested loop to examine that, for each element a in time, whether there is another element b such that (a + b) % 60 == 0. Note that this approach might be too brutal to pass an interview.

|  |
| --- |
| class Solution {  public int numPairsDivisibleBy60(int[] time) {  int count = 0, n = time.length;  for (int i = 0; i < n; i++) {  // j starts with i+1 so that i is always to the left of j  // to avoid repetitive counting  for (int j = i + 1; j < n; j++) {  if ((time[i] + time[j]) % 60 == 0) {  count++;  }  }  }  return count;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(n^2)O(*n*2), when n*n* is the length of the input array. For each item in time, we iterate through the rest of the array to find a qualified complement taking \mathcal{O}(n)O(*n*) time.
* Space complexity: \mathcal{O}(1)O(1).

#### Approach 2: Use an Array to Store Frequencies

**Intuition**

Let's dive deep into the condition (time[i] + time[j]) % 60 == 0 to examine the relation between time[i] and time[j]. Assuming that a and b are two elements in the input array time, we have:

(a+b)\space \% \space60=0 \\ \Downarrow \\ ((a \space \% \space 60)+(b \space \% \space 60))\space \% \space 60=0 \\ \Downarrow \\ \text{Therefore, either }\begin{cases} a \space \% \space60 &= 0\\ b \space \% \space60 &= 0 \end{cases} \text{, or } (a\space\%\space60)+(b\space\%\space60)=60 \\(*a*+*b*) % 60=0⇓((*a* % 60)+(*b* % 60)) % 60=0⇓Therefore, either {*a* % 60*b* % 60​=0=0​, or (*a* % 60)+(*b* % 60)=60

You can learn more about the modulo operation [here](https://en.wikipedia.org/wiki/Modulo_operation#Properties_(identities)).

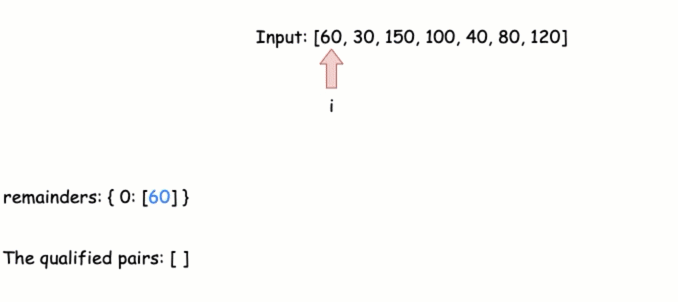
Hence, all we need would be finding the pairs of elements in time so they meet these conditions.

**Algorithm**

We would iterate through the input array time and for each element a, we want to know the number of elements b such that:

1. b \space\%\space 60=0*b* % 60=0, \space \text{if } a \space \% \space 60=0 if *a* % 60=0
2. b \space \% \space 60=60-a \space\% \space60*b* % 60=60−*a* % 60, \space \text{if } a\space\% \space 60\neq0 if *a* % 60​=0

We can use Approach 1 to implement this logic by repeatedly examining the rest of time again and again for each element a. However, we are able to improve the time complexity by consuming more space - we can store the frequencies of the remainder a % 60, so that we can find the number of the complements in \mathcal{O}(1)O(1) time.



We would initiate an array remainders with size 6060 to record the frequencies of each remainder - as the range of remainders is [0,59][0,59]. Then we can loop through the array once and for each element a we would:

1. if a \space \% \space 60=0*a* % 60=0, add remainders[0] to the result; else, add remainders[60 - t % 60] to the result;
2. update remainders[a % 60].

|  |
| --- |
| class Solution {  public int numPairsDivisibleBy60(int[] time) {  int remainders[] = new int[60];  int count = 0;  for (int t: time) {  if (t % 60 == 0) { // check if a%60==0 && b%60==0  count += remainders[0];  } else { // check if a%60+b%60==60  count += remainders[60 - t % 60];  }  remainders[t % 60]++; // remember to update the remainders  }  return count;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(n)O(*n*), when n*n* is the length of the input array, because we would visit each element in time once.
* Space complexity: \mathcal{O}(1)O(1), because the size of the array remainders is fixed with 6060.

**Remove Duplicates from Sorted Array II**

Given a sorted array nums, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that duplicates appeared at most twice and return the new length.

Do not allocate extra space for another array; you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

**Clarification:**

Confused why the returned value is an integer, but your answer is an array?

Note that the input array is passed in by **reference**, which means a modification to the input array will be known to the caller.

Internally you can think of this:

// **nums** is passed in by reference. (i.e., without making a copy)

int len = removeDuplicates(nums);

// any modification to **nums** in your function would be known by the caller.

// using the length returned by your function, it prints the first **len** elements.

for (int i = 0; i < len; i++) {

    print(nums[i]);

}

**Example 1:**

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

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

**Explanation:** Your function should return length = **5**, with the first five elements of *nums* being **1, 1, 2, 2** and **3** respectively. It doesn't matter what you leave beyond the returned length.

**Example 2:**

**Input:** nums = [0,0,1,1,1,1,2,3,3]

**Output:** 7, nums = [0,0,1,1,2,3,3]

**Explanation:** Your function should return length = **7**, with the first seven elements of *nums* being modified to **0**, **0**, **1**, **1**, **2**, **3** and **3** respectively. It doesn't matter what values are set beyond the returned length.

**Constraints:**

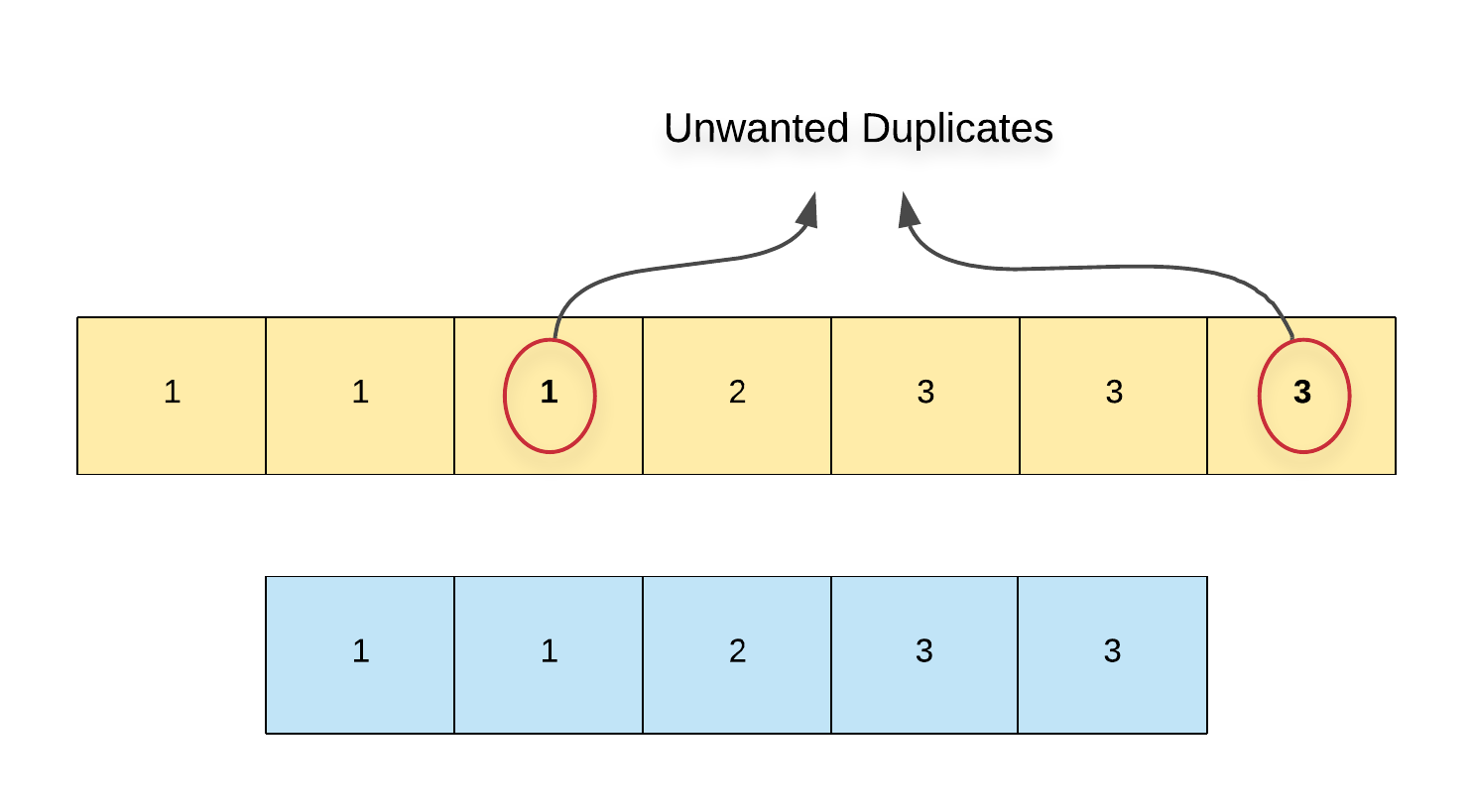
* 1 <= nums.length <= 3 \* 104
* -104 <= nums[i] <= 104
* nums is sorted in ascending order.

## Solution

#### Approach 1: Popping Unwanted Duplicates

**Intuition**

The input array is already sorted and hence, all the duplicates appear next to each other. The problem statement mentions that we are not allowed to use any additional space and we have to modify the array in-place. The easiest approach for in-place modifications would be to get rid of all the unwanted duplicates. For every number in the array, if we detect > 2 duplicates, we simply remove them from the list of elements and we do this for all the elements in the array.

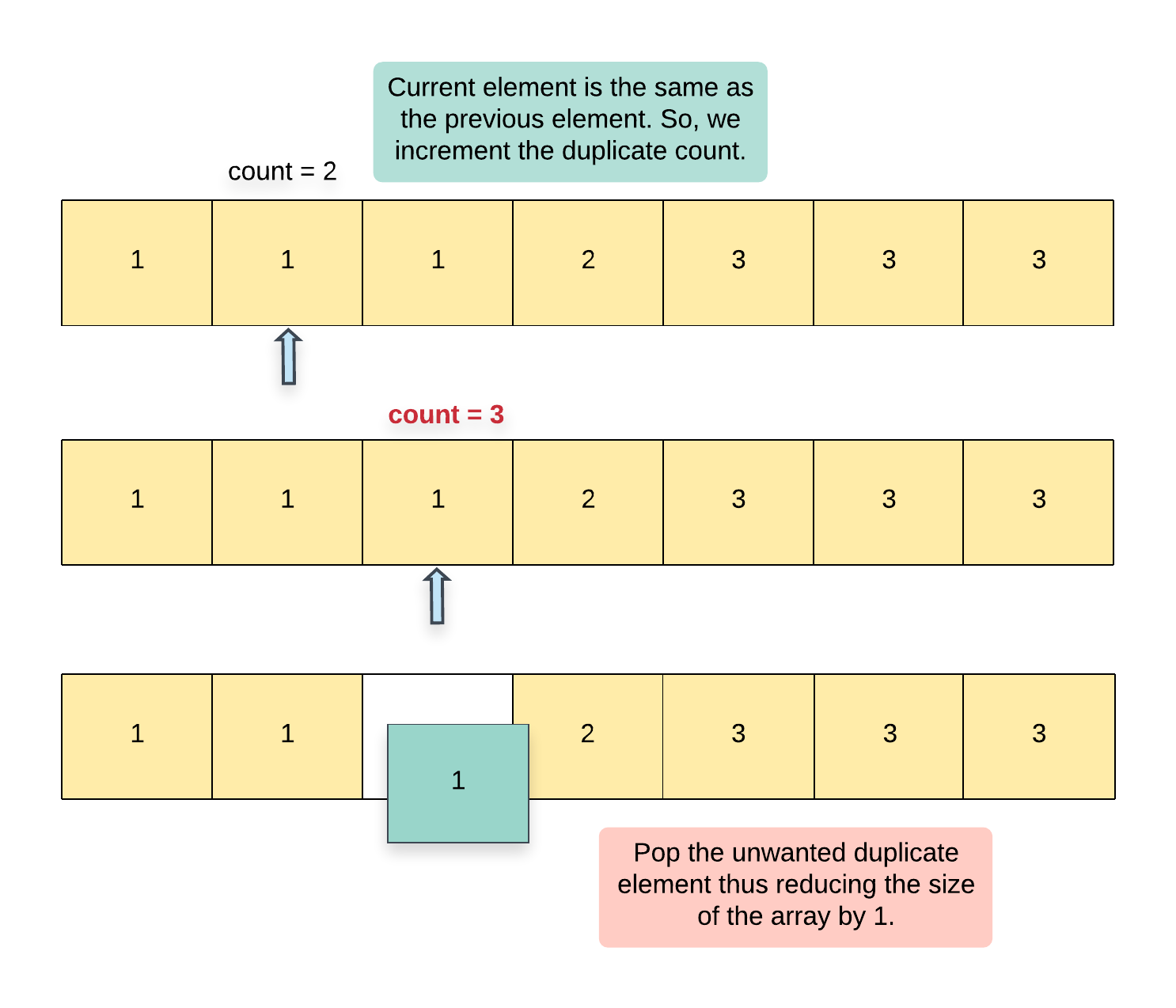


**Algorithm**

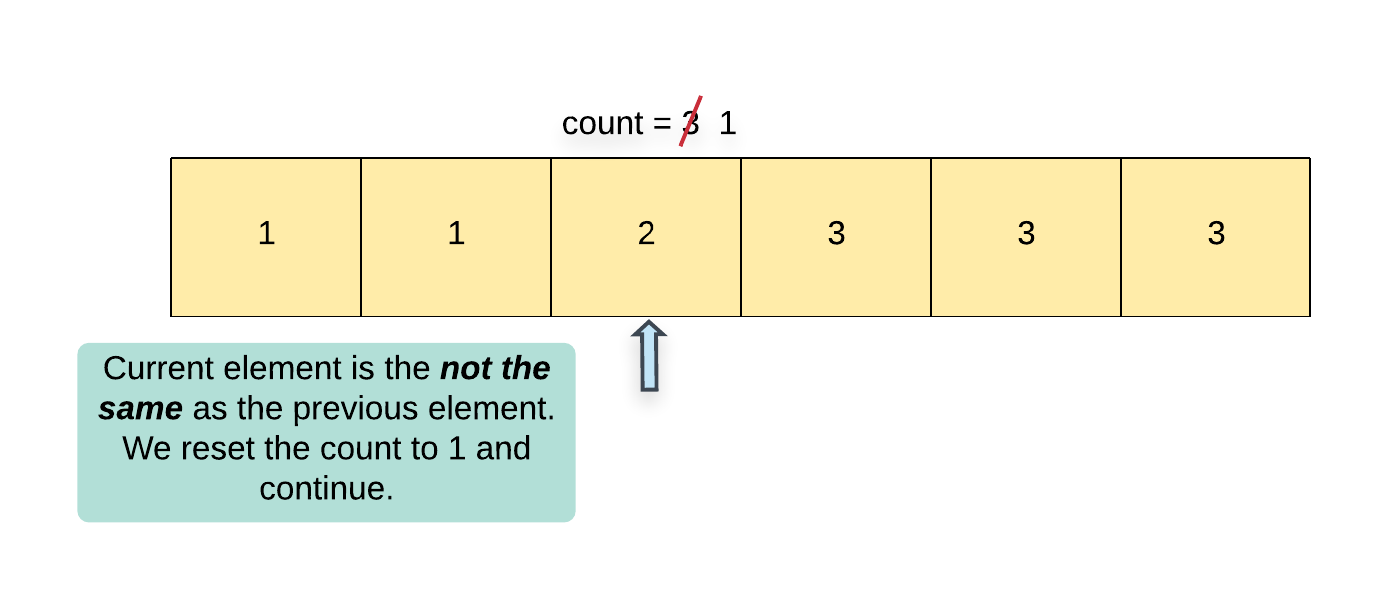
1. The implementation is slightly tricky so to say since we will be removing elements from the array and iterating over it at the same time. So, we need to keep updating the array's indexes as and when we pop an element else we'll be accessing invalid indexes.
2. Say we have two variables, i which is the array pointer and count which keeps track of the count of a particular element in the array. Note that the minimum count would always be 1.



1. We start with index 1 and process one element at a time in the array.
2. If we find that the current element is the same as the previous element i.e. nums[i] == nums[i - 1], then we increment the count. If the value of count > 2, then we have encountered an unwanted duplicate element and we can remove it from the array. Since we know the index of this element, we can use the del or pop or remove operation (or whatever corresponding operation is supported in your language of choice) to delete the element at index i from the array. Since we popped an element, we decrement the index by 1 as well.



1. If we encounter that the current element is not the same as the previous element i.e. nums[i] != nums[i - 1], then it means we have a new element at hand and so accordingly, we update count = 1.



1. Since we are removing all the unwanted duplicates from the original array, the final array that remains after process all the elements will only contain the valid elements and hence we simply return the length of this array.

|  |
| --- |
| class Solution {    public int[] remElement(int[] arr, int index) {    //  // Overwrite the element at the given index by  // moving all the elements to the right of the  // index, one position to the left.  //  for (int i = index + 1; i < arr.length; i++) {  arr[i - 1] = arr[i];  }    return arr;  }    public int removeDuplicates(int[] nums) {    // Initialize the counter and the array index.  int i = 1, count = 1, length = nums.length;    //  // Start from the second element of the array and process  // elements one by one.  //  while (i < length) {    //  // If the current element is a duplicate,  // increment the count.  //  if (nums[i] == nums[i - 1]) {    count++;    //  // If the count is more than 2, this is an unwanted duplicate element  // and hence we remove it from the array.  //  if (count > 2) {    this.remElement(nums, i);    //  // Note that we have to decrement the array index value to  // keep it consistent with the size of the array.  //  i--;    //  // Since we have a fixed size array and we can't actually  // remove an element, we reduce the length of the array as  // well.  //  length--;  }  } else {    //  // Reset the count since we encountered a different element  // than the previous one.  //  count = 1;  }    // Move on to the next element in the array  i++;  }    return length;  }  } |

**Complexity Analysis**

* Time Complexity: Let's see what the costly operations in our array are:
  + We have to iterate over all the elements in the array. Suppose that the original array contains N elements, the time taken here would be O(N)*O*(*N*).
  + Next, for every unwanted duplicate element, we will have to perform a delete operation and deletions in arrays are also O(N)*O*(*N*).
  + The worst case would be when all the elements in the array are the same. In that case, we would be performing N - 2*N*−2 deletions thus giving us O(N^2)*O*(*N*2) complexity for deletions
  + Overall complexity = O(N) + O(N^2) \equiv O(N^2)*O*(*N*)+*O*(*N*2)≡*O*(*N*2).
* Space Complexity: O(1)*O*(1) since we are modifying the array in-place.

#### Approach 2: Overwriting unwanted duplicates

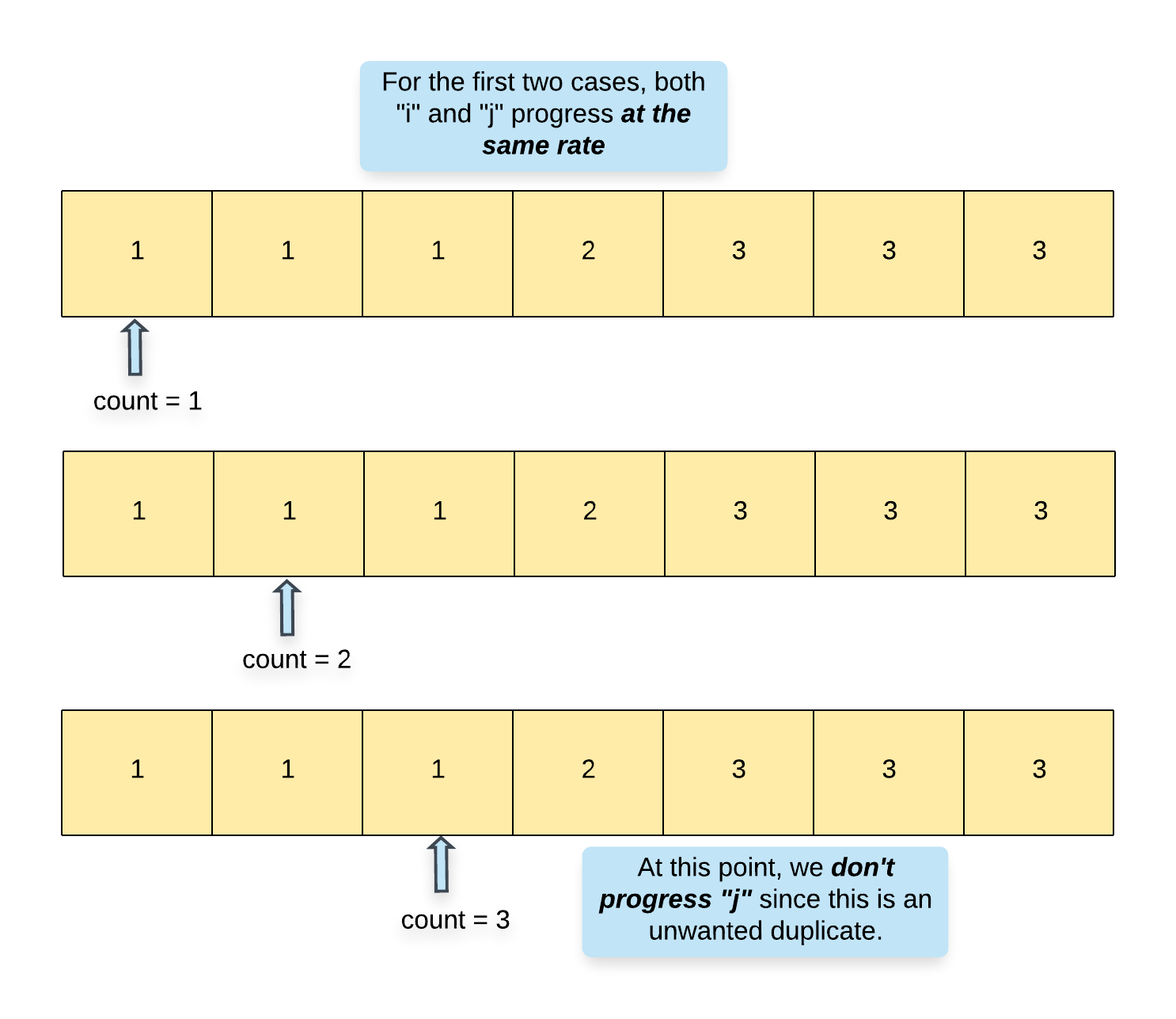
**Intuition**

The second approach is really inspired by the fact that the problem statement asks us to return the new length of the array from the function. If all we had to do was remove elements, the function would not really ask us to return the updated length. However, in our scenario, this is really an indication that we don't need to actually remove elements from the array. Instead, we can do something better and simply overwrite the duplicate elements that are unwanted.

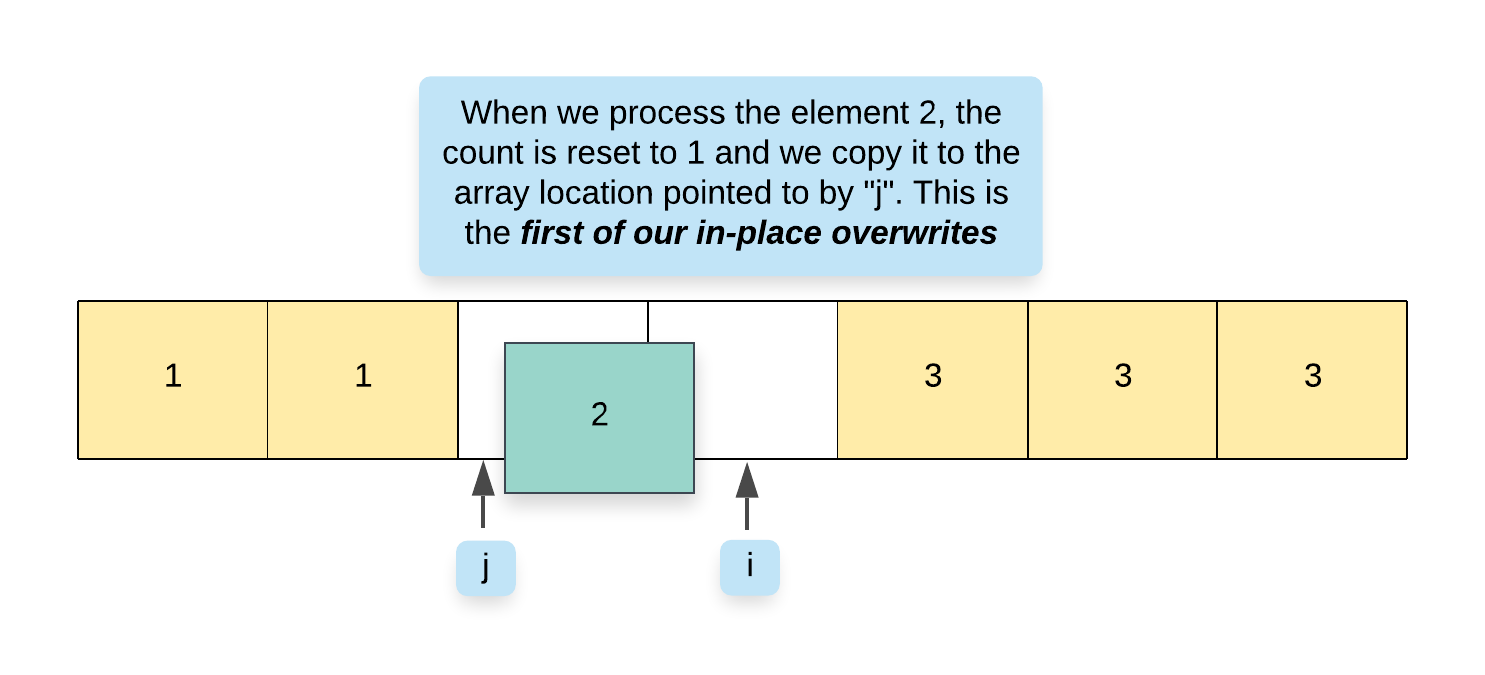
We won't be able to achieve this using a single pointer. We will be using a two-pointer approach where one pointer iterates over the original set of elements and another one that keeps track of the next "empty" location in the array or the next location that can be overwritten in the array.

**Algorithm**

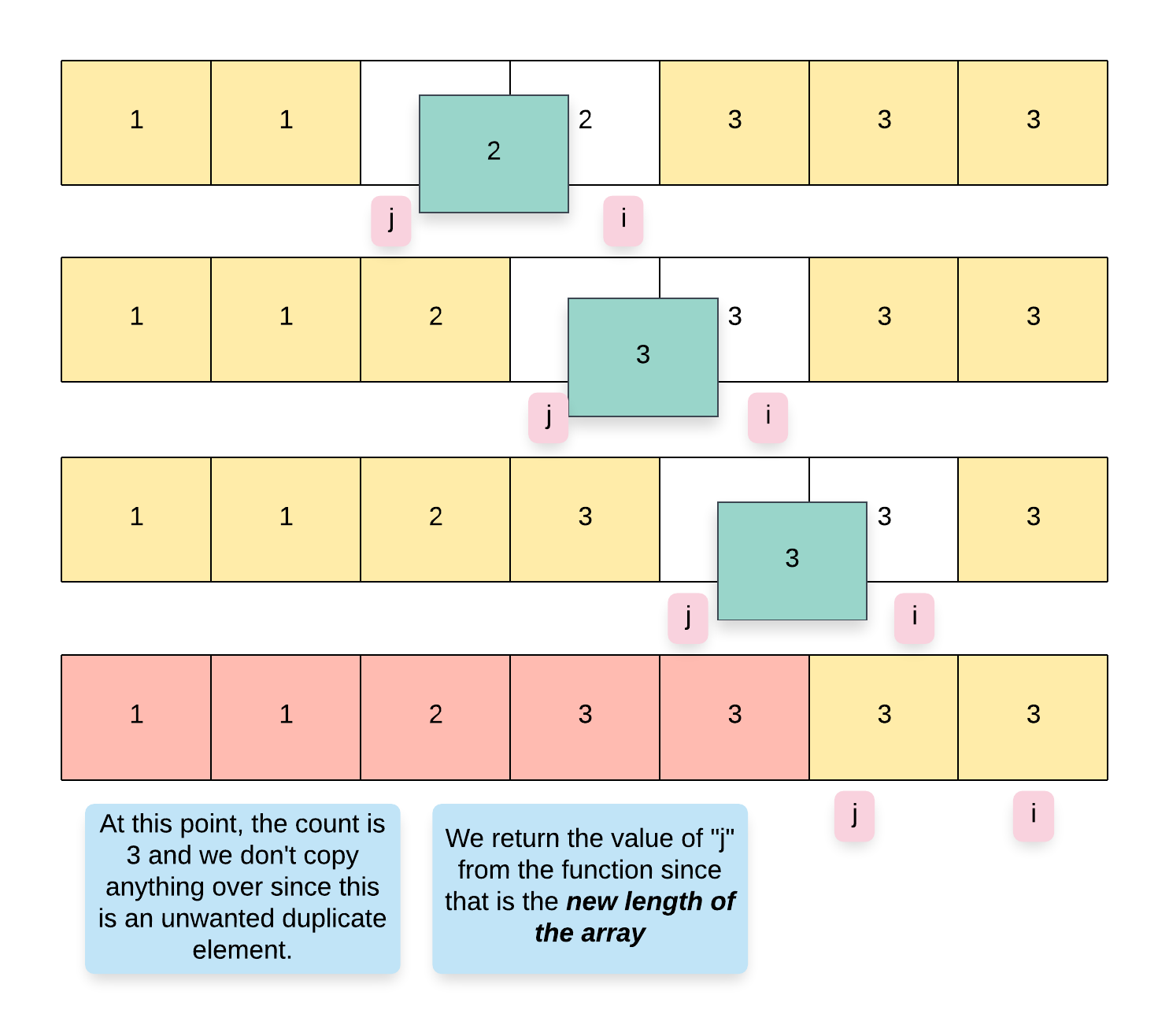
1. We define two pointers, i and j for our algorithm. The pointer i iterates of the array processing one element at a time and j keeps track of the next location in the array where we can overwrite an element.
2. We also keep a variable count which keeps track of the count of a particular element in the array. Note that the minimum count would always be 1.
3. We start with index 1 and process one element at a time in the array.
4. If we find that the current element is the same as the previous element i.e. nums[i] == nums[i - 1], then we increment the count. If the value of count > 2, then we have encountered an unwanted duplicate element. In this case, we simply move forward i.e. we increment i but not j.
5. However, if the count is <= 2, then we can move the element from index i to index j.



1. If we encounter that the current element is not the same as the previous element i.e. nums[i] != nums[i - 1], then it means we have a new element at hand and so accordingly, we update count = 1 and also move this element to index j.



1. It goes without saying that whenever we copy a new element to nums[j], we have to update the value of j as well since j always points to the location where the next element can be copied to in the array.



|  |
| --- |
| class Solution {    public int removeDuplicates(int[] nums) {    //  // Initialize the counter and the second pointer.  //  int j = 1, count = 1;    //  // Start from the second element of the array and process  // elements one by one.  //  for (int i = 1; i < nums.length; i++) {    //  // If the current element is a duplicate, increment the count.  //  if (nums[i] == nums[i - 1]) {    count++;    } else {    //  // Reset the count since we encountered a different element  // than the previous one.  //  count = 1;  }    //  // For a count <= 2, we copy the element over thus  // overwriting the element at index "j" in the array  //  if (count <= 2) {  nums[j++] = nums[i];  }  }  return j;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*) since we process each element exactly once.
* Space Complexity: O(1)*O*(1).

**Smallest Subtree with all the Deepest Nodes**

Given the root of a binary tree, the depth of each node is **the shortest distance to the root**.

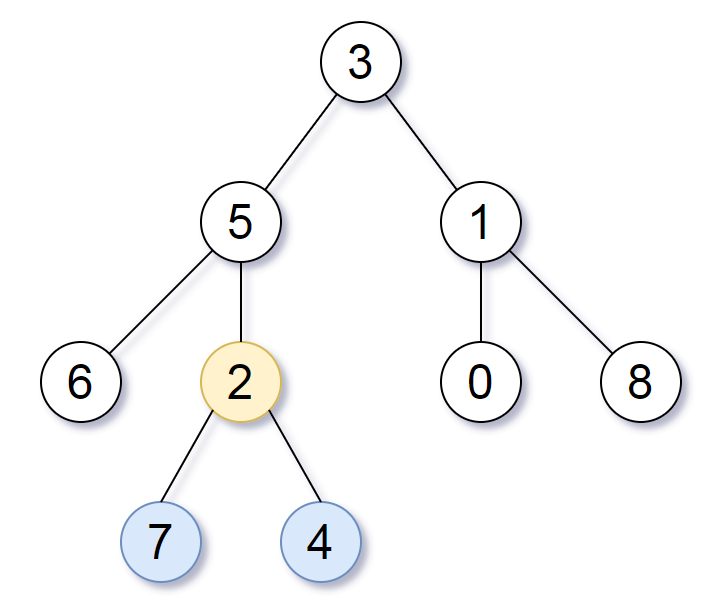
Return the smallest subtree such that it contains **all the deepest nodes** in the original tree.

A node is called **the deepest** if it has the largest depth possible among any node in the entire tree.

The **subtree** of a node is tree consisting of that node, plus the set of all descendants of that node.

**Note:** This question is the same as 1123: <https://leetcode.com/problems/lowest-common-ancestor-of-deepest-leaves/>

**Example 1:**



**Input:** root = [3,5,1,6,2,0,8,null,null,7,4]

**Output:** [2,7,4]

**Explanation:** We return the node with value 2, colored in yellow in the diagram.

The nodes coloured in blue are the deepest nodes of the tree.

Notice that nodes 5, 3 and 2 contain the deepest nodes in the tree but node 2 is the smallest subtree among them, so we return it.

**Example 2:**

**Input:** root = [1]

**Output:** [1]

**Explanation:** The root is the deepest node in the tree.

**Example 3:**

**Input:** root = [0,1,3,null,2]

**Output:** [2]

**Explanation:** The deepest node in the tree is 2, the valid subtrees are the subtrees of nodes 2, 1 and 0 but the subtree of node 2 is the smallest.

**Constraints:**

* The number of nodes in the tree will be in the range [1, 500].
* 0 <= Node.val <= 500
* The values of the nodes in the tree are **unique**.

## Solution

#### Approach 1: Paint Deepest Nodes

**Intuition**

We try a straightforward approach that has two phases.

The first phase is to identify the nodes of the tree that are deepest. To do this, we have to annotate the depth of each node. We can do this with a depth first search.

Afterwards, we will use that annotation to help us find the answer:

* If the node in question has maximum depth, it is the answer.
* If both the left and right child of a node have a deepest descendant, then the answer is this parent node.
* Otherwise, if some child has a deepest descendant, then the answer is that child.
* Otherwise, the answer for this subtree doesn't exist.

**Algorithm**

In the first phase, we use a depth first search dfs to annotate our nodes.

In the second phase, we also use a depth first search answer(node), returning the answer for the subtree at that node, and using the rules above to build our answer from the answers of the children of node.

Note that in this approach, the answer function returns answers that have the deepest nodes of the entire tree, not just the subtree being considered.

|  |
| --- |
| class Solution {  Map<TreeNode, Integer> depth;  int max\_depth;  public TreeNode subtreeWithAllDeepest(TreeNode root) {  depth = new HashMap();  depth.put(null, -1);  dfs(root, null);  max\_depth = -1;  for (Integer d: depth.values())  max\_depth = Math.max(max\_depth, d);  return answer(root);  }  public void dfs(TreeNode node, TreeNode parent) {  if (node != null) {  depth.put(node, depth.get(parent) + 1);  dfs(node.left, node);  dfs(node.right, node);  }  }  public TreeNode answer(TreeNode node) {  if (node == null || depth.get(node) == max\_depth)  return node;  TreeNode L = answer(node.left),  R = answer(node.right);  if (L != null && R != null) return node;  if (L != null) return L;  if (R != null) return R;  return null;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of nodes in the tree.
* Space Complexity: O(N)*O*(*N*).

#### Approach 2: Recursion

**Intuition**

We can combine both depth first searches in Approach #1 into an approach that does both steps in one pass. We will have some function dfs(node) that returns both the answer for this subtree, and the distance from node to the deepest nodes in this subtree.

**Algorithm**

The Result (on some subtree) returned by our (depth-first search) recursion will have two parts:

* Result.node: the largest depth node that is equal to or an ancestor of all the deepest nodes of this subtree.
* Result.dist: the number of nodes in the path from the root of this subtree, to the deepest node in this subtree.

We can calculate these answers disjointly for dfs(node):

* To calculate the Result.node of our answer:
  + If one childResult has deeper nodes, then childResult.node will be the answer.
  + If they both have the same depth nodes, then node will be the answer.
* The Result.dist of our answer is always 1 more than the largest childResult.dist we have.

|  |
| --- |
| class Solution {  public TreeNode subtreeWithAllDeepest(TreeNode root) {  return dfs(root).node;  }  // Return the result of the subtree at this node.  public Result dfs(TreeNode node) {  if (node == null) return new Result(null, 0);  Result L = dfs(node.left),  R = dfs(node.right);  if (L.dist > R.dist) return new Result(L.node, L.dist + 1);  if (L.dist < R.dist) return new Result(R.node, R.dist + 1);  return new Result(node, L.dist + 1);  }  }  /\*\*  \* The result of a subtree is:  \* Result.node: the largest depth node that is equal to or  \* an ancestor of all the deepest nodes of this subtree.  \* Result.dist: the number of nodes in the path from the root  \* of this subtree, to the deepest node in this subtree.  \*/  class Result {  TreeNode node;  int dist;  Result(TreeNode n, int d) {  node = n;  dist = d;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of nodes in the tree.
* Space Complexity: O(N)*O*(*N*).

**Plus One Linked List**

Given a non-negative integer represented as a linked list of digits, plus one to the integer.

The digits are stored such that the most significant digit is at the head of the list.

**Example 1:**

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

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

**Example 2:**

**Input:** head = [0]

**Output:** [1]

**Constraints:**

* The number of nodes in the linked list is in the range [1, 100].
* 0 <= Node.val <= 9
* The number represented by the linked list does not contain leading zeros except for the zero itself.

## Solution Article

#### Overview.

"Plus One" is a subset of a problem set "Add Number", and the solution patterns are the same.

All these problems could be solved in linear time, and the question here is how to solve without using addition operation or how to fit into constant space complexity.

The choice of algorithm should be based on input format:

1. Integers. Usually addition operation is not allowed for such a case. Use Bit Manipulation Approach. Here is an example: [Add Binary](https://leetcode.com/articles/add-binary/).
2. Strings. Use schoolbook bit by bit computation. Note, that to fit into constant space is not possible for languages with immutable strings, for ex. for Java and Python. Here is an example: [Add Binary](https://leetcode.com/articles/add-binary/).
3. Arrays. The same textbook addition. Here is an example: [Add to Array Form of Integer](https://leetcode.com/articles/add-to-array-form-of-integer/).
4. Linked Lists, current problem. Sentinel Head + Textbook Addition.

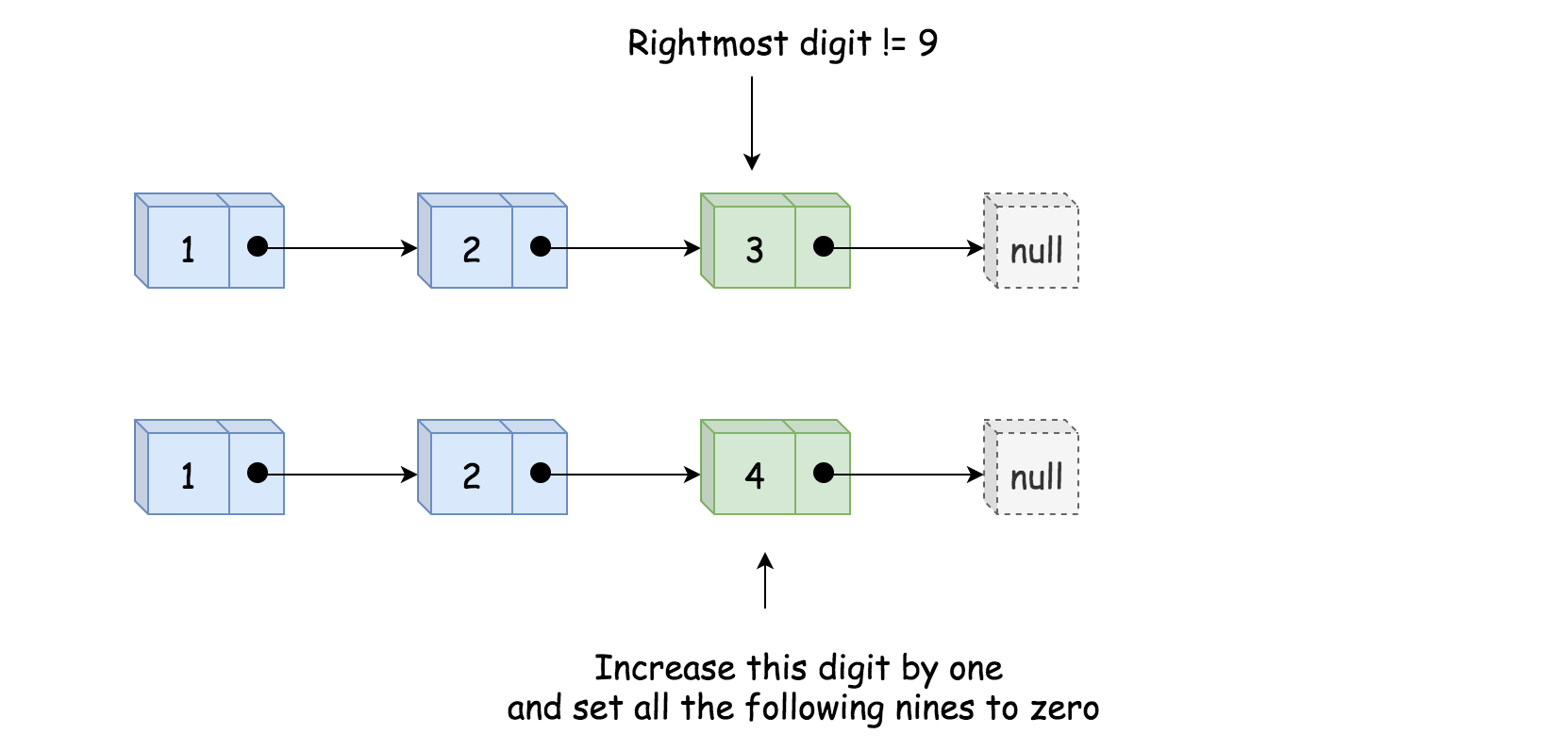
Note, that straightforward idea to convert everything into integers and then use addition could be risky for Java interviews because of possible overflow issues, [here is in more details](https://leetcode.com/articles/add-binary/).

#### Approach 1: Sentinel Head + Textbook Addition.

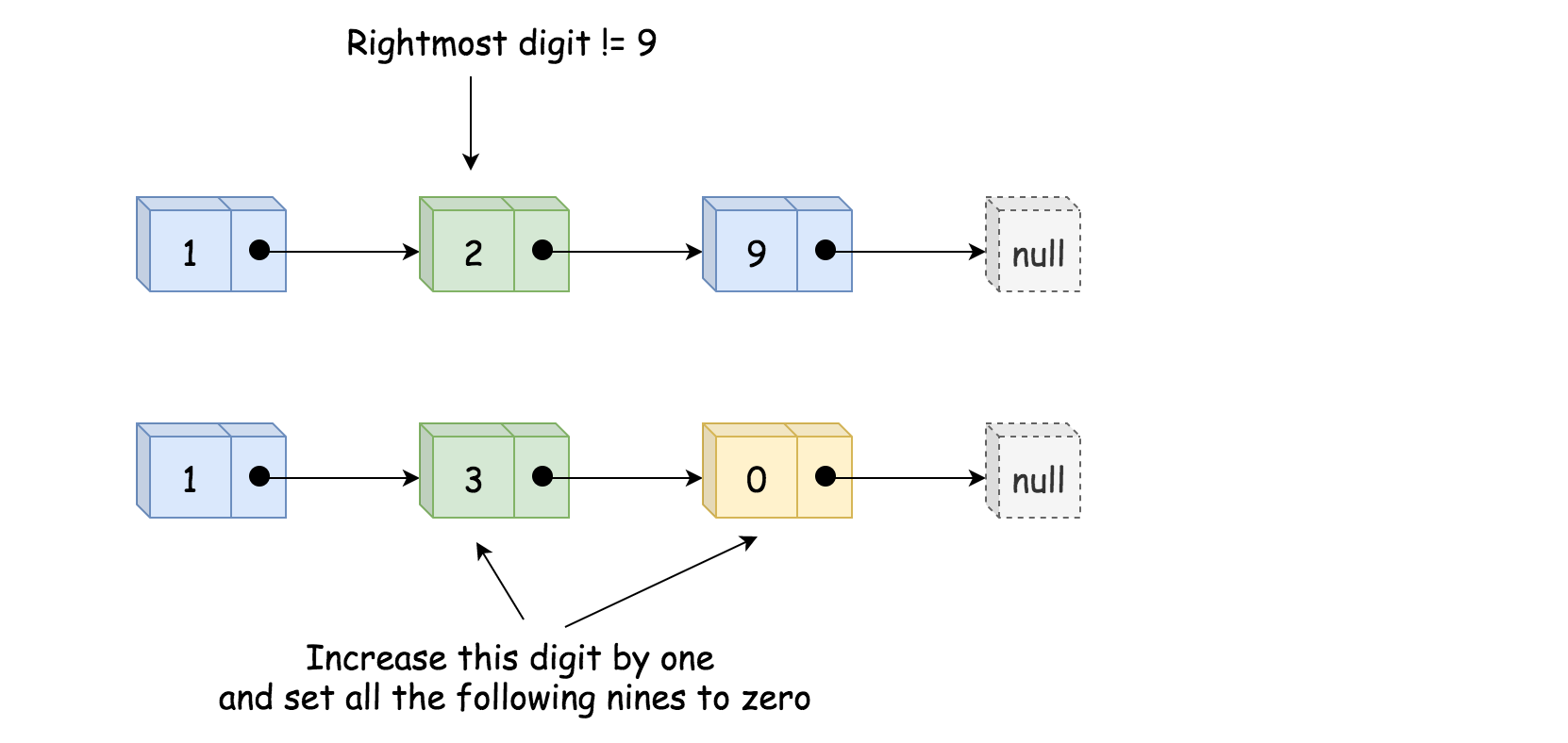
**Textbook Addition**

Let's identify the rightmost digit which is not equal to nine and increase that digit by one. All the following nines should be set to zero.

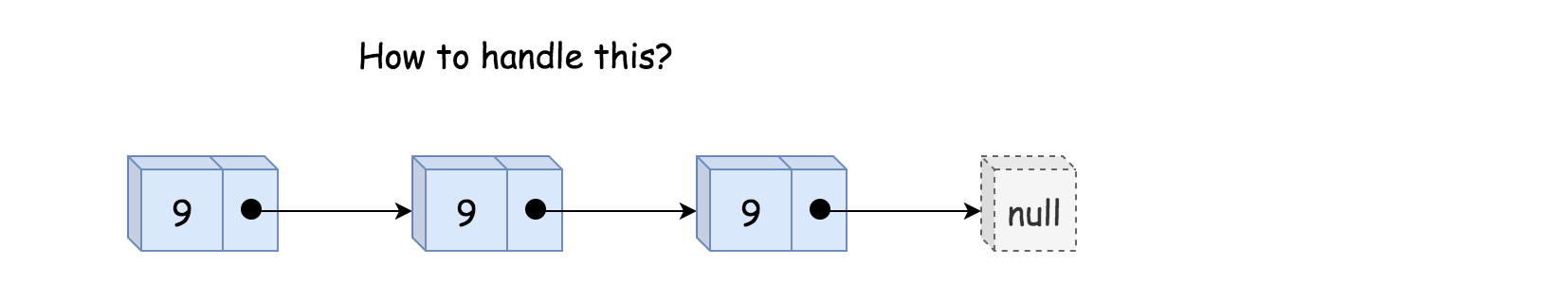
Here is the simplest use case which works fine.



Here is more difficult case which still passes.



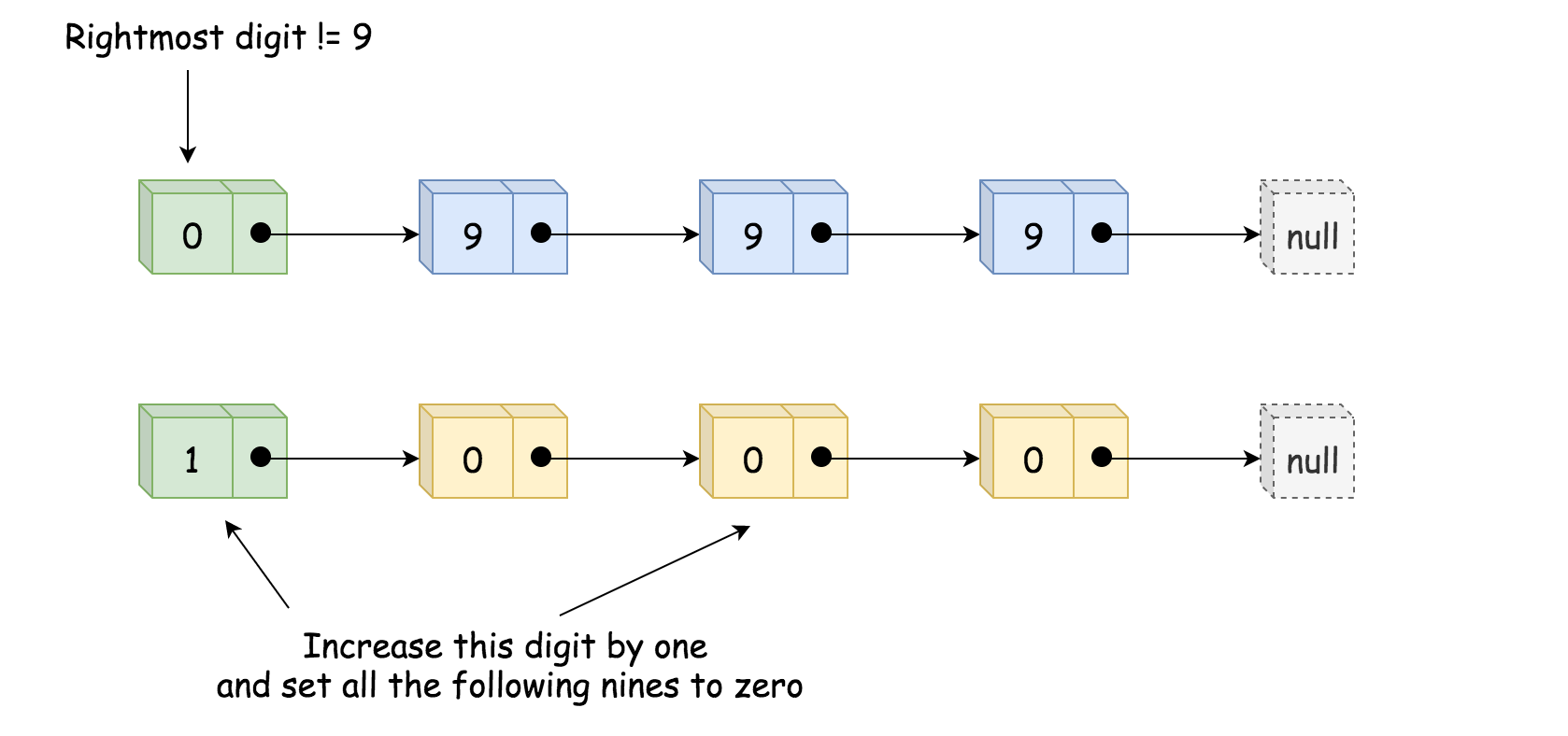
And here is the case which breaks everything.



**Sentinel Head**

To handle the last use case, one needs so called [Sentinel Node](https://en.wikipedia.org/wiki/Sentinel_node). Sentinel nodes are widely used for trees and linked lists as pseudo-heads, pseudo-tails, etc. They are purely functional, and usually don't hold any data. Their main purpose is to standardize the situation to avoid edge case handling.

For example, here one could add pseudo-head with zero value, and hence there will always be not-nine node.



**Algorithm**

* Initialize sentinel node as ListNode(0) and set it to be the new head: sentinel.next = head.
* Find the rightmost digit not equal to nine.
* Increase that digit by one.
* Set all the following nines to zero.
* Return sentinel node if it was set to 1, and head sentinel.next otherwise.

**Implementation**

|  |
| --- |
| class Solution {  public ListNode plusOne(ListNode head) {  // sentinel head  ListNode sentinel = new ListNode(0);  sentinel.next = head;  ListNode notNine = sentinel;  // find the rightmost not-nine digit  while (head != null) {  if (head.val != 9) {  notNine = head;  }  head = head.next;  }  // increase this rightmost not-nine digit by 1  notNine.val++;  notNine = notNine.next;  // set all the following nines to zeros  while (notNine != null) {  notNine.val = 0;  notNine = notNine.next;  }  return sentinel.val != 0 ? sentinel : sentinel.next;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since it's not more that two passes along the input list.
* Space complexity : \mathcal{O}(1)O(1).

**Cherry Pickup II**

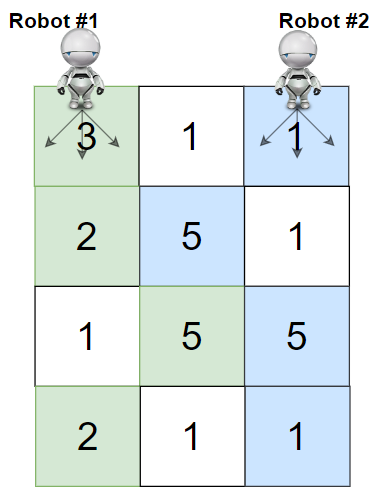
Given a rows x cols matrix grid representing a field of cherries. Each cell in grid represents the number of cherries that you can collect.

You have two robots that can collect cherries for you, Robot #1 is located at the top-left corner (0,0) , and Robot #2 is located at the top-right corner (0, cols-1) of the grid.

Return the maximum number of cherries collection using both robots  by following the rules below:

* From a cell (i,j), robots can move to cell (i+1, j-1) , (i+1, j) or (i+1, j+1).
* When any robot is passing through a cell, It picks it up all cherries, and the cell becomes an empty cell (0).
* When both robots stay on the same cell, only one of them takes the cherries.
* Both robots cannot move outside of the grid at any moment.
* Both robots should reach the bottom row in the grid.

**Example 1:**

****

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

**Output:** 24

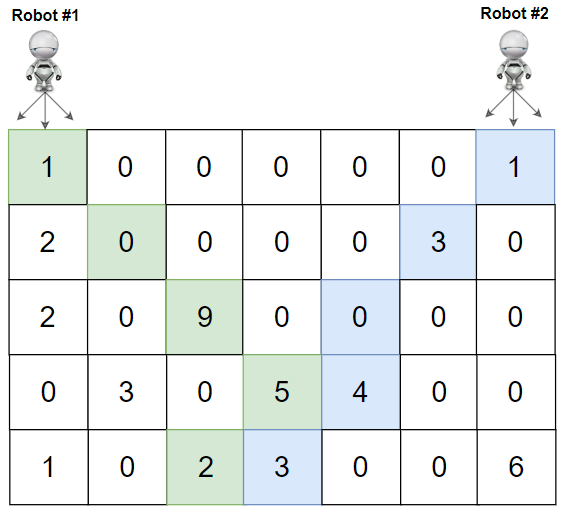
**Explanation:** Path of robot #1 and #2 are described in color green and blue respectively.

Cherries taken by Robot #1, (3 + 2 + 5 + 2) = 12.

Cherries taken by Robot #2, (1 + 5 + 5 + 1) = 12.

Total of cherries: 12 + 12 = 24.

**Example 2:**

****

**Input:** grid = [[1,0,0,0,0,0,1],[2,0,0,0,0,3,0],[2,0,9,0,0,0,0],[0,3,0,5,4,0,0],[1,0,2,3,0,0,6]]

**Output:** 28

**Explanation:** Path of robot #1 and #2 are described in color green and blue respectively.

Cherries taken by Robot #1, (1 + 9 + 5 + 2) = 17.

Cherries taken by Robot #2, (1 + 3 + 4 + 3) = 11.

Total of cherries: 17 + 11 = 28.

**Example 3:**

**Input:** grid = [[1,0,0,3],[0,0,0,3],[0,0,3,3],[9,0,3,3]]

**Output:** 22

**Example 4:**

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

**Output:** 4

**Constraints:**

* rows == grid.length
* cols == grid[i].length
* 2 <= rows, cols <= 70
* 0 <= grid[i][j] <= 100

   Hide Hint #1

Use dynammic programming, define DP[i][j][k]: The maximum cherries that both robots can take starting on the ith row, and column j and k of Robot 1 and 2 respectively.

#### Overview

You probably can guess from the problem title that this problem is an extension of the original [Cherry Pickup](https://leetcode.com/problems/cherry-pickup/).

In this problem, we do not need to return to the starting point but have two robots instead of one. However, the essence of this problem does not change, and the same method is still available.

Below, we will discuss two similar approaches: Dynamic Programming (Top Down) and Dynamic Programming (Bottom Up).

The first one is also known as dfs with memoization or memoization dp, and the second one is also known as tabulation dp. They have the same main idea but different coding approaches.

#### Approach #1: Dynamic Programming (Top Down)

**Intuition**

In this part, we will explain how to think of this approach step by step.

If you are only interested in the pure algorithm, you can jump to the algorithm part.

We need to move two robots! Note that the order of moving robot1 or robot2 does not matter since it would not impact the cherries we can pick. The number of cherries we can pick only depends on the tracks of our robots.

Therefore, we can move the robot1 and robot2 in any order we want. We aim to apply DP, so we are looking for an order that suitable for DP.

Let's try a few possible moving orders.

Can we move robot1 firstly to the bottom row, and then move robot2?

Maybe not. In this case, the movement of robot1 will impact the movement of robot2. In other words, the optimal track of robot2 depends on the track of robot1. If we want to apply DP, we need to record the whole track of robot1 as the state. The number of sub-problems is too much.

In fact, in any case, when we move one robot several steps earlier than the other, the movement of the first robot will impact the movement of the second robot.

Unless we move them **synchronously** (i.e., move one step of robot1 and robot2 at the same time).

Let's define the DP state as (row1, col1, row2, col2), where (row1, col1) represents the location of robot1, and (row2, col2) represents the location of robot2.

If we move them synchronously, robot1 and robot2 will always on the same row. Therefore, row1 == row2.

Let row = row1. The DP state is simplified to (row, col1, col2), where (row, col1) represents the location of robot1, and (row, col2) represents the location of robot2.

OK, time to define the DP function.

Since it's a top-down DP approach, we try to solve the problem with the DP function. Check approach 2 for DP array (bottom-up).

Let dp(row, col1, col2) return the maximum cherries we can pick if robot1 starts at (row, col1) and robot2 starts at (row, col2).

You can try changing different dp meaning to yield some other similar approaches. For example, let dp(row, col1, col2) mean the maximum cherries we can pick if robot1 **ends** at (row, col1) and robot2 **ends** at (row, col2).

The base cases are that robot1 and robot2 both start at the bottom line. In this case, they do not need to move. All we need to do is to collect the cherries at current cells. Remember not to double count if robot1 and robot2 are at exactly the same cell.

In other cases, we need to add the maximum cherries robots can pick in the future. Here comes the transition function.

Since we move robots synchronously, and each robot has three different movements for one step, we totally have 3\*3 = 93∗3=9 possible movements for two robots:

ROBOT1 | ROBOT2

------------------------

LEFT DOWN | LEFT DOWN

LEFT DOWN | DOWN

LEFT DOWN | RIGHT DOWN

DOWN | LEFT DOWN

DOWN | DOWN

DOWN | RIGHT DOWN

RIGHT DOWN | LEFT DOWN

RIGHT DOWN | DOWN

RIGHT DOWN | RIGHT DOWN

The maximum cherries robots can pick in the future would be the max of those 9 movements, which is the maximum of dp(row+1, new\_col1, new\_col2), where new\_col1 can be col1, col1+1, or col1-1, and new\_col2 can be col2, col2+1, or col2-1.

Remember to use a map or an array to store the results of our dp function to prevent redundant calculating.

**Algorithm**

Define a dp function that takes three integers row, col1, and col2 as input.

(row, col1) represents the location of robot1, and (row, col2) represents the location of robot2.

The dp function returns the maximum cherries we can pick if robot1 starts at (row, col1) and robot2 starts at (row, col2).

In the dp function:

* Collect the cherry at (row, col1) and (row, col2). Do not double count if col1 == col2.
* If we do not reach the last row, we need to add the maximum cherries we can pick in the future.
* The maximum cherries we can pick in the future is the maximum of dp(row+1, new\_col1, new\_col2), where new\_col1 can be col1, col1+1, or col1-1, and new\_col2 can be col2, col2+1, or col2-1.
* Return the total cherries we can pick.

Finally, return dp(row=0, col1=0, col2=last\_column) in the main function.

**Implementation**

|  |
| --- |
| class Solution {  public int cherryPickup(int[][] grid) {  int m = grid.length;  int n = grid[0].length;  int[][][] dpCache = new int[m][n][n];  // initial all elements to -1 to mark unseen  for (int i = 0; i < m; i++) {  for (int j = 0; j < n; j++) {  for (int k = 0; k < n; k++) {  dpCache[i][j][k] = -1;  }  }  }  return dp(0, 0, n - 1, grid, dpCache);  }  private int dp(int row, int col1, int col2, int[][] grid, int[][][] dpCache) {  if (col1 < 0 || col1 >= grid[0].length || col2 < 0 || col2 >= grid[0].length) {  return 0;  }  // check cache  if (dpCache[row][col1][col2] != -1) {  return dpCache[row][col1][col2];  }  // current cell  int result = 0;  result += grid[row][col1];  if (col1 != col2) {  result += grid[row][col2];  }  // transition  if (row != grid.length - 1) {  int max = 0;  for (int newCol1 = col1 - 1; newCol1 <= col1 + 1; newCol1++) {  for (int newCol2 = col2 - 1; newCol2 <= col2 + 1; newCol2++) {  max = Math.max(max, dp(row + 1, newCol1, newCol2, grid, dpCache));  }  }  result += max;  }  dpCache[row][col1][col2] = result;  return result;  }  } |

**Complexity Analysis**

Let M*M* be the number of rows in grid and N*N* be the number of columns in grid.

* Time Complexity: \mathcal{O}(MN^2)O(*MN*2), since our helper function have three variables as input, which have M*M*, N*N*, and N*N* possible values respectively. In the worst case, we have to calculate them all once, so that would cost \mathcal{O}(M \cdot N \cdot N) = \mathcal{O}(MN^2)O(*M*⋅*N*⋅*N*)=O(*MN*2). Also, since we save the results after calculating, we would not have repeated calculation.
* Space Complexity: \mathcal{O}(MN^2)O(*MN*2), since our helper function have three variables as input, and they have M*M*, N*N*, and N*N* possible values respectively. We need a map with size of \mathcal{O}(M \cdot N \cdot N) = \mathcal{O}(MN^2)O(*M*⋅*N*⋅*N*)=O(*MN*2) to store the results.

#### Approach #2: Dynamic Programming (Bottom Up)

**Intuition**

Similarly, we need a three-dimension array dp[row][col1][col2] to store calculated results:

dp[row][col1][col2] represents the maximum cherries we can pick if robot1 starts at (row, col1) and robot2 starts at (row, col2).

Remember, we move robot1 and robot2 synchronously, so they are always on the same row.

The base cases are that robot1 and robot2 both start at the bottom row. In this case, we only need to calculate the cherry at current cells.

Otherwise, apply the transition equation to get the maximum cherries we can pick in the future.

Since the base case is at the bottom row, we need to iterate from the bottom row to the top row when filling the dp array.

You can use state compression to save the first dimension since we only need dp[row+1] when calculating dp[row].

You can change the meaning of dp[row][col1][col2] and some corresponding codes to get some other similar approaches. For example, let dp[row][col1][col2] mean the maximum cherries we can pick if robot1 **ends** at (row, col1) and robot2 **ends** at (row, col2).

**Algorithm**

Define a three-dimension dp array where each dimension has a size of m, n, and n respectively.

dp[row][col1][col2] represents the maximum cherries we can pick if robot1 starts at (row, col1) and robot2 starts at (row, col2).

To compute dp[row][col1][col2] (transition equation):

* Collect the cherry at (row, col1) and (row, col2). Do not double count if col1 == col2.
* If we are not in the last row, we need to add the maximum cherries we can pick in the future.
* The maximum cherries we can pick in the future is the maximum of dp[row+1][new\_col1][new\_col2], where new\_col1 can be col1, col1+1, or col1-1, and new\_col2 can be col2, col2+1, or col2-1.

Finally, return dp[0][0][last\_column].

State compression can be used to save the first dimension: dp[col1][col2]. Just reuse the dp array after iterating one row.

**Implementation**

|  |
| --- |
| class Solution {  public int cherryPickup(int[][] grid) {  int m = grid.length;  int n = grid[0].length;  int dp[][][] = new int[m][n][n];  for (int row = m - 1; row >= 0; row--) {  for (int col1 = 0; col1 < n; col1++) {  for (int col2 = 0; col2 < n; col2++) {  int result = 0;  // current cell  result += grid[row][col1];  if (col1 != col2) {  result += grid[row][col2];  }  // transition  if (row != m - 1) {  int max = 0;  for (int newCol1 = col1 - 1; newCol1 <= col1 + 1; newCol1++) {  for (int newCol2 = col2 - 1; newCol2 <= col2 + 1; newCol2++) {  if (newCol1 >= 0 && newCol1 < n && newCol2 >= 0 && newCol2 < n) {  max = Math.max(max, dp[row + 1][newCol1][newCol2]);  }  }  }  result += max;  }  dp[row][col1][col2] = result;  }  }  }  return dp[0][0][n - 1];  }  } |

**Complexity Analysis**

Let M*M* be the number of rows in grid and N*N* be the number of columns in grid.

* Time Complexity: \mathcal{O}(MN^2)O(*MN*2), since our dynamic programming has three nested for-loops, which have M*M*, N*N*, and N*N* iterations respectively. In total, it costs \mathcal{O}(M \cdot N \cdot N) = \mathcal{O}(MN^2)O(*M*⋅*N*⋅*N*)=O(*MN*2).
* Space Complexity: \mathcal{O}(MN^2)O(*MN*2) if not use state compression, since our dp array has \mathcal{O}(M \cdot N \cdot N) = \mathcal{O}(MN^2)O(*M*⋅*N*⋅*N*)=O(*MN*2) elements. \mathcal{O}(N^2)O(*N*2) if use state compression, since we can reuse the first dimension, and our dp array only has \mathcal{O}(N \cdot N) = \mathcal{O}(N^2)O(*N*⋅*N*)=O(*N*2) elements.

**Decoded String at Index**

An encoded string S is given.  To find and write the decoded string to a tape, the encoded string is read **one character at a time** and the following steps are taken:

* If the character read is a letter, that letter is written onto the tape.
* If the character read is a digit (say d), the entire current tape is repeatedly written d-1 more times in total.

Now for some encoded string S, and an index K, find and return the K-th letter (1 indexed) in the decoded string.

**Example 1:**

**Input:** S = "leet2code3", K = 10

**Output:** "o"

**Explanation:**

The decoded string is "leetleetcodeleetleetcodeleetleetcode".

The 10th letter in the string is "o".

**Example 2:**

**Input:** S = "ha22", K = 5

**Output:** "h"

**Explanation:**

The decoded string is "hahahaha". The 5th letter is "h".

**Example 3:**

**Input:** S = "a2345678999999999999999", K = 1

**Output:** "a"

**Explanation:**

The decoded string is "a" repeated 8301530446056247680 times. The 1st letter is "a".

**Constraints:**

* 2 <= S.length <= 100
* S will only contain lowercase letters and digits 2 through 9.
* S starts with a letter.
* 1 <= K <= 10^9
* It's guaranteed that K is less than or equal to the length of the decoded string.
* The decoded string is guaranteed to have less than 2^63 letters.

## Solution

#### Approach 1: Work Backwards

**Intuition**

If we have a decoded string like appleappleappleappleappleapple and an index like K = 24, the answer is the same if K = 4.

In general, when a decoded string is equal to some word with size length repeated some number of times (such as apple with size = 5 repeated 6 times), the answer is the same for the index K as it is for the index K % size.

We can use this insight by working backwards, keeping track of the size of the decoded string. Whenever the decoded string would equal some word repeated d times, we can reduce K to K % (word.length).

**Algorithm**

First, find the length of the decoded string. After, we'll work backwards, keeping track of size: the length of the decoded string after parsing symbols S[0], S[1], ..., S[i].

If we see a digit S[i], it means the size of the decoded string after parsing S[0], S[1], ..., S[i-1] will be size / Integer(S[i]). Otherwise, it will be size - 1.

|  |
| --- |
| class Solution {  public String decodeAtIndex(String S, int K) {  long size = 0;  int N = S.length();  // Find size = length of decoded string  for (int i = 0; i < N; ++i) {  char c = S.charAt(i);  if (Character.isDigit(c))  size \*= c - '0';  else  size++;  }  for (int i = N-1; i >= 0; --i) {  char c = S.charAt(i);  K %= size;  if (K == 0 && Character.isLetter(c))  return Character.toString(c);  if (Character.isDigit(c))  size /= c - '0';  else  size--;  }  throw null;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of S.
* Space Complexity: O(1)*O*(1).

**Smallest Range II**

Given an array A of integers, for each integer A[i] we need to choose **either x = -K or x = K**, and add x to A[i] **(only once)**.

After this process, we have some array B.

Return the smallest possible difference between the maximum value of B and the minimum value of B.

**Example 1:**

**Input:** A = [1], K = 0

**Output:** 0

**Explanation**: B = [1]

**Example 2:**

**Input:** A = [0,10], K = 2

**Output:** 6

**Explanation**: B = [2,8]

**Example 3:**

**Input:** A = [1,3,6], K = 3

**Output:** 3

**Explanation**: B = [4,6,3]

**Note:**

1. 1 <= A.length <= 10000
2. 0 <= A[i] <= 10000
3. 0 <= K <= 10000

## Solution

#### Approach 1: Linear Scan

**Intuition**

As in Smallest Range I, smaller A[i] will choose to increase their value ("go up"), and bigger A[i] will decrease their value ("go down").

**Algorithm**

We can formalize the above concept: if A[i] < A[j], we don't need to consider when A[i] goes down while A[j] goes up. This is because the interval (A[i] + K, A[j] - K) is a subset of (A[i] - K, A[j] + K) (here, (a, b) for a > b denotes (b, a) instead.)

That means that it is never worse to choose (up, down) instead of (down, up). We can prove this claim that one interval is a subset of another, by showing both A[i] + K and A[j] - K are between A[i] - K and A[j] + K.

For sorted A, say A[i] is the largest i that goes up. Then A[0] + K, A[i] + K, A[i+1] - K, A[A.length - 1] - K are the only relevant values for calculating the answer: every other value is between one of these extremal values.

|  |
| --- |
| class Solution {  public int smallestRangeII(int[] A, int K) {  int N = A.length;  Arrays.sort(A);  int ans = A[N-1] - A[0];  for (int i = 0; i < A.length - 1; ++i) {  int a = A[i], b = A[i+1];  int high = Math.max(A[N-1] - K, a + K);  int low = Math.min(A[0] + K, b - K);  ans = Math.min(ans, high - low);  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the length of the A.
* Space complexity : \mathcal{O}(N)O(*N*) or \mathcal{O}(\log{N})O(log*N*)
  + The space complexity of the sorting algorithm depends on the implementation of each program language.
  + For instance, the list.sort() function in Python is implemented with the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm whose space complexity is \mathcal{O}(N)O(*N*).
  + In Java, the [Arrays.sort()](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html" \l "sort-byte:A-) is implemented as a variant of quicksort algorithm whose space complexity is \mathcal{O}(\log{N})O(log*N*).

**Find Nearest Right Node in Binary Tree**

**Solution**

Given the root of a binary tree and a node u in the tree, return *the****nearest****node on the****same level****that is to the****right****of* u*, or return* null *if*u *is the rightmost node in its level*.

**Example 1:**



**Input:** root = [1,2,3,null,4,5,6], u = 4

**Output:** 5

**Explanation:** The nearest node on the same level to the right of node 4 is node 5.

**Example 2:**

****

**Input:** root = [3,null,4,2], u = 2

**Output:** null

**Explanation:** There are no nodes to the right of 2.

**Example 3:**

**Input:** root = [1], u = 1

**Output:** null

**Example 4:**

**Input:** root = [3,4,2,null,null,null,1], u = 4

**Output:** 2

**Constraints:**

* The number of nodes in the tree is in the range [1, 105].
* 1 <= Node.val <= 105
* All values in the tree are **distinct**.
* u is a node in the binary tree rooted at root.

Hide Hint #1

Use BFS, traverse the tree level by level and always push the left node first

   Hide Hint #2

When you reach the target node, mark a boolean variable true

   Hide Hint #3

If you meet another node in the same level after marking the boolean true, return this node.

   Hide Hint #4

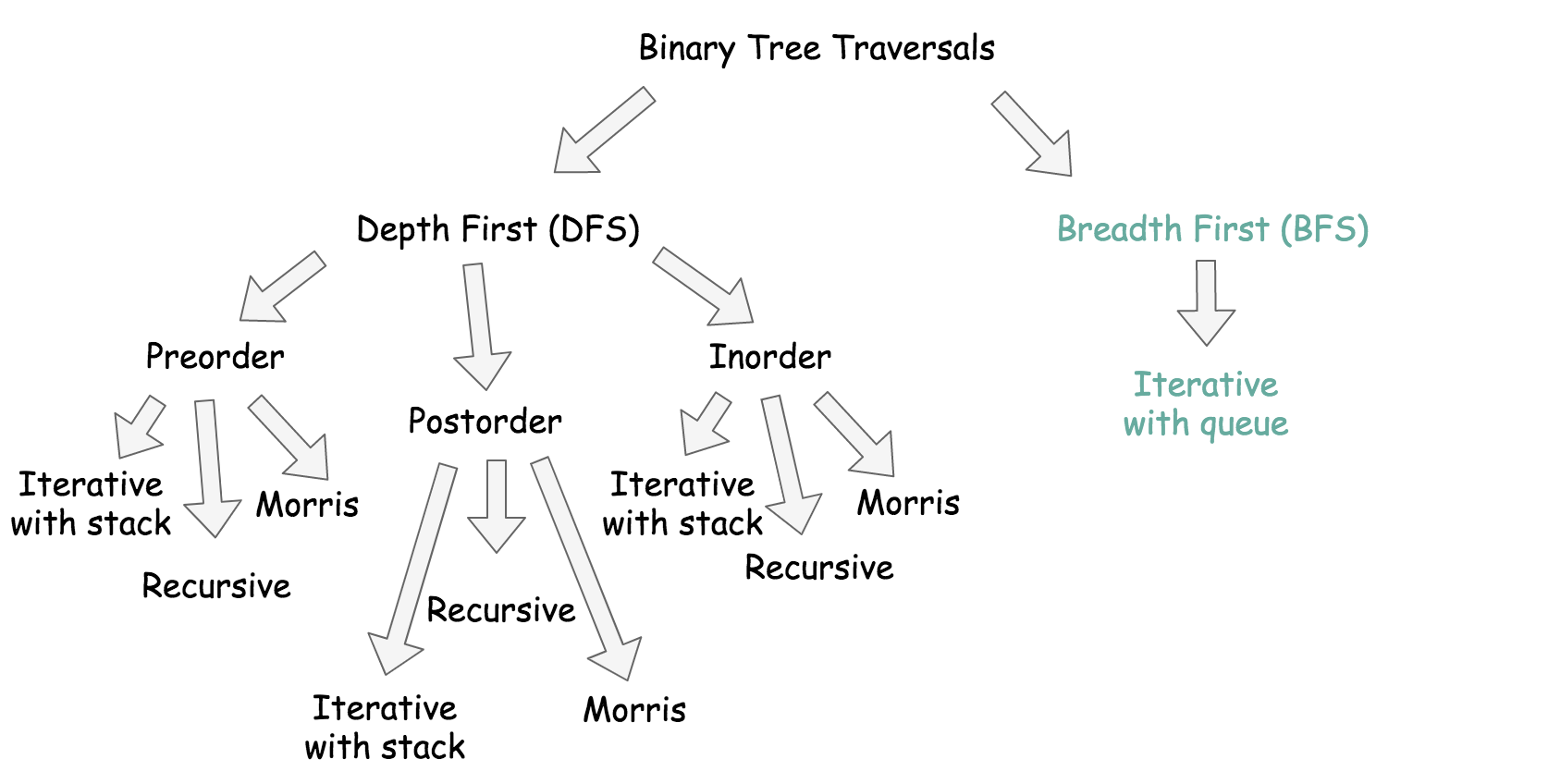
If you did not meet new nodes in the same level and started traversing a new level, return Null

## Solution Article

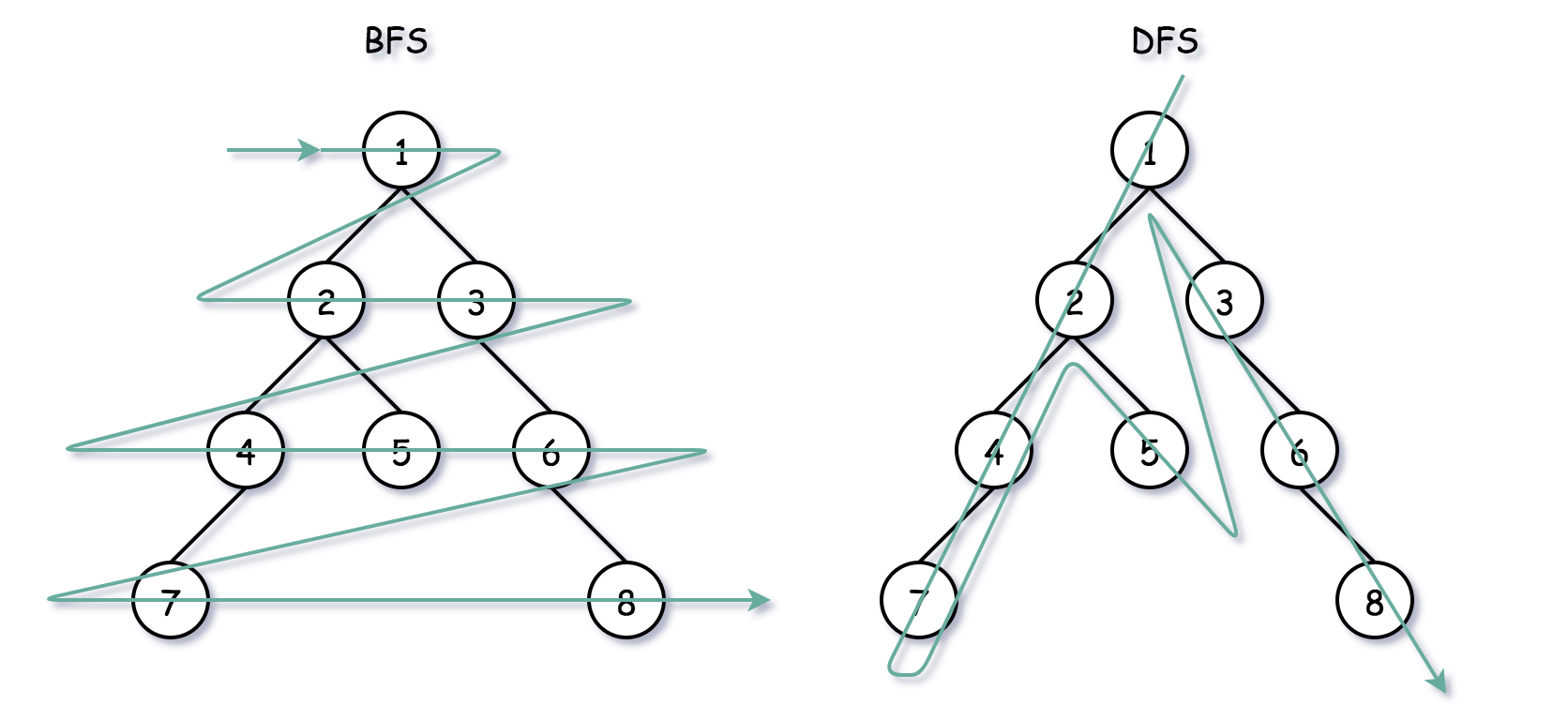
#### Overview

**DFS vs. BFS**

There are two ways to traverse the tree: DFS depth first search and BFS breadth first search. Here is a small summary



BFS traverses level by level, and DFS first goes to the leaves.



Which approach to choose, BFS or DFS?

* The problem is to return the nearest node on the same level that is to the right of u, so it's the way more natural to implement BFS here.
* Time complexity is the same \mathcal{O}(N)O(*N*) both for DFS and BFS since one has to visit all nodes.
* Space complexity is \mathcal{O}(H)O(*H*) for DFS and \mathcal{O}(D)O(*D*) for BFS, where H*H* is a tree height, and D*D* is a tree diameter. They both result in \mathcal{O}(N)O(*N*) space in the worst-case scenarios: skewed tree for DFS and complete tree for BFS.

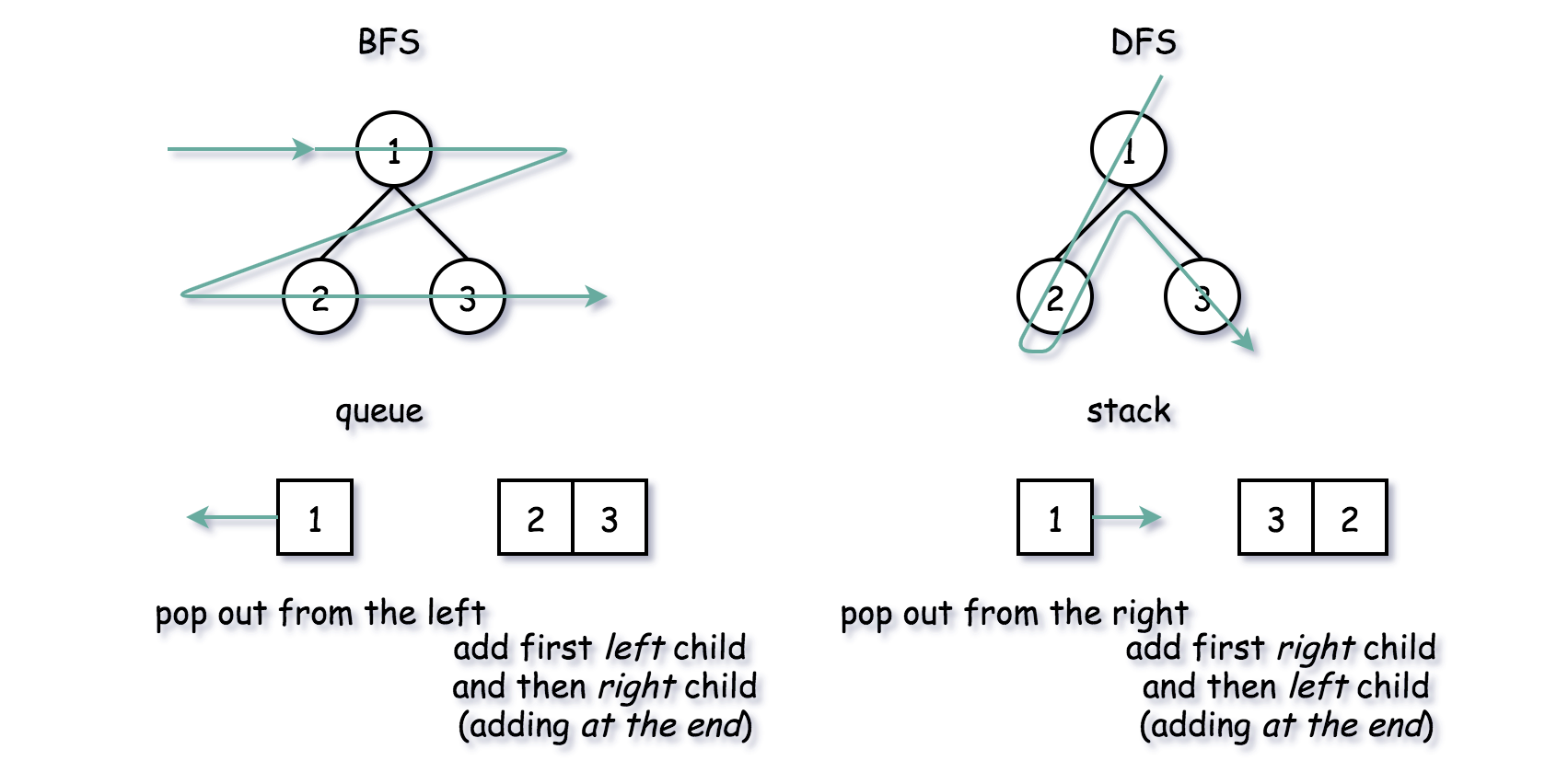
Let's use the opportunity to check out three different BFS implementations with the queue, Approach 1 - Approach 3.

If you prefer to use DFS on the interviews - check the Approach 4.

**BFS implementation**

All three implementations use the queue in a standard BFS way:

* Push the root into the queue.
* Pop-out a node from the left.
* Push the left child into the queue, and then push the right child.



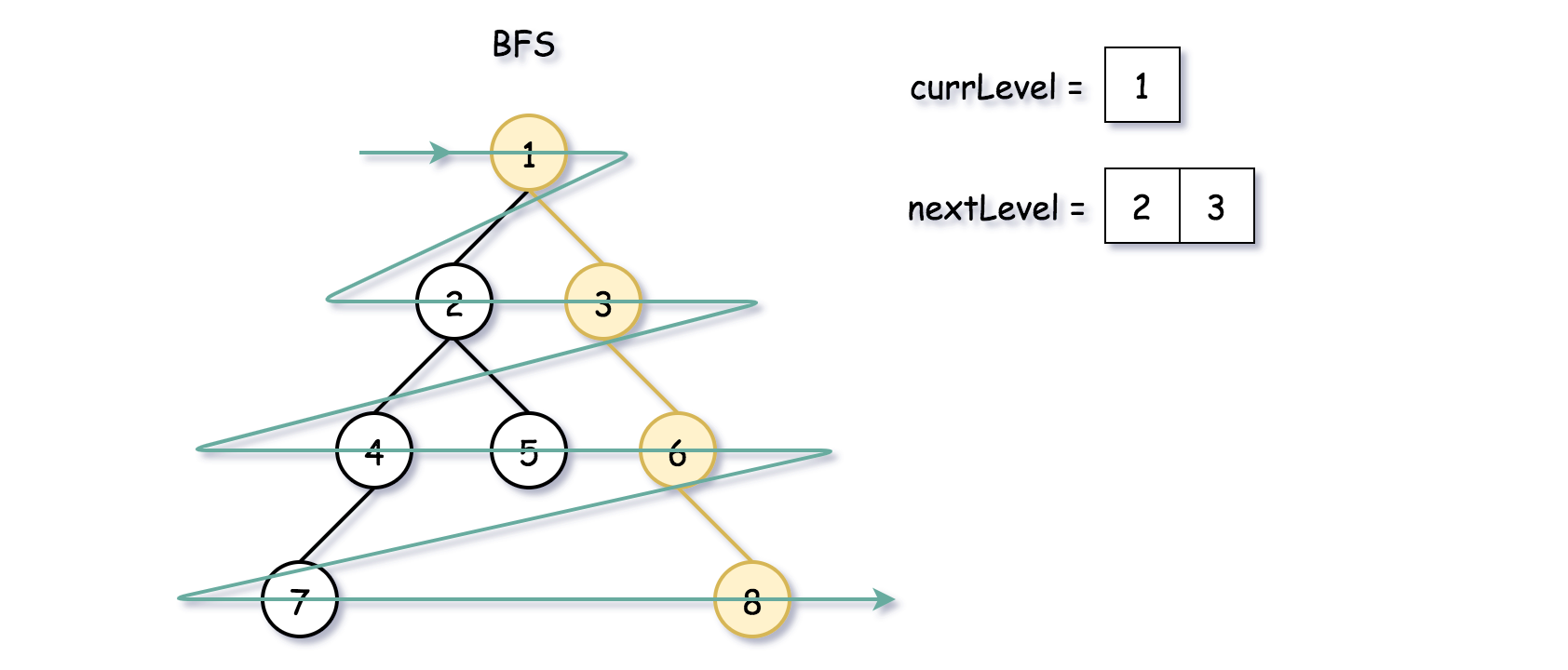
**Three BFS approaches**

The difference is how to identify the end of the level:

* Two queues, one for the previous level and one for the current.
* One queue with sentinel to mark the end of the level.
* One queue + level size measurement.

#### Approach 1: BFS: Two Queues

Let's use two queues: one for the current level, and one for the next. The idea is to pop the nodes one by one from the current level and push their children into the next level queue.



**Algorithm**

* Initiate two queues: one for the current level, and one for the next. Add root into nextLevel queue.
* While nextLevel queue is not empty:
  + Initiate the current level: currLevel = nextLevel, and empty the next level nextLevel.
  + While current level queue is not empty:
    - Pop out a node from the current level queue.
    - If this node is u, return the next node from the queue. If there is no more nodes in nextLevel queue, return null.
    - Add first left and then right child node into nextLevel queue.

**Implementation**

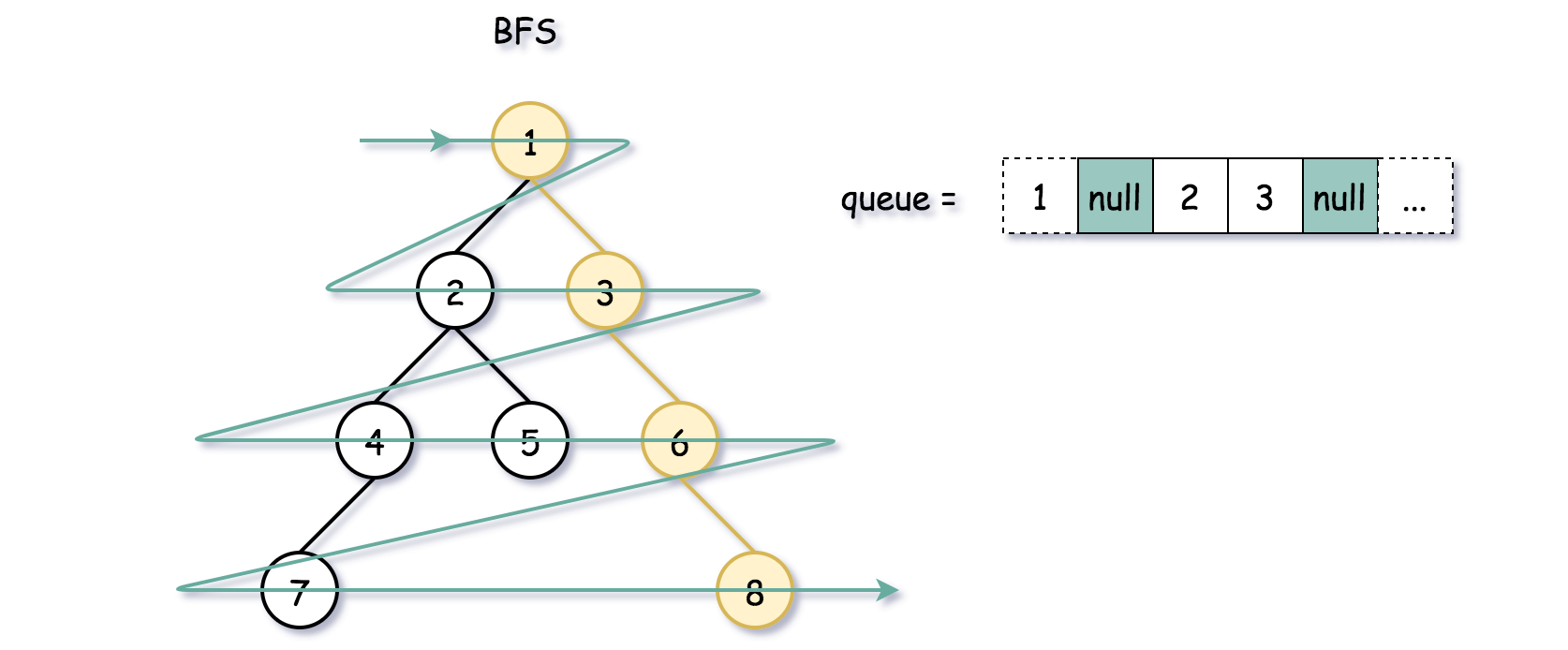
|  |
| --- |
| class Solution {  public TreeNode findNearestRightNode(TreeNode root, TreeNode u) {  if (root == null) return null;  ArrayDeque<TreeNode> nextLevel = new ArrayDeque() {{ offer(root); }};  ArrayDeque<TreeNode> currLevel = new ArrayDeque();  TreeNode node = null;  while (!nextLevel.isEmpty()) {  // prepare for the next level  currLevel = nextLevel.clone();  nextLevel.clear();  while (!currLevel.isEmpty()) {  node = currLevel.poll();  if (node == u)  return currLevel.poll();  // add child nodes of the current level  // in the queue for the next level  if (node.left != null)  nextLevel.offer(node.left);  if (node.right != null)  nextLevel.offer(node.right);  }  }  return null;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*) since one has to visit each node.
* Space complexity: \mathcal{O}(D)O(*D*) to keep the queues, where D*D* is a tree diameter. Let's use the last level to estimate the queue size. This level could contain up to N/2*N*/2 tree nodes in the case of [complete binary tree](https://leetcode.com/problems/count-complete-tree-nodes/).

#### Approach 2: BFS: One Queue + Sentinel

Another approach is to push all the nodes in one queue and to use a [sentinel node](https://en.wikipedia.org/wiki/Sentinel_node) to separate the levels. Typically, one could use null as a sentinel.



The first step is to initiate the first level: root + null as a sentinel. Once it's done, continue to pop the nodes one by one from the left and push their children to the right. Stop each time the current node is null because it means we hit the end of the current level. Each stop is a time to push null in the queue to mark the end of the next level.

**Algorithm**

* Initiate the queue by adding a root. Add null sentinel to mark the end of the first level.
* Initiate the current node as root.
* While queue is not empty:
  + Pop the current node from the queue curr = queue.poll().
  + If this node is u, return the next node from the queue. If there is no more nodes in the queue, return null.
  + If the current node is not null:
    - Add first left and then right child node into the queue.
    - Update the current node: curr = queue.poll().
  + Now, the current node is null, i.e. we reached the end of the current level. If the queue is not empty, push the null node as a sentinel, to mark the end of the next level.

**Implementation**

Note that ArrayDeque in Java doesn't support null elements, and hence the data structure to use here is LinkedList.

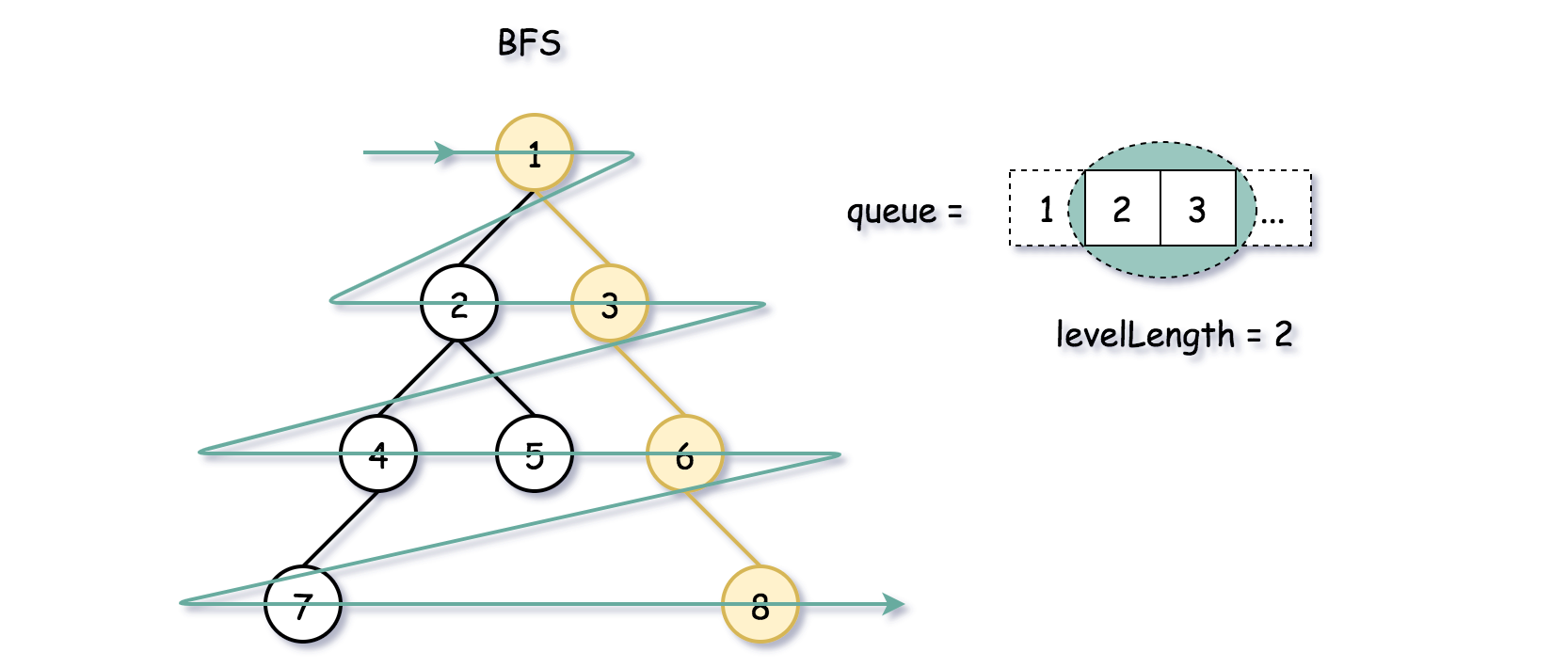
|  |
| --- |
| class Solution {  public TreeNode findNearestRightNode(TreeNode root, TreeNode u) {  if (root == null) return null;  Queue<TreeNode> queue = new LinkedList(){{ offer(root); offer(null); }};  TreeNode curr = null;  while (!queue.isEmpty()) {  curr = queue.poll();  if (curr != null) {  // if it's the given node  if (curr == u)  return queue.poll();  // add child nodes in the queue  if (curr.left != null) {  queue.offer(curr.left);  }  if (curr.right != null) {  queue.offer(curr.right);  }  } else {  // add a sentinel to mark end of level  if (!queue.isEmpty())  queue.offer(null);  }  }  return null;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*) since one has to visit each node.
* Space complexity: \mathcal{O}(D)O(*D*) to keep the queues, where D*D* is a tree diameter. Let's use the last level to estimate the queue size. This level could contain up to N/2*N*/2 tree nodes in the case of [complete binary tree](https://leetcode.com/problems/count-complete-tree-nodes/).

#### Approach 3: BFS: One Queue + Level Size Measurements

Instead of using the sentinel, we could write down the length of the current level.



**Algorithm**

* Initiate the queue by adding a root.
* While the queue is not empty:
  + Write down the length of the current level: levelLength = queue.size().
  + Iterate over i from 0 to level\_length - 1:
    - Pop the current node from the queue: node = queue.poll().
    - If this node is u, return the next node from the queue. Check that the next node is on the same level: i != levelLength - 1, otherwise return null.
    - Add first left and then right child node into the queue.

**Implementation**

|  |
| --- |
| class Solution {  public TreeNode findNearestRightNode(TreeNode root, TreeNode u) {  if (root == null) return null;  Queue<TreeNode> queue = new LinkedList<>();  queue.offer(root);  while (!queue.isEmpty()) {  int levelSize = queue.size();  for (int i = 0; i < levelSize; i++) {  TreeNode curr = queue.poll();  // if it's the given node  if (curr == u) {  if (i == levelSize - 1) {  return null;  }  else {  return queue.poll();  }  }  if (curr.left != null) queue.offer(curr.left);  if (curr.right != null) queue.offer(curr.right);  }  }  return null;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*) since one has to visit each node.
* Space complexity: \mathcal{O}(D)O(*D*) to keep the queues, where D*D* is a tree diameter. Let's use the last level to estimate the queue size. This level could contain up to N/2*N*/2 tree nodes in the case of [complete binary tree](https://leetcode.com/problems/count-complete-tree-nodes/).

#### Approach 4: Recursive DFS: Preorder Traversal

Everyone likes recursive DFS because of its simplicity, so let's add it here as well. The idea is straightforward: to perform standard preorder traversal of the tree, starting each time from the leftmost child.

|  |
| --- |
| class Solution {  private int uDepth;  private TreeNode nextNode, targetNode;  public TreeNode findNearestRightNode(TreeNode root, TreeNode u) {  uDepth = - 1;  targetNode = u;  nextNode = null;  dfs(root, 0);  return nextNode;  }  public void dfs(TreeNode currNode, int depth) {  // the depth to look for next node is identified  if (currNode == targetNode) {  uDepth = depth;  return;  }  // we're on the level to look for the next node  if (depth == uDepth) {  if (nextNode == null) nextNode = currNode;  return;  }  // continue to traverse the tree  if (currNode.left != null) dfs(currNode.left, depth + 1);  if (currNode.right != null) dfs(currNode.right, depth + 1);  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*) since one has to visit each node.
* Space complexity: \mathcal{O}(H)O(*H*) to keep the recursion stack, where H*H* is the tree height. The worst-case situation is a skewed tree when H = N*H*=*N*.

**Next Greater Element III**

Given a positive integer n, find *the smallest integer which has exactly the same digits existing in the integer* n *and is greater in value than* n. If no such positive integer exists, return -1.

**Note** that the returned integer should fit in **32-bit integer**, if there is a valid answer but it does not fit in **32-bit integer**, return -1.

**Example 1:**

**Input:** n = 12

**Output:** 21

**Example 2:**

**Input:** n = 21

**Output:** -1

**Constraints:**

* 1 <= n <= 231 - 1

## Solution

#### Approach #1 Brute Force

To solve the given problem, we treat the given number as a string, s*s*. In this approach, we find out every possible permutation of list formed by the elements of the string s*s* formed. We form a list of strings, list*list*, containing all the permutations possible. Then, we sort the given list*list* to find out the permutation which is just larger than the given one. But this one will be a very naive approach, since it requires us to find out every possible permutation which will take really long time.

|  |
| --- |
| public class Solution {  public String swap(String s, int i0, int i1) {  if (i0 == i1)  return s;  String s1 = s.substring(0, i0);  String s2 = s.substring(i0 + 1, i1);  String s3 = s.substring(i1 + 1);  return s1 + s.charAt(i1) + s2 + s.charAt(i0) + s3;  }  ArrayList < String > list = new ArrayList < > ();  void permute(String a, int l, int r) {  int i;  if (l == r)  list.add(a);  else {  for (i = l; i <= r; i++) {  a = swap(a, l, i);  permute(a, l + 1, r);  a = swap(a, l, i);  }  }  }  public int nextGreaterElement(int n) {  String s = "" + n;  permute(s, 0, s.length() - 1);  Collections.sort(list);  int i;  for (i = list.size() - 1; i >= 0; i--) {  if (list.get(i).equals("" + n))  break;  }  return i == list.size() - 1 ? -1 : Integer.parseInt(list.get(i + 1));  }  } |

**Complexity Analysis**

* Time complexity : O(n!)*O*(*n*!). A total of n!*n*! permutations are possible for a number consisting of n*n* digits.
* Space complexity : O(n!)*O*(*n*!). A total of n!*n*! permutations are possible for a number consisting of n*n* digits, with each permutation consisting of n*n* digits.

#### Approach #2 Linear Solution

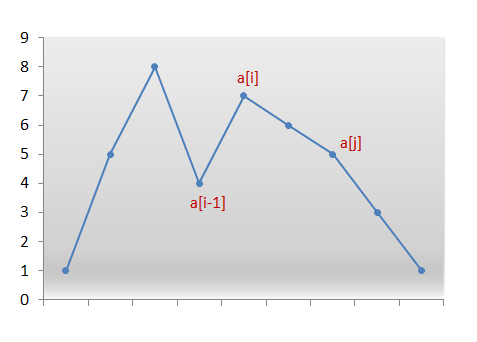
**Algorithm**

In this case as well, we consider the given number n*n* as a character array a*a*. First, we observe that for any given sequence that is in descending order, no next larger permutation is possible. For example, no next permutation is possible for the following array:

[9, 5, 4, 3, 1]

We need to find the first pair of two successive numbers a[i]*a*[*i*] and a[i-1]*a*[*i*−1], from the right, which satisfy a[i] > a[i-1]*a*[*i*]>*a*[*i*−1]. Now, no rearrangements to the right of a[i-1]*a*[*i*−1] can create a larger permutation since that subarray consists of numbers in descending order. Thus, we need to rearrange the numbers to the right of a[i-1]*a*[*i*−1] including itself.

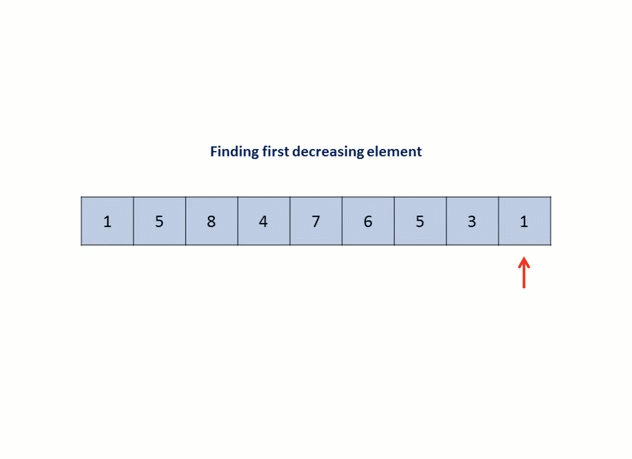
Now, what kind of rearrangement will produce the next larger number? We want to create the permutation just larger than the current one. Therefore, we need to replace the number a[i-1]*a*[*i*−1] with the number which is just larger than itself among the numbers lying to its right section, say a[j]*a*[*j*].



We swap the numbers a[i-1]*a*[*i*−1] and a[j]*a*[*j*]. We now have the correct number at index i-1*i*−1. But still the current permutation isn't the permutation that we are looking for. We need the smallest permutation that can be formed by using the numbers only to the right of a[i-1]*a*[*i*−1]. Therefore, we need to place those numbers in ascending order to get their smallest permutation.

But, recall that while scanning the numbers from the right, we simply kept decrementing the index until we found the pair a[i]*a*[*i*] and a[i-1]*a*[*i*−1] where, a[i] > a[i-1]*a*[*i*]>*a*[*i*−1]. Thus, all numbers to the right of a[i-1]*a*[*i*−1] were already sorted in descending order. Furthermore, swapping a[i-1]*a*[*i*−1] and a[j]*a*[*j*] didn't change that order. Therefore, we simply need to reverse the numbers following a[i-1]*a*[*i*−1] to get the next smallest lexicographic permutation.

The following animation will make things clearer:



|  |
| --- |
| public class Solution {  public int nextGreaterElement(int n) {  char[] a = ("" + n).toCharArray();  int i = a.length - 2;  while (i >= 0 && a[i + 1] <= a[i]) {  i--;  }  if (i < 0)  return -1;  int j = a.length - 1;  while (j >= 0 && a[j] <= a[i]) {  j--;  }  swap(a, i, j);  reverse(a, i + 1);  try {  return Integer.parseInt(new String(a));  } catch (Exception e) {  return -1;  }  }  private void reverse(char[] a, int start) {  int i = start, j = a.length - 1;  while (i < j) {  swap(a, i, j);  i++;  j--;  }  }  private void swap(char[] a, int i, int j) {  char temp = a[i];  a[i] = a[j];  a[j] = temp;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). In worst case, only two scans of the whole array are needed. Here, n*n* refers to the number of digits in the given number.
* Space complexity : O(n)*O*(*n*). An array a*a* of size n*n* is used, where n*n* is the number of digits in the given number.

**Jump Game IV**

Given an array of integers arr, you are initially positioned at the first index of the array.

In one step you can jump from index i to index:

* i + 1 where: i + 1 < arr.length.
* i - 1 where: i - 1 >= 0.
* j where: arr[i] == arr[j] and i != j.

Return *the minimum number of steps* to reach the **last index** of the array.

Notice that you can not jump outside of the array at any time.

**Example 1:**

**Input:** arr = [100,-23,-23,404,100,23,23,23,3,404]

**Output:** 3

**Explanation:** You need three jumps from index 0 --> 4 --> 3 --> 9. Note that index 9 is the last index of the array.

**Example 2:**

**Input:** arr = [7]

**Output:** 0

**Explanation:** Start index is the last index. You don't need to jump.

**Example 3:**

**Input:** arr = [7,6,9,6,9,6,9,7]

**Output:** 1

**Explanation:** You can jump directly from index 0 to index 7 which is last index of the array.

**Example 4:**

**Input:** arr = [6,1,9]

**Output:** 2

**Example 5:**

**Input:** arr = [11,22,7,7,7,7,7,7,7,22,13]

**Output:** 3

**Constraints:**

* 1 <= arr.length <= 5 \* 104
* -108 <= arr[i] <= 108

   Hide Hint #1

Build a graph of n nodes where nodes are the indices of the array and edges for node i are nodes i+1, i-1, j where arr[i] == arr[j].

   Hide Hint #2

Start bfs from node 0 and keep distance, answer is the distance when you reach onode n-1.

## Solution

### **Overview**

You probably can guess from the problem title, this is the fourth problem in the series of [Jump Game](https://leetcode.com/problems/jump-game/) problems. Those problems are similar, but have considerable differences, making their solutions quite different.

Here, two approaches are introduced: Breadth-First Search approach and Bidirectional BFS approach.

#### Approach 1: Breadth-First Search

Most solutions start from a brute force approach and are optimized by removing unnecessary calculations. Same as this one.

A naive brute force approach is to iterate all the possible routes and check if there is one reaches the last index. However, if we already checked one index, we do not need to check it again. We can mark the index as visited by storing them in a visited set.

From convenience, we can store nodes with the same value together in a graph dictionary. With this method, when searching, we do not need to iterate the whole list to find the nodes with the same value as the next steps, but only need to ask the precomputed dictionary. However, to prevent stepping back, we need to clear the dictionary after we get to that value.

|  |
| --- |
| class Solution {  public int minJumps(int[] arr) {  int n = arr.length;  if (n <= 1) {  return 0;  }  Map<Integer, List<Integer>> graph = new HashMap<>();  for (int i = 0; i < n; i++) {  graph.computeIfAbsent(arr[i], v -> new LinkedList<>()).add(i);  }  List<Integer> curs = new LinkedList<>(); // store current layer  curs.add(0);  Set<Integer> visited = new HashSet<>();  int step = 0;  // when current layer exists  while (!curs.isEmpty()) {  List<Integer> nex = new LinkedList<>();  // iterate the layer  for (int node : curs) {  // check if reached end  if (node == n - 1) {  return step;  }  // check same value  for (int child : graph.get(arr[node])) {  if (!visited.contains(child)) {  visited.add(child);  nex.add(child);  }  }  // clear the list to prevent redundant search  graph.get(arr[node]).clear();  // check neighbors  if (node + 1 < n && !visited.contains(node + 1)) {  visited.add(node + 1);  nex.add(node + 1);  }  if (node - 1 >= 0 && !visited.contains(node - 1)) {  visited.add(node - 1);  nex.add(node - 1);  }  }  curs = nex;  step++;  }  return -1;  }  } |

**Complexity Analysis**

Assume N*N* is the length of arr.

* Time complexity: \mathcal{O}(N)O(*N*) since we will visit every node at most once.
* Space complexity: \mathcal{O}(N)O(*N*) since it needs curs and nex to store nodes.

#### Approach 2: Bidirectional BFS

In the later part of our original BFS method, the layer may be long and takes a long time to compute the next layer. In this situation, we can compute the layer from the end, which may be short and takes less time.

|  |
| --- |
| class Solution {  public static int minJumps(int[] arr) {  int n = arr.length;  if (n <= 1) {  return 0;  }  Map<Integer, List<Integer>> graph = new HashMap<>();  for (int i = 0; i < n; i++) {  graph.computeIfAbsent(arr[i], v -> new LinkedList<>()).add(i);  }  HashSet<Integer> curs = new HashSet<>(); // store layers from start  curs.add(0);  Set<Integer> visited = new HashSet<>();  visited.add(0);  visited.add(n - 1);  int step = 0;  HashSet<Integer> other = new HashSet<>(); // store layers from end  other.add(n - 1);  // when current layer exists  while (!curs.isEmpty()) {  // search from the side with fewer nodes  if (curs.size() > other.size()) {  HashSet<Integer> tmp = curs;  curs = other;  other = tmp;  }  HashSet<Integer> nex = new HashSet<>();  // iterate the layer  for (int node : curs) {  // check same value  for (int child : graph.get(arr[node])) {  if (other.contains(child)) {  return step + 1;  }  if (!visited.contains(child)) {  visited.add(child);  nex.add(child);  }  }  // clear the list to prevent redundant search  graph.get(arr[node]).clear();  // check neighbors  if (other.contains(node + 1) || other.contains(node - 1)) {  return step + 1;  }  if (node + 1 < n && !visited.contains(node + 1)) {  visited.add(node + 1);  nex.add(node + 1);  }  if (node - 1 >= 0 && !visited.contains(node - 1)) {  visited.add(node - 1);  nex.add(node - 1);  }  }  curs = nex;  step++;  }  return -1;  }  } |

**Complexity Analysis**

Assume N*N* is the length of arr.

* Time complexity: \mathcal{O}(N)O(*N*) since we will visit every node at most once, but usually faster than approach 1.
* Space complexity: \mathcal{O}(N)O(*N*) since it needs curs, other and nex to store nodes.

**Reach a Number**

You are standing at position 0 on an infinite number line. There is a goal at position target.

On each move, you can either go left or right. During the *n*-th move (starting from 1), you take *n* steps.

Return the minimum number of steps required to reach the destination.

**Example 1:**

**Input:** target = 3

**Output:** 2

**Explanation:**

On the first move we step from 0 to 1.

On the second step we step from 1 to 3.

**Example 2:**

**Input:** target = 2

**Output:** 3

**Explanation:**

On the first move we step from 0 to 1.

On the second move we step from 1 to -1.

On the third move we step from -1 to 2.

**Note:**

 target will be a non-zero integer in the range [-10^9, 10^9].

#### Approach #1: Mathematical [Accepted]

**Intuition**

The crux of the problem is to put + and - signs on the numbers 1, 2, 3, ..., k so that the sum is target.

When target < 0 and we made a sum of target, we could switch the signs of all the numbers so that it equals Math.abs(target). Thus, the answer for target is the same as Math.abs(target), and so without loss of generality, we can consider only target > 0.

Now let's say k is the smallest number with S = 1 + 2 + ... + k >= target. If S == target, the answer is clearly k.

If S > target, we need to change some number signs. If delta = S - target is even, then we can always find a subset of {1, 2, ..., k} equal to delta / 2 and switch the signs, so the answer is k. (This depends on T = delta / 2 being at most S.) [The proof is simple: either T <= k and we choose it, or we choose k in our subset and try to solve the same instance of the problem for T -= k and the set {1, 2, ..., k-1}.]

Otherwise, if delta is odd, we can't do it, as every sign change from positive to negative changes the sum by an even number. So let's consider a candidate answer of k+1, which changes delta by k+1. If this is odd, then delta will be even and we can have an answer of k+1. Otherwise, delta will be odd, and we will have an answer of k+2.

For concrete examples of the above four cases, consider the following:

* If target = 3, then k = 2, delta = 0 and the answer is k = 2.
* If target = 4, then k = 3, delta = 2, delta is even and the answer is k = 3.
* If target = 7, then k = 4, delta = 3, delta is odd and adding k+1 makes delta even. The answer is k+1 = 5.
* If target = 5, then k = 3, delta = 1, delta is odd and adding k+1 keeps delta odd. The answer is k+2 = 5.

**Algorithm**

Subtract ++k from target until it goes non-positive. Then k will be as described, and target will be delta as described. We can output the four cases above: if delta is even then the answer is k, if delta is odd then the answer is k+1 or k+2 depending on the parity of k.

|  |
| --- |
| class Solution {  public int reachNumber(int target) {  target = Math.abs(target);  int k = 0;  while (target > 0)  target -= ++k;  return target % 2 == 0 ? k : k + 1 + k%2;  }  } |

**Complexity Analysis**

* Time Complexity: O(\sqrt{\text{target}})*O*(target​). Our while loop needs this many steps, as 1 + 2 + \dots + k = \frac{k(k+1)}{2}1+2+⋯+*k*=2*k*(*k*+1)​.
* Space Complexity: O(1)*O*(1).

**Longest Substring with At Most K Distinct Characters**

Given a string s and an integer k, return *the length of the longest substring of* s *that contains at most* k ***distinct****characters*.

**Example 1:**

**Input:** s = "eceba", k = 2

**Output:** 3

**Explanation:** The substring is "ece" with length 3.

**Example 2:**

**Input:** s = "aa", k = 1

**Output:** 2

**Explanation:** The substring is "aa" with length 2.

**Constraints:**

* 1 <= s.length <= 5 \* 104
* 0 <= k <= 50

## Solution Article

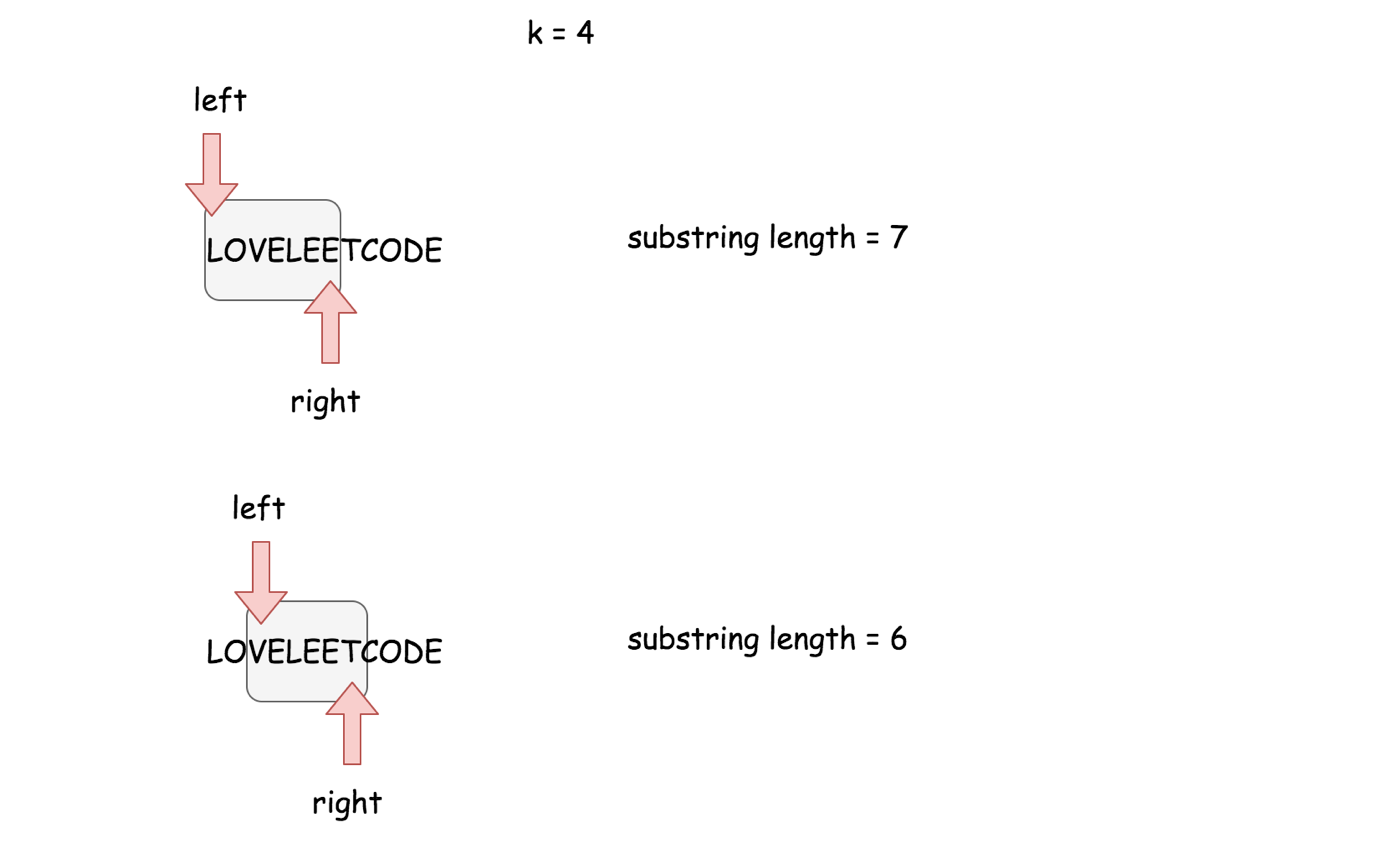
#### Approach 1: Sliding Window + Hashmap.

**Intuition**

We could take some inspiration from a simpler problem called [longest substring with at most two distinct characters](https://leetcode.com/problems/longest-substring-with-at-most-two-distinct-characters/).

To solve the problem in one pass let's use here sliding window approach with two set pointers left and right serving as the window boundaries.

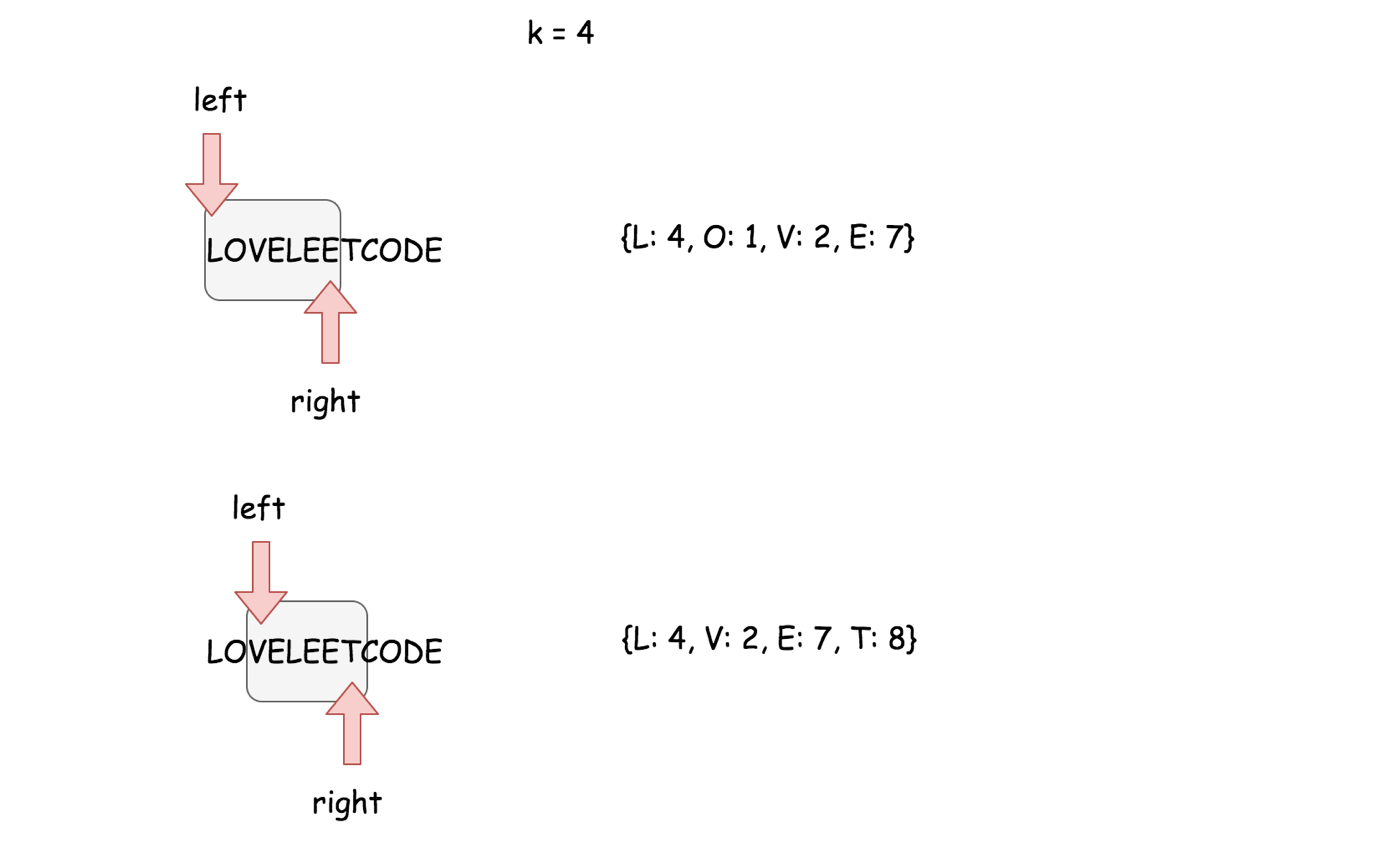
The idea is to set both pointers in the position 0 and then move right pointer to the right while the window contains not more than k distinct characters. If at some point we've got k + 1 distinct characters, let's move left pointer to keep not more than k + 1 distinct characters in the window.



Basically that's the algorithm : to move sliding window along the string, to keep not more than k distinct characters in the window, and to update max substring length at each step.

There is just one more question to reply - how to move the left pointer to keep only k distinct characters in the string?

Let's use for this purpose hashmap containing all characters in the sliding window as keys and their rightmost positions as values. At each moment, this hashmap could contain not more than k + 1 elements.



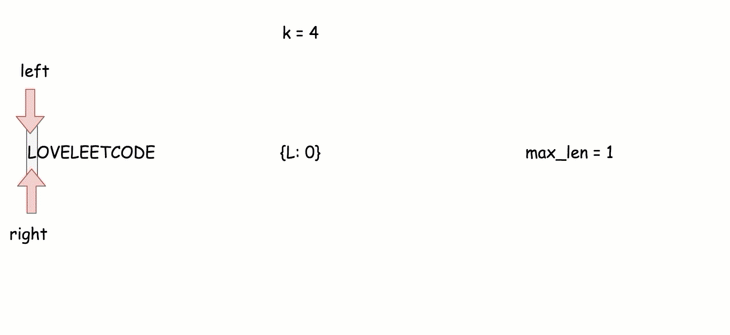
For example, using this hashmap one knows that the rightmost position of character O in "LOVELEE" window is 1 and so one has to move left pointer in the position 1 + 1 = 2 to exclude the character O from the sliding window.

**Algorithm**

Now one could write down the algortihm.

* Return 0 if the string is empty or k is equal to zero.
* Set both set pointers in the beginning of the string left = 0 and right = 0 and init max substring length max\_len = 1.
* While right pointer is less than N:
  + Add the current character s[right] in the hashmap and move right pointer to the right.
  + If hashmap contains k + 1 distinct characters, remove the leftmost character from the hashmap and move the left pointer so that sliding window contains again k distinct characters only.
  + Update max\_len.

**Implementation**



|  |
| --- |
| class Solution {  public int lengthOfLongestSubstringKDistinct(String s, int k) {  int n = s.length();  if (n \* k == 0) {  return 0;  }  int left = 0;  int right = 0;  Map<Character, Integer> rightmostPosition = new HashMap<>();  int maxLength = 1;  while (right < n) {  rightmostPosition.put(s.charAt(right), right++);  if (rightmostPosition.size() == k + 1) {  int lowestIndex = Collections.min(rightmostPosition.values());  rightmostPosition.remove(s.charAt(lowestIndex));  left = lowestIndex + 1;  }  maxLength = Math.max(maxLength, right - left);  }  return maxLength;  }  } |

**Complexity Analysis**

Do we have here the best possible time complexity \mathcal{O}(N)O(*N*) as it was for more simple [problem with at most two distinct characters](https://leetcode.com/articles/longest-substring-with-at-most-two-distinct-charac/)?

For the best case when input string contains not more than k distinct characters the answer is yes. It's the only one pass along the string with N characters and the time complexity is \mathcal{O}(N)O(*N*).

For the worst case when the input string contains n distinct characters, the answer is no. In that case at each step one uses \mathcal{O}(k)O(*k*) time to find a minimum value in the hashmap with k elements and so the overall time complexity is \mathcal{O}(N k)O(*Nk*).

* Time complexity : \mathcal{O}(N)O(*N*) in the best case of k distinct characters in the string and \mathcal{O}(N k)O(*Nk*) in the worst case of N distinct characters in the string.
* Space complexity : \mathcal{O}(k)O(*k*) since additional space is used only for a hashmap with at most k + 1 elements.

#### Approach 2: Sliding Window + Ordered Dictionary.

**How to achieve \mathcal{O}(N)O(*N*) time complexity**

Approach 1 with a standard hashmap couldn't ensure \mathcal{O}(N)O(*N*) time complexity.

To have \mathcal{O}(N)O(*N*) algorithm performance, one would need a structure, which provides four operations in \mathcal{O}(1)O(1) time :

* Insert the key
* Get the key and check if the key exists
* Delete the key
* Return the first or last added key/ value

The first three operations in \mathcal{O}(1)O(1) time are provided by the standard hashmap, and the forth one - by linked list.

There is a structure called ordered dictionary, it combines behind both hashmap and linked list. In Python this structure is called [OrderedDict](https://docs.python.org/3/library/collections.html" \l "collections.OrderedDict) and in Java [LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html).

Ordered dictionary is quite popular for interviews. for example, check out the [Implementing a LRU Cache](https://leetcode.com/problems/lru-cache/) question by Google.

**Algorithm**

Let's use ordered dictionary instead of standard hashmap to trim the algorithm from approach 1 :

* Return 0 if the string is empty or k is equal to zero.
* Set both pointers to the beginning of the string left = 0 and right = 0 and initialize max substring length max\_len = 1.
* While right pointer is less than N:
  + If the current character s[right] is already in the ordered dictionary hashmap -- delete it, to ensure that the first key in hashmap is the leftmost character.
  + Add the current character s[right] in the ordered dictionary and move right pointer to the right.
  + If ordered dictionary hashmap contains k + 1 distinct characters, remove the leftmost one and move the left pointer so that sliding window contains again k distinct characters only.
  + Update max\_len.

**Implementation**

|  |
| --- |
| class Solution {  public int lengthOfLongestSubstringKDistinct(String s, int k) {  int n = s.length();  if (n \* k == 0) {  return 0;  }  int left = 0;  int right = 0;  Map<Character, Integer> rightmostPosition = new LinkedHashMap<>();  int maxLength = 1;  while (right < n) {  Character character = s.charAt(right);  if (rightmostPosition.containsKey(character)) {  rightmostPosition.remove(character);  }  rightmostPosition.put(character, right++);  if (rightmostPosition.size() == k + 1) {  Map.Entry<Character, Integer> leftmost =  rightmostPosition.entrySet().iterator().next();  rightmostPosition.remove(leftmost.getKey());  left = leftmost.getValue() + 1;  }  maxLength = Math.max(maxLength, right - left);  }  return maxLength;  }  } |

**Complexity Analysis**

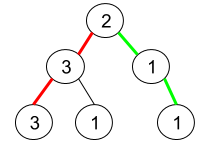
* Time complexity : \mathcal{O}(N)O(*N*) since all operations with ordered dictionary : insert/get/delete/popitem (put/containsKey/remove) are done in a constant time.
* Space complexity : \mathcal{O}(k)O(*k*) since additional space is used only for an ordered dictionary with at most k + 1 elements.

**Pseudo-Palindromic Paths in a Binary Tree**

Given a binary tree where node values are digits from 1 to 9. A path in the binary tree is said to be **pseudo-palindromic** if at least one permutation of the node values in the path is a palindrome.

*Return the number of****pseudo-palindromic****paths going from the root node to leaf nodes.*

**Example 1:**

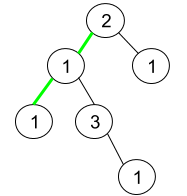


**Input:** root = [2,3,1,3,1,null,1]

**Output:** 2

**Explanation:** The figure above represents the given binary tree. There are three paths going from the root node to leaf nodes: the red path [2,3,3], the green path [2,1,1], and the path [2,3,1]. Among these paths only red path and green path are pseudo-palindromic paths since the red path [2,3,3] can be rearranged in [3,2,3] (palindrome) and the green path [2,1,1] can be rearranged in [1,2,1] (palindrome).

**Example 2:**

****

**Input:** root = [2,1,1,1,3,null,null,null,null,null,1]

**Output:** 1

**Explanation:** The figure above represents the given binary tree. There are three paths going from the root node to leaf nodes: the green path [2,1,1], the path [2,1,3,1], and the path [2,1]. Among these paths only the green path is pseudo-palindromic since [2,1,1] can be rearranged in [1,2,1] (palindrome).

**Example 3:**

**Input:** root = [9]

**Output:** 1

**Constraints:**

* The number of nodes in the tree is in the range [1, 105].
* 1 <= Node.val <= 9

   Hide Hint #1

Note that the node values of a path form a palindrome if at most one digit has an odd frequency (parity).

   Hide Hint #2

Use a Depth First Search (DFS) keeping the frequency (parity) of the digits. Once you are in a leaf node check if at most one digit has an odd frequency (parity).

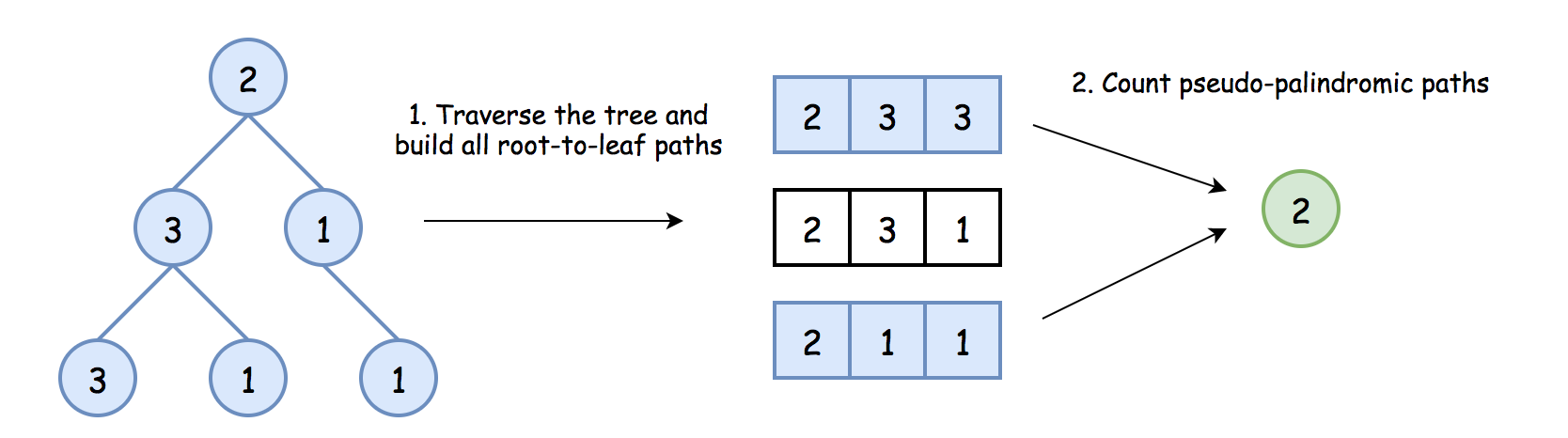
## Solution

#### Overview

**Two subproblems**

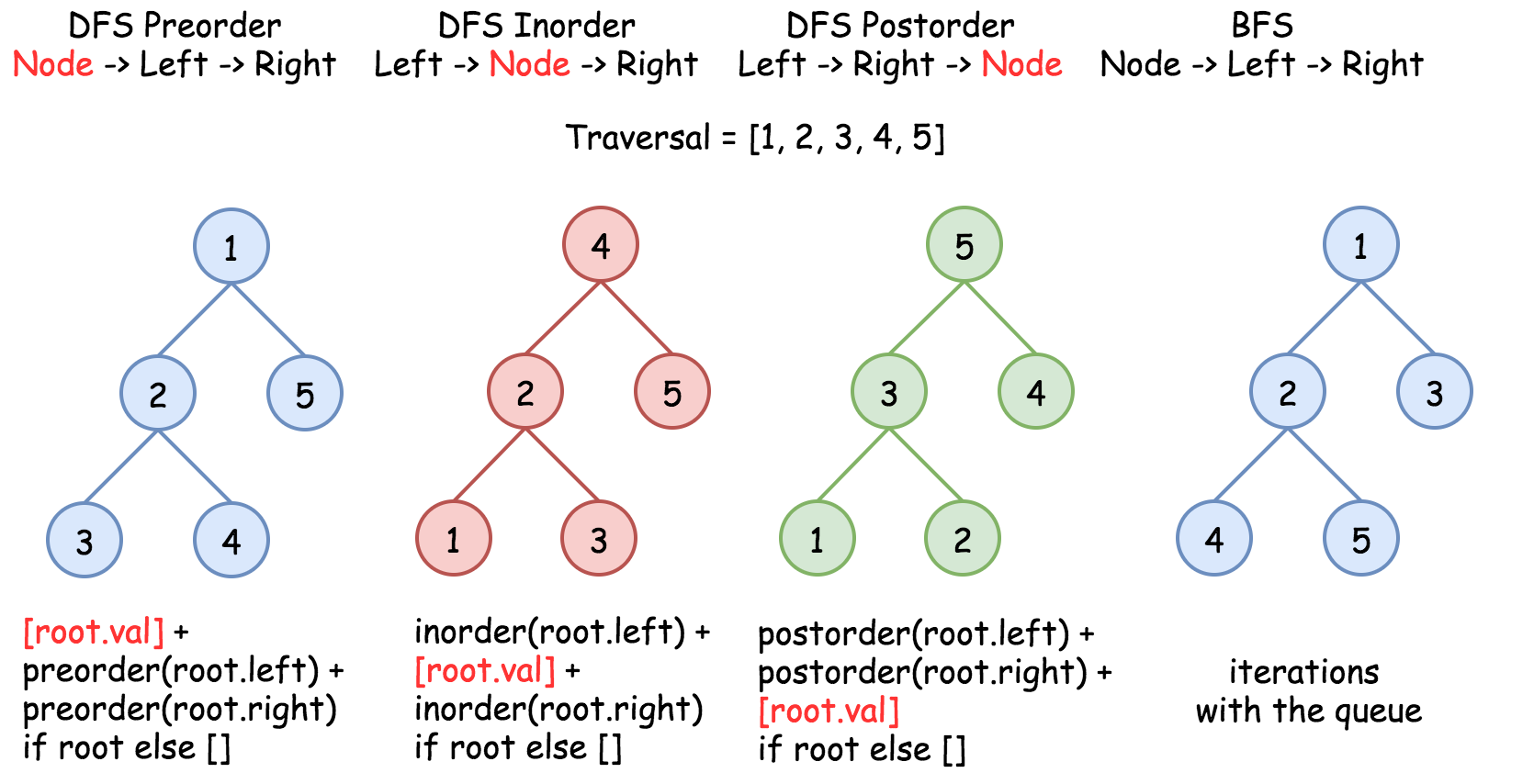
The problem consists of two subproblems:

* Traverse the tree to build all root-to-leaf paths.
* For each root-to-leaf path, check if it's a pseudo-palindromic path or not.

 Figure 1. Two subproblems.

**How to traverse the tree to build all root-to-leaf paths**

There are three DFS ways to traverse the tree: preorder, postorder and inorder. Please check two minutes picture explanation if you don't remember them quite well: [here is Python version](https://leetcode.com/problems/binary-tree-inorder-traversal/discuss/283746/all-dfs-traversals-preorder-inorder-postorder-in-python-in-1-line) and [here is Java version](https://leetcode.com/problems/binary-tree-inorder-traversal/discuss/328601/all-dfs-traversals-preorder-postorder-inorder-in-java-in-5-lines).

 Figure 2. The nodes are enumerated in the order of visit. To compare different DFS strategies, follow *1-2-3-4-5* direction.

Root-to-leaf traversal is so-called DFS preorder traversal. To implement it, one has to follow the straightforward strategy Root->Left->Right.

There are three ways to implement preorder traversal: iterative, recursive, and Morris. Here we're going to implement the first two.

Iterative and recursive approaches here do the job in one pass, but they both need up to \mathcal{O}(H)O(*H*) space to keep the stack, where H*H* is a tree height.

**How to check if the path is pseudo-palindromic or not**

It's quite evident that the path is pseudo-palindromic, if it has at most one digit with an odd frequency.

How to check that?

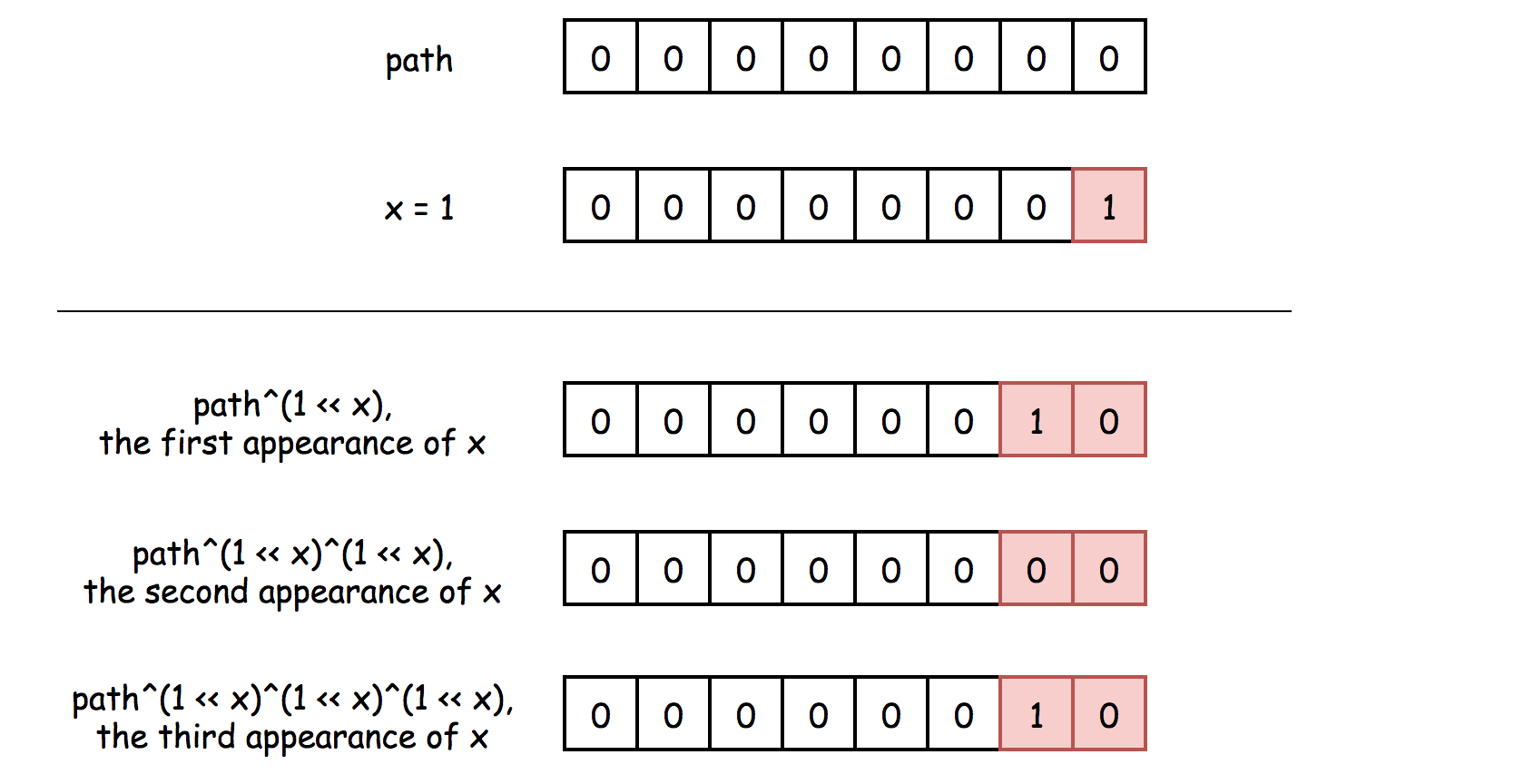
The straightforward way is to save each root-to-leaf path into a list and then to check each digit for parity.

|  |
| --- |
| public boolean checkPalindrom(ArrayList<Integer> nums) {  int isPalindrom = 0;  for (int i = 1; i < 10; ++i) {  if (Collections.frequency(nums, i) % 2 == 1) {  ++isPalindrom;  if (isPalindrom > 1) {  return false;  }  }  }  return true;  } |

This method requires to keep each root-to-leaf path, and that becomes space-consuming for the large trees. To save the space, let's compute the parity on-the-fly using bitwise operators.

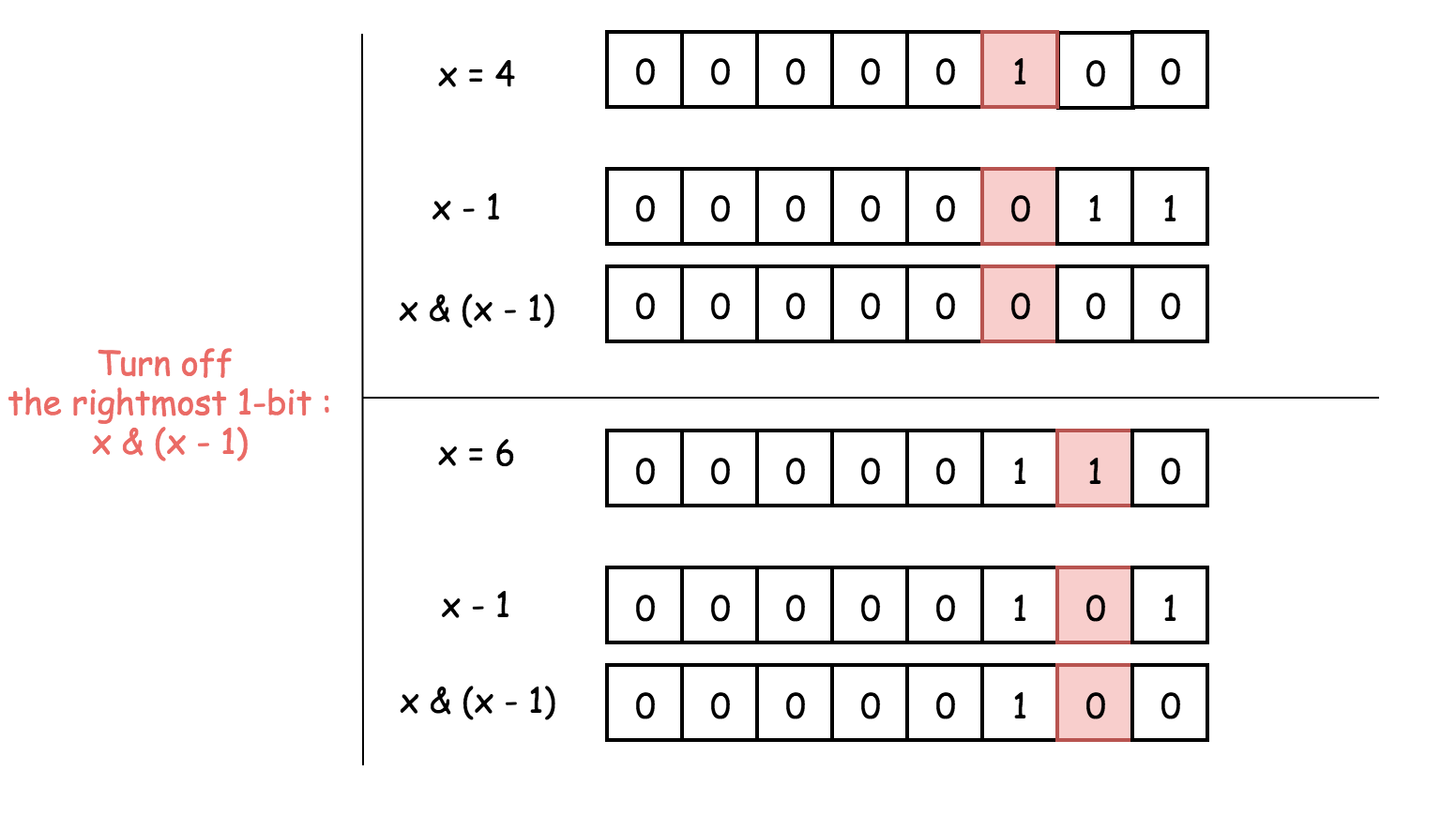
The idea is to keep the frequency of digit 1 in the first bit, 2 in the second bit, etc: path ^= (1 << node.val).

[Left shift operator](https://leetcode.com/problems/pseudo-palindromic-paths-in-a-binary-tree/solution/(https:/wiki.python.org/moin/BitwiseOperators)) is used to define the bit, and [XOR operator](https://leetcode.com/problems/single-number-ii/solution/) - to compute the digit frequency.

 Figure 3. XOR of zero and a bit results in that bit. XOR of two equal bits (even if they are zeros) results in a zero. Hence, one could see the bit in a path only if it appears odd number of times.

|  |
| --- |
| // compute occurences of each digit  // in the corresponding bit  path = path ^ (1 << node.val); |

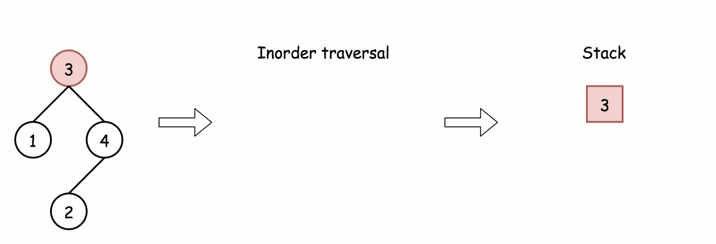
Now, to ensure that at most one digit has an odd frequency, one has to check that path is a [power of two](https://leetcode.com/problems/power-of-two/solution/), i.e., at most one bit is set to one. That could be done by turning off (= setting to 0) the rightmost 1-bit: path & (path - 1) == 0. You might want to check the article [Power of Two](https://leetcode.com/problems/power-of-two/solution/) for the detailed explanation of this bitwise trick.

 Figure 4. *x & (x - 1)* is a way to set the rightmost 1-bit to zero, i.e., *x & (x - 1) == 0* for the power of two. To subtract 1 means to change the rightmost 1-bit to 0 and to set all the lower bits to 1. Now AND operator: the rightmost 1-bit will be turned off because *1 & 0 = 0*, and all the lower bits as well.

|  |
| --- |
| // if it's a leaf,  // check that at most one digit has an odd frequency  if ((path & (path - 1)) == 0) {  ++count;  } |

#### Approach 1: Iterative Preorder Traversal.

**Intuition**



Here we implement standard iterative preorder traversal with the stack:

* Initialize the counter to zero.
* Push root into stack.
* While stack is not empty:
  + Pop out a node from the stack and update the current number.
  + If the node is a leaf, update the root-to-leaf path, check it for being pseudo-palindromic, and update the count.
  + Push right and left child nodes into stack.
* Return count.

**Implementation**

Note, that [Javadocs recommends to use ArrayDeque, and not Stack as a stack implementation](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html).

|  |
| --- |
| class Solution {  public int pseudoPalindromicPaths (TreeNode root) {  int count = 0, path = 0;    Deque<Pair<TreeNode, Integer>> stack = new ArrayDeque();  stack.push(new Pair(root, 0));  while (!stack.isEmpty()) {  Pair<TreeNode, Integer> p = stack.pop();  TreeNode node = p.getKey();  path = p.getValue();  if (node != null) {  // compute occurences of each digit  // in the corresponding register  path = path ^ (1 << node.val);  // if it's a leaf check if the path is pseudo-palindromic  if (node.left == null && node.right == null) {  // check if at most one digit has an odd frequency  if ((path & (path - 1)) == 0) {  ++count;  }  } else {  stack.push(new Pair(node.right, path));  stack.push(new Pair(node.left, path));  }  }  }  return count;  }  } |

**Complexity Analysis**

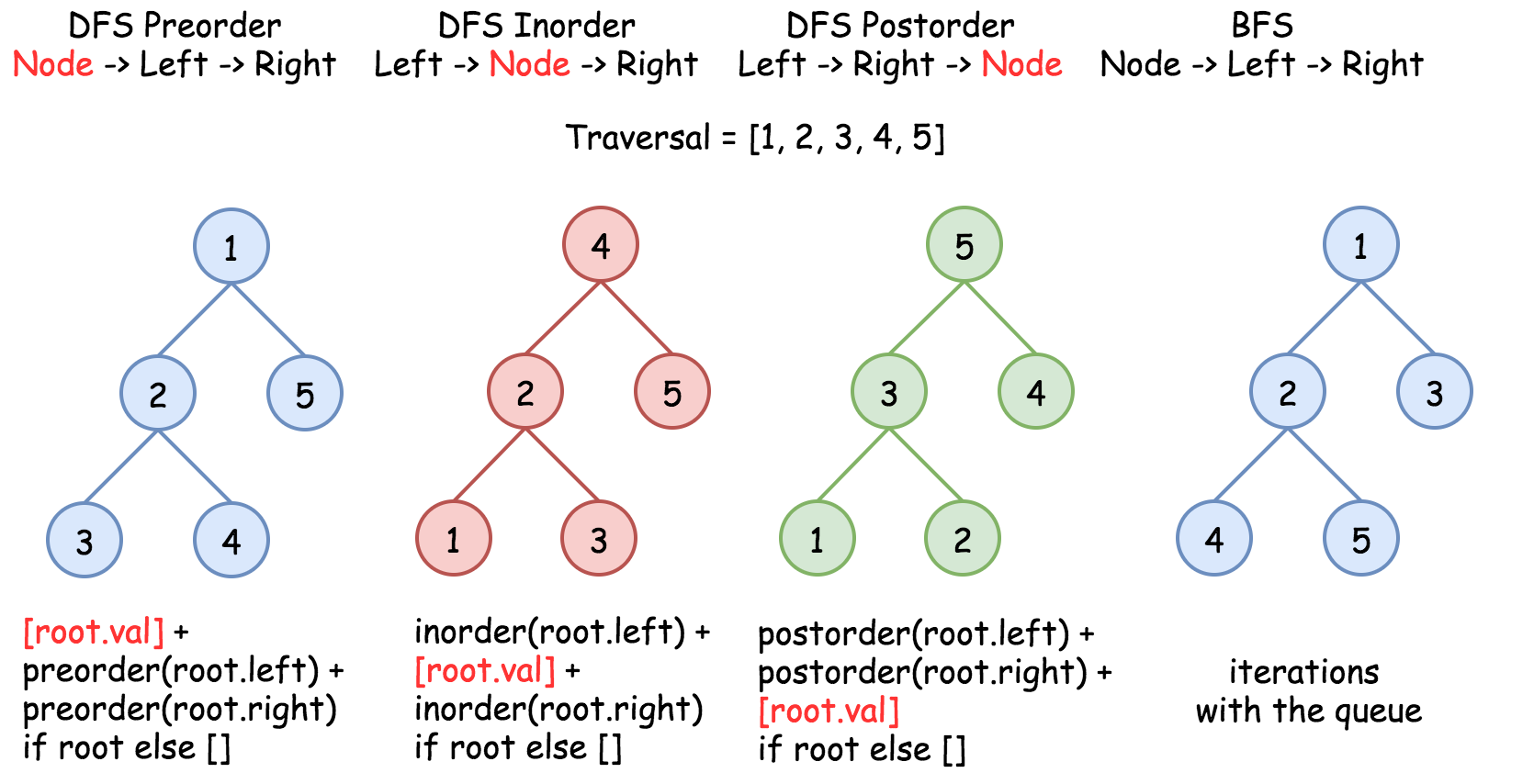
* Time complexity: \mathcal{O}(N)O(*N*) since one has to visit each node, where N*N* is a number of nodes.
* Space complexity: up to \mathcal{O}(H)O(*H*) to keep the stack, where H*H* is a tree height.

#### Approach 2: Recursive Preorder Traversal.

Iterative approach 1 could be converted into a recursive one.

Recursive preorder traversal is extremely simple: follow Root->Left->Right direction, i.e., do all the business with the node (i.e., update the current path and the counter), and then do the recursive calls for the left and right child nodes.

P.S. Here is the difference between preorder and the other DFS recursive traversals.

 Figure 5. The nodes are enumerated in the order of visit. To compare different DFS strategies, follow *1-2-3-4-5* direction.

**Implementation**

|  |
| --- |
| class Solution {  int count = 0;    public void preorder(TreeNode node, int path) {  if (node != null) {  // compute occurences of each digit  // in the corresponding register  path = path ^ (1 << node.val);  // if it's a leaf check if the path is pseudo-palindromic  if (node.left == null && node.right == null) {  // check if at most one digit has an odd frequency  if ((path & (path - 1)) == 0) {  ++count;  }  }  preorder(node.left, path);  preorder(node.right, path) ;  }  }  public int pseudoPalindromicPaths (TreeNode root) {  preorder(root, 0);  return count;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*) since one has to visit each node, check if at most one digit has an odd frequency.
* Space complexity: up to \mathcal{O}(H)O(*H*) to keep the recursion stack, where H*H* is a tree height.

#### Further Reading

The problem could be solved in constant space using the Morris inorder traversal algorithm, as it was done in [Sum Root-to-Leaf Numbers](https://leetcode.com/problems/sum-root-to-leaf-numbers/solution/). It is unlike that one can come up a Morris solution during an interview, but it is worth to know anyway.