27. Minimum Depth of Binary Tree

Code it now: https://oj.leetcode.com/problems/minimum-depth-of-binary-tree/ Difficulty: Easy, Frequency: High

Question:

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Similar to Question [26. Maximum Depth of Binary Tree], here we need to find the *minimum* depth instead.

Solution:

***O*(*n*) runtime, *O*(log *n*) space – Depth-first traversal:**

Similar to the [Recursion] approach to find the maximum depth, but make sure you consider these cases:

i. The node itself is a leaf node. The minimum depth is one.

ii. Node that has one empty sub-tree while the other one is non-empty. Return the minimum depth of that non-empty sub-tree.

```
public int minDepth(TreeNode root) {
   if (root == null) return 0;
   if (root.left == null) return minDepth(root.right) + 1;
   if (root.right == null) return minDepth(root.left) + 1;
   return Math.min(minDepth(root.left), minDepth(root.right)) + 1;
}
```

***O*(*n*) runtime, *O*(*n*) space – Breadth-first traversal:**

Note that the previous approach traverses all the nodes even for a highly unbalanced tree. In fact, we could optimize this scenario by doing a breadth-first traversal (also known as level-order traversal). When we encounter the first leaf node, we immediately stop the traversal.

We also keep track of the current depth and increment it when we reach the end of level. We know that we have reached the end of level when the current node is the right-most node.

Compared to the recursion approach, the breadth-first traversal works well for highly unbalanced tree because it does not need to traverse all nodes. The worst case is when the tree is a full binary tree with a total of *n* nodes. In this case, we have to traverse all nodes. The worst case space complexity is *O*(*n*), due to the extra space needed to store current level nodes in the queue.

```
public int minDepth(TreeNode root) {
   if (root == null) return 0;
   Queue<TreeNode> q = new LinkedList<>();
   q.add(root);
   TreeNode rightMost = root;
   int depth = 1;
   while (!q.isEmpty()) {
      TreeNode node = q.poll();
      if (node.left == null && node.right == null) break;
      if (node.left != null) q.add(node.left);
      if (node.right != null) q.add(node.right);
      if (node == rightMost) {
         depth++;
         rightMost = (node.right != null) ? node.right : node.left;
      }
   }
   return depth;
}
```

28. Balanced Binary Tree

Code it now: https://oj.leetcode.com/problems/balanced-binary-tree/ Difficulty: Easy, Frequency: High

Question:

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differs by more than 1.

Solution:

O(n^2) runtime, *O*(n) stack space – Brute force top-down recursion:

We could devise a brute force algorithm directly based on the above definition. We also reused the [Recursion] approach to find the maximum depth of a subtree. The brute force algorithm worst case runtime complexity is $O(n^2)$ when the input tree is degenerated.

```
public boolean isBalanced(TreeNode root) {
  if (root == null) return true;
  return Math.abs(maxDepth(root.left) - maxDepth(root.right)) <= 1
      && isBalanced(root.left)
      && isBalanced(root.right);
}

public int maxDepth(TreeNode root) {
  if (root == null) return 0;
  return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
}
```

O(n) runtime, *O*(n) stack space – Bottom-up recursion:

It seems that the above approach is recalculating max depth repeatedly for each node. We could avoid the recalculation by passing the depth *bottom-up*. We use a sentinel value –1 to represent that the tree is unbalanced so we could avoid unnecessary calculations.

In each step, we look at the left subtree's depth (*L*), and ask: "Is the left subtree unbalanced?" If it is indeed unbalanced, we return –1 right away. Otherwise, *L* represents the left subtree's depth. We then repeat the same process for the right subtree's depth (*R*).

We calculate the absolute difference between *L* and *R*. If the subtrees' depth difference is less than one, we could return the height of the current node, otherwise return –1 meaning the current tree is unbalanced.

```java
public boolean isBalanced(TreeNode root) {
   return maxDepth(root) != -1;
}

private int maxDepth(TreeNode root) {
   if (root == null) return 0;
   int L = maxDepth(root.left);
   if (L == -1) return -1;
   int R = maxDepth(root.right);
   if (R == -1) return -1;
   return (Math.abs(L - R) <= 1) ? (Math.max(L, R) + 1) : -1;
}
```

29. Convert Sorted Array to Balanced Binary Search Tree

Code it now: https://oj.leetcode.com/problems/convert-sorted-array-to-binary-search-tree/ Difficulty: Medium, Frequency: Low

Question:

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

Hint:

This question is highly recursive in nature. Think of how binary search works.

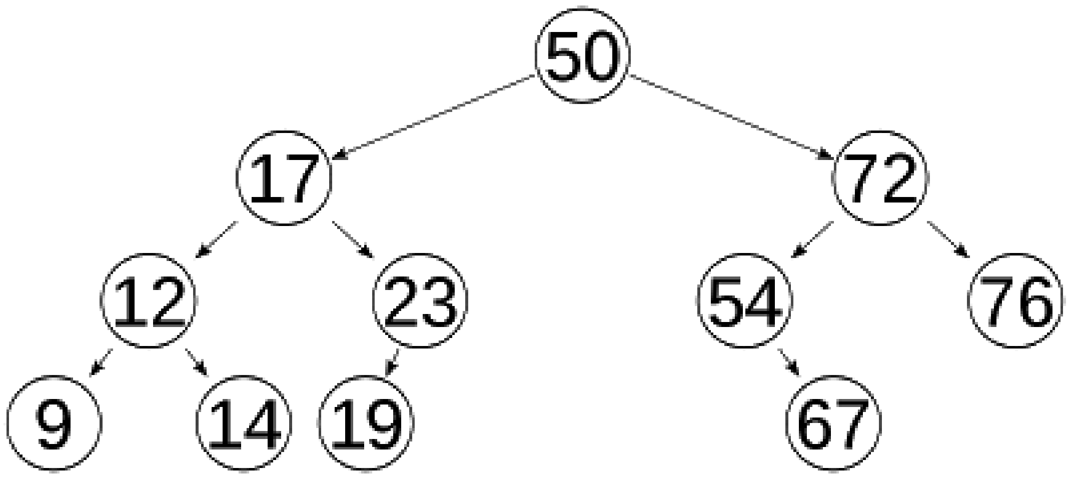

Figure 2: An example of a height-balanced tree. A height-balanced tree is a tree whose subtrees differ in height by no more than one and the subtrees are height-balanced, too.

Solution:

***O*(*n*) runtime, *O*(log *n*) stack space – Divide and conquer:**

If you would have to choose an array element to be the root of a balanced BST, which element would you pick? The root of a balanced BST should be the middle element from the sorted array.

You would pick the middle element from the sorted array in each iteration. You then create a node in the tree initialized with this element. After the element is chosen, what is left? Could you identify the sub-problems within the problem?

There are two arrays left — The one on its left and the one on its right. These two arrays are the sub-problems of the original problem, since both of them are sorted. Furthermore, they are subtrees of the current node's left and right child.

The code below creates a balanced BST from the sorted array in *O*(*n*) time (*n* is the number of elements in the array). Compare how similar the code is to a binary search algorithm. Both are using the divide and conquer methodology. Because the input array could be subdivided in at most log(*n*) times, the extra stack space used by the recursion is in *O*(log *n*).

```java
public TreeNode sortedArrayToBST(int[] num) {
   return sortedArrayToBST(num, 0, num.length-1);
}

private TreeNode sortedArrayToBST(int[] arr, int start, int end) {
   if (start > end) return null;
   int mid = (start + end) / 2;
   TreeNode node = new TreeNode(arr[mid]);
   node.left = sortedArrayToBST(arr, start, mid-1);
   node.right = sortedArrayToBST(arr, mid+1, end);
   return node;
}
```

Further Thoughts:

Consider changing the problem statement to "Converting a singly linked list to a balanced BST". How would your implementation change from the above? See Question [30. Convert Sorted List to Balanced Binary Search Tree].

30. Convert Sorted List to Balanced Binary Search Tree

Code it now: https://oj.leetcode.com/problems/convert-sorted-list-to-binary-search-tree/ Difficulty: Hard, Frequency: Low

Question:

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Hint:

Things get a little more complicated when you have a singly linked list instead of an array. Please note that in linked list, you no longer have random access to an element in $O(1)$ time.

How about inserting nodes following the list's order? If we can achieve this, we no longer need to find the middle element, as we are able to traverse the list while inserting nodes to the tree.

***O*(*n* log *n*) runtime, O(log *n*) stack space – Brute force:**

A naive way is to apply the previous solution from Question [29. Convert Sorted Array to Balanced Binary Search Tree] directly. In each recursive call, you would have to traverse half of the list's length to find the middle element. The run time complexity is clearly $O(n \log n)$, where n is the total number of elements in the list. This is because each level of recursive call requires a total of $\frac{n}{2}$ traversal steps in the list, and there are a total of $\log(n)$ number of levels (ie, the height of the balanced tree).

***O*(*n*) runtime, *O*(log *n*) stack space – Bottom-up recursion:**

As usual, the best solution requires you to think from another perspective. In other words, we no longer create nodes in the tree using the top-down approach. We create nodes bottom-up, and assign them to its parents. The bottom-up approach enables us to access the list in its order while creating nodes.

Isn't the bottom-up approach neat? Each time you are stuck with the top-down approach, give bottom-up a try. Although bottom-up approach is not the most natural way we think, it is extremely helpful in some cases. However, you should prefer top-down instead of bottom-up in general, since the latter is more difficult to verify in correctness.

Below is the code for converting a singly linked list to a balanced BST. Please note that the algorithm requires the list's length to be passed in as the function's parameters. The list's length could be found in $O(n)$ time by traversing the entire list's once. The recursive calls traverse the list and create tree's nodes by the list's order, which also takes $O(n)$ time. Therefore, the overall run time complexity is still $O(n)$.

```java
private ListNode list;

private TreeNode sortedListToBST(int start, int end) {
   if (start > end) return null;
   int mid = (start + end) / 2;
   TreeNode leftChild = sortedListToBST(start, mid-1);
   TreeNode parent = new TreeNode(list.val);
   parent.left = leftChild;
   list = list.next;
   parent.right = sortedListToBST(mid+1, end);
   return parent;
}

public TreeNode sortedListToBST(ListNode head) {
   int n = 0;
   ListNode p = head;
   while (p != null) {
      p = p.next;
      n++;
   }
   list = head;
   return sortedListToBST(0, n - 1);
}
```

31. Binary Tree Maximum Path Sum

Code it now: https://oj.leetcode.com/problems/binary-tree-maximum-path-sum/ Difficulty: Hard, Frequency: Medium

Question:

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

For example, given the below binary tree,

```
      1
     / \
    2   4
   / \
  2   3
```

The highlighted path yields the maximum sum 10.

Example Questions Candidate Might Ask:

Q: What if the tree is empty?
A: Assume the tree is non-empty.

Q: How about a tree that contains only a single node?
A: Then the maximum path sum starts and ends at the same node.

Q: What if every node contains negative value?
A: Then you should return the single node value that is the least negative.

Q: Does the maximum path have to go through the root node?
A: Not necessarily. For example, the below tree yield 6 as the maximum path sum and does not go through root.

```
     -5
     / \
    2   3
   / \
 -1   4
```

Hint:

Anytime when you found that doing top down approach uses a lot of repeated calculation, bottom up approach usually is able to be more efficient.

```
      _____Node____
     /             \
    /               \
left subtree   right subtree
```

Try the bottom up approach. At each node, the potential maximum path could be one of these cases:

i. max(left subtree) + node

ii. max(right subtree) + node

iii. max(left subtree) + max(right subtree) + node

iv. the node itself

Then, we need to return the maximum path sum that goes through this node and to either one of its left or right subtree to its parent. There's a little trick here: If this maximum happens to be negative, we should return 0, which means: "Do not include this subtree as part of the maximum path of the parent node", which greatly simplifies our code.

```java
private int maxSum;

public int maxPathSum(TreeNode root) {
   maxSum = Integer.MIN_VALUE;
   findMax(root);
   return maxSum;
}

private int findMax(TreeNode p) {
   if (p == null) return 0;
   int left = findMax(p.left);
   int right = findMax(p.right);
   maxSum = Math.max(p.val + left + right, maxSum);
   int ret = p.val + Math.max(left, right);
   return ret > 0 ? ret : 0;
}
```

32. Binary Tree Upside Down

Code it now: https://oj.leetcode.com/problems/binary-tree-upside-down/ Difficulty: Medium, Frequency: N/A

Question:

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

Solution:

At each node you want to assign:

```
p.left = parent.right;
p.right = parent;
```

Top down approach:

We need to be very careful when reassigning current node's left and right children. Besides making a copy of the parent node, you would also need to make a copy of the parent's right child too. The reason is the current node becomes the parent node in the next iteration.

```
public TreeNode UpsideDownBinaryTree(TreeNode root) {
   TreeNode p = root, parent = null, parentRight = null;
   while (p != null) {
      TreeNode left = p.left;
      p.left = parentRight;
      parentRight = p.right;
      p.right = parent;
      parent = p;
      p = left;
   }
   return parent;
}
```

The above code is actually very similar to the algorithm in reversing a linked list.

Bottom up approach:

Although the code for the top-down approach seems concise, it is actually subtle and there are a lot of hidden traps if you are not careful. The other approach is thinking recursively in a bottom-up fashion. If we reassign the bottom-level nodes before the upper ones, we won’t have to make copies and worry about overwriting something. We know the new root will be the left-most leaf node, so we begin the reassignment here.

```
public TreeNode UpsideDownBinaryTree(TreeNode root) {
   return dfsBottomUp(root, null);
}

private TreeNode dfsBottomUp(TreeNode p, TreeNode parent) {
   if (p == null) return parent;
   TreeNode root = dfsBottomUp(p.left, p);
   p.left = (parent == null) ? parent : parent.right;
   p.right = parent;
   return root;
}
```

Chapter 5: Bit Manipulation

33. Single Number

Code it now: https://oj.leetcode.com/problems/single-number/ Difficulty: Medium, Frequency: High

Question:

Given an array of integers, every element appears twice except for one. Find that single one.

Example Questions Candidate Might Ask:

Q: Does the array contain both positive and negative integers?
A: Yes.

Q: Could any element appear more than twice?
A: No.

Solution:

We could use a map to keep track of the number of times an element appears. In a second pass, we could extract the single number by consulting the hash map. As a hash map provides constant time lookup, the overall complexity is O(n), where n is the total number of elements.

```
public int singleNumber(int[] A) {
   Map<Integer, Integer> map = new HashMap<>();
   for (int x : A) {
      int count = map.containsKey(x) ? map.get(x) : 0;
      map.put(x, count + 1);
   }
   for (int x : A) {
      if (map.get(x) == 1) {
         return x;
      }
   }
   throw new IllegalArgumentException("No single element");
}
```

Although the map approach works, we are not taking advantage of the “every elements appears twice except one” property. Could we do better in one pass?

How about inserting the elements into a set instead? If an element already exists, we discard the element from the set knowing that it will not appear again. After the first pass, the set must contain only the single element.

```java
public int singleNumber(int[] A) {
   Set<Integer> set = new HashSet<>();
   for (int x : A) {
      if (set.contains(x)) {
         set.remove(x);
      } else {
         set.add(x);
      }
   }
   return set.iterator().next();
}
```

The set is pretty efficient and runs in one pass. However, it uses extra space of O(n).

XOR-ing a number with itself is zero. If we XOR all numbers together, it would effectively cancel out all elements that appear twice leaving us with only the single number. As the XOR operation is both commutative and associative, the order in how you XOR them does not matter.

```java
public int singleNumber(int[] A) {
   int num = 0;
   for (int x : A) {
      num ^= x;
   }
   return num;
}
```

Further Thoughts:

Let us change the question a little: “If every element appears even number of times except for one element that appears odd number of times, find that one element”, would the XOR approach work? Why?

34. Single Number II

Code it now: https://oj.leetcode.com/problems/single-number-ii/ Difficulty: Hard, Frequency: Medium

Question:

Given an array of integers, every element appears *three* times except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Solution:

To solve this problem using only constant space, you have to rethink how the numbers are being represented in computers – using bits.

If you sum the i^{th} bit of all numbers and mod 3, it must be either 0 or 1 due to the constraint of this problem where each number must appear either three times or once. This will be the i^{th} bit of that "single number".

A straightforward implementation is to use an array of size 32 to keep track of the total count of i^{th} bit.

```
int singleNumber(int A[], int n) {
    int count[32] = {0};
    int result = 0;
    for (int i = 0; i < 32; i++) {
        for (int j = 0; j < n; j++) {
            if ((A[j] >> i) & 1) {
                count[i]++;
            }
        }
        result |= ((count[i] % 3) << i);
    }
    return result;
}
```

We can improve this based on the previous solution using three bitmask variables:

1. ones as a bitmask to represent the i^{th} bit had appeared once.
2. twos as a bitmask to represent the i^{th} bit had appeared twice.
3. threes as a bitmask to represent the i^{th} bit had appeared three times.

When the i^{th} bit had appeared for the third time, clear the i^{th} bit of both ones and twos to 0. The final answer will be the value of ones.

```
int singleNumber(int A[], int n) {
    int ones = 0, twos = 0, threes = 0;
    for (int i = 0; i < n; i++) {
        twos |= ones & A[i];
        ones ^= A[i];
        threes = ones & twos;
        ones &= ~threes;
        twos &= ~threes;
    }
    return ones;
}
```

Further Thoughts:

If we extend the problem to:

> Given an array of integers, every element appears *k* times except for one. Find that single one which appears *l* times.

How would you solve it?

Please see the excellent answer by @ranmocy in LeetCode Discuss:

https://oj.leetcode.com/discuss/857/constant-space-solution?show=2542#a2542

Chapter 6: Misc

35. Spiral Matrix

Code it now: https://oj.leetcode.com/problems/spiral-matrix/ Difficulty: Medium, Frequency: Medium

Question:

Given a matrix of m × n elements (m rows, n columns), return all elements of the matrix in spiral order.

For example, given the following matrix:

```
[
  [ 1, 2, 3 ],
  [ 4, 5, 6 ],
  [ 7, 8, 9 ]
]
```

You should return [1,2,3,6,9,8,7,4,5].

Solution:

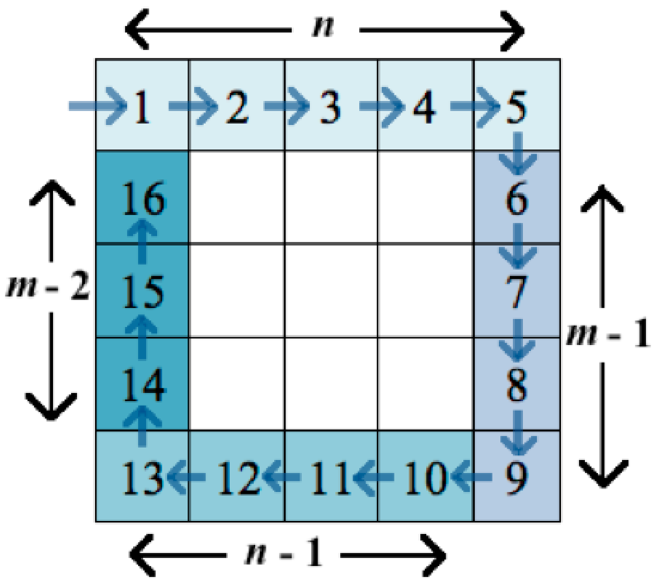

Figure 3: A m × n matrix. The arrows show the direction of traversal in spiral order.

We simulate walking the matrix from the top left corner in a spiral manner. In the outmost level, we traverse n steps right, m – 1 steps down, n – 1 steps left, and m – 2 steps up, then we continue traverse into its next inner level.

As the traversal spiral toward the matrix's center, we stop by determining if we have reached the "center". However, defining the "center" is difficult since the matrix is not necessarily a square. Think of edge cases such as 1×1, 1×10 and 10×1 matrices, where is the “center”? These cases had to be dealt separately and are messy.

A cleaner solution is to keep track of our current position and the number of steps in both horizontal and vertical directions. As we change direction we decrement the steps in that direction. When the number of steps in a direction becomes zero, we know that we have finished traversing the entire matrix.

```
public List<Integer> spiralOrder(int[][] matrix) {
   List<Integer> elements = new ArrayList<>();
   if (matrix.length == 0) return elements;
   int m = matrix.length, n = matrix[0].length;
   int row = 0, col = -1;
   while (true) {
      for (int i = 0; i < n; i++) {
         elements.add(matrix[row][++col]);
      }
      if (--m == 0) break;
      for (int i = 0; i < m; i++) {
         elements.add(matrix[++row][col]);
      }
      if (--n == 0) break;
      for (int i = 0; i < n; i++) {
         elements.add(matrix[row][--col]);
      }
      if (--m == 0) break;
      for (int i = 0; i < m; i++) {
         elements.add(matrix[--row][col]);
      }
      if (--n == 0) break;
   }
   return elements;
}
```

36. Integer to Roman

Code it now: https://oj.leetcode.com/problems/integer-to-roman/ Difficulty: Medium, Frequency: Low

Question:

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

Hint:

What is the range of the numbers?

Solution:

Roman Literal	Decimal
I	**1**
V	**5**
X	**10**
L	**50**
C	**100**
D	**500**
M	**1000**

Table 1: Roman literals and its decimal representations.

First, let's understand how to read roman numerals. The rule of roman numerals is simple: Symbols are placed from left to right starting with the largest, and we add the values according to the additive notation. However, there is an exception to avoid four symbols being repeated in succession, also known as the subtractive notation.

The additive notation:

We combine the symbols and add the values. For example, III is three ones, which is 3. Another example XV means ten followed by a five, which is 15.

The subtractive notation:

Four characters are avoided being repeated in succession (such as IIII). Instead, the symbol I could appear before V and X to signify 4 (IV) and 9 (IX) respectively. Using the same pattern, we observe that X could appear before L and C to signify 40 (XL) and 90 (XC) respectively. The same pattern could be applied to C that is placed before D and M.

With our understanding of roman numerals, we have to decide how to extract the digits from the integer. Should we extract from right to left (from the least significant digit) or from left to right (from the most significant digit)?

If digits are extracted from right to left, we have to append the symbols in reversed order. Extracting digits from left to right seem more natural. It is also slightly trickier but not if we know the maximum number of digits could the number have in advanced, which we do – The number is within the range from 1 to 3999.

Using the additive notation, we convert to roman numerals by breaking it so each chunk can be represented by the symbol entity. For example, 11 = 10 + 1 = “X” + “I”. Similarly, 6 = 5 + 1 = “V” + “I”. Let’s take a look of an example which uses the subtractive notation: 49 = 40 + 9 = “XL” + “IX”. Note that we treat “XL” and “IX” as one single entity to avoid dealing with these special cases to greatly simplify the code.

```java
private static final int[] values = {
   1000, 900, 500, 400,
   100,  90,  50,  40,
   10,   9,   5,   4,
   1
};
private static final String[] symbols = {
   "M", "CM", "D", "CD",
   "C", "XC", "L", "XL",
   "X", "IX", "V", "IV",
   "I"
};

public String intToRoman(int num) {
   StringBuilder roman = new StringBuilder();
   int i = 0;
   while (num > 0) {
      int k = num / values[i];
      for (int j = 0; j < k; j++) {
         roman.append(symbols[i]);
         num -= values[i];
      }
      i++;
   }
   return roman.toString();
}
```

Follow up:

Implement Roman to Integer. See Question [37. Roman to Integer].

37. Roman to Integer

Code it now: https://oj.leetcode.com/problems/roman-to-integer/ Difficulty: Medium, Frequency: Low

Question:

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

Solution:

Roman Literal	**Decimal**
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Table 2: Roman literals and its decimal representations.

Let’s work through some examples. Assume the input is “VII”, using the [**additive notation**], we could simply add up each roman literal, ‘V’ + ‘I’ + ‘I’ = 5 + 1 + 1 = 7.

Now let’s look at another example input “IV”. Now we need to use the [**subtractive notation**]. We first look at ‘I’, and we add 1 to it. Then we look at ‘V’ and since a *smaller* roman literal ‘I’ appears before it, we need to subtract ‘I’ from ‘V’. Remember that we already added another ‘I’ before it, so we need to subtract a total of two one’s from it.

Below is a more complex example that involves both additive and subtractive notation: “MXCVI”.

Roman literals from left to right	**Accumulated total**
M	**1000**
M**X**	1000 + **10** = 1010
MX**C**	1010 + (**100 – 2 * 10**) = 1010 + 80 = 1090
MXC**V**	1090 + **5** = 1095
MXCV**I**	1095 + **1** = 1096

Table 3: Step by step calculation of roman numeral "MXCVI".

```
private Map<Character, Integer> map =
      new HashMap<Character, Integer>() {{
            put('I', 1);  put('V', 5);   put('X', 10);
            put('L', 50); put('C', 100); put('D', 500);
            put('M', 1000);
      }};

public int romanToInt(String s) {
   int prev = 0, total = 0;
   for (char c : s.toCharArray()) {
      int curr = map.get(c);
      total += (curr > prev) ? (curr - 2 * prev) : curr;
      prev = curr;
   }
   return total;
}
```

38. Clone graph

Code it now: https://oj.leetcode.com/problems/clone-graph/ Difficulty: Medium, Frequency: Medium

Question:

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

Solution:

There are two main ways to traverse a graph: *breadth-first* or *depth-first*. Let's try the depth-first approach first, which is a recursion algorithm. Then we will look at the breadth-first approach, which is an iterative algorithm that uses a queue.

***O*(*n*) runtime, *O*(*n*) space – Depth-first traversal:**

A graph is simply represented by a graph node that serves as its starting point. In fact, the starting point could be any other graph nodes and it does not affect the cloning algorithm.

As each of its neighbors is a graph node too, we could recursively clone each of its neighbors and assign it to each neighbor of the cloned node. We can easily see that it is doing a depth-first traversal of each node.

Note that the graph could contain cycles, for example a node could have a neighbor that points back to it. Therefore, we should use a map that records each node's copy to avoid infinite recursion.

```
public UndirectedGraphNode cloneGraph(UndirectedGraphNode graph) {
   if (graph == null) return null;
   Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();
   return DFS(graph, map);
}

private UndirectedGraphNode DFS(UndirectedGraphNode graph,
      Map<UndirectedGraphNode, UndirectedGraphNode> map) {
   if (map.containsKey(graph)) {
      return map.get(graph);
   }
   UndirectedGraphNode graphCopy = new UndirectedGraphNode(graph.label);
   map.put(graph, graphCopy);
   for (UndirectedGraphNode neighbor : graph.neighbors) {
      graphCopy.neighbors.add(DFS(neighbor, map));
   }
   return graphCopy;
}
```

***O*(*n*) runtime, *O*(*n*) space – Breadth-first traversal:**

How does the breadth-first traversal works? Easy, as we pop a node off the queue, we copy each of its neighbors, and push them to the queue.

A straight forward breadth-first traversal seemed to work. But some details are still missing. For example, how do we connect the nodes of the cloned graph?

The fact that B can traverse back to A implies that the graph may contain a cycle. You must take extra care to handle this case or else your code could have an infinite loop.

Let’s analyze this further by using the below example:

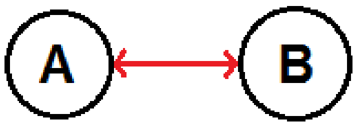

Figure 4: A simple graph

Assume that the starting point of the graph is A. First, you make a copy of node A (A2), and found that A has only one neighbor B. You make a copy of B (B2) and connect A2→B2 by pushing B2 as A2’s neighbor. Next, you find that B has A as neighbor, which you have already made a copy of. Here, we have to be careful not to make a copy of A again, but to connect B2→A2 by pushing A2 as B2’s neighbor. But, how do we know if a node has already been copied?

Easy, we could use a hash table! As we copy a node, we insert it into the table. If we later find that one of a node’s neighbors is already in the table, we do not make a copy of that neighbor, but to push its neighbor’s copy to its copy instead. Therefore, the hash table would need to store a mapping of key-value pairs, where the key is a node in the original graph and its value is the node’s copy.

```
public UndirectedGraphNode cloneGraph(UndirectedGraphNode graph) {
   if (graph == null) return null;
   Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();
   Queue<UndirectedGraphNode> q = new LinkedList<>();
   q.add(graph);
   UndirectedGraphNode graphCopy = new UndirectedGraphNode(graph.label);
   map.put(graph, graphCopy);
   while (!q.isEmpty()) {
      UndirectedGraphNode node = q.poll();
      for (UndirectedGraphNode neighbor : node.neighbors) {
         if (map.containsKey(neighbor)) {
            map.get(node).neighbors.add(map.get(neighbor));
         } else {
            UndirectedGraphNode neighborCopy =
                  new UndirectedGraphNode(neighbor.label);
            map.get(node).neighbors.add(neighborCopy);
            map.put(neighbor, neighborCopy);
            q.add(neighbor);
         }
      }
   }
   return graphCopy;
}
```

Chapter 7: Stack

39. Min Stack

Code it now: https://oj.leetcode.com/problems/min-stack/ Difficulty: Easy, Frequency: N/A

Question:

Design a stack that supports push, pop, top, and retrieving the minimum element in *constant* time.

- *push*(*x*) – Push element *x* onto stack.
- *pop*() – Removes the element on top of the stack.
- *top*() – Get the top element.
- *getMin*() – Retrieve the minimum element in the stack.

Hints:

- ➢ Consider space-time tradeoff. How would you keep track of the minimums using extra space?
- ➢ Make sure to consider duplicate elements.

Solution:

O(*n*) runtime, *O*(*n*) space – Extra stack:

Consider using an extra stack to keep track of the current minimum value. During the push operation we choose the new element or the current minimum, whichever that is smaller to push onto the min stack.

For the pop operation, we would pop from both stacks. *getMin*() is then reflected by the top element of min stack.

To illustrate this idea, we push the elements 1, 4, 3, 0, 3 in that order.

Main stack	Min stack
3	0
0	0
3	1
4	1
1	1

After popping two elements from the stack it becomes:

Main stack	Min stack
3	1
4	1
1	1