

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.

```
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.

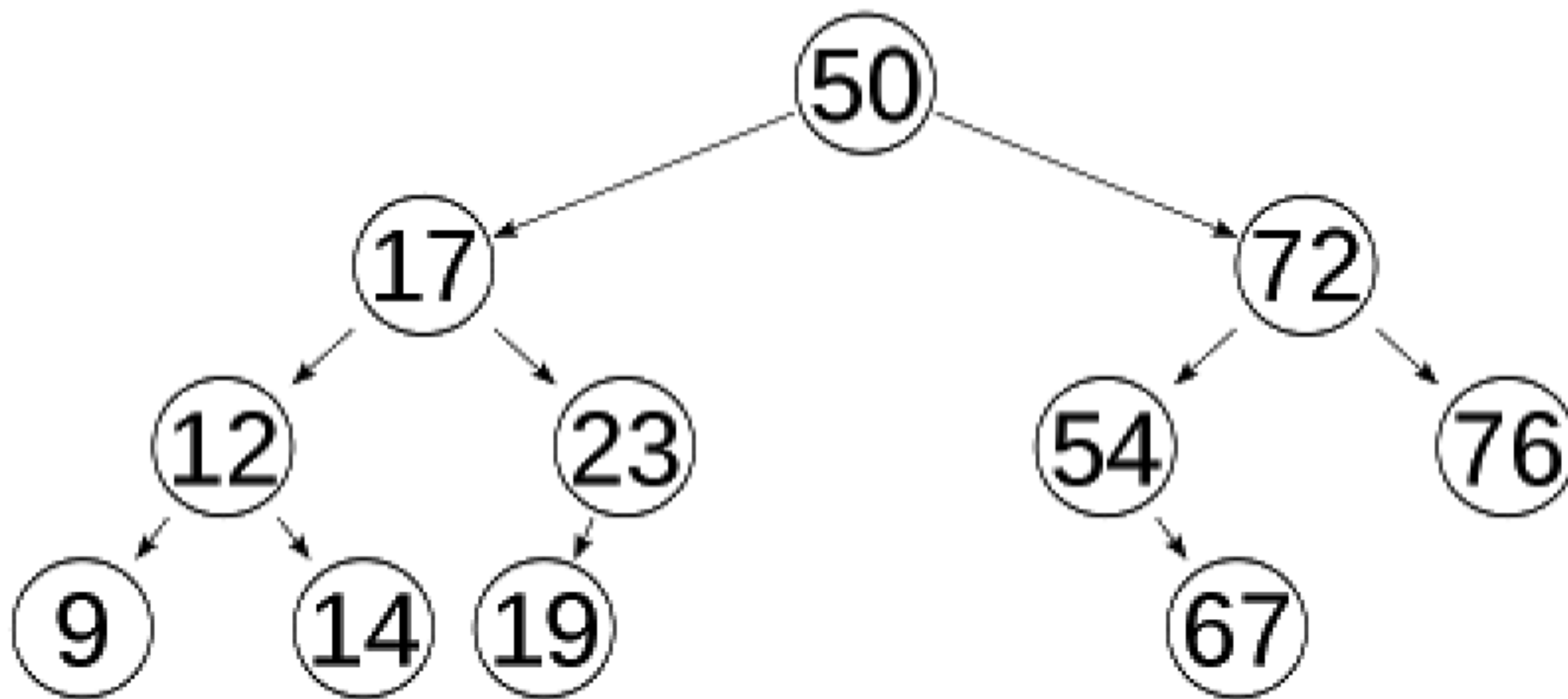


Figure 2: An example of a height-balanced tree. A height-balanced tree is a tree whose subtrees differ in height by no more than one and the subtrees are height-balanced, too.

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 $O(\log n)$.

```

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 $O(n)$.

```

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,



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?

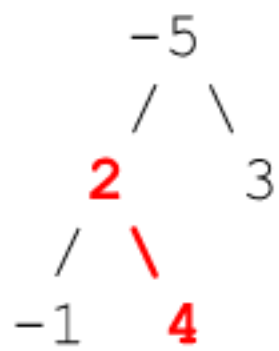
A: Then the maximum path sum starts and ends at the same node.

Q: What if every node contains negative value?

A: Then you should return the single node value that is the least negative.

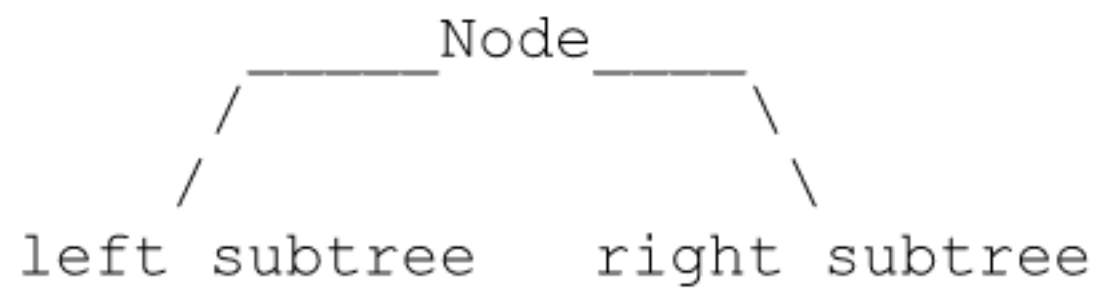
Q: Does the maximum path have to go through the root node?

A: Not necessarily. For example, the below tree yield 6 as the maximum path sum and does not go through root.



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.



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

- i. $\text{max}(\text{left subtree}) + \text{node}$
- ii. $\text{max}(\text{right subtree}) + \text{node}$
- iii. $\text{max}(\text{left subtree}) + \text{max}(\text{right subtree}) + \text{node}$
- 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.

```

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: Coming soon!

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 = null, 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.

```

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.

```

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 $m \times n$ 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:

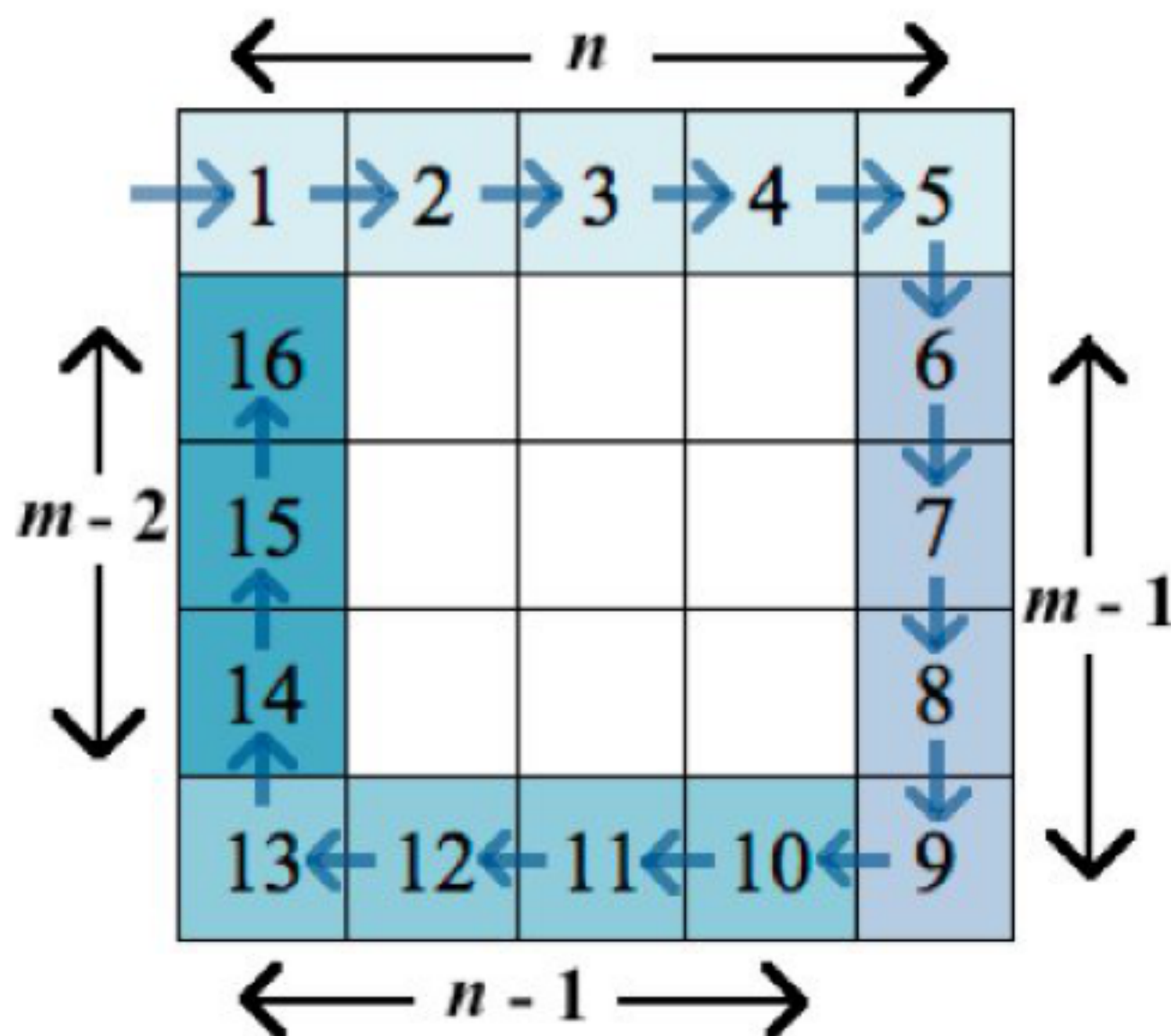


Figure 3: A $m \times n$ matrix. The arrows show the direction of traversal in spiral order.

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, $11 = 10 + 1 = \text{"X"} + \text{"I"}$. Similarly, $6 = 5 + 1 = \text{"V"} + \text{"I"}$. Let's take a look of an example which uses the subtractive notation: $49 = 40 + 9 = \text{"XL"} + \text{"IX"}$. Note that we treat "XL" and "IX" as one single entity to avoid dealing with these special cases to greatly simplify the code.

```
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
MX	1000 + 10 = 1010
MXC	1010 + (100 - 2 * 10) = 1010 + 80 = 1090
MXCV	1090 + 5 = 1095
MXCVI	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

$O(n)$ runtime, $O(n)$ space – Minor space optimization:

If a new element is larger than the current minimum, we do not need to push it on to the min stack. When we perform the pop operation, check if the popped element is the same as the current minimum. If it is, pop it off the min stack too.

```
class MinStack {
    private Stack<Integer> stack = new Stack<>();
    private Stack<Integer> minStack = new Stack<>();

    public void push(int x) {
        stack.push(x);
        if (minStack.isEmpty() || x <= minStack.peek()) {
            minStack.push(x);
        }
    }

    public void pop() {
        if (stack.pop().equals(minStack.peek())) minStack.pop();
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return minStack.peek();
    }
}
```

40. Evaluate Reverse Polish Notation

Code it now: <https://oj.leetcode.com/problems/evaluate-reverse-polish-notation/>

Difficulty: Medium, Frequency: Low

Question:

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another expression.

Some examples:

`["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9`

`["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6`

Example Questions Candidate Might Ask:

Q: Is an empty array a valid input?

A: No.

Solution:

The Reverse Polish Notation (RPN) is also known as the postfix notation, because each operator appears *after* its operands. For example, the infix notation “3 + 4” is expressed as “3 4 +” in RPN.

The brute force approach:

We look for the simplest RPN sequence that could be evaluated immediately, that is: Two successive operands followed by an operator such as “4 2 +”. We replace this sequence with the expression’s value and repeat until there is only one operand left.

For example,

`["4", "13", "5", "/", "+"]`

→ `["4", "2", "+"]`

→ `["6"]`

How would you do the replacement? You could do it in-place with the input array, but it would result in quadratic runtime as elements have to be shifted every time a replacement occurs.

A workaround is to copy the input array to a doubly linked list. The replacement is efficient and the next scan begins with the replaced value’s previous element. This results in an algorithm with linear runtime and linear space. Although this works, the implementation is complex and is far from ideal in an interview session.

The optimal approach:

Observe that every time we see an operator, we need to evaluate the last two operands. Stack fits perfectly as it is a Last-In-First-Out (LIFO) data structure.

We evaluate the expression left-to-right and push operands onto the stack until we encounter an operator, which we pop the top two values from the stack. We then evaluate the operator, with the values as arguments and push the result back onto the stack.

For example, the infix expression “ $8 - ((1 + 2) * 2)$ ” in RPN is:

8 1 2 + 2 * -

Input	Operation	Stack	Notes
8	Push operand	8	
1	Push operand	1 8	
2	Push operand	2 1 8	
+	Add	3 8	Pop two values 1, 2 and push result 3
2	Push operand	2 3 8	
*	Multiply	6 8	Pop two values 3, 2 and push result 6
-	Subtract	2	Pop two values 8, 6 and push result 2

After the algorithm finishes, the stack contains only one value which is the RPN expression's result; in this case, 2.


```

private static final Set<String> OPERATORS =
    new HashSet<>(Arrays.asList("+", "-", "*", "/"));

public int evalRPN(String[] tokens) {
    Stack<Integer> stack = new Stack<>();
    for (String token : tokens) {
        if (OPERATORS.contains(token)) {
            int y = stack.pop();
            int x = stack.pop();
            stack.push(eval(x, y, token));
        } else {
            stack.push(Integer.parseInt(token));
        }
    }
    return stack.pop();
}

private int eval(int x, int y, String operator) {
    switch (operator) {
        case "+": return x + y;
        case "-": return x - y;
        case "*": return x * y;
        default: return x / y;
    }
}

```

Further Thoughts:

The above code contains duplication. For example, if we decide to add a new operator, we would need to update the code in two places – in the set's initialization and the switch statement. Could you refactor the code so it is more extensible?

You are probably not expected to write this refactored code during an interview session. However, it will make you a stronger candidate if you could make this observation and point this out, as it shows to the interviewer that you care about clean code.

In Java, create an interface called `Operator` and map each operator string to an implementation of the `Operator` interface. For other languages such as C++, each operator will be mapped to a function pointer instead.

```

interface Operator {
    int eval(int x, int y);
}

private static final Map<String, Operator> OPERATORS =
    new HashMap<String, Operator>() {{
        put("+", new Operator() {
            public int eval(int x, int y) { return x + y; }
        });
        put("-", new Operator() {
            public int eval(int x, int y) { return x - y; }
        });
        put("*", new Operator() {
            public int eval(int x, int y) { return x * y; }
        });
        put("/", new Operator() {
            public int eval(int x, int y) { return x / y; }
        });
    }};

public int evalRPN(String[] tokens) {
    Stack<Integer> stack = new Stack<>();
    for (String token : tokens) {
        if (OPERATORS.containsKey(token)) {
            int y = stack.pop();
            int x = stack.pop();
            stack.push(OPERATORS.get(token).eval(x, y));
        } else {
            stack.push(Integer.parseInt(token));
        }
    }
    return stack.pop();
}

```

41. Valid Parentheses

Code it now: <https://oj.leetcode.com/problems/valid-parentheses/>

Difficulty: Easy, Frequency: High

Question:

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "[)]" are not.

Example Questions Candidate Might Ask:

Q: Is the empty string valid?

A: Yes.

Solution:

To validate the parentheses, we need to match each closing parenthesis with its opening counterpart. A Last-In-First-Out (LIFO) data structure such as stack is the perfect fit.

As we see an opening parenthesis, we push it onto the stack. On the other hand, when we encounter a closing parenthesis, we pop the last inserted opening parenthesis from the stack and check if the pair is a valid match.

It would be wise to avoid writing multiple if statements when matching parentheses, as your interviewer may think that you are writing sloppy code. You could use a map, which is more maintainable.

```
private static final Map<Character, Character> map =
    new HashMap<Character, Character>() {{
        put('(', ')');
        put('{', '}');
        put('[', ']');
    }};

public boolean isValid(String s) {
    Stack<Character> stack = new Stack<>();
    for (char c : s.toCharArray()) {
        if (map.containsKey(c)) {
            stack.push(c);
        } else if (stack.isEmpty() || map.get(stack.pop()) != c) {
            return false;
        }
    }
    return stack.isEmpty();
}
```

Chapter 8: Dynamic Programming

42. Climbing Stairs

Code it now: <https://oj.leetcode.com/problems/climbing-stairs/>

Difficulty: Easy, Frequency: High

Question:

You are climbing a staircase. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Solution:

$O(n)$ runtime, $O(1)$ space – Dynamic programming:

This is a classic Dynamic Programming problem.

Define:

$f(n)$ = number of ways you can climb to the n th step.

To reach to the n^{th} step, you have only two choices:

1. Advance one step from the $n - 1^{\text{th}}$ step.
2. Advance two steps from the $n - 2^{\text{th}}$ step.

Therefore, $f(n) = f(n - 1) + f(n - 2)$, which is the exact same recurrence formula defined by the [Fibonacci sequence](#) (with different base cases, though).

Set base cases $f(1) = 1, f(2) = 2$ and you are almost done.

Now, we could calculate $f(n)$ easily by storing previous values in an one dimension array and work our way up to n . Heck, we can even optimize this further by storing just the previous two values.

```
public int climbStairs(int n) {  
    int p = 1, q = 1;  
    for (int i = 2; i <= n; i++) {  
        int temp = q;  
        q += p;  
        p = temp;  
    }  
    return q;  
}
```


Combinatorics:

Interestingly, this problem could also be solved using combinatorics.

Warning: Math-y stuff ahead, feel free to skip this section if you are not interested.

For example, let's assume $n = 6$.

Let:

x = number of 1's,

y = number of 2's.

We could reach the top using one of the four combinations below:

x	y	
6	0	=> 1) Six single steps.
4	1	=> 2) Four single steps and one double step.
2	2	=> 3) Two single steps and two double steps.
0	3	=> 4) Three double steps.

For the first combination pair $(x,y) = (6,0)$, there's obviously only one way of arranging six single steps.

For the second combination pair $(4,1)$, there's five ways of arranging (think of it as slotting the double step between the single steps).

Similarly, there are six ways $C(4,2)$ and one way $\binom{3}{3}$ of arranging the third and fourth combination pairs respectively.

Generally, for pair (x,y) , there are a total of $\binom{x+y}{y} = \frac{(x+y)!}{x!y!}$ ways of arranging the 1's and 2's.

The total number of possible ways is the sum of all individual terms,

$$f(6) = 1 + 5 + 6 + 1 = 13.$$

Generalizing for all n 's (including odd n),

$$f(n) = \binom{n}{0} + \binom{n-1}{1} + \binom{n-2}{2} + \dots + \binom{\text{ceil}(\frac{n}{2})}{\text{floor}(\frac{n}{2})}$$

43. Unique Paths

Code it now: <https://oj.leetcode.com/problems/unique-paths/>

Difficulty: Medium, Frequency: Low

Question:

A robot is located at the top-left corner of a $m \times n$ grid (marked 'Start' in the diagram below). The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below). How many possible unique paths are there?

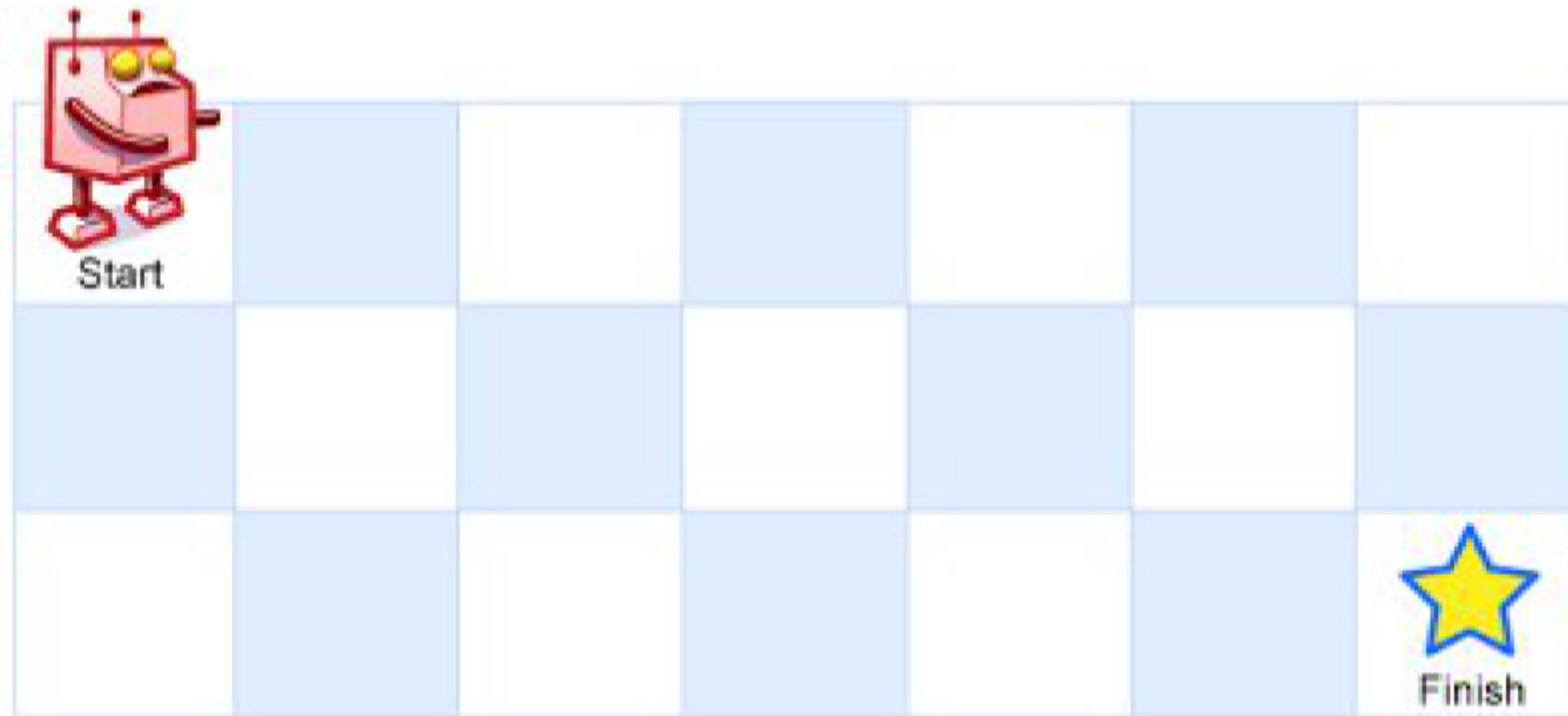


Figure 5: The grid above is 7×3, and is used to illustrate the problem.

$O\left(\binom{m+n}{m}\right)$ runtime, $O(m + n)$ space – Backtracking:

The most direct way is to write code that traverses each possible path, which can be done using backtracking. When you reach $row = m$ and $col = n$, you know you've reached the bottom-right corner, and there is one additional unique path to it. However, when you reach $row > m$ or $col > n$, then it's an invalid path and you should stop traversing. For any grid at $row = r$ and $col = c$, you have two choices: Traverse to the right or traverse to the bottom. Therefore, the total unique paths at grid (r, c) are equal to the sum of total unique paths from the grid to the right and the grid below.

Deriving the runtime complexity is slightly tricky. Observe that the robot must go right exactly m times and go down exactly n times. Assume that the right movement is 0 and the down movement is 1. We could then represent the robot's path as a binary string of length $= m + n$, where the string contains m zeros and n ones. Since the backtracking algorithm is just trying to explore all possibilities, its runtime complexity is equivalent to the total permutations of a binary string that contains m zeros and n ones, which is $\binom{m+n}{m}$.

On the other hand, the space complexity is $O(m + n)$ due to the recursion that goes at most $m + n$ level deep.

Below is the backtracking code in just five lines of code:

```

private int backtrack(int r, int c, int m, int n) {
    if (r == m - 1 && c == n - 1)
        return 1;
    if (r >= m || c >= n)
        return 0;

    return backtrack(r + 1, c, m, n) + backtrack(r, c + 1, m, n);
}

public int uniquePaths(int m, int n) {
    return backtrack(0, 0, m, n);
}

```

Improved Backtracking Solution using Memoization:

Although the above backtracking solution is easy to code, it is very inefficient in the sense that it recalculates the same solution for a grid over and over again. By caching the results, we prevent recalculation and only calculate when necessary. Here, we are using a dynamic programming (DP) technique called memoization.

```

private int backtrack(int r, int c, int m, int n, int[][] mat) {
    if (r == m - 1 && c == n - 1)
        return 1;
    if (r >= m || c >= n)
        return 0;

    if (mat[r + 1][c] == -1)
        mat[r + 1][c] = backtrack(r + 1, c, m, n, mat);
    if (mat[r][c + 1] == -1)
        mat[r][c + 1] = backtrack(r, c + 1, m, n, mat);

    return mat[r + 1][c] + mat[r][c + 1];
}

public int uniquePaths(int m, int n) {
    int[][] mat = new int[m + 1][n + 1];
    for (int i = 0; i < m + 1; i++) {
        for (int j = 0; j < n + 1; j++) {
            mat[i][j] = -1;
        }
    }
    return backtrack(0, 0, m, n, mat);
}

```

$O(mn)$ runtime, $O(mn)$ space – Bottom-up dynamic programming:

If you notice closely, the above DP solution is using a *top-down* approach. Now let's try a *bottom-up* approach. Remember this important relationship that is necessary for this DP solution to work:

The total unique paths at grid (r, c) are equal to the sum of total unique paths from grid to the right $(r, c + 1)$ and the grid below $(r + 1, c)$.

How can this relationship help us solve the problem? We observe that all grids of the bottom edge and right edge must all have only one unique path to the bottom-right

corner. Using this as the base case, we can build our way up to our solution at grid (1, 1) using the relationship above.

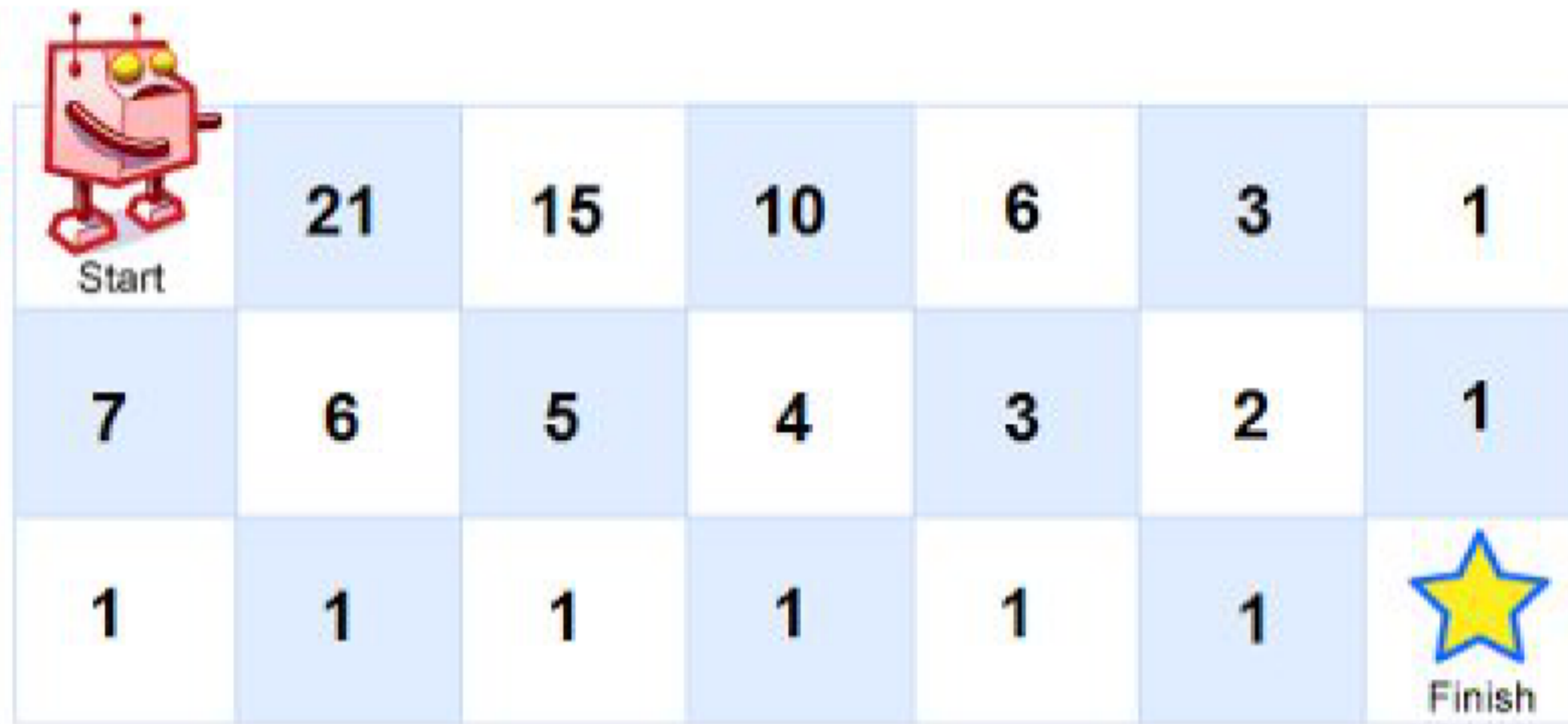


Figure 6: The total unique paths at grid (r, c) are equal to the sum of total unique paths from grid to the right (r, c + 1) and the grid below (r + 1, c).

```
public int uniquePaths(int m, int n) {
    int[][] mat = new int[m + 1][n + 1];
    mat[m - 1][n] = 1;
    for (int r = m - 1; r >= 0; r--) {
        for (int c = n - 1; c >= 0; c--) {
            mat[r][c] = mat[r + 1][c] + mat[r][c + 1];
        }
    }
    return mat[0][0];
}
```

Combinatorial Solution:

It turns out this problem could be solved using combinatorics, which no doubt would be the most efficient solution. In order to see it as a combinatorial problem, there are some necessary observations. Look at the 7×3 sample grid in the picture above. Notice that no matter how you traverse the grids, you always traverse a total of 8 steps. To be more exact, you always have to choose 6 steps to the right (R) and 2 steps to the bottom (B). Therefore, the problem can be transformed to a question of how many ways can you choose 6R's and 2B's in these 8 steps. The answer is $\binom{8}{2}$ or $\binom{8}{6}$. Therefore, the general solution for an $m \times n$ grid is $\binom{m+n-2}{m-1}$.

Further Thoughts:

Now consider if some obstacles are added to the grids marked as 'X'. How many unique paths would there be? A combinatorial solution is difficult to obtain, but the DP solution can be modified easily to accommodate this constraint. See Question [44. Unique Paths II].

44. Unique Paths II

Code it now: <https://oj.leetcode.com/problems/unique-paths-ii/>

Difficulty: Medium, Frequency: High

Question:

Similar to Question [43. Unique Paths], but now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space are marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3×3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is 2.

Solution:

$O(mn)$ runtime, $O(mn)$ space – Dynamic programming:

It turns out to be really easy to extend from the [Bottom-up dynamic programming] approach above. Just set the total paths to 0 when you encounter an obstacle.

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    int m = obstacleGrid.length;
    if (m == 0) return 0;
    int n = obstacleGrid[0].length;
    int[][] mat = new int[m + 1][n + 1];
    mat[m - 1][n] = 1;
    for (int r = m - 1; r >= 0; r--) {
        for (int c = n - 1; c >= 0; c--) {
            mat[r][c] = (obstacleGrid[r][c] == 1) ? 0 : mat[r][c+1] + mat[r+1][c];
        }
    }
    return mat[0][0];
}
```

45. Maximum Sum Subarray

Code it now: <https://oj.leetcode.com/problems/maximum-subarray/>

Difficulty: Medium, Frequency: High

Question:

Find the contiguous subarray within an array (containing at least one number) that has the largest sum.

For example, given the array $[2, 1, -3, 4, -1, 2, 1, -5, 4]$,

The contiguous array $[4, -1, 2, 1]$ has the largest sum = 6.

Solution:

$O(n \log n)$ runtime, $O(\log n)$ stack space – Divide and Conquer:

Assume we partition the array A into two smaller arrays S and T at the middle index, M . Then, $S = A_1 \dots A_{M-1}$, and $T = A_{M+1} \dots A_N$.

The contiguous subarray that has the largest sum could either:

- i. Contain the middle element:
 - a. The largest sum is the maximum suffix sum of $S + A_M$ + the maximum prefix sum of T .
- ii. Does not contain the middle element:
 - a. The largest sum is in S , which we could apply the same algorithm to S .
 - b. The largest sum is in T , which we could apply the same algorithm to T .

```
public int maxSubArray(int[] A) {
    return maxSubArrayHelper(A, 0, A.length - 1);
}

private int maxSubArrayHelper(int[] A, int L, int R) {
    if (L > R) return Integer.MIN_VALUE;
    int M = (L + R) / 2;
    int leftAns = maxSubArrayHelper(A, L, M - 1);
    int rightAns = maxSubArrayHelper(A, M + 1, R);
    int lMaxSum = 0;
    int sum = 0;
    for (int i = M - 1; i >= L; i--) {
        sum += A[i];
        lMaxSum = Math.max(sum, lMaxSum);
    }
    int rMaxSum = 0;
    sum = 0;
    for (int i = M + 1; i <= R; i++) {
        sum += A[i];
        rMaxSum = Math.max(sum, rMaxSum);
    }
    return Math.max(lMaxSum + A[M] + rMaxSum, Math.max(leftAns, rightAns));
}
```

The runtime complexity could be expressed as $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, which is $O(n \log n)$. We will not attempt to prove it here; you could read up any advanced algorithm textbooks to learn the proof.

$O(n)$ runtime, $O(1)$ space – Dynamic programming:

To devise a dynamic programming formula, let us assume that we are calculating the maximum sum of subarray that ends at a specific index.

Let us denote that:

$f(k)$ = Maximum sum of subarray ending at index k .

Then,

$f(k) = \max(f(k-1) + A[k], A[k])$

Using an array of size n , We could deduce the final answer by as $f(n - 1)$, with the initial state of $f(0) = A[0]$. Since we only need to access its previous element at each step, two variables are sufficient. Notice the difference between the two: *maxEndingHere* and *maxSoFar*; the former is the maximum sum of subarray that *must* end at index k , while the latter is the global maximum subarray sum.

```
public int maxSubArray(int[] A) {
    int maxEndingHere = A[0], maxSoFar = A[0];
    for (int i = 1; i < A.length; i++) {
        maxEndingHere = Math.max(maxEndingHere + A[i], A[i]);
        maxSoFar = Math.max(maxEndingHere, maxSoFar);
    }
    return maxSoFar;
}
```


46. Maximum Product Subarray

Code it now: <https://oj.leetcode.com/problems/maximum-product-subarray/>

Difficulty: Medium, Frequency: High

Question:

Find the contiguous subarray within an array of integers that has the largest product. For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

Example Questions Candidate Might Ask:

Q: Could the subarray be empty?

A: No, the subarray must contain at least one number.

Solution:

This problem is very similar to Question [45. Maximum Sum Subarray]. There is a slight twist though. Besides keeping track of the largest product, we also need to keep track of the smallest product. Why? The smallest product, which is the largest in the negative sense could become the maximum when being multiplied by a negative number.

Let us denote that:

$f(k)$ = Largest product subarray, from index 0 up to k .

Similarly,

$g(k)$ = Smallest product subarray, from index 0 up to k .

Then,

$f(k) = \max(f(k-1) * A[k], A[k], g(k-1) * A[k])$

$g(k) = \min(g(k-1) * A[k], A[k], f(k-1) * A[k])$

There we have a dynamic programming formula. Using two arrays of size n , we could deduce the final answer as $f(n-1)$. Since we only need to access its previous elements at each step, two variables are sufficient.

```
public int maxProduct(int[] A) {  
    assert A.length > 0;  
    int max = A[0], min = A[0], maxAns = A[0];  
    for (int i = 1; i < A.length; i++) {  
        int mx = max, mn = min;  
        max = Math.max(Math.max(A[i], mx * A[i]), mn * A[i]);  
        min = Math.min(Math.min(A[i], mx * A[i]), mn * A[i]);  
        maxAns = Math.max(max, maxAns);  
    }  
    return maxAns;  
}
```


47. Coins in a Line

Code it now: Coming soon!

Difficulty: Hard, Frequency: N/A

Question:

There are n coins in a line. (Assume n is even). Two players take turns to take a coin from one of the ends of the line until there are no more coins left. The player with the larger amount of money wins.

1. Would you rather go first or second? Does it matter?
2. Assume that you go first, describe an algorithm to compute the maximum amount of money you can win.



Figure 7: U.S. coins in various denominations in a line. Two players take turn to pick a coin from one of the ends until no more coins are left. Whoever with the larger amount of money wins.

Hints:

If you go first, is there a strategy you can follow which prevents you from losing? Try to consider how it matters when the number of coins is odd vs. even.

Solution:

Going first will guarantee that you will not lose. By following the strategy below, you will always win the game (or get a possible tie).

1. Count the sum of all coins that are odd-numbered. (Call this X)
2. Count the sum of all coins that are even-numbered. (Call this Y)
3. If $X > Y$, take the left-most coin first. Choose all odd-numbered coins in subsequent moves.
4. If $X < Y$, take the right-most coin first. Choose all even-numbered coins in subsequent moves.
5. If $X == Y$, you will guarantee to get a tie if you stick with taking only even-numbered/odd-numbered coins.

You might be wondering how you can always choose odd-numbered/even-numbered coins. Let me illustrate this using an example where you have 10 coins:

If you take the coin numbered 1 (the left-most coin), your opponent can only have the choice of taking coin numbered 2 or 10 (which are both even-numbered coins). On the other hand, if you choose to take the coin numbered 10 (the right-most coin), your opponent can only take coin numbered 1 or 9 (which are odd-numbered coins).

Notice that the total number of coins change from even to odd and vice-versa when player takes turn each time. Therefore, by going first and depending on the coin you choose, you are essentially forcing your opponent to take either only even-numbered or odd-numbered coins.

Now that you have found a non-losing strategy, could you compute the maximum amount of money you can win?

Hints:

One misconception is to think that the above non-losing strategy would generate the maximum amount of money as well. This is probably incorrect. Could you find a counter example? (You might need at least 6 coins to find a counter example).

Assume that you are finding the maximum amount of money in a certain range (ie, from coins numbered i to j , inclusive). Could you express it as a recursive formula? Find ways to make it as efficient as possible.

Solution for (2):

Although the simple strategy illustrated in Solution (1) guarantees you not to lose, it does not guarantee that it is optimal in any way.

Here, we use a good counter example to better see why this is so. Assume the coins are laid out as below:

{ 3, 2, 2, 3, 1, 2 }

Following our previous non-losing strategy, we would count the sum of odd-numbered coins, $X = 3 + 2 + 1 = 6$, and the sum of even-numbered coins, $Y = 2 + 3 + 2 = 7$. As $Y > X$, we would take the last coin first and end up winning with the total amount of 7 by taking only even-numbered coins.

However, let us try another way by taking the first coin (valued at 3, denote by (3)) instead. The opponent is left with two possible choices, the left coin (2) and the right coin (2), both valued at 2. No matter which coin the opponent chose, you can always take the other coin (2) next and the configuration of the coins becomes: { 2, 3, 1 }. Now, the coin in the middle (3) would be yours to keep for sure. Therefore, you win the game by a total amount of $3 + 2 + 3 = 8$, which proves that the previous non-losing strategy is not necessarily optimal.

To solve this problem in an optimal way, we need to find efficient means in enumerating all possibilities. This is when Dynamic Programming (DP) kicks in and become so powerful that you start to feel magical.

First, we would need some observations to establish a recurrence relation, which is essential as our first step in solving DP problems.

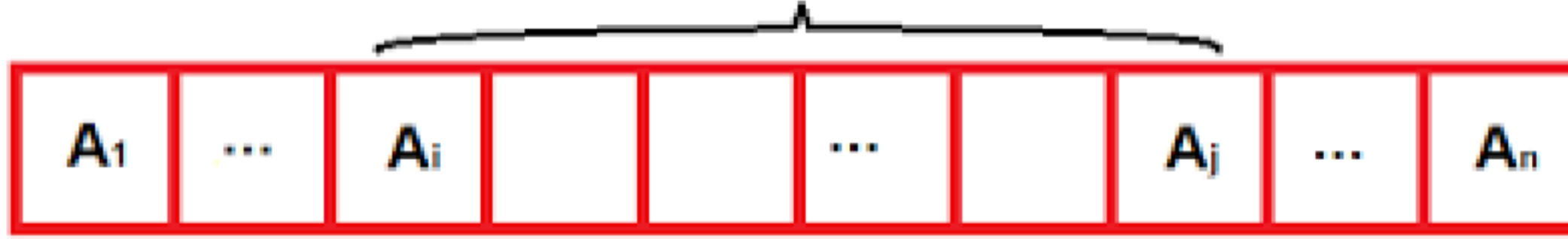


Figure 8: The remaining coins are $\{A_i \dots A_j\}$ and it is your turn. Let $P(i, j)$ denotes the maximum amount of money you can get. Should you choose A_i or A_j ?

Assume that $P(i, j)$ denotes the maximum amount of money you can win when the remaining coins are $\{A_i, \dots, A_j\}$, and it is your turn now. You have two choices, either take A_i or A_j . First, let us focus on the case where you take A_i , so that the remaining coins become $\{A_{i+1} \dots A_j\}$. Since the opponent is as smart as you, he must choose the best way that yields the maximum for him, where the maximum amount he can get is denoted by $P(i + 1, j)$.

Therefore, if you choose A_i , the maximum amount you can get is:

$$P_1 = \sum_{k=i}^j A_k - P(i + 1, j)$$

Similarly, if you choose A_j , the maximum amount you can get is:

$$P_2 = \sum_{k=i}^j A_k - P(i, j - 1)$$

Therefore,

$$\begin{aligned} P(i, j) &= \max(P_1, P_2) \\ &= \max\left(\sum_{k=i}^j A_k - P(i + 1, j), \sum_{k=i}^j A_k - P(i, j - 1)\right) \end{aligned}$$

In fact, we are able to simplify the above relation further to (Why?):

$$P(i, j) = \sum_{k=i}^j A_k - \min(P(i + 1, j), P(i, j - 1))$$

Although the above recurrence relation is easy to understand, we need to compute the value of $\sum_{k=i}^j A_k$ in each step, which is not very efficient. To avoid this problem, we can store values of $\sum_{k=i}^j A_k$ in a table and avoid re-computations by computing in a certain order. Try to figure this out by yourself. (Hint: You would first compute $P(1,1)$, $P(2,2)$, \dots $P(n, n)$ and work your way up).

A Better Solution:

There is another solution that does not rely on computing and storing results of $\sum_{k=i}^j A_k$, therefore is more efficient in terms of time and space. Let us rewind back to the case where you take A_i , and the remaining coins become $\{ A_{i+1} \dots A_j \}$.



Figure 9: You took A_i from the coins $\{ A_i \dots A_j \}$. The opponent will choose either A_{i+1} or A_j . Which one would he choose?

Let us look one extra step ahead this time by considering the two coins the opponent will possibly take, A_{i+1} and A_j . If the opponent takes A_{i+1} , the remaining coins are $\{ A_{i+2} \dots A_j \}$, which our maximum is denoted by $P(i+2, j)$. On the other hand, if the opponent takes A_j , our maximum is $P(i+1, j-1)$. Since the opponent is as smart as you, he would have chosen the choice that yields the minimum amount to you.

Therefore, the maximum amount you can get when you choose A_i is:

$$P_1 = A_i + \min(P(i+2, j), P(i+1, j-1))$$

Similarly, the maximum amount you can get when you choose A_j is:

$$P_2 = A_j + \min(P(i+1, j-1), P(i, j-2))$$

Therefore,

$$\begin{aligned} P(i, j) &= \max(P_1, P_2) \\ &= \max\left(A_i + \min(P(i+2, j), P(i+1, j-1)), A_j + \min(P(i+1, j-1), P(i, j-2))\right) \end{aligned}$$

Although the above recurrence relation could be implemented in few lines of code, its complexity is exponential. The reason is that each recursive call branches into a total of four separate recursive calls, and it could be n levels deep from the very first call).

Memoization provides an efficient way by avoiding re-computations using intermediate results stored in a table. Below is the code which runs in $O(n^2)$ time and takes $O(n^2)$ space.

The code contains a function *printMoves* which prints out all the moves you and the opponent make (assuming both of you are taking the coins in an optimal way).


```

const int MAX_N = 100;

void printMoves(int P[][MAX_N], int A[], int N) {
    int sum1 = 0, sum2 = 0;
    int m = 0, n = N-1;
    bool myTurn = true;
    while (m <= n) {
        int P1 = P[m+1][n]; // If take A[m], opponent can get...
        int P2 = P[m][n-1]; // If take A[n]
        cout << (myTurn ? "I" : "You") << " take coin no. ";
        if (P1 <= P2) {
            cout << m+1 << " (" << A[m] << ")";
            m++;
        } else {
            cout << n+1 << " (" << A[n] << ")";
            n--;
        }
        cout << (myTurn ? ", " : ".\n");
        myTurn = !myTurn;
    }
    cout << "\nThe total amount of money (maximum) I get is " << P[0][N-1] << ".\n";
}

int maxMoney(int A[], int N) {
    int P[MAX_N][MAX_N] = {0};
    int a, b, c;
    for (int i = 0; i < N; i++) {
        for (int m = 0, n = i; n < N; m++, n++) {
            assert(m < N); assert(n < N);
            a = ((m+2 <= N-1) ? P[m+2][n] : 0);
            b = ((m+1 <= N-1 && n-1 >= 0) ? P[m+1][n-1] : 0);
            c = ((n-2 >= 0) ? P[m][n-2] : 0);
            P[m][n] = max(A[m] + min(a,b),
                          A[n] + min(b,c));
        }
    }
    printMoves(P, A, N);
    return P[0][N-1];
}

```

Further Thoughts:

Assume that your opponent is so dumb that you are able to manipulate him into choosing the coins you want him to choose. Now, what is the maximum possible amount of money you can win?

Chapter 9: Binary Search

48. Search Insert Position

Code it now: <https://oj.leetcode.com/problems/search-insert-position/>

Difficulty: Medium, Frequency: Low

Question:

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Here are few examples.

[1,3,5,6], 5 → 2

[1,3,5,6], 2 → 1

[1,3,5,6], 7 → 4

[1,3,5,6], 0 → 0

Solution:

This problem is a direct application of Binary Search, as you can spot it easily by the keywords *sorted* and *finding target*. The requirements seem complex, but let's first start with something we're already familiar with – The raw binary search algorithm.

Let's start with defining two variables, L and R representing its lowest and highest inclusive indices that are searched, which are initialized to 0 and $n - 1$ respectively.

```
int L = 0, R = A.length - 1;
while (L < R) {
    int M = (L + R) / 2;
    // TODO: Implement conditional checks.
}
```

Code 1: Getting started with a Binary Search template.

Now, the key part of the binary search – We look at the middle element, and ask: “Is the middle element smaller than the target element?” If this is true, then it means all elements from L up to M inclusive could be excluded from the search. Otherwise, the middle element is greater or equal to the target element and that means all elements from $M + 1$ up to R could be excluded.

```

int L = 0, R = A.length - 1;
while (L < R) {
    int M = (L + R) / 2;
    if (A[M] < target) {
        L = M + 1;
    } else {
        R = M;
    }
}

```

Code 2: Filling out the key part of the Binary Search algorithm.

A good thing to verify your above binary search does not stuck in an infinite loop is to test with input containing two elements, e.g., [1,3] and test with *target* = 0 and 1. Here, our binary search works properly, but if we were to define *M* as the *upper* middle, that is: $M = (L + R + 1) / 2$, then it will stuck in an infinite loop.

We've now reached the final step. When the while loop ends, *L* must be equal to *R* and it is a valid index. Obviously, if *A[L]* is equal to *target*, we return *L*. If *A[L]* is greater than *target*, that means we are inserting *target* **before** *A[L]*, so we return *L*. If *A[L]* is less than *target*, that means we insert *target* **after** *A[L]*, so we return *L* + 1.

```

public int searchInsert(int[] A, int target) {
    int L = 0, R = A.length - 1;
    while (L < R) {
        int M = (L + R) / 2;
        if (A[M] < target) {
            L = M + 1;
        } else {
            R = M;
        }
    }
    return (A[L] < target) ? L + 1 : L;
}

```


49. Find Minimum in Sorted Rotated Array

Code it now: <https://oj.leetcode.com/problems/find-minimum-in-rotated-sorted-array/>

Difficulty: Medium, Frequency: High

Question:

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

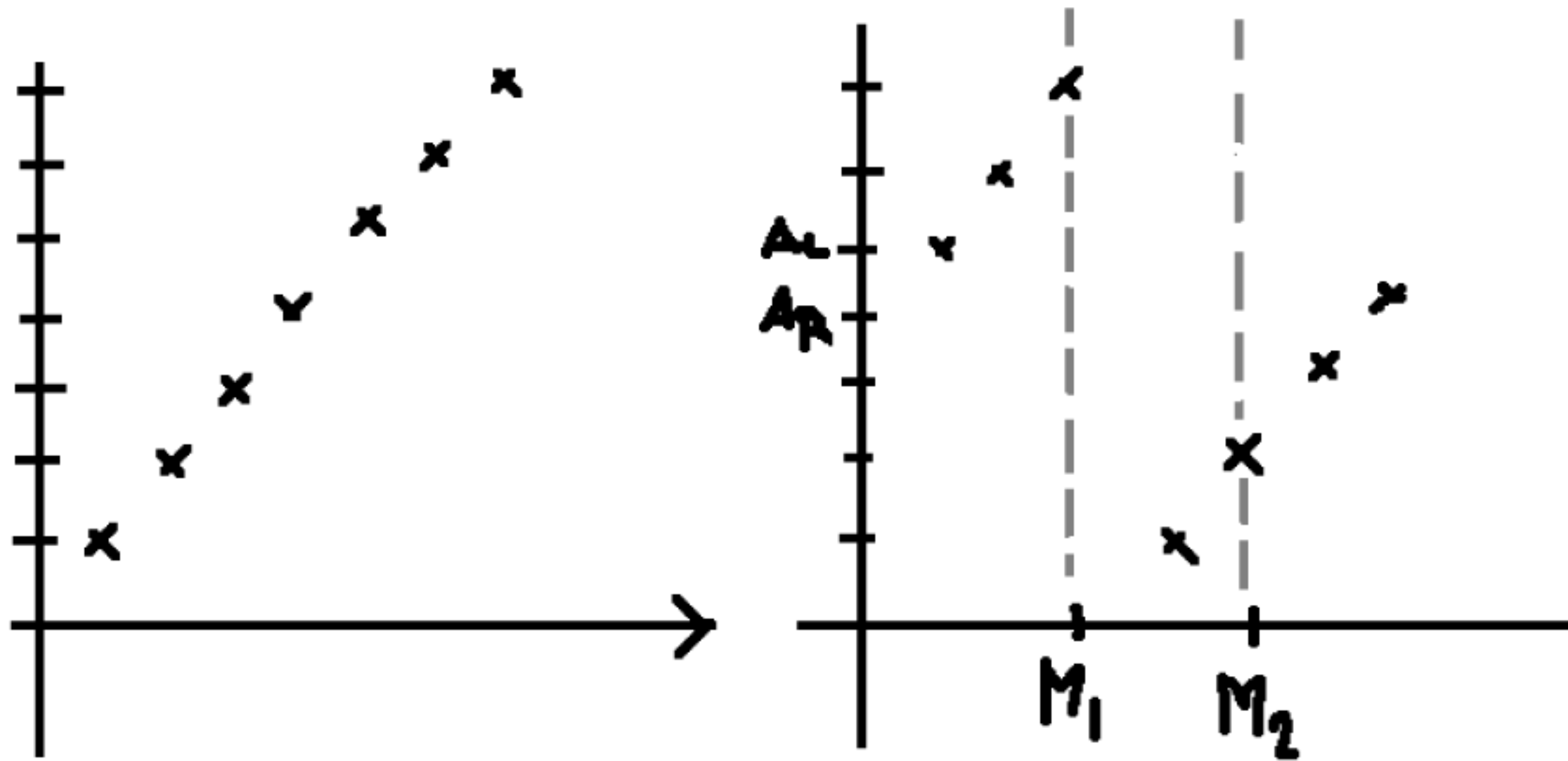
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

You may assume no duplicate exists in the array.

Solution:

The minimum is at A_i where $A_{i-1} > A_i$. Notice that if we subdivide the array into two, one will always be sorted, while the other contains the minimum.



Imagine we have an array $[1, 2, 3, 4, 5, 6, 7]$ (See graph 1) which was being rotated 3 steps to the right $[5, 6, 7, 1, 2, 3, 4]$ (See graph 2). Let's say we subdivide the array at point k to two subarrays $[A_L, A_{L+1}, \dots, A_k], [A_{k+1}, \dots, A_R]$.

If the sorted array is not rotated, then $A_L < A_R$ then we could return A_L as the minimum immediately.

Otherwise for a sorted array that was rotated at least one step, A_L must always be greater than A_R .

Let's assume we choose M_1 as the dividing point. Since $A_{M_1} > A_R$, we know that each element in $[A_L \dots A_{M_1}]$ is greater than A_R (Remember that $A_L > A_R$?). Therefore, the minimum value must locate in $[A_{M_1+1} \dots A_R]$.

On the other hand, let's assume we choose M_2 as the dividing point. Since $A_{M_2} \leq A_R$, we know that each element in $[A_{M_2+1} \dots A_R]$ is greater than A_{M_2} . Therefore, the minimum point must locate in $[A_L \dots A_{M_2}]$.

As we are discarding half of the elements at each step, the runtime complexity is $O(\log n)$.

To understand the correct terminating condition, we look at two elements. Let us choose the lower median as $M = (L + R) / 2$. Therefore, if there are two elements, it will choose A_L as the first element.

There are two cases for two elements:

$A = [1, 2]$

$B = [2, 1]$

For A, $1 < 2 \Rightarrow A_M < A_R$, and therefore it will set $R = M \Rightarrow R = 0$.

For B, $2 > 1 \Rightarrow A_M > A_R$, and therefore it will set $L = M + 1 \Rightarrow L = 1$.

Therefore, it is clear that when $L == R$, we have found the minimum element.

```
public int findMin(int[] A) {
    int L = 0, R = A.length - 1;
    while (L < R && A[L] >= A[R]) {
        int M = (L + R) / 2;
        if (A[M] > A[R]) {
            L = M + 1;
        } else {
            R = M;
        }
    }
    return A[L];
}
```

Further Thoughts:

If the rotated sorted array could contain duplicates? Is your algorithm still $O(\log n)$ in runtime complexity?

50. Find Minimum in Rotated Sorted Array II – with duplicates

Code it now: <https://oj.leetcode.com/problems/find-minimum-in-rotated-sorted-array-ii/>

Difficulty: Hard, Frequency: Medium

Question:

If the rotated sorted array could contain duplicates? Is your algorithm still $O(\log n)$ in runtime complexity?

Solution:

For case where $A_L == A_M == A_R$, the minimum could be on A_M 's left or right side (eg, [1, 1, 1, 0, 1] or [1, 0, 1, 1, 1]). In this case, we could not discard either subarrays and therefore such worst case degenerates to the order of $O(n)$.

```
public int findMin(int[] A) {
    int L = 0, R = A.length - 1;
    while (L < R && A[L] >= A[R]) {
        int M = (L + R) / 2;
        if (A[M] > A[R]) {
            L = M + 1;
        } else if (A[M] < A[L]) {
            R = M;
        } else { // A[L] == A[M] == A[R]
            L = L + 1;
        }
    }
    return A[L];
}
```