Leet Code

July Coding Challenge

E-Mail	sunil016@yahoo.com
HackerRank	https://www.hackerrank.com/atworksunil
GitHub	https://github.com/Ysunil016
Linkedin	https://www.linkedin.com/in/sunil016/

July - Week 1

Arranging Coins

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

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

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

Example 1:

```
n = 5

The coins can form the following rows:
¤
¤ ¤
¤ ¤

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

Example 2:

```
n = 8

The coins can form the following rows:
¤
¤ ¤
¤ ¤ ¤
¤ ¤

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

Coding

```java
public static int arrangeCoins(int n) {
        int Count = 0;
        int Res = 1;
        while (n - Res >= 0) {
            n -= Res;
            Res += 1;
            Count++;
        }
        return Count;
    }
```

Binary Tree Level Order Traversal II

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

For example:
Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its bottom-up level order traversal as:

```
[
  [15,7],
  [9,20],
  [3]
]
```

Coding

```
static void findBottomUpTraversal(Node root, List<List<Integer>> res) {
        Queue<Node> Q = new LinkedList<>();
        Q.add(root);
        int rC = 0;
        res.add(rC, Arrays.asList(root.Val));
        rC++;
        while (true) {
            int Counter = Q.size();
            if (Counter == 0) {
                break;
            }

            List<Integer> xRes = new ArrayList<>();
            while (Counter > 0) {
                Node pVal = Q.poll();
                if (pVal.left != null) {
                    xRes.add(pVal.left.Val);
                    Q.add(pVal.left);
                }
                if (pVal.right != null) {
                    xRes.add(pVal.right.Val);
                    Q.add(pVal.right);
                }
                Counter--;
            }
            if (xRes.size() != 0) {
                res.add(rC, xRes);
                rC++;
            }
        }
    }
```

Prison Cells After N Days

There are 8 prison cells in a row, and each cell is either occupied or vacant.

Each day, whether the cell is occupied or vacant changes according to the following rules:

- If a cell has two adjacent neighbors that are both occupied or both vacant, then the cell becomes occupied.
- Otherwise, it becomes vacant.

(Note that because the prison is a row, the first and the last cells in the row can't have two adjacent neighbors.)

We describe the current state of the prison in the following way: `cells[i] == 1` if the `i`-th cell is occupied, else `cells[i] == 0`.

Given the initial state of the prison, return the state of the prison after `N` days (and `N` such changes described above.)

Example 1:

```
Input: cells = [0,1,0,1,1,0,0,1], N = 7
Output: [0,0,1,1,0,0,0,0]
Explanation:
The following table summarizes the state of the prison on each day:
Day 0: [0, 1, 0, 1, 1, 0, 0, 1]
Day 1: [0, 1, 1, 0, 0, 0, 0, 0]
Day 2: [0, 0, 0, 0, 1, 1, 1, 0]
Day 3: [0, 1, 1, 0, 0, 1, 0, 0]
Day 4: [0, 0, 0, 0, 0, 1, 0, 0]
Day 5: [0, 1, 1, 1, 0, 1, 0, 0]
Day 6: [0, 0, 1, 0, 1, 1, 0, 0]
Day 7: [0, 0, 1, 1, 0, 0, 0, 0]
```

Example 2:

```
Input: cells = [1,0,0,1,0,0,1,0], N = 1000000000
Output: [0,0,1,1,1,1,1,0]
```

Coding

```java
public static int[] prisonAfterNDays(int[] cells, int N) {
        if (N == 0) {
            return cells;
        }
        HashSet<Integer> hSet = new HashSet<>();
        List<int[]> aStore = new ArrayList<>();
        int T = 0;
        for (T = 0; T < N; T++) {
            int prevElement = cells[0];
            cells[0] = 0;
            int calV = 0;
            int i = 1;
            for (i = 1; i < 7; i++) {
                if (prevElement == cells[i + 1]) {
                    prevElement = cells[i];
                    cells[i] = 1;
                    calV += Math.pow(2, (8 - i));
                } else {
                    prevElement = cells[i];
                    cells[i] = 0;
                }
            }
            cells[i] = 0;
            aStore.add(cells.clone());
            if (hSet.contains(calV)) {
                break;
            }

            hSet.add(calV);
        }

        if (N < 8) {
            return cells;
        }
        int Rem = N % T;
        if (Rem == 0) {
            return aStore.get(T - 1);
        }
        return aStore.get(Rem - 1);
    }
```

Ugly Number II

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

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

Example:

```
Input: n = 10
Output: 12
Explanation: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers.
```

Note:

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

Coding

```java
public static int nthUglyNumber(int n) {
        if (n == 1) {
            return 1;
        }
        List<Integer> stack = new ArrayList<>();
        stack.add(1);
        int i = 0, j = 0, k = 0;
        while (stack.size() < n) {
            stack.add(Math.min(stack.get(i) * 2,
Math.min(stack.get(j) * 3, stack.get(k) * 5)));
            if (stack.get(stack.size() - 1) == stack.get(i) * 2)
                i++;
            if (stack.get(stack.size() - 1) == stack.get(j) * 3)
                j++;
            if (stack.get(stack.size() - 1) == stack.get(k) * 5)
                k++;
        }
        return stack.get(stack.size() - 1);
    }
```

Hamming Distance

The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Given two integers x and y, calculate the Hamming distance.

Note:
0 ≤ x, y < 231.

Example:

```
Input: x = 1, y = 4

Output: 2

Explanation:
1   (0 0 0 1)
4   (0 1 0 0)
      ↑   ↑

The above arrows point to positions where the corresponding bits are different.
```

Coding

```
private static int getHammingDistance(int X, int Y) {
        int t = 33;
        int r = 0;
        while (--t != 0) {
            int _X = X & 1;
            int _Y = Y & 1;
            if (_X != _Y) {
                r++;
            }
            X = X >> 1;
            Y = Y >> 1;
        }
        return r;
    }
```

Plus One

Given a **non-empty** array of digits representing a non-negative integer, plus one to the integer.

The digits are stored such that the most significant digit is at the head of the list, and each element in the array contain a single digit.

You may assume the integer does not contain any leading zero, except the number 0 itself.

Example 1:

```
Input: [1,2,3]
Output: [1,2,4]
Explanation: The array represents the integer 123.
```

Example 2:

```
Input: [4,3,2,1]
Output: [4,3,2,2]
Explanation: The array represents the integer 4321.
```

Coding

```
private static int[] plusOne(int[] digits) {
        int carry = 1;
        for (int i = digits.length - 1; i >= 0; i--) {
            if (digits[i] + carry == 10) {
                digits[i] = 0;
            } else {
                digits[i] += carry;
                carry = 0;
                return digits;
            }
        }
        int[] nArray = new int[digits.length + 1];
        for (int i = digits.length - 1; i >= 0; i--) {
            nArray[i + 1] = digits[i];
        }
        nArray[0] = carry;

        return nArray;
    }
```

Island Perimeter

You are given a map in form of a two-dimensional integer grid where 1 represents land and 0 represents water.

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

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

Coding

```
public static int islandPerimeter(int[][] grid) {
        int parameter = 0;
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                if (grid[i][j] == 1) {

                    if (j - 1 < 0 || grid[i][j - 1] == 0) {
                        parameter += 1;
                    }
                    if (j + 1 > grid[i].length - 1 || grid[i][j + 1] == 0) {
                        parameter += 1;
                    }
                    if (i - 1 < 0 || grid[i - 1][j] == 0) {
                        parameter += 1;
                    }
                    if (i + 1 > grid.length - 1 || grid[i + 1][j] == 0) {
                        parameter += 1;
                    }

                }
            }
        }
        return parameter;
    }
```

July - Week 2

3Sum

Given an array `nums` of *n* integers, are there elements *a*, *b*, *c* in `nums` such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

Note:

The solution set must not contain duplicate triplets.

Example:

```
Given array nums = [-1, 0, 1, 2, -1, -4],

A solution set is:
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

Coding

```
public static List<List<Integer>> threeSum(int[] num) {
        List<List<Integer>> res = new ArrayList<>();
        Arrays.sort(num);
        for (int i = 0; i < num.length - 2; i++) {
            if (i == 0 || (i > 0 && num[i] != num[i - 1])) {
                int lo = i + 1, hi = num.length - 1, sum = 0 - num[i];
                while (lo < hi) {
                    if (num[lo] + num[hi] == sum) {
                        res.add(Arrays.asList(num[i], num[lo], num[hi]));
                        while (lo < hi && num[lo] == num[lo + 1])
                            lo++;
                        while (lo < hi && num[hi] == num[hi - 1])
                            hi--;
                        lo++;
                        hi--;
                    } else if (num[lo] + num[hi] < sum)
                        lo++;
                    else
                        hi--;
                }
            }
        }
        return res;
    }
```

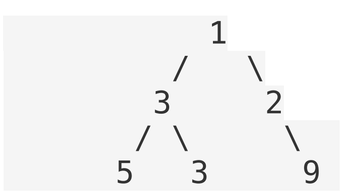
Maximum Width of Binary Tree

Given a binary tree, write a function to get the maximum width of the given tree. The width of a tree is the maximum width among all levels. The binary tree has the same structure as a **full binary tree**, but some nodes are null.

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

Example 1:

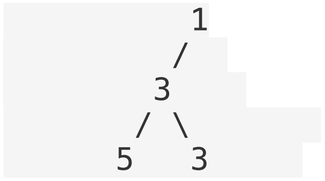
Input:



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

Example 2:

Input:



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

Coding

```
    static int MaxWidth;
    static HashMap<Integer, Integer> hash;

    public static int widthOfBinaryTree(TreeNode root) {
        hash = new HashMap<>();
        MaxWidth = 0;
        getWidth(root, 0, 0);
        return MaxWidth;
    }

    private static void getWidth(TreeNode root, int depth, int
position) {
        if (root == null) {
            return;
        }
        hash.computeIfAbsent(depth, x -> position);
        MaxWidth = Math.max(MaxWidth, position -
hash.get(depth) + 1);
        getWidth(root.left, depth + 1, position * 2);
        getWidth(root.right, depth + 1, (position * 2) + 1);
    }
```

Flatten a Multilevel Doubly Linked List

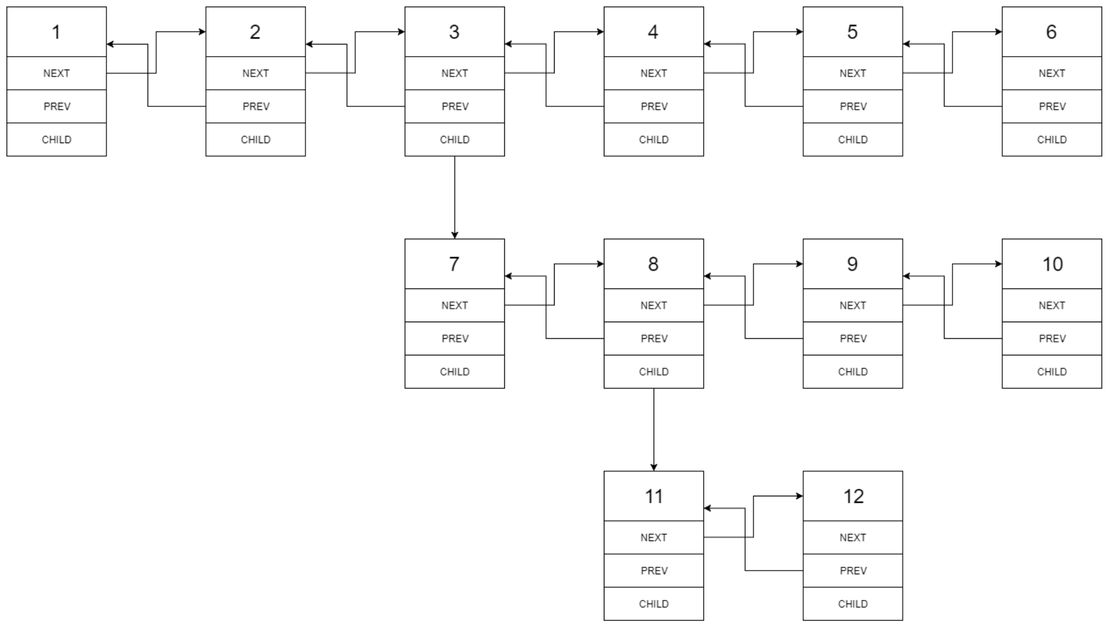
You are given a doubly linked list which in addition to the next and previous pointers, it could have a child pointer, which may or may not point to a separate doubly linked list. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure, as shown in the example below.

Flatten the list so that all the nodes appear in a single-level, doubly linked list. You are given the head of the first level of the list.

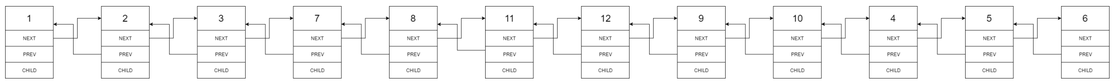
Example 1:

Input: head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]
Output: [1,2,3,7,8,11,12,9,10,4,5,6]
Explanation:

The multilevel linked list in the input is as follows:



After flattening the multilevel linked list it becomes:



Example 2:

Input: head = [1,2,null,3]
Output: [1,3,2]

Coding

```
public static Node flatten(Node head) {
    if (head == null) {
        return head;
    }
    makeFlatten(head, new Stack<Node>());
    return head;
}

private static void makeFlatten(Node head, Stack<Node> store) {
    if (head.next == null) {
        if (head.child != null) {
            head.next = head.child;
            head.child.prev = head;
            head.child = null;
            makeFlatten(head.next, store);
        }
        if (!store.isEmpty()) {
            Node found = store.pop();
            head.next = found;
            found.prev = head;
            makeFlatten(head.next, store);
        }
        return;
    }

    if (head.child != null) {
        store.push(head.next);
        head.next = head.child;
        head.child.prev = head;
        head.child = null;
    }
    makeFlatten(head.next, store);
}
```

Subsets

Given a set of **distinct** integers, *nums*, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

Example:

```
Input: nums = [1,2,3]
Output:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

Coding

```
private static List<List<Integer>> getSubset(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        generateSubset(0, nums, new ArrayList<>(), res);
        return res;
    }

    private static void generateSubset(int index, int[] nums,
List<Integer> cur, List<List<Integer>> subset) {
        subset.add(new ArrayList<>(cur));
        for (int i = index; i < nums.length; i++) {
            cur.add(nums[i]);
            generateSubset(i + 1, nums, cur, subset);
            cur.remove(cur.size() - 1);
        }

    }
```

Reverse Bits

Reverse bits of a given 32 bits unsigned integer.

Example 1:

Input: 00000010100101000001111010011100
Output: 00111001011110000010100101000000
Explanation: The input binary string **00000010100101000001111010011100** represents the unsigned integer 43261596, so return 964176192 which its binary representation is **00111001011110000010100101000000.**

Example 2:

Input: 11111111111111111111111111111101
Output: 10111111111111111111111111111111
Explanation: The input binary string **11111111111111111111111111111101** represents the unsigned integer 4294967293, so return 3221225471 which its binary representation is **10111111111111111111111111111111.**

Note:

- Note that in some languages such as Java, there is no unsigned integer type. In this case, both input and output will be given as signed integer type and should not affect your implementation, as the internal binary representation of the integer is the same whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in **Example 2** above the input represents the signed integer `-3` and the output represents the signed integer `-1073741825`.

Follow up:

If this function is called many times, how would you optimize it?

Coding

```java
public static int reverseBits(int n) {
        int res = 0;
        int t = 32;
        while (--t > 0) {
            res |= (n & 1);
            n = n >> 1;
            res = res << 1;
        }
        res |= (n & 1);
        return res;
    }
```

Same Tree

Given two binary trees, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical and the nodes have the same value.

Example 1:

```
Input:     1          1
          / \        / \
         2   3      2   3

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

Output: true
```

Example 2:

```
Input:     1          1
          /            \
         2              2

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

Output: false
```

Example 3:

```
Input:     1          1
          / \        / \
         2   1      1   2

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

Output: false
```

Coding

```java
public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {
        }

        TreeNode(int val) {
            this.val = val;
        }

        TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
        }
    }

    private static boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == null && q == null) {
            return true;
        }
        if (p == null || q == null)
            return false;

        if (p.val != q.val) {
            return false;
        }
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
```

Angle Between Hands of a Clock

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

Example 1:

```
Input: hour = 12, minutes = 30
Output: 165
```

Example 2:

```
Input: hour = 3, minutes = 30
Output: 75
```

Coding

```
public static double angleClock(int hour, int minutes) {
        if (hour == 12)
            hour = 0;
        double angleDifference = Math.abs(hour * 30 + minutes *
0.5 - minutes * 6);
        return Math.min(angleDifference, 360 - angleDifference);
    }
```

July - Week 3

Reverse Words in a String

Given an input string, reverse the string word by word.

Example 1:

```
Input: "the sky is blue"
Output: "blue is sky the"
```

Example 2:

```
Input: "  hello world!  "
Output: "world! hello"
Explanation: Your reversed string should not contain leading or trailing spaces.
```

Example 3:

```
Input: "a good   example"
Output: "example good a"
Explanation: You need to reduce multiple spaces between two words to a single space in the reversed string.
```

Note:

- A word is defined as a sequence of non-space characters.
- Input string may contain leading or trailing spaces. However, your reversed string should not contain leading or trailing spaces.
- You need to reduce multiple spaces between two words to a single space in the reversed string.

Follow up:

For C programmers, try to solve it *in-place* in *O*(1) extra space.

Coding

```java
public static String reverseWords(String s) {
        if (s == null) {
            return new String();
        }
        s = s.trim();
        String[] list = s.split(" ");
        StringBuilder strBuilder = new StringBuilder();
        for (int i = list.length - 1; i > 0; i--) {
            System.out.println(list[i].length());
            if (list[i].length() != 0) {
                strBuilder.append(list[i].trim());
                strBuilder.append(" ");
            }
        }
        strBuilder.append(list[0]);
        return new String(strBuilder);
    }
```

Pow(x, n)

Implement pow(*x*, *n*), which calculates *x* raised to the power *n* (xn).

Example 1:

```
Input: 2.00000, 10
Output: 1024.00000
```

Example 2:

```
Input: 2.10000, 3
Output: 9.26100
```

Example 3:

```
Input: 2.00000, -2
Output: 0.25000
Explanation: 2-2 = 1/22 = 1/4 = 0.25
```

Coding

```
public static double myPow(double x, int n) {
        if (n < 0)
                return getPowerNegative(x, n);
        else
                return getPowerPositive(x, n);
    }

    private static double getPowerNegative(double x, inpublic static double myPow(double x, int n) {
        if (n < 0)
                return getPowerNegative(x, n);
        else
                return getPowerPositive(x, n);
    }

    private static double getPowerNegative(double x, int n) {
        if (n == 0) {
                return 1;
        }
        double XXX = getPowerNegative(x, n / 2);
        if (n % 2 == 0) {
                return XXX * XXX;
        } else {
                return 1 / x * XXX * XXX;
        }
    }

    private static double getPowerPositive(double x, int n) {
        if (n == 0) {
                return 1;
        }
        double XXX = getPowerPositive(x, n / 2);
        if (n % 2 == 0) {
                return XXX * XXX;
        } else {
                return x * XXX * XXX;
        }
    }
t n) {
        if (n == 0) {
                return 1;
        }
        double XXX = getPowerNegative(x, n / 2);
        if (n % 2 == 0) {
                return XXX * XXX;
        } else {
                return 1 / x * XXX * XXX;
        }
    }

    private static double getPowerPositive(double x, int n) {
        if (n == 0) {
                return 1;
        }
        double XXX = getPowerPositive(x, n / 2);
        if (n % 2 == 0) {
                return XXX * XXX;
        } else {
                return x * XXX * XXX;
        }
    }
```

Top K Frequent Elements

Given a non-empty array of integers, return the ***k*** most frequent elements.

Example 1:

```
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]
```

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

Note:

- You may assume *k* is always valid, 1 ≤ *k* ≤ number of unique elements.
- Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size.
- It's guaranteed that the answer is unique, in other words the set of the top k frequent elements is unique.
- You can return the answer in any order.

Coding

```
public static int[] topKFrequent(int[] nums, int k) {
        HashMap<Integer, Integer> hashMap = new HashMap<>();
        for (int number : nums) {
            hashMap.put(number, hashMap.getOrDefault(number,
0) + 1);
        }
        Queue<Integer> queue = new PriorityQueue<>((a, b) ->
hashMap.get(a) - hashMap.get(b));
        for (int n : hashMap.keySet()) {
            queue.add(n);
            if (queue.size() > k) {
                queue.poll();
            }
        }
        int[] result = new int[k];
        int i = 0;
        for (int n : queue) {
            result[i++] = n;
        }
        return result;
    }
```

Course Schedule II

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

Example 1:

```
Input: 2, [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished
             course 0. So the correct course order
is [0,1] .
```

Example 2:

```
Input: 4, [[1,0],[2,0],[3,1],[3,2]]
Output: [0,1,2,3] or [0,2,1,3]
Explanation: There are a total of 4 courses to take.
To take course 3 you should have finished both
             courses 1 and 2. Both courses 1 and 2
should be taken after you finished course 0.
             So one correct course order is
[0,1,2,3]. Another correct ordering is [0,2,1,3] .
```

Coding

```java
    public static int[] findOrder(int numCourses, int[][] prerequisites) {
        LinkedList<Integer>[] adj = new LinkedList[numCourses];
        for (int i = 0; i < numCourses; i++) {
            adj[i] = new LinkedList<Integer>();
        }
        for (int[] courses : prerequisites) {
            adj[courses[1]].add(courses[0]);
        }
        LinkedList<Integer> stack = new LinkedList<>();
        int[] visited = new int[numCourses];
        for (int i = 0; i < numCourses; i++) {
            if (visited[i] == 0 && dfs(i, adj, stack, visited)) {
                return new int[0];
            }
        }
        Collections.reverse(stack);
        int[] result = new int[stack.size()];
        int i = 0;
        for (int stk : stack) {
            result[i++] = stk;
        }
        return result;
    }

    private static boolean dfs(int index, LinkedList<Integer>[] adj, LinkedList<Integer> stackStore,
int[] isVisited) {
        isVisited[index] = 1;

        for (int neighbours : adj[index]) {
            if (isVisited[neighbours] == 1)
                return true;
            if (isVisited[neighbours] == 0 && dfs(neighbours, adj, stackStore, isVisited))
                return true;
        }
        isVisited[index] = 2;
        stackStore.add(index);
        return false;
    }
```

Add Binary

Given two binary strings, return their sum (also a binary string).

The input strings are both **non-empty** and contains only characters `1` or `0`.

Example 1:

```
Input: a = "11", b = "1"
Output: "100"
```

Example 2:

```
Input: a = "1010", b = "1011"
Output: "10101"
```

Constraints:

- Each string consists only of `'0'` or `'1'` characters.
- `1 <= a.length, b.length <= 10^4`
- Each string is either `"0"` or doesn't contain any leading zero.

Coding

```
static String addBinary(String a, String b) {
        char[] aArray = a.toCharArray(); char[] bArray = b.toCharArray();
        int counter = 0;
        int carry = 0;
        StringBuilder strBuilder = new StringBuilder();
        while (counter < aArray.length && counter < bArray.length) {
                char x = aArray[aArray.length - 1 - counter];
                char y = bArray[bArray.length - 1 - counter];
                if (carry == 1) {
                        if (x == '1' && y == '1')
                                strBuilder.append("1");
                        else if (x == '0' && y == '0') {
                                strBuilder.append("1");
                                carry = 0;
                        } else
                                strBuilder.append("0");
                } else {
                        if (x == '1' && y == '1') {
                                strBuilder.append("0");
                                carry = 1;
                        } else if (x == '0' && y == '0')
                                strBuilder.append("0");
                        else
                                strBuilder.append("1");
                }
                counter++;
        }
        while (counter < aArray.length) {
                char x = aArray[aArray.length - 1 - counter];
                if (carry == 1) {
                        if (x == '0') {
                                strBuilder.append("1");
                                carry = 0;
                        } else {
                                strBuilder.append("0");
                                carry = 1;
                        }
                } else
                        strBuilder.append(x);
                counter++;
        }
        while (counter < bArray.length) {
                char y = bArray[bArray.length - 1 - counter];
                if (carry == 1) {
                        if (y == '0') {
                                strBuilder.append("1");
                                carry = 0;
                        } else {
                                strBuilder.append("0");
                                carry = 1;
                        }
                } else
                        strBuilder.append(y);
                counter++;
        }
        if (carry == 1)
                strBuilder.append("1");
        strBuilder.reverse();
        return new String(strBuilder);
    }
```

Remove Linked List Elements

Remove all elements from a linked list of integers that have value ***val***.

Example:

```
Input:  1->2->6->3->4->5->6, val = 6
Output: 1->2->3->4->5
```

Coding

```
public static ListNode removeElements(ListNode head, int val) {
        ListNode result = head;
        if (head == null)
            return null;
        if (head.val == val) {
            result = head.next;
            head = head.next;
        }
        ListNode PrevNode = null;
        while (head != null) {
            if (head.val == val) {
                if (PrevNode == null) {
                    result = head.next;
                } else {
                    PrevNode.next = head.next;
                }
            } else
                PrevNode = head;
            head = head.next;
        }

        return result;
    }
```

Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example:

```
board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]

Given word = "ABCCED", return true.
Given word = "SEE", return true.
Given word = "ABCB", return false.
```

Coding

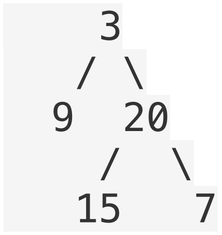
```
public static boolean exist(char[][] board, String word) {
        char x = word.charAt(0);
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (board[i][j] == x) {
                    if (dfs(board, i, j, word, 0)) {
                        return true;
                    }
                }
            }
        }
        return false;
    }

    public static boolean dfs(char[][] board, int i, int j, String word, int sIndex) {
        if (i > board.length - 1 || j > board[0].length - 1 || i < 0 || j < 0 || sIndex > word.length() - 1) {
            sIndex--;
            return false;
        }
        char foundChar = board[i][j];
        if (foundChar != word.charAt(sIndex) || foundChar == '1') {
            sIndex--;
            return false;
        }

        if (sIndex == word.length() - 1)
            return true;

        board[i][j] = '1';
        boolean result = dfs(board, i, j + 1, word, sIndex + 1) || dfs(board, i + 1, j, word, sIndex + 1)
                || dfs(board, i, j - 1, word, sIndex + 1) || dfs(board, i - 1, j, word, sIndex + 1);

        board[i][j] = foundChar;

        return result;

    }
```

July - Week 4

Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:
Given binary tree `[3,9,20,null,null,15,7]`,



return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

Coding

```java
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        boolean isEven = true;
        Queue<TreeNode> queue = new LinkedList<>();
        Deque<TreeNode> qStore = new LinkedList<>();

        queue.add(root);
        while (true) {
            int nodeCount = queue.size();
            if (nodeCount == 0)
                break;
            while (nodeCount > 0) {
                TreeNode currNode = queue.poll();
                qStore.add(currNode);
                if (currNode.left != null)
                    queue.add(currNode.left);
                if (currNode.right != null)
                    queue.add(currNode.right);
                nodeCount--;
            }
            ArrayList<Integer> levelOrderStore = new ArrayList<>();
            while (!qStore.isEmpty()) {
                if (isEven) {
                    levelOrderStore.add(qStore.removeFirst().val);
                } else {
                    levelOrderStore.add(qStore.removeLast().val);
                }
            }
            res.add(levelOrderStore);
            if (isEven) {
                isEven = false;
            } else {
                isEven = true;
            }
        }
        return res;
    }
```

Single Number III

Given an array of numbers `nums`, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

Example:

```
Input:  [1,2,1,3,2,5]
Output: [3,5]
```

Note:

1. The order of the result is not important. So in the above example, `[5, 3]` is also correct.
2. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

Coding

```
public static int[] singleNumber(int[] nums) {
        int[] result = { 0, 0 };

        int difference = 0;
        for (int num : nums) {
            difference ^= num;
        }
        difference &= -difference;

        for (int num : nums) {
            if ((num & difference) == 0) {
                result[0] ^= num;
            } else {
                result[1] ^= num;
            }

        }

        return result;
    }
```

All Paths From Source to Target

Given a directed, acyclic graph of `N` nodes. Find all possible paths from node `0` to node `N-1`, and return them in any order.

The graph is given as follows: the nodes are 0, 1, ..., graph.length - 1. graph[i] is a list of all nodes j for which the edge (i, j) exists.

Example:
Input: `[[1,2], [3], [3], []]`
Output: `[[0,1,3],[0,2,3]]`
Explanation: The graph looks like this:

```
0--->1
|    |
v    v
2--->3
```

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

Note:

- The number of nodes in the graph will be in the range `[2, 15]`.
- You can print different paths in any order, but you should keep the order of nodes inside one path.

Coding

```
static List<List<Integer>> resultList = new LinkedList<>();

    public static List<List<Integer>> allPathsSourceTarget(int[][] graph) {
        List<Integer> ar = new ArrayList<>();
        dfsSearch(graph, 0, graph.length - 1, ar);
        return resultList;
    }

    private static void dfsSearch(int[][] graph, int currentIndex, int
finalIndex, List<Integer> currentList) {
        if (currentIndex == finalIndex) {
            @SuppressWarnings("unchecked")
            ArrayList<Integer> foundPath = (ArrayList<Integer>)
((ArrayList<Integer>) currentList).clone();
            resultList.add(foundPath);
            return;
        }
        if (currentIndex == 0) {
            currentList.add(0);
        }

        for (int num : graph[currentIndex]) {
            currentList.add(num);
            dfsSearch(graph, num, finalIndex, currentList);
            currentList.remove(currentList.size() - 1);
        }
    }
```

Find Minimum in Rotated Sorted Array II

Suppose an array sorted in ascending order 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.

The array may contain duplicates.

Example 1:

```
Input: [1,3,5]
Output: 1
```

Example 2:

```
Input: [2,2,2,0,1]
Output: 0
```

Note:

- This is a follow up problem to Find Minimum in Rotated Sorted Array.
- Would allow duplicates affect the run-time complexity? How and why?

Coding

```
static int findMinOfAll(int[] rArr) {
        return bst(rArr, 0, rArr.length - 1);
    }

    static int bst(int[] rArr, int start, int end) {
        while (start < end) {
            int middleElement = start + (end - start) / 2;
            if (rArr[middleElement] == rArr[end]) {
                end--;
            } else if (rArr[middleElement] > rArr[end]) {
                start = middleElement + 1;
            } else {
                end = middleElement;
            }
        }
        return rArr[end];
    }
```

Add Digits

Given a non-negative integer `num`, repeatedly add all its digits until the result has only one digit.

Example:

```
Input: 38
Output: 2
Explanation: The process is like: 3 + 8 = 11,
1 + 1 = 2.
             Since 2 has only one digit,
return it.
```

Follow up:

Could you do it without any loop/recursion in O(1) runtime?

Coding

```
public static int addDigits(int num) {
        while (num > 9) {
            int Sum = 0;
            while (num != 0) {
                Sum += num % 10;
                num = num / 10;
            }
            num = Sum;
        }
        return num;
    }

int addDigitsOptmized(int num) {
        if(num==0) return 0;

        return (num %9 !=0)? num%9 :9;
    }
```

Construct Binary Tree from Inorder and Postorder Traversal

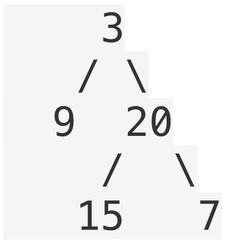
Given inorder and postorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

For example, given

```
inorder = [9,3,15,20,7]
postorder = [9,15,7,20,3]
```

Return the following binary tree:



Coding

```
static TreeNode buildTree(int[] inOrder, int[] postOrder, int n) {
    Index iIndex = new Index();
    iIndex.index = n - 1;
    return buildUtils(inOrder, postOrder, 0, n - 1, iIndex);
}

static class Index {
    int index;
}

static TreeNode buildUtils(int[] inOrder, int[] postOrder, int inStart, int inEnd, Index pIndex) {
    if (inStart > inEnd)
        return null;

    TreeNode tNode = new TreeNode(postOrder[pIndex.index]);
    (pIndex.index)--;

    if (inStart == inEnd)
        return tNode;

    int iIndex = searchIndex(inOrder, inStart, inEnd, tNode.val);

    tNode.right = buildUtils(inOrder, postOrder, iIndex + 1, inEnd, pIndex);
    tNode.left = buildUtils(inOrder, postOrder, inStart, iIndex - 1, pIndex);

    return tNode;
}

static int searchIndex(int[] inOrder, int inStart, int inEnd, int val) {
    for (int i = inStart; i <= inEnd; i++) {
        if (inOrder[i] == val)
            return i;
    }
    return -1;
}
```

Task Scheduler

You are given a char array representing tasks CPU need to do. It contains capital letters A to Z where each letter represents a different task. Tasks could be done without the original order of the array. Each task is done in one unit of time. For each unit of time, the CPU could complete either one task or just be idle.

However, there is a non-negative integer `n` that represents the cooldown period between two **same tasks** (the same letter in the array), that is that there must be at least `n` units of time between any two same tasks.

You need to return the **least** number of units of times that the CPU will take to finish all the given tasks.

Example 1:

```
Input: tasks = ["A","A","A","B","B","B"], n = 2
Output: 8
Explanation:
A -> B -> idle -> A -> B -> idle -> A -> B
There is at least 2 units of time between any two same tasks.
```

Example 2:

```
Input: tasks = ["A","A","A","B","B","B"], n = 0
Output: 6
Explanation: On this case any permutation of size 6 would work since n = 0.
["A","A","A","B","B","B"]
["A","B","A","B","A","B"]
["B","B","B","A","A","A"]
...
And so on.
```

Example 3:

```
Input: tasks = ["A","A","A","A","A","A","B","C","D","E","F","G"], n = 2
Output: 16
Explanation:
One possible solution is
A -> B -> C -> A -> D -> E -> A -> F -> G -> A -> idle -> idle -> A -> idle -> idle -> A
```

Coding

```java
public static int leastInterval(char[] tasks, int n) {
        char[] task_frequncy = new char[26];
        for (char x : tasks) {
            task_frequncy[x - 'A']++;
        }
        Arrays.sort(task_frequncy);
        int maxFreq = task_frequncy[25];
        int idleSlots = maxFreq * n;

        for (int i = 24; i >= 0; i--) {
            idleSlots -= Math.min(maxFreq, task_frequncy[i]);
        }
        return (idleSlots > 0) ? idleSlots + tasks.length :
tasks.length;

    }
```

July - Week 5

Best Time to Buy and Sell Stock with Cooldown

Say you have an array for which the *i*th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

- You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
- After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

```
Input: [1,2,3,0,2]
Output: 3
Explanation: transactions = [buy, sell, cooldown, buy, sell]
```

Coding

```
static private int maxProfit(int[] prices) {
        HashMap<String, Integer> hashStore = new HashMap<>();
        return profit(prices, false, 0, hashStore);
    }

    static private int profit(int[] prices, boolean buyOrSell, int runningIndex,
HashMap<String, Integer> hm) {
        if (runningIndex >= prices.length)
            return 0;

        String key = runningIndex + "Codigo" + String.valueOf(buyOrSell);
        if (hm.containsKey(key)) {
            return hm.get(key);
        }

        int result = 0;
        if (!buyOrSell) {
            int buy = profit(prices, true, runningIndex + 1, hm) - prices[runningIndex];
            int noBuy = profit(prices, false, runningIndex + 1, hm);
            result = Math.max(buy, noBuy);
        } else {
            int sell = profit(prices, false, runningIndex + 2, hm) + prices[runningIndex];
            int noSell = profit(prices, true, runningIndex + 1, hm);
            result = Math.max(sell, noSell);
        }

        hm.put(key, result);

        return result;
    }
```

Word Break II

Given a **non-empty** string *s* and a dictionary *wordDict* containing a list of **non-empty** words, add spaces in *s* to construct a sentence where each word is a valid dictionary word. Return all such possible sentences.

Note:

- The same word in the dictionary may be reused multiple times in the segmentation.
- You may assume the dictionary does not contain duplicate words.

Example 1:

```
Input:
s = "catsanddog"
wordDict = ["cat", "cats", "and", "sand", "dog"]
Output:
[
  "cats and dog",
  "cat sand dog"
]
```

Example 2:

```
Input:
s = "pineapplepenapple"
wordDict = ["apple", "pen", "applepen", "pine",
"pineapple"]
Output:
[
  "pine apple pen apple",
  "pineapple pen apple",
  "pine applepen apple"
]
Explanation: Note that you are allowed to reuse a
dictionary word.
```

Coding

```
private static List<String> wordBreakHelper(String s, List<String> wordDict,
            HashMap<String, List<String>> memorize) {
        if (memorize.containsKey(s))
            return memorize.get(s);

        List<String> result = new ArrayList<String>();

        if (s.length() == 0) {
            result.add("");
            return result;
        }

        for (String word : wordDict) {
            if (s.startsWith(word)) {
                String sub = s.substring(word.length());
                List<String> subString = wordBreakHelper(sub, wordDict,
memorize);

                for (String subStr : subString) {
                    String spaceOptional = subStr.isEmpty() ? "" : " ";
                    result.add(word + spaceOptional + subStr);
                }
            }
        }

        memorize.put(s, result);

        return result;
    }
```

Climbing Stairs

You are climbing a stair case. 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?

Example 1:

```
Input: 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

Constraints:

- `1 <= n <= 45`

Coding

```java
public static int climbStairs(int n) {
        HashMap<Integer, Integer> hashMap = new HashMap<>();
        return stair(n, 0, 0, hashMap);
    }

    private static int stair(int n, int pos, int path, HashMap<Integer, Integer> hashMap) {
        if (hashMap.containsKey(pos))
            return hashMap.get(pos);
        if (pos > n)
            return 0;
        if (pos == n)
            return 1;
        int result = stair(n, pos + 1, 1, hashMap) + stair(n, pos + 2, path + 2, hashMap);
        hashMap.putIfAbsent(pos, result);
        return result;
    }
```