Leet Code

April 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/

April - Week 1

Single Number

Given a **non-empty** array of integers, every element appears *twice* except for one. Find that single one.

Note:

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

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

Code

```java
public static int singleNumberBits(int[] nums) {
    int x = 0;
    for (int num : nums)
        x ^= num;
    return x;
}
```

Happy Number

Write an algorithm to determine if a number `n` is "happy".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1. Those numbers for which this process **ends in 1** are happy numbers.

Return True if `n` is a happy number, and False if not.

Input: 19
Output: true
Explanation:
12 + 92 = 82
82 + 22 = 68
62 + 82 = 100
12 + 02 + 02 = 1

Code

```
static int firstSum = Integer.MAX_VALUE;
static boolean happyNumber(int n) {
        int temp = n;
        int sum = 0;
        while (temp != 0) {
                sum += Math.pow(temp % 10, 2);
                temp /= 10;
        }
        if (sum == 4)
                return false;
        if (sum < firstSum)
                firstSum = sum;
        if (sum == 1)
                return true;
        else
                return happyNumber(sum);
}
```

Maximum Subarray

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example:

`Input:` `[-2,1,-3,4,-1,2,1,-5,4],`

`Output:` `6`

`Explanation:`
`[4,-1,2,1] has the largest sum = 6.`

Follow up:

If you have figured out the O(*n*) solution, try coding another solution using the divide and conquer approach, which is more subtle.

Code

```java
static int findMaxSubArray(int[] arr) {
    int sum = 0;
    int result = Integer.MIN_VALUE;
    for (int num : arr) {
        sum += num;
        result = Math.max(sum, result);
        if (sum < 0)
            sum = 0;
    }
    return (result == Integer.MIN_VALUE) ? -1 : result;
}
```

Move Zeroes

Given an array `nums`, write a function to move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

Example:

```
Input: [0,1,0,3,12]
Output: [1,3,12,0,0]
```

Note:

1. You must do this **in-place** without making a copy of the array.
2. Minimize the total number of operations.

Code

```
public static void moveZeroes(int[] nums) {
    int I = 0;
    int J = 1;

    for (int Z = 0; Z < nums.length-1; Z++) {
        if (nums[J] != 0) {
            if (nums[I] == 0) {
                nums[I] = nums[J];
                nums[J] = 0;
                I++;
            }
        }
        J++;
    }
}
```

Best Time to Buy and Sell Stock II

Say you have an array `prices` 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 (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: [7,1,5,3,6,4]
Output: 7
Explanation:
Buy on day 2 (price = 1) and sell on day 3
(price = 5), profit = 5-1 = 4.
Then buy on day 4 (price = 3) and sell on
day 5 (price = 6), profit = 6-3 = 3.
```

Code

```java
static int findMax(int[] arr) {
    if (arr.length == 0 || arr == null) {
        return 0;
    }
    int profit = 0;
    for (int i = 0; i < arr.length - 1; i++) {
        if (arr[i + 1] > arr[i]) {
            profit += arr[i + 1] - arr[i];
        }
    }
    return profit;
}
```

Group Anagrams

Given an array of strings, group anagrams together.

Example:

```
Input: ["eat", "tea", "tan", "ate", "nat", "bat"],

Output:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

Note:

- All inputs will be in lowercase.
- The order of your output does not matter.

Code

```
  private static List<List<String>> getAnagramList(String[] str) {
        List<List<String>> anagramGroup = new ArrayList<>();
        HashMap<String, ArrayList<String>> hashStore = new
HashMap<String, ArrayList<String>>();
        for (String current : str) {
            char[] characters = current.toCharArray();
            Arrays.sort(characters);
            String sortedString = new String(characters);
            if (!hashStore.containsKey(sortedString)) {
                ArrayList<String> a = new ArrayList<String>() {
                                {
                        add(current);
                    }
                };
                hashStore.put(sortedString, a);
            } else {
                ArrayList<String> a = hashStore.get(sortedString);
                a.add(current);
                hashStore.put(sortedString, a);
            }
        }

        for (Map.Entry<String, ArrayList<String>> val :
hashStore.entrySet()) {
            anagramGroup.add(val.getValue());
        }

        return anagramGroup;
```

Counting Elements

Given an integer array arr, count element x such that x + 1 is also in arr. If there're duplicates in arr, count them separately.

Example :

Input: arr = [1,2,3]

Output: 2

Explanation: 1 and 2 are counted cause 2 and 3 are in arr.

Code

```java
public static int countElements(int[] arr) {
        int count = 0;
        HashSet<Integer> uStore = new HashSet<Integer>();
        for (int a : arr)
            uStore.add(a);

        for (int a : arr) {
            if (uStore.contains(a + 1)) {
                count++;
            }
        }

        return count;
    }
```

April - Week 2

Middle of the Linked List

Given a non-empty, singly linked list with head node head, return a middle node of linked list.
If there are two middle nodes, return the second middle node.

Example 1:
Input: [1,2,3,4,5]
Output: Node 3 from this list (Serialization: [3,4,5])

The returned node has value 3. (The judge's serialization of this node is [3,4,5]).
Note that we returned a ListNode object ans, such that: ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, and ans.next.next.next = NULL.

Code

```
public ListNode middleNode(ListNode head) {
    ListNode slow = head;
    ListNode fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}
```

Backspace String Compare

Given two strings S and T, return if they are equal when both are typed into empty text editors. # means a backspace character.

Note that after backspacing an empty text, the text will continue empty.

Example 1:

Input: S = "ab#c", T = "ad#c"
Output: true
Explanation: Both S and T become "ac".

Code

```
public static boolean backspaceCompare(String S, String T) {
        Stack<Character> charStoreS = new Stack<>();
        Stack<Character> charStoreT = new Stack<>();
        char[] S_Array = S.toCharArray();
        char[] T_Array = T.toCharArray();

        for (char key : S_Array) {
            if (key == '#') {
                if (!charStoreS.isEmpty())
                    charStoreS.pop();
            } else {
                charStoreS.push(key);
            }
        }
        for (char key : T_Array) {
            if (key == '#') {
                if (!charStoreT.isEmpty())
                    charStoreT.pop();
            } else {
                charStoreT.push(key);
            }
        }
        // Comparing Both the Stacks
        while (!charStoreS.isEmpty() && !charStoreT.isEmpty()) {
            if (!(charStoreS.pop() == charStoreT.pop()))
                return false;
        }

        if (!charStoreS.isEmpty() || !charStoreT.isEmpty())
            return false;

        return true;
    }
```

Min Stack

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.

Example 1:

Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();   // return 0
minStack.getMin(); // return -2

Constraints:

- Methods pop, top and getMin operations will always be called on **non-empty** stacks.

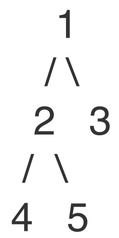
Code

```java
class MinStackSolution {

    Stack<Integer> stack;
    Stack<Integer> minStack;

    public MinStackSolution() {
        stack = new Stack<Integer>();
        minStack = new Stack<Integer>();
    }

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

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

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

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

Diameter of Binary Tree

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the **longest** path between any two nodes in a tree. This path may or may not pass through the root.

Example:
Given a binary tree



Return **3**, which is the length of the path [4,2,1,3] or [5,2,1,3].

Note: The length of path between two nodes is represented by the number of edges between them.

Code

```
    private static int longestPath = 0;

    public static int diameterOfBinaryTree(TreeNode root) {
        longestPath(root);
        return longestPath;
    }

    // Since Height of a Node is actually its longest path... so
collaborating these two will give out the result.
    private static int longestPath(TreeNode root) {
        if (root == null)
            return 0;

        int leftHeight = longestPath(root.left);
        int rightHeight = longestPath(root.right);
        longestPath = Math.max(longestPath, (leftHeight +
rightHeight));
        return Math.max(leftHeight, rightHeight) + 1;
    }
```

Last Stone Weight

We have a collection of stones, each stone has a positive integer weight.

Each turn, we choose the two **heaviest** stones and smash them together. Suppose the stones have weights x and y with x <= y. The result of this smash is:

- If x == y, both stones are totally destroyed;
- If x != y, the stone of weight x is totally destroyed, and the stone of weight y has new weight y-x.

At the end, there is at most 1 stone left. Return the weight of this stone (or 0 if there are no stones left.)

Example 1:

Input: [2,7,4,1,8,1]
Output: 1
Explanation:
We combine 7 and 8 to get 1 so the array converts to [2,4,1,1,1] then,
we combine 2 and 4 to get 2 so the array converts to [2,1,1,1] then,
we combine 2 and 1 to get 1 so the array converts to [1,1,1] then,
we combine 1 and 1 to get 0 so the array converts to [1] then that's the value of last stone.

Code

```java
    static int getStoned(int[] arr) {
        // Priority Queue is an Implementation of Min Heap, we will use -ve for Making it Max Heap
        PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>();
        for (int x : arr) {
            maxHeap.add(-x);
        }

        while (maxHeap.size() > 1) {
            int One = -maxHeap.remove();
            int Two = -maxHeap.remove();
            int diff = Math.abs(Two - One);
            if (diff != 0) {
                maxHeap.add(-diff);
            }
        }

        if (maxHeap.size() == 0)
            return 0;

        return -maxHeap.remove();
    }
```

Contiguous Array

Given a binary array, find the maximum length of a contiguous subarray with equal number of 0 and 1.

Example 1:

```
Input: [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with equal number of 0 and 1.
```

Code

```
    public static int findMaxLength(int[] nums) {
        HashMap<Integer, Integer> hashStore = new
HashMap<Integer, Integer>();
        int count = 0;
        int maxEqual = 0;
        for (int J = 0; J < nums.length; J++) {
            if (nums[J] == 0)
                count--;
            if (nums[J] == 1)
                count++;

            if (count == 0)
                maxEqual = J + 1;

            if (hashStore.containsKey(count)) {
                maxEqual = Math.max(maxEqual, J -
hashStore.get(count));
            } else
                hashStore.put(count, J);

        }
        return maxEqual;
    }
```

String Shift

Given a string s containing lowercase English letters, and a matrix shift,
where shift[i] = [direction, amount]:

** Direction can be 0 (for left shift) or 1 (for right shift).
** Amount is the amount by which string s is to be shifted.
** A left shift by 1 means remove the first character of s and append it to the end.
** Similarly, a right shift by 1 means remove the last character of s and add it to the beginning.

Return the final string after all operations.

For Example -

Example 1:

Input: s = "abc", shift = [[0,1], [1,2]]
Output: "cab"

Code

```
public static String stringShift(String s, int[][] shift) {
    Deque<Character> queue = new LinkedList<Character>();
    char[] strA = s.toCharArray();
    for (char x : strA)
        queue.add(x);

    for (int[] arr : shift) {
        switch (arr[0]) {
        // Left Shift
        case 0:
            for (int i = 0; i < arr[1]; i++)
                queue.addLast(queue.poll());
            break;
        case 1:
            for (int i = 0; i < arr[1]; i++)
                queue.addFirst(queue.pollLast());
            break;
        }
    }

    StringBuilder builder = new StringBuilder();
    while (!queue.isEmpty()) {
        builder.append(queue.poll());
    }

    return new String(builder);
}
```

April - Week 3

Product of Array Except Self

Given an array `nums` of *n* integers where *n* > 1, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Example:

```
Input:  [1,2,3,4]
Output: [24,12,8,6]
```

Constraint: It's guaranteed that the product of the elements of any prefix or suffix of the array (including the whole array) fits in a 32 bit integer.

Note: Please solve it **without division** and in O(*n*).

Follow up:
Could you solve it with constant space complexity? (The output array **does not** count as extra space for the purpose of space complexity analysis.)

Code

```java
public static int[] productExceptSelf(int[] nums) {

    int[] output = new int[nums.length];
    int prod = 1;
    for (int i = 0; i < nums.length; i++) {
        output[i] = 1;
        output[i] *= prod;
        prod *= nums[i];
    }

    prod = 1;
    for (int i = nums.length - 1; i >= 0; i--) {
        output[i] *= prod;
        prod *= nums[i];
    }

    for (int x : output)
        System.out.print(x + " ");

    return nums;
}
```

Valid Parenthesis String

Given a string containing only three types of characters: '(', ')' and '*', write a function to check whether this string is valid. We define the validity of a string by these rules:

1. Any left parenthesis `'('` must have a corresponding right parenthesis `')'`.
2. Any right parenthesis `')'` must have a corresponding left parenthesis `'('`.
3. Left parenthesis `'('` must go before the corresponding right parenthesis `')'`.
4. `'*'` could be treated as a single right parenthesis `')'` or a single left parenthesis `'('` or an empty string.
5. An empty string is also valid.

Example 1:

```
Input: "()"
Output: True
```

Code

```
public static boolean checkValidString(String s) {
        char[] arr = s.toCharArray();

        int leftCount = 0;
        int rightCount = 0;

        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == ')')
                leftCount--;
            else
                leftCount++;
            if(leftCount<0)
                return false;
        }
        if(leftCount==0)
            return true;

        for (int i = arr.length-1; i >=0; i--) {
            if (arr[i] == '(')
                rightCount--;
            else
                rightCount++;
            if(rightCount<0)
                return false;
        }
        if(rightCount==0)
            return true;


        return true;
    }
```

Number of Islands

Given a 2d grid map of `'1'`s (land) and `'0'`s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input:
11110
11010
11000
00000

Output: 1
```

Code

```
static int findIsland(char[][] arr) {
    if (arr == null || arr.length == '0')
        return '0';
    int islandCounter = 0;
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr[i].length; j++) {
            if (arr[i][j] == '1')
                islandCounter += dfs(arr, i, j);
        }
    }
    return islandCounter;
}

// Making Depth Search to Converst all Connected 1't to Zero
static int dfs(char arr[][], int i, int j) {
    if (i < 0 || i >= arr.length || j < 0 || j >= arr[0].length || arr[i][j] ==
'0') {
        return 0;
    }
    if (arr[i][j] == '1')
        arr[i][j] = '0';
    dfs(arr, i + 1, j);
    dfs(arr, i - 1, j);
    dfs(arr, i, j + 1);
    dfs(arr, i, j - 1);
    return 1;
}
```

Minimum Path Sum

Given a *m* x *n* grid filled with non-negative numbers, find a path from top left to bottom right which *minimizes* the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example:

```
Input:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
Output: 7
Explanation: Because the path 1→3→1→1→1 minimizes the sum.
```

Code

```
static int getSum(int[][] arr) {
    if (arr == null || arr.length == 0)
        return 0;
    int[][] dynamic = new int[arr.length][arr[0].length];
    for (int i = 0; i < dynamic.length; i++) {
        for (int j = 0; j < dynamic[i].length; j++) {
            // Making Sum;
            dynamic[i][j] += arr[i][j];
            if (i > 0 && j > 0) {
                dynamic[i][j] += Math.min(dynamic[i - 1][j],
dynamic[i][j - 1]);
            } else if (i > 0) {
                dynamic[i][j] += dynamic[i - 1][j];
            } else if (j > 0) {
                dynamic[i][j] += dynamic[i][j - 1];
            }
        }
    }
    return dynamic[dynamic.length - 1][dynamic[0].length - 1];
}
```

Search in Rotated Sorted Array

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]`).

You are given a target value to search. If found in the array return its index, otherwise return `-1`.

You may assume no duplicate exists in the array.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

Example 1:

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
```

Code

```
static int search(int[] arr, int l, int h, int key) {
    if (l > h)
        return -1;
    // Finding Middle Element
    int mid = (l + h) / 2;
    if (key == arr[mid])
        return mid;
    // Checking of Low<Mid i.e Array is Sorted
    if (arr[l] <= arr[mid]) {
        if (key >= arr[l] && key <= arr[mid])
            return search(arr, l, mid - 1, key);
        return search(arr, mid + 1, h, key);
    }
    // Not Sorted
    if (key >= arr[mid] && key <= arr[h])
        return search(arr, mid + 1, h, key);
    return search(arr, l, mid - 1, key);
}
```

Construct Binary Search Tree from Preorder Traversal

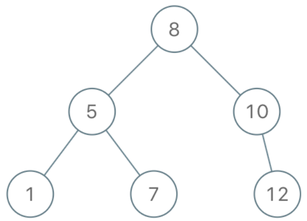
Return the root node of a binary **search** tree that matches the given `preorder` traversal.

(Recall that a binary search tree is a binary tree where for every `node`, any descendant of `node.left` has a value `<` `node.val`, and any descendant of `node.right` has a value `>` `node.val`. Also recall that a preorder traversal displays the value of the `node` first, then traverses `node.left`, then traverses `node.right`.)

It's guaranteed that for the given test cases there is always possible to find a binary search tree with the given requirements.

Example 1:

```
Input: [8,5,1,7,10,12]
Output: [8,5,10,1,7,null,12]
```



Code

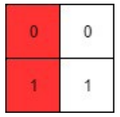
```
static TreeNode buildBST(int[] preorder) {
    TreeNode root = new TreeNode(preorder[0]);
    for (int i = 1; i < preorder.length; i++)
        insertInBST(root, preorder[i]);
    return root;
}

static void insertInBST(TreeNode root, int data) {
    if (data < root.data) {
        if (root.left == null) {
            root.left = new TreeNode(data);
            System.out.println("Data Left " + data);
        } else
            insertInBST(root.left, data);
    } else {
        if (root.right == null) {
            root.right = new TreeNode(data);
            System.out.println("Data Right " + data);
        } else
            insertInBST(root.right, data);
    }
}
```

Leftmost Column with at Least a One

A binary matrix means that all elements are `0` or `1`. For each **individual** row of the matrix, this row is sorted in non-decreasing order.
Given a row-sorted binary matrix binaryMatrix, return leftmost column index(0-indexed) with at least a `1` in it. If such index doesn't exist, return `-1`.
You can't access the Binary Matrix directly. You may only access the matrix using a `BinaryMatrix` interface:

- `BinaryMatrix.get(x, y)` returns the element of the matrix at index (x, y) (0-indexed).
- `BinaryMatrix.dimensions()` returns a list of 2 elements [m, n], which means the matrix is `m * n`.

Submissions making more than `1000` calls to `BinaryMatrix.get` will be judged *Wrong Answer*. Also, any solutions that attempt to circumvent the judge will result in disqualification.
For custom testing purposes you're given the binary matrix `mat` as input in the following four examples. You will not have access the binary matrix directly.
Example 1:



Input: mat = [[0,0],[1,1]]
Output: 0

Code

```java
public static int leftMostColumnWithOne(BinaryMatrix binaryMatrix) {
    List<Integer> list = binaryMatrix.dimensions();
    int Y = list.get(1) - 1;

    int Xr = 0;
    int Yr = Y;

    int rPointer = binaryMatrix.get(Xr, Yr);
    int rC = -1;
    boolean isNotEnd = true;
    while (isNotEnd) {
        // Traverse Left
        if (rPointer == 1) {
            rC = Yr;
            if (Yr != 0)
                rPointer = binaryMatrix.get(Xr, --Yr);
            else
                isNotEnd = false;
        } else {
            if (Xr != list.get(0) - 1)
                rPointer = binaryMatrix.get(++Xr, Yr);
            else
                isNotEnd = false;
        }
    }
    return rC;
}
```

April - Week 4

Subarray Sum Equals K

Given an array of integers and an integer **k**, you need to find the total number of continuous subarrays whose sum equals to **k**.

Example 1:

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

Note:

1. The length of the array is in range [1, 20,000].
2. The range of numbers in the array is [-1000, 1000] and the range of the integer **k** is [-1e7, 1e7].

Code

```
static int getCount(int[] nums, int key) {
        HashMap<Integer, Integer> hash = new HashMap<Integer,
Integer>();
        int count = 0;
        int prefixSum = 0;
        if (nums.length == 1) {
            if (nums[0] == key)
                return 1;
            else
                return 0;
        }

        for (int i = 0; i < nums.length; i++) {
            prefixSum += nums[i];
            if (prefixSum == key)
                count++;

            if (hash.containsKey(prefixSum - key))
                count += hash.get(prefixSum - key);

            if (hash.containsKey(prefixSum))
                hash.put(prefixSum, hash.get(prefixSum) + 1);
            else
                hash.put(prefixSum, 1);

        }
        return count;
    }
```

Bitwise AND of Numbers Range

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

Example 1:

```
Input: [5,7]
Output: 4
```

Code

```
static int getAnd(int m, int n) {

        int counter = 0;

        // Checking for Common Bits in M and N
        while (m != n) {
            counter++;
            m = m >> 1;
            n = n >> 1;
        }
        // Found Common Bits as Counter, Now Shifting Left
        n = n << counter;

        return n;
    }
```

LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: `get` and `put`.

`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
`put(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

The cache is initialized with a **positive** capacity.

Follow up:
Could you do both operations in **O(1)** time complexity?

Example:

```
LRUCache cache = new LRUCache( 2 /* capacity */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.put(4, 4);    // evicts key 1
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

Code

```
class LRUCache {
    LinkedHashMap<Integer, Integer> hash;

    public LRUCache(int capacity) {
        hash = new LinkedHashMap<Integer, Integer>(capacity,
0.75f, true) {
            private static final long serialVersionUID = 1L;
            @Override
            protected boolean
removeEldestEntry(Map.Entry<Integer, Integer> entry) {
                return this.size() > capacity;
            }
        };
    }

    public int get(int key) {
        return (hash.containsKey(key)) ? hash.get(key) : -1;
    }

    public void put(int key, int value) {
        hash.put(key, value);
    }
}
```

Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

Example 1:

```
Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

Code

```java
public boolean optimalSolution(int[] nums) {
    int rechability = 0;
    for (int i = 0; i < nums.length; i++) {
        if (rechability < i)
            return false;
        rechability = Math.max(rechability, i + nums[i]);
    }
    return true;
}
```

Longest Common Subsequence

Given two strings `text1` and `text2`, return the length of their longest common subsequence.

A *subsequence* of a string is a new string generated from the original string with some characters(can be none) deleted without changing the relative order of the remaining characters. (eg, "ace" is a subsequence of "abcde" while "aec" is not). A *common subsequence* of two strings is a subsequence that is common to both strings.

If there is no common subsequence, return 0.

Example 1:

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
```

Code

```
private static int longestCommonSubsequence(char[] a, char[] b,
int m, int n) {
        int[][] L = new int[m + 1][n + 1];
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0 || j == 0)
                    L[i][j] = 0;
                else if (a[i - 1] == b[j - 1]) {
                    L[i][j] = 1 + L[i - 1][j - 1];
                } else
                    L[i][j] = Math.max(L[i][j - 1], L[i - 1][j]);
            }
        }
        return L[m][n];
    }
```

Maximal Square

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing only 1's and return its area.

Example:

Input:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

Output: 4

Code

```
static int getMaxSq(char[][] arr) {
    if (arr == null || arr.length == 0)
        return 0;
    int maxFound = 0;
    int[][] dp = new int[arr.length][arr[0].length];
    for (int i = 0; i < arr[0].length; i++) {
        dp[0][i] = Character.getNumericValue(arr[0][i]);
        maxFound = Math.max(maxFound, dp[0][i]);
    }
    for (int i = 0; i < arr.length; i++) {
        dp[i][0] = Character.getNumericValue(arr[i][0]);
        maxFound = Math.max(maxFound, dp[i][0]);
    }
    for (int i = 0; i < arr.length; i++)
        for (int j = 0; j < arr[0].length; j++) {
            if (i == 0 || j == 0)
                continue;
            if (arr[i][j] == '1') {
                dp[i][j] = Math.min(Math.min(dp[i][j - 1], dp[i - 1][j]),
dp[i - 1][j - 1]) + 1;
                maxFound = Math.max(maxFound, dp[i][j]);
            }
        }
    return maxFound * maxFound;
}
```

Find First Unique Element

You have a queue of integers, you need to retrieve the first unique integer in the queue.

Implement the FirstUnique class:

- FirstUnique(int[] nums) Initializes the object with the numbers in the queue.
- int showFirstUnique() returns the value of the first unique integer of the queue, and returns -1 if there is no such integer.
- void add(int value) insert value to the queue.

Example:

```
Input: [2, 3, 5]
add(5)
add(2)
add(3)
Output: -1
```

Code

```
class FirstUnique {

    Deque<Integer> uniqueStore = new LinkedList<Integer>();
    HashMap<Integer, Integer> store = new HashMap<Integer, Integer>();

    public FirstUnique(int[] nums) {
        for (int x : nums) {
            if (store.containsKey(x)) {
                store.put(x, store.get(x) + 1);
            } else {
                store.put(x, 1);
                uniqueStore.add(x);
            }
        }
    }

    public int showFirstUnique() {
        while (!uniqueStore.isEmpty() && store.get(uniqueStore.getFirst()) > 1) {
            uniqueStore.pop();
        }

        if (uniqueStore.isEmpty())
            return -1;

        return uniqueStore.getFirst();
    }

    public void add(int x) {
        if (store.containsKey(x)) {
            store.put(x, store.get(x) + 1);
        } else {
            store.put(x, 1);
            System.out.println("Adding" + x);
            uniqueStore.add(x);
        }
    }
```

April - Week 5

Binary Tree Maximum Path Sum

Given a **non-empty** binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain **at least one node** and does not need to go through the root.

Example 1:

Input: [1,2,3]

```
   1
  / \
 2   3
```

Output: 6

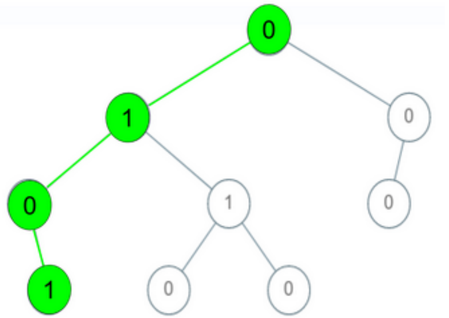
Code

```
static int dfs(TreeNode root) {
    if (root == null)
        return 0;

    int x = dfs(root.left);
    int y = dfs(root.right);
    MaxAnswer = Math.max(MaxAnswer, x + y + root.val);
    return Math.max(0, Math.max(x, y) + root.val);
}
```

Check If a String Is a Valid Sequence from Root to Leaves Path in a Binary Tree

Given a binary tree where each path going from the root to any leaf form a **valid sequence**, check if a given string is a **valid sequence** in such binary tree.
We get the given string from the concatenation of an array of integers `arr` and the concatenation of all values of the nodes along a path results in a **sequence** in the given binary tree.

Example 1:



Input:
root = [0,1,0,0,1,0,null,null,1,0,0],
arr = [0,1,0,1]
Output: true
Explanation:
The path 0 -> 1 -> 0 -> 1 is a valid sequence (green color in the figure).
Other valid sequences are:
0 -> 1 -> 1 -> 0
0 -> 0 -> 0

Code

```
 static public boolean isValidSequence_(TreeNode root, int[] arr)
{
        if (root == null)
            return (arr.length == 0) ? true : false;
        return dfs(root, arr, 0);
    }

    static boolean dfs(TreeNode root, int[] arr, int index) {
        if (arr[index] != root.val)
            return false;

        if (index == (arr.length - 1)) {
            return (root.left == null && root.right == null);
        }

        return ((root.left != null && dfs(root.left, arr, index + 1))
                || (root.right != null && dfs(root.right, arr, index + 1)));
    }
```