Leet Code

June 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/

June - Week 1

Invert Binary Tree

Invert a binary tree.

Example:

Input:

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

Output:

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

Trivia:

This problem was inspired by this original tweet by Max Howell:

Google: 90% of our engineers use the software you wrote (Homebrew), but you can’t invert a binary tree on a whiteboard so f*** off.

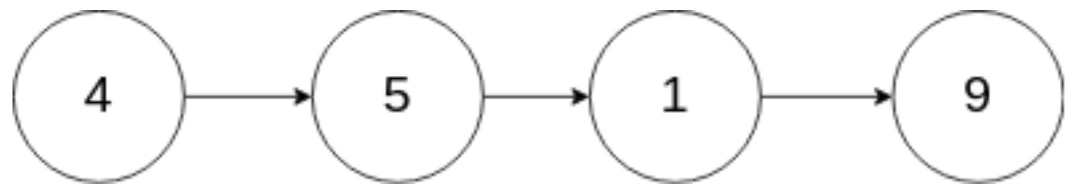
Coding

```java
private static Node invertTree(Node root) {
        invertNode(root);
        return root;
    }

    private static void invertNode(Node root) {
        if (root == null)
            return;

        if (root.left == null && root.right == null)
            return;

        Node x = root.right;
        root.right = root.left;
        root.left = x;

        invertNode(root.left);
        invertNode(root.right);
    }
```

Delete Node in a Linked List

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Given linked list -- head = [4,5,1,9], which looks like following:



Example 1:

```
Input: head = [4,5,1,9], node = 5
Output: [4,1,9]
Explanation: You are given the second node
with value 5, the linked list should become 4
-> 1 -> 9 after calling your function.
```

Example 2:

```
Input: head = [4,5,1,9], node = 1
Output: [4,5,9]
Explanation: You are given the third node
with value 1, the linked list should become 4
-> 5 -> 9 after calling your function.
```

Coding

```
public void deleteNode(ListNode node) {
        ListNode prev = null;
        while (node.next != null) {
            node.val = node.next.val;
            prev = node;
            node = node.next;
        }
        prev.next = null;
    }
```

Two City Scheduling

There are `2N` people a company is planning to interview. The cost of flying the `i`-th person to city `A` is `costs[i][0]`, and the cost of flying the `i`-th person to city `B` is `costs[i][1]`.

Return the minimum cost to fly every person to a city such that exactly `N` people arrive in each city.

Example 1:

```
Input: [[10,20],[30,200],[400,50],[30,20]]
Output: 110
Explanation:
The first person goes to city A for a cost of 10.
The second person goes to city A for a cost of
30.
The third person goes to city B for a cost of 50.
The fourth person goes to city B for a cost of
20.

The total minimum cost is 10 + 30 + 50 + 20 = 110
to have half the people interviewing in each
city.
```

Coding

```
public static int twoCitySchedCost(int[][] costs) {
  // Sorting Int[], based on their Differences
 Arrays.sort(costs, new Comparator<int[]>() {
         @Override
         public int compare(int[] A, int[] B) {
            return Integer.compare((A[0] - A[1]), (B[0] - B[1]));
         }
     });
     int Result = 0;
   // Fetching from Array, for N/2 from A and N/2 from B
     for (int i = 0; i < costs.length; i++) {
         if (i < costs.length / 2) {
            Result += costs[i][0];
         } else {
            Result += costs[i][1];
         }
     }
     return Result;
 }
```

Reverse String

Write a function that reverses a string. The input string is given as an array of characters `char[]`.

Do not allocate extra space for another array, you must do this by **modifying the input array in-place** with O(1) extra memory.

You may assume all the characters consist of printable ascii characters.

Example 1:

```
Input: ["h","e","l","l","o"]
Output: ["o","l","l","e","h"]
```

Example 2:

```
Input: ["H","a","n","n","a","h"]
Output: ["h","a","n","n","a","H"]
```

Coding

```
public static void reverseString(char[] s) {
        int e = s.length;
        for (int i = 0; i < e / 2; i++) {
            char x = s[i];
            s[i] = s[e - 1 - i];
            s[e - 1 - i] = x;
        }
    }
```

Random Pick with Weight

Given an array `w` of positive integers, where `w[i]` describes the weight of index `i`, write a function `pickIndex` which randomly picks an index in proportion to its weight.

Note:

1. `1 <= w.length <= 10000`
2. `1 <= w[i] <= 10^5`
3. `pickIndex` will be called at most `10000` times.

Example 1:

```
Input: 
["Solution","pickIndex"]
[[[1]],[]]
Output: [null,0]
```

Example 2:

```
Input: 
["Solution","pickIndex","pickIndex","pickIndex","pic
kIndex","pickIndex"]
[[[1,3]],[],[],[],[],[]]
Output: [null,0,1,1,1,0]
```

Explanation of Input Syntax:

The input is two lists: the subroutines called and their arguments. `Solution`'s constructor has one argument, the array `w`. `pickIndex` has no arguments. Arguments are always wrapped with a list, even if there aren't any.

Coding

```java
class Solution {
    int[] ActualArray;
    int[] weightArray;
    int TotalSum = 0;

    public Solution(int[] w) {
        ActualArray = w.clone();
        weightArray = new int[w.length];
        int TSum = 0;
        for (int i = 0; i < w.length; i++) {
            TSum += ActualArray[i];
            weightArray[i] = TSum;
        }
        TotalSum = TSum;
    }

    public int pickIndex() {
        double RandomNumber = Math.random() * TotalSum;
        int low = 0;
        int high = weightArray.length;
        while (low < high) {
            int mid = (low + high) / 2;
            if (RandomNumber > weightArray[mid]) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return low;
    }
}
```

Queue Reconstruction by Height

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers `(h, k)`, where `h` is the height of the person and `k` is the number of people in front of this person who have a height greater than or equal to `h`. Write an algorithm to reconstruct the queue.

Note:
The number of people is less than 1,100.

Example

```
Input:
[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:
[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]
```

Coding

```
public static int[][] reconstructQueue(int[][] people) {
        // Sorting People Array First - Descending Order
        List<int[]> result = new ArrayList<int[]>();
        Arrays.sort(people, new Comparator<int[]>() {
            public int compare(int[] a, int[] b) {
                if (a[0] != b[0])
                    return Integer.compare(b[0], a[0]);
                else
                    return Integer.compare(a[1], b[1]);
            }
        });
        for (int[] person : people) {
            result.add(person[1], person);
        }
        return result.toArray(new int[result.size()][]);
    }
```

Coin Change 2

You are given coins of different denominations and a total amount of money. Write a function to compute the number of combinations that make up that amount. You may assume that you have infinite number of each kind of coin.

Example 1:

```
Input: amount = 5, coins = [1, 2, 5]
Output: 4
Explanation: there are four ways to make up the amount:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

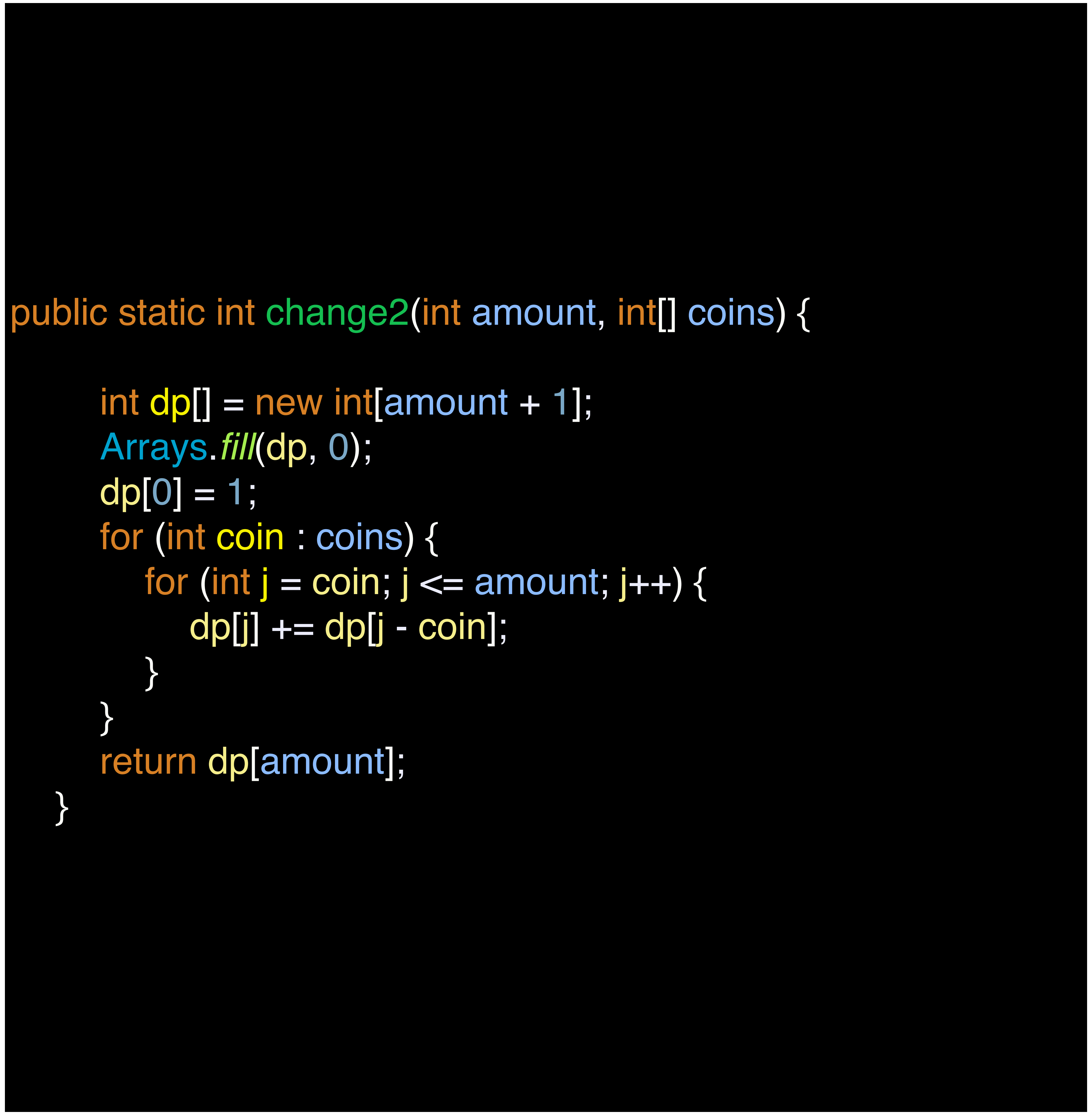
Example 2:

```
Input: amount = 3, coins = [2]
Output: 0
Explanation: the amount of 3 cannot be made up just with coins of 2.
```

Example 3:

```
Input: amount = 10, coins = [10]
Output: 1
```

Coding



```java
public static int change2(int amount, int[] coins) {

        int dp[] = new int[amount + 1];
        Arrays.fill(dp, 0);
        dp[0] = 1;
        for (int coin : coins) {
            for (int j = coin; j <= amount; j++) {
                dp[j] += dp[j - coin];
            }
        }
        return dp[amount];
    }
```

June - Week 2

Power of Two

Given an integer, write a function to determine if it is a power of two.

Example 1:

```
Input: 1
Output: true
Explanation: 20 = 1
```

Example 2:

```
Input: 16
Output: true
Explanation: 24 = 16
```

Example 3:

```
Input: 218
Output: false
```

Coding

```
public static boolean isPowerOfTwo(int n) {
        if (n < 0) {
            return false;
        }
        int counter = 0;
        while (n != 0) {
            if (n % 2 != 0)
                counter++;
            n = n >> 1;
        }
        return (counter == 1) ? true : false;
    }
```

Is Subsequence

Given a string **s** and a string **t**, check if **s** is subsequence of **t**.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, `"ace"` is a subsequence of `"abcde"` while `"aec"` is not).

Follow up:
If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B, and you want to check one by one to see if T has its subsequence. In this scenario, how would you change your code?

Credits:
Special thanks to @pbrother for adding this problem and creating all test cases.

Example 1:

```
Input: s = "abc", t = "ahbgdc"
Output: true
```

Example 2:

```
Input: s = "axc", t = "ahbgdc"
Output: false
```

Constraints:

- `0 <= s.length <= 100`
- `0 <= t.length <= 10^4`
- Both strings consists only of lowercase characters.

Coding

```
public static boolean isSubsequence(String s, String t) {
        if (s.length() == 0) {
            return true;
        }
        if (t.length() == 0) {
            return false;
        }
        char[] S = s.toCharArray();
        char[] T = t.toCharArray();
        int sCounter = 0;
        for (int i = 0; i < T.length; i++) {
            if (T[i] == S[sCounter]) {
                System.out.println(T[i]);
                sCounter++;
            }
            if (sCounter == S.length) {
                return true;
            }
        }
        return false;
    }
```

Search Insert Position

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.

Example 1:

```
Input: [1,3,5,6], 5
Output: 2
```

Example 2:

```
Input: [1,3,5,6], 2
Output: 1
```

Example 3:

```
Input: [1,3,5,6], 7
Output: 4
```

Example 4:

```
Input: [1,3,5,6], 0
Output: 0
```

Coding

```
public static int searchInsert(int[] nums, int target) {
    int low = 0;
    int high = nums.length - 1;
    if (target < nums[0]) {
        return 0;
    }
    if (target > nums[nums.length - 1]) {
        return nums.length;
    }

    while (low < high) {
        int mid = low + (high - low) / 2;
        if (target > nums[mid]) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }

    return low;
}
```

Sort Colors

Given an array with *n* objects colored red, white or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note: You are not suppose to use the library's sort function for this problem.

Example:

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

Follow up:

- A rather straight forward solution is a two-pass algorithm using counting sort.
  First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.
- Could you come up with a one-pass algorithm using only constant space?

Coding

```
public static void sortColors(int[] nums) {
        int start = 0;
        int end = nums.length - 1;
        int index = 0;
        while (index <= end && start < end) {
            if (nums[index] == 0) {
                nums[index] = nums[start];
                nums[start] = 0;
                start++;
                index++;
            } else if (nums[index] == 2) {
                nums[index] = nums[end];
                nums[end] = 2;
                end--;
            } else
                index++;
        }

    }
```

Insert Delete GetRandom O(1)

Design a data structure that supports all following operations in *average* **O(1)** time.

1. `insert(val)`: Inserts an item val to the set if not already present.
2. `remove(val)`: Removes an item val from the set if present.
3. `getRandom`: Returns a random element from current set of elements. Each element must have the **same probability** of being returned.
4.

Coding

```java
class RandomizedSet {
    HashMap<Integer, Integer> hash;
    ArrayList<Integer> array;

    /** Initialize your data structure here. */
    public RandomizedSet() {
        hash = new HashMap<Integer, Integer>();
        array = new ArrayList<Integer>();
    }

    public boolean insert(int val) {
        if (hash.containsKey(val)) {
            return false;
        }
        hash.put(val, array.size());
        array.add(val);
        return true;
    }
    public boolean remove(int val) {
        if (hash.containsKey(val)) {
            int last_element = array.get(array.size() - 1);
            int key = hash.get(val);
            array.set(key, last_element);
            hash.put(last_element, key);
            array.remove(array.size() - 1);
            hash.remove(val);
            return true;
        }
        return false;
    }
    public int getRandom() {
        int Size = array.size();
        int Prob = (int) (Math.random() * (Size));
        return array.get(Prob);
    }
}
```

Largest Divisible Subset

Given a set of **distinct** positive integers, find the largest subset such that every pair (Si, Sj) of elements in this subset satisfies:

Si % Sj = 0 or Sj % Si = 0.

If there are multiple solutions, return any subset is fine.

Example 1:

```
Input: [1,2,3]
Output: [1,2] (of course, [1,3] will also be
ok)
```

Example 2:

```
Input: [1,2,4,8]
Output: [1,2,4,8]
```

Coding

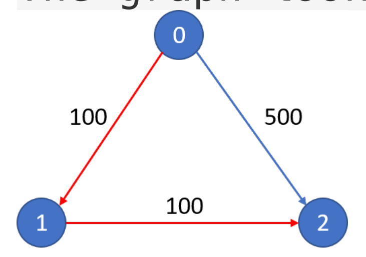
```java
public static List<Integer> largestDivisibleSubset(int[] nums) {
        if (nums.length == 0) {
            return new ArrayList<Integer>();
        }
        List<Integer> res = new ArrayList<Integer>();
        Arrays.sort(nums);
        int prev[] = new int[nums.length];
        Arrays.fill(prev, -1);
        int divCount[] = new int[nums.length];
        Arrays.fill(divCount, 1);
        int MaxCOunt = 0;
        for (int i = 0; i < nums.length; i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] % nums[j] == 0 && divCount[i] < divCount[j] + 1) {
                    prev[i] = j;
                    divCount[i] = divCount[j] + 1;
                }
            }
            if (divCount[i] > divCount[MaxCOunt])
                MaxCOunt = i;
        }
        int k = MaxCOunt;
        while (k >= 0) {
            res.add(nums[k]);
            k = prev[k];
        }
        return res;
    }
```

Cheapest Flights Within K Stops

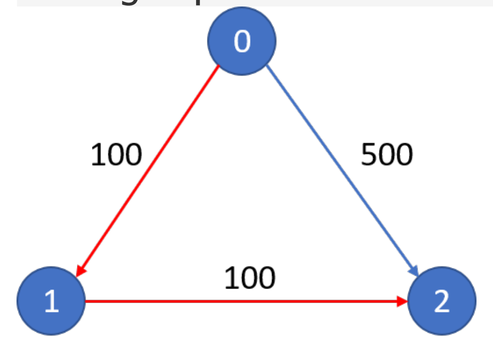
There are `n` cities connected by `m` flights. Each flight starts from city `u` and arrives at `v` with a price `w`.

Now given all the cities and flights, together with starting city `src` and the destination `dst`, your task is to find the cheapest price from `src` to `dst` with up to `k` stops. If there is no such route, output `-1`.

```
Example 1:
Input:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 1
Output: 200
Explanation:
The graph looks like this:
```



```
The cheapest price from city 0 to city 2 with at most 1 stop costs 200, as marked red in
the picture.
Example 2:
Input:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 0
Output: 500
Explanation:
The graph looks like this:
```

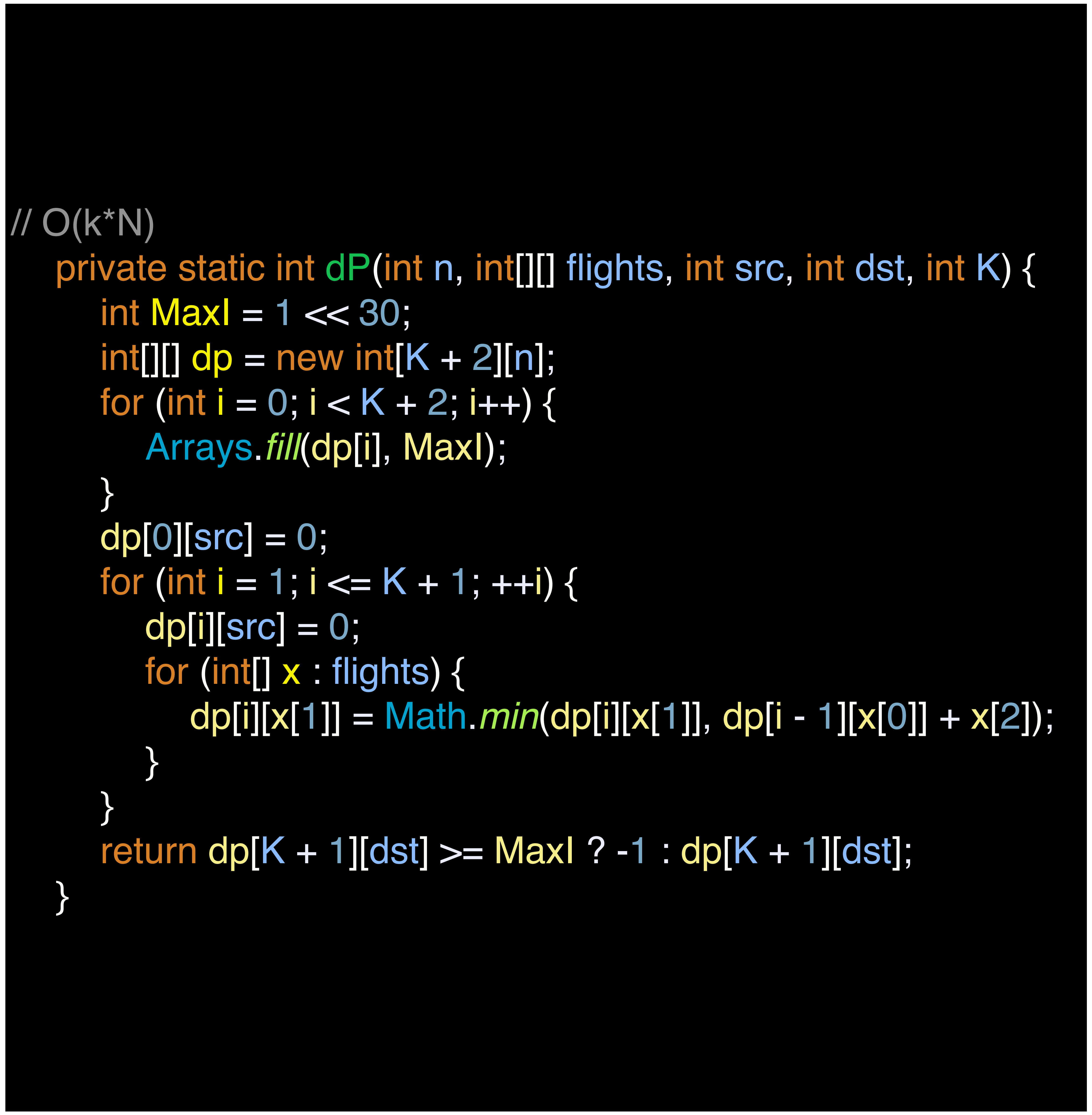


```
The cheapest price from city 0 to city 2 with at most 0 stop costs 500, as marked blue
in the picture.
```

Constraints:

- The number of nodes `n` will be in range `[1, 100]`, with nodes labeled from `0` to `n - 1`.
- The size of `flights` will be in range `[0, n * (n - 1) / 2]`.
- The format of each flight will be `(src, dst, price)`.
- The price of each flight will be in the range `[1, 10000]`.
- `k` is in the range of `[0, n - 1]`.
- There will not be any duplicated flights or self cycles.

Coding



```java
// O(k*N)
    private static int dP(int n, int[][] flights, int src, int dst, int K) {
        int MaxI = 1 << 30;
        int[][] dp = new int[K + 2][n];
        for (int i = 0; i < K + 2; i++) {
            Arrays.fill(dp[i], MaxI);
        }
        dp[0][src] = 0;
        for (int i = 1; i <= K + 1; ++i) {
            dp[i][src] = 0;
            for (int[] x : flights) {
                dp[i][x[1]] = Math.min(dp[i][x[1]], dp[i - 1][x[0]] + x[2]);
            }
        }
        return dp[K + 1][dst] >= MaxI ? -1 : dp[K + 1][dst];
    }
```

June - Week 3

Search in a Binary Search Tree

Given the root node of a binary search tree (BST) and a value. You need to find the node in the BST that the node's value equals the given value. Return the subtree rooted with that node. If such node doesn't exist, you should return NULL.

For example,

```
Given the tree:
        4
       / \
      2   7
     / \
    1   3

And the value to search: 2
```

You should return this subtree:

```
      2
     / \
    1   3
```

In the example above, if we want to search the value `5`, since there is no node with value `5`, we should return `NULL`.

Note that an empty tree is represented by `NULL`, therefore you would see the expected output (serialized tree format) as `[]`, not `null`.

Coding

```
public static TreeNode searchBST(TreeNode root, int val) {
        if (root == null) {
            return null;
        }
        return hasFound(root, val);
    }

    private static TreeNode hasFound(TreeNode root, int val) {
        if (root == null) {
            return null;
        }
        if (root.val == val) {
            return root;
        }
        if (val > root.val) {
            return hasFound(root.right, val);
        } else {
            return hasFound(root.left, val);
        }
    }
```

Validate IP Address

Write a function to check whether an input string is a valid IPv4 address or IPv6 address or neither. **IPv4** addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots ("."), e.g.,`172.16.254.1`; Besides, leading zeros in the IPv4 is invalid. For example, the address `172.16.254.01` is invalid. **IPv6** addresses are represented as eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons (":"). For example, the address `2001:0db8:85a3:0000:0000:8a2e:0370:7334` is a valid one. Also, we could omit some leading zeros among four hexadecimal digits and some low-case characters in the address to upper-case ones, so `2001:db8:85a3:0:0:8A2E:0370:7334` is also a valid IPv6 address(Omit leading zeros and using upper cases). However, we don't replace a consecutive group of zero value with a single empty group using two consecutive colons (::) to pursue simplicity. For example, `2001:0db8:85a3::8A2E:0370:7334` is an invalid IPv6 address. Besides, extra leading zeros in the IPv6 is also invalid. For example, the address `02001:0db8:85a3:0000:0000:8a2e:0370:7334` is invalid. **Note:** You may assume there is no extra space or special characters in the input string.

Example 1:

```
Input: "172.16.254.1"
Output: "IPv4"
Explanation: This is a valid IPv4 address, return "IPv4".
```

Example 2:

```
Input: "2001:0db8:85a3:0:0:8A2E:0370:7334"
Output: "IPv6"
Explanation: This is a valid IPv6 address, return "IPv6".
```

Example 3:

```
Input: "256.256.256.256"
Output: "Neither"
Explanation: This is neither a IPv4 address nor a IPv6 address.
```

Coding

```
class Solution {
        String chunkIPv4 = "([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|
25[0-5])";
          Pattern pattenIPv4 =
                Pattern.compile("^(" + chunkIPv4 + "\\.){3}" +
chunkIPv4 + "$");

        String chunkIPv6 = "([0-9a-fA-F]{1,4})";
        Pattern pattenIPv6 =
                Pattern.compile("^(" + chunkIPv6 + "\\:){7}" +
chunkIPv6 + "$");

        public String validIPAddress(String IP) {
          if (pattenIPv4.matcher(IP).matches()) return "IPv4";
          return (pattenIPv6.matcher(IP).matches()) ? "IPv6" :
"Neither";
        }
      }
```

Surrounded Regions

Given a 2D board containing `'X'` and `'O'` (the letter O), capture all regions surrounded by `'X'`.

A region is captured by flipping all `'O'`s into `'X'`s in that surrounded region.

Example:

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

Explanation:

Surrounded regions shouldn’t be on the border, which means that any `'O'` on the border of the board are not flipped to `'X'`. Any `'O'` that is not on the border and it is not connected to an `'O'` on the border will be flipped to `'X'`. Two cells are connected if they are adjacent cells connected horizontally or vertically.

Coding

```
public static void solve(char[][] board) {
        if (board.length == 0 || board[0].length == 0) {
                return;
        }
        for (int i = 0; i < board.length; i++) {
                if (board[i][0] == 'O')
                        DFS(board, i, 0);
                if (board[i][board[0].length - 1] == 'O')
                        DFS(board, i, board[0].length - 1);
        }

        for (int i = 0; i < board[0].length; i++) {
                if (board[0][i] == 'O')
                        DFS(board, 0, i);
                if (board[board.length - 1][i] == 'O')
                        DFS(board, board.length - 1, i);
        }

        for (int i = 0; i < board.length; i++) {
                for (int j = 0; j < board[0].length; j++) {
                        if (board[i][j] == 'O') {
                                board[i][j] = 'X';
                        } else if (board[i][j] == '*') {
                                board[i][j] = 'O';
                        }
                }
        }

}
private static void DFS(char[][] board, int I, int J) {
        if (I > board.length - 1 || I < 0 || J > board[0].length || J < 0)
                return;
        if (board[I][J] == 'O')
                board[I][J] = '*';
        if (I > 0 && board[I - 1][J] == 'O') {
                DFS(board, I - 1, J);
        }
        if (I < board.length - 1 && board[I + 1][J] == 'O') {
                DFS(board, I + 1, J);
        }
        if (J > 0 && board[I][J - 1] == 'O') {
                DFS(board, I, J - 1);
        }
        if (J < board[0].length - 1 && board[I][J + 1] == 'O') {
                DFS(board, I, J + 1);
        }
}
```

H-Index II

Given an array of citations **sorted in ascending order** (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

According to the definition of h-index on Wikipedia: "A scientist has index *h* if *h* of his/her *N* papers have **at least** *h* citations each, and the other *N − h* papers have **no more than** *h* citations each."

Example:

```
Input: citations = [0,1,3,5,6]
Output: 3
Explanation: [0,1,3,5,6] means the researcher has
5 papers in total and each of them had
             received 0, 1, 3, 5, 6 citations
respectively.
             Since the researcher has 3 papers
with at least 3 citations each and the remaining
             two with no more than 3 citations
each, her h-index is 3.
```

Note:

If there are several possible values for *h*, the maximum one is taken as the h-index.

Follow up:

- This is a follow up problem to H-Index, where `citations` is now guaranteed to be sorted in ascending order.
- Could you solve it in logarithmic time complexity?
-

Coding

```java
public static int hIndex(int[] x) {
    if (x.length == 0)
        return 0;

    int l = 0;
    int r = x.length - 1;
    int res = 0;
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (x[m] < x.length - m) {
            res = Math.max(res, x[m]);
            l = m + 1;
        }
        if (x[m] >= x.length - m) {
            r = m - 1;
        }
    }
    return x.length - l;
}
```

Longest Duplicate Substring

Given a string S, consider all *duplicated substrings*: (contiguous) substrings of S that occur 2 or more times. (The occurrences may overlap.)

Return **any** duplicated substring that has the longest possible length. (If S does not have a duplicated substring, the answer is `""`.)

Example 1:

```
Input: "banana"
Output: "ana"
```

Example 2:

```
Input: "abcd"
Output: ""
```

Coding

```
public static String longestDupSubstring(String S) {
        int len = S.length();
        int[] nums = new int[len];

        for (int i = 0; i < len; i++)
            nums[i] = (int) S.charAt(i) - (int) 'a';

        int a = 26;
        long mod = (long) Math.pow(2, 32);
        int l = 1;
        int r = len;
        int m;
        while (l < r) {
            m = l + (r - l) / 2;
            if (search(m, a, mod, len, nums) != -1) {
                l = m + 1;
            } else {
                r = m;
            }
        }
        int StartingPoint = search(l - 1, a, mod, len, nums);
        return (StartingPoint != -1) ? S.substring(StartingPoint, StartingPoint + l - 1) : "";
    }

    private static int search(int L, int a, long mod, int n, int[] nums) {
        long h = 0;
        for (int i = 0; i < L; i++) {
            h = (h * a + nums[i]) % mod;
        }
        HashSet<Long> hS = new HashSet<Long>();
        hS.add(h);
        long aL = 1;
        for (int i = 1; i <= L; i++) {
            aL = (aL * a) % mod;
        }
        for (int start = 1; start < n - L + 1; start++) {
            h = (h * a - nums[start - 1] * aL % mod + mod) % mod;
            h = (h + nums[start + L - 1]) % mod;
            if (hS.contains(h))
                return start;
            hS.add(h);
        }
        return -1;
    }
```

Permutation Sequence

The set `[1,2,3,...,n]` contains a total of *n*! unique permutations.

By listing and labeling all of the permutations in order, we get the following sequence for *n* = 3:

```
1."123"
2."132"
3."213"
4."231"
5."312"
6."321"
```

Given *n* and *k*, return the *k*th permutation sequence.

Note:

- Given *n* will be between 1 and 9 inclusive.
- Given *k* will be between 1 and *n*! inclusive.

Example 1:

```
Input: n = 3, k = 3
Output: "213"
```

Example 2:

```
Input: n = 4, k = 9
Output: "2314"
```

Coding

```
private static String findPermutation(int n, int k) {
        int[] fact = new int[n];
        fact[0] = 1;
        ArrayList<Integer> l = new ArrayList<Integer>();
        for (int i = 1; i < n; i++) {
            fact[i] = i * fact[i - 1];
        }
        for (int i = 1; i <= n; i++)
            l.add(i);

        k--;

        StringBuilder res = new StringBuilder();
        for (int i = n - 1; i >= 0; i--) {
            int index = k / fact[i];
            res.append("").append(l.remove(index));
            k = k % fact[i];
        }
        return new String(res);
    }
```

Dungeon Game

The demons had captured the princess (**P**) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (**K**) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (*negative* integers) upon entering these rooms; other rooms are either empty (*0's*) or contain magic orbs that increase the knight's health (*positive* integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.

For example, given the dungeon below, the initial health of the knight must be at least **7** if he follows the optimal path `RIGHT-> RIGHT -> DOWN -> DOWN`.

Note:

- The knight's health has no upper bound.
- Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.
-

Coding

```
    private static int Optimal(int[][] arr) {
        int[][] dp = new int[arr.length + 1][arr[0].length + 1];
        for (int i = 0; i < dp.length; i++) {
            dp[i][arr[0].length] = Integer.MAX_VALUE;
        }
        for (int i = 0; i < dp[0].length; i++) {
            dp[arr.length][i] = Integer.MAX_VALUE;
        }

        dp[arr.length][arr[0].length - 1] = 1;
        dp[arr.length - 1][arr[0].length] = 1;
        dp[arr.length][arr[0].length] = 1;
        if (arr[arr.length - 1][arr[0].length - 1] <= 0) {
            dp[arr.length - 1][arr[0].length - 1] = -1 * arr[arr.length - 1][arr[0].length -
1] + 1;
        } else {
            dp[arr.length - 1][arr[0].length - 1] = 1;
        }

        for (int i = arr.length - 1; i >= 0; i--) {
            for (int j = arr[0].length - 1; j >= 0; j--) {
                dp[i][j] = Math.min(dp[i + 1][j], dp[i][j + 1]) - arr[i][j];
                if (dp[i][j] < 1)
                    dp[i][j] = 1;
            }
        }

        return dp[0][0];
    }
```

June - Week 4

Single Number II

Given a **non-empty** array of integers, every element appears *three* times except for one, which appears exactly once. Find that single one.

Note:

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

Example 1:

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

Example 2:

```
Input: [0,1,0,1,0,1,99]
Output: 99
```

Coding

```
public static int singleNumber(int[] nums) {
        int Ones = 0;
        int Twos = 0;
        int commonBits;
        for (int i = 0; i < nums.length; i++) {
            Twos = Twos | (Ones & nums[i]);
            Ones = Ones ^ nums[i];
            commonBits = ~(Ones & Twos);
            Ones &= commonBits;
            Twos &= commonBits;
        }
        return Ones;
    }
```

Count Complete Tree Nodes

Given a **complete** binary tree, count the number of nodes.

Note:

Definition of a complete binary tree from Wikipedia:
In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

Example:

```
Input:
    1
   / \
  2   3
 / \  /
4  5 6

Output: 6
```

Coding

```
public static int countNodes(TreeNode root) {
        if (root == null) {
            return 0;
        }
        return 1 + countNodes(root.left) + countNodes(root.right);
    }
```

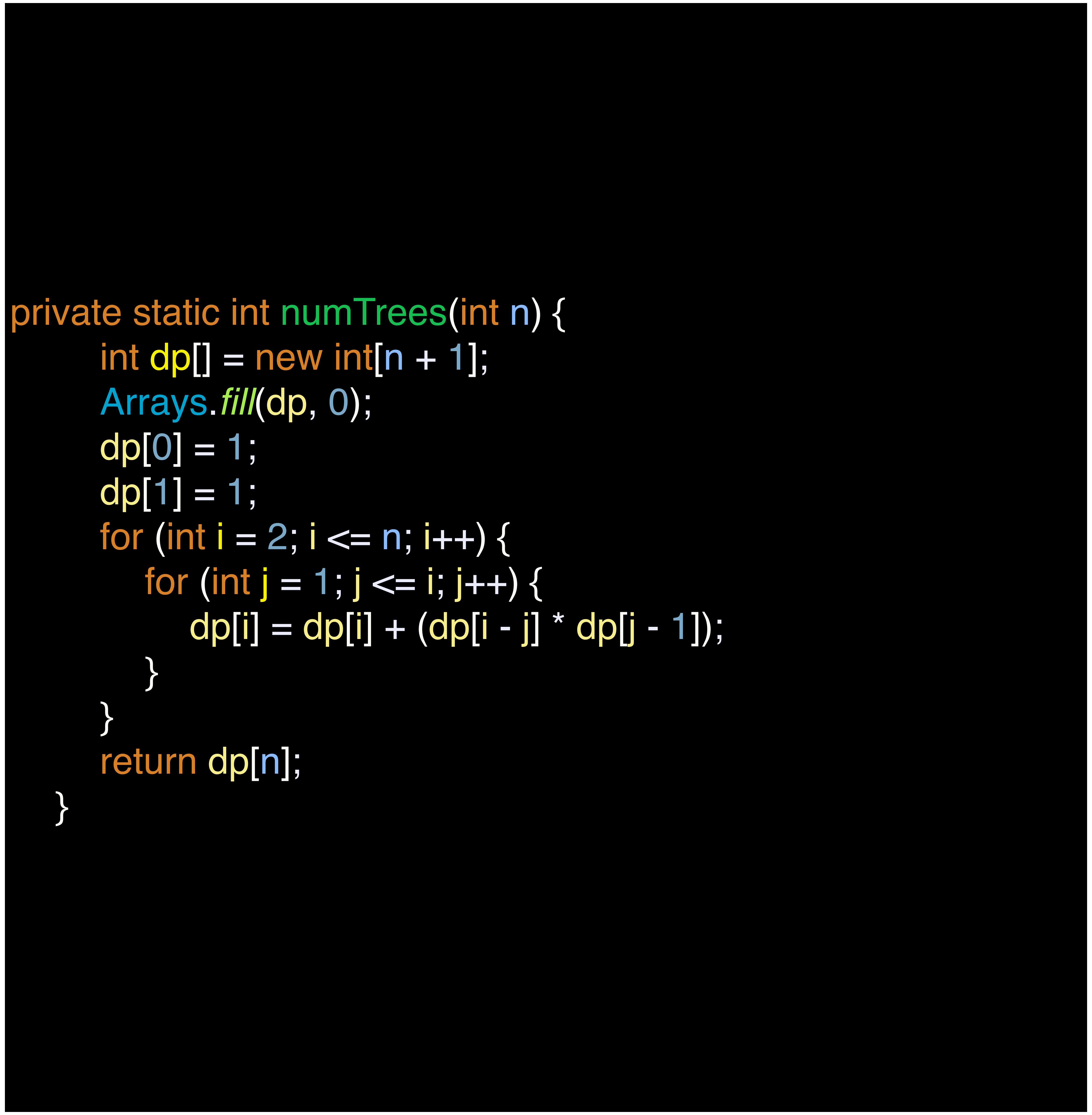
Unique Binary Search Trees

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1 ... *n*?

Example:

```
Input: 3
Output: 5
Explanation:
Given n = 3, there are a total of 5 unique
BST's:

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

Coding



```
private static int numTrees(int n) {
        int dp[] = new int[n + 1];
        Arrays.fill(dp, 0);
        dp[0] = 1;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            for (int j = 1; j <= i; j++) {
                dp[i] = dp[i] + (dp[i - j] * dp[j - 1]);
            }
        }
        return dp[n];
    }
```

Find the Duplicate Number

Given an array *nums* containing *n* + 1 integers where each integer is between 1 and *n* (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

Example 1:

```
Input: [1,3,4,2,2]
Output: 2
```

Example 2:

```
Input: [3,1,3,4,2]
Output: 3
```

Note:

1. You **must not** modify the array (assume the array is read only).
2. You must use only constant, *O*(1) extra space.
3. Your runtime complexity should be less than *O*(*n*2).
4. There is only one duplicate number in the array, but it could be repeated more than once.

Coding

```
static public int findDuplicate(int[] nums) {
        int slow = nums[0];
        int fast = nums[0];
        slow = nums[slow];
        fast = nums[nums[fast]];
        while (slow != fast) {
            slow = nums[slow];
            fast = nums[nums[fast]];
        }
        int x = nums[0];
        int y = slow;
        while (x != y) {
            x = nums[x];
            y = nums[y];
        }
        return x;
    }
```

Sum Root to Leaf Numbers

Given a binary tree containing digits from `0-9` only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path `1->2->3` which represents the number `123`.

Find the total sum of all root-to-leaf numbers.

Note: A leaf is a node with no children.

Example:

```
Input: [1,2,3]
    1
   / \
  2   3
Output: 25
Explanation:
The root-to-leaf path 1->2 represents the number 12.
The root-to-leaf path 1->3 represents the number 13.
Therefore, sum = 12 + 13 = 25.
```

Example 2:

```
Input: [4,9,0,5,1]
    4
   / \
  9   0
 / \
5   1
Output: 1026
Explanation:
The root-to-leaf path 4->9->5 represents the number 495.
The root-to-leaf path 4->9->1 represents the number 491.
The root-to-leaf path 4->0 represents the number 40.
Therefore, sum = 495 + 491 + 40 = 1026.
```

Coding

```
private static int Result;

private static int sumNumbers(TreeNode root) {
    Result = 0;
    traverse(root, 0);
    return Result;
}

private static void traverse(TreeNode root, int Sum) {
    if (root == null) {
        return;
    }
    Sum *= 10;
    Sum += root.val;
    if (root.left == null && root.right == null) {
        Result += Sum;
    }
    traverse(root.left, Sum);
    traverse(root.right, Sum);
}
```

Perfect Squares

Given a positive integer *n*, find the least number of perfect square numbers (for example, `1, 4, 9, 16, ...`) which sum to *n*.

Example 1:

```
Input: n = 12
Output: 3
Explanation: 12 = 4 + 4 + 4.
```

Example 2:

```
Input: n = 13
Output: 2
Explanation: 13 = 4 + 9.
```

Coding

```
public static int make(int n) {
        int[] dp = new int[n + 1];
        Arrays.fill(dp, 0);
        for (int i = 1; i <= n; i++) {
            int minV = i;
            int y = 1;
            int sq = 1;
            while (sq <= i) {
                minV = Math.min(minV, 1 + dp[i - sq]);
                y++;
                sq = y * y;
            }
            dp[i] = minV;
        }
        return dp[n];
    }
```

Reconstruct Itinerary

Given a list of airline tickets represented by pairs of departure and arrival airports `[from, to]`, reconstruct the itinerary in order. All of the tickets belong to a man who departs from `JFK`. Thus, the itinerary must begin with `JFK`.

Note:

1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary `["JFK", "LGA"]` has a smaller lexical order than `["JFK", "LGB"]`.
2. All airports are represented by three capital letters (IATA code).
3. You may assume all tickets form at least one valid itinerary.
4. One must use all the tickets once and only once.

Example 1:

```
Input: [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]
Output: ["JFK", "MUC", "LHR", "SFO", "SJC"]
```

Example 2:

```
Input: [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]
Output: ["JFK","ATL","JFK","SFO","ATL","SFO"]
Explanation: Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SFO"].
             But it is larger in lexical order.
```

Coding

```
public static List<String> findItinerary(List<List<String>> tickets) {
        List<String> fResult = new ArrayList<>();
        HashMap<String, PriorityQueue<String>> list = new HashMap<>();

        for (int i = 0; i < tickets.size(); i++) {
            if (!list.containsKey(tickets.get(i).get(0))) {
                PriorityQueue<String> ad = new PriorityQueue<>();
                ad.add(tickets.get(i).get(1));
                list.put(tickets.get(i).get(0), ad);
            } else {
                list.get(tickets.get(i).get(0)).add(tickets.get(i).get(1));
            }
        }
//      fResult.add("JFK");
        dfs(list, "JFK", fResult);
        return fResult;
    };

    private static void dfs(HashMap<String, PriorityQueue<String>> list,
String currentIndex, List<String> str) {
        PriorityQueue<String> arivals = list.get(currentIndex);
        while (arivals != null && !arivals.isEmpty()) {
            String X = arivals.poll();
            dfs(list, X, str);
        }
        str.add(0, currentIndex);
    }
```

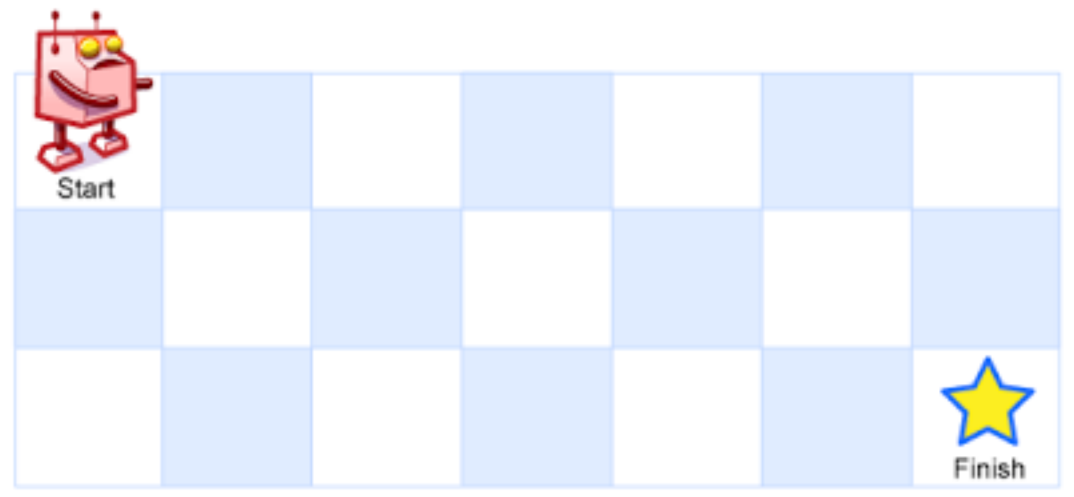
June - Week 5

Unique Paths

A robot is located at the top-left corner of a *m* x *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?



Above is a 7 x 3 grid. How many possible unique paths are there?

Example 1:

```
Input: m = 3, n = 2
Output: 3
Explanation:
From the top-left corner, there are a total of 3 ways to
reach the bottom-right corner:
1. Right -> Right -> Down
2. Right -> Down -> Right
3. Down -> Right -> Right
```

Example 2:

```
Input: m = 7, n = 3
Output: 28
```

Coding

```
public static int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];
        for (int i = 0; i < m; i++) {
            dp[i][0] = 1;
        }
        for (int i = 0; i < n; i++) {
            dp[0][i] = 1;
        }
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = dp[i][j - 1] + dp[i - 1][j];
            }
        }
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                System.out.print(dp[i][j] + " ");
            }
            System.out.println();
        }
        return dp[m - 1][n - 1];
    }
```

Word Search II

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must 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 in a word.

Example:

```
Input:
board = [
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]
words = ["oath","pea","eat","rain"]

Output: ["eat","oath"]
```

Note:

1. All inputs are consist of lowercase letters `a-z`.
2. The values of `words` are distinct.

Coding

```java
List<String> res = new ArrayList<>();

    Trie root = new Trie();
    buildTries(root, words);

    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[i].length; j++) {
            char C = board[i][j];
            if (root.Children[C - 'a'] != null)
                dfs(board, i, j, root, res);
        }
    }
    return res;
}

private static void dfs(char[][] board, int i, int j, Trie cur, List<String> res) {
    if (i < 0 || j < 0 || i > board.length - 1 || j > board[0].length - 1) {
        return;
    }

    if (board[i][j] == '#') {
        return;
    }
    char C = board[i][j];
    if (cur.Children[C - 'a'] == null) {
        return;
    }
    cur = cur.Children[C - 'a'];
    if (cur.word != null) {
        res.add(cur.word);
        cur.word = null;
    }

    board[i][j] = '#';
    dfs(board, i + 1, j, cur, res);
    dfs(board, i - 1, j, cur, res);
    dfs(board, i, j + 1, cur, res);
    dfs(board, i, j - 1, cur, res);
    board[i][j] = C;
}

private static void buildTries(Trie root, String[] words) {
    for (String w : words) {
        Trie Cur = root;
        for (char C : w.toCharArray()) {
            int index = (int) (C - 'a');
            if (Cur.Children[index] == null) {
                Cur.Children[index] = new Trie();
            }
            Cur = Cur.Children[index];
        }
        Cur.word = w;
    }
}
```