Algorithms

Last updated March 2, 2020

Contents

1 Knowledge Base **1**

1.1 Union Find . . . 1
1.2 Quick Sort . . . 1
1.3 Traverse Binary Tree . . . 2
1.3.1 Iterative Preorder Traversal . . . 2
1.3.2 Iterative Inorder Traversal . . . 2
1.3.3 Iterative Postorder Traversal . . . 3

2 Leetcode **3**

2.1 10. Regular Expression Matching . . . 3
2.2 11. Container With Most Water . . . 5
2.3 22. Generate Parentheses . . . 6
2.4 33. Search in Rotated Sorted Array . . . 7
2.5 37. Sudoku Solver . . . 8
2.6 48. Rotate Image . . . 10
2.7 51. N-Queens . . . 11
2.8 74. Search a 2D Matrix . . . 12
2.9 76. Minimum Window Substring . . . 13
2.10 98. Validate Binary Search Tree . . . 15
2.11 115. Distinct Subsequences . . . 16
2.12 124. Binary Tree Maximum Path Sum . . . 17
2.13 128. Longest Consecutive Sequence . . . 18
2.14 131. Palindrome Partitioning . . . 19
2.15 132. Palindrome Partitioning II . . . 20
2.16 134. Gas Station . . . 21
2.17 137. Single Number II . . . 22
2.18 140. Word Break II . . . 23
2.19 142. Linked List Cycle II . . . 24
2.20 146. LRU Cache . . . 25
2.21 169. Majority Element . . . 27
2.22 221. Maximal Square . . . 28
2.23 224. Basic Calculator . . . 29
2.24 236. Lowest Common Ancestor of a Binary Tree . . . 32
2.25 264. Ugly Number II . . . 33
2.26 273. Integer to English Words . . . 34
2.27 295. Find Median from Data Stream . . . 35
2.28 307. Range Sum Query - Mutable . . . 36
2.29 315. Count of Smaller Numbers After Self . . . 37
2.30 322. Coin Change . . . 39
2.31 343. Integer Break . . . 40
2.32 363. Max Sum of Rectangle No Larger Than K . . . 40
2.33 384. Shuffle an Array . . . 42
2.34 402. Remove K Digits . . . 43
2.35 403. Frog Jump . . . 43
2.36 435. Non-overlapping Intervals . . . 44

2.37 452. Minimum Number of Arrows to Burst Balloons 45
2.38 460. LFU Cache 46
2.39 470. Implement Rand10() Using Rand7() 48
2.40 480. Sliding Window Median 48
2.41 510. Inorder Successor in BST II 49
2.42 581. Shortest Unsorted Continuous Subarray 51
2.43 727. Minimum Window Subsequence 51
2.44 737. Sentence Similarity II 53
2.45 768. Max Chunks To Make Sorted II 54
2.46 850. Rectangle Area II 55
2.47 851. Loud and Rich 56
2.48 854. K-Similar Strings 57
2.49 855. Exam Room 58
2.50 857. Minimum Cost to Hire K Workers 60
2.51 858. Mirror Reflection 61
2.52 873. Length of Longest Fibonacci Subsequence 62
2.53 879. Profitable Schemes 63
2.54 947. Most Stones Removed with Same Row or Column 64
2.55 956. Tallest Billboard 65
2.56 968. Binary Tree Cameras 66
2.57 974. Subarray Sums Divisible by K 68
2.58 975. Odd Even Jump 69
2.59 984. String Without AAA or BBB 71
2.60 986. Interval List Intersections 71
2.61 992. Subarrays with K Different Integers 73
2.62 1000. Minimum Cost to Merge Stones 73
2.63 1022. Smallest Integer Divisible by K 75
2.64 1024. Video Stitching 75
2.65 1025. Divisor Game 77
2.66 1027. Longest Arithmetic Sequence 77
2.67 1029. Two City Scheduling 78
2.68 1049. Last Stone Weight II 79

3 Cracking the Code Interview 80

3.1 BST Sequences 80
3.2 Insertion 81
3.3 Count number of bits to be flipped to convert A to B 82
3.4 Magic Index 82
3.5 Boolean Parenthesization Problem 83
3.6 Find Duplicates 85
3.7 Rank of An Eelement in A Stream 86
3.8 Diving Board 87
3.9 Sum Swap 88
3.10 Number of 2s 89
3.11 The Masseuse 90

1 Knowledge Base

1.1 Union Find

```
public class UnionFind {
  int[] id;

  public UnionFind(int N) {
    this.id = new int[N];
    for (int i = 0; i < id.length; i++) {
      id[i] = i;
    }
  }

  public int find(int i) {
    if (id[i] != i)
      return find(id[i]);
    return i;
  }


  public void union(int x, int y) {
    x = find(x);
    y = find(y);
    id[x] = y;
  }
}
```

1.2 Quick Sort

```
class QuickSort {

  public static void main(String[] args) {
    int[] arr = {4, 5, 1, 2, 3, 3};
    quickSort(arr, 0, arr.length - 1);
    System.out.println(Arrays.toString(arr));
  }

  public static void quickSort(int[] arr, int start, int end) {

    int partition = partition(arr, start, end);

    if (partition - 1 > start) {
      quickSort(arr, start, partition - 1);
    }
    if (partition + 1 < end) {
      quickSort(arr, partition + 1, end);
    }
  }

  public static int partition(int[] arr, int start, int end) {
    int pivot = arr[end];

    for (int i = start; i < end; i++) {
      if (arr[i] < pivot) {
        int temp = arr[start];
        arr[start] = arr[i];
        arr[i] = temp;
```

```
        start++;
      }
    }

    int temp = arr[start];
    arr[start] = pivot;
    arr[end] = temp;

    return start;
  }
}
```

1.3 Traverse Binary Tree

```
static class Node {
  int data;
  Node left;
  Node right;

  public Node(int data) {
    this.data = data;
  }
}
```

1.3.1 Iterative Preorder Traversal

```
public static List<Integer> iterativePreorder(Node root) {
  List<Integer> res = new LinkedList<>();
  if (root == null) return res;
  Stack<Node> stack = new Stack<>();
  stack.push(root);
  while (!stack.isEmpty()) {
    Node node = stack.peek();
    res.add(node.data);
    stack.pop();
    if (node.right != null) {
      stack.push(node.right);
    }
    if (node.left != null) {
      stack.push(node.left);
    }
  }
  return res;
}
```

1.3.2 Iterative Inorder Traversal

```
public static List<Integer> iterativeInorder(Node root) {
  List<Integer> res = new LinkedList<>();
  if (root == null) return res;
  Stack<Node> stack = new Stack();
  Node curr = root;
  while (curr != null || stack.size() > 0) {
    while (curr != null) {
      stack.push(curr);
      curr = curr.left;
```

```
    }
    curr = stack.pop();
    res.add(curr.data);
    curr = curr.right;
  }
  return res;
}
```

1.3.3 Iterative Postorder Traversal

```
/**
 * Traverse the binary Tree with the following steps:
 * 1. Push root to first stack.
 * 2. Loop while first stack is not empty
 * 2.1 Pop a node from first stack and push it to second stack
 * 2.2 Push left and right children of the popped node to first stack
 * 3. Print contents of second stack
 */
public static List<Integer> postOrderIterative(Node root) {
  List<Integer> res = new LinkedList<>();
  Stack<Node> s1, s2;

  s1 = new Stack<>();
  s2 = new Stack<>();

  if (root == null)
    return res;
  s1.push(root);

  while (!s1.isEmpty()) {
    Node temp = s1.pop();
    s2.push(temp);

    if (temp.left != null)
      s1.push(temp.left);
    if (temp.right != null)
      s1.push(temp.right);
  }

  while (!s2.isEmpty()) {
    Node temp = s2.pop();
    res.add(temp.data);
  }
  return res;
}
```

2 Leetcode

2.1 10. Regular Expression Matching

Given an input string (s) and a pattern (p), implement regular expression matching with support for `'.'` and `'*'`.

`'.'` Matches any single character.

`'*'` Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).
Note:
a. s could be empty and contains only lowercase letters `'a-z'`.
b. p could be empty and contains only lowercase letters `'a-z'`, and characters like `'.'` or `'*'`.
Examples:

```
Input:
s = "aa"
p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".

Input:
s = "aa"
p = "a*"
Output: true
Explanation: '*' means zero or more of the precedeng element, 'a'. Therefore, by
repeating 'a' once, it becomes "aa".

Input:
s = "ab"
p = ".*"
Output: true
Explanation: ".*" means "zero or more (*) of any character (.)".

Input:
s = "aab"
p = "c*a*b"
Output: true
Explanation: c can be repeated 0 times, a can be repeated 1 time. Therefore it matches
"aab".

Input:
s = "mississippi"
p = "mis*is*p*."
Output: false
```

Here are some conditions to figure out, then the logic can be very straightforward.

1. `p.charAt(j) == s.charAt(i)`: `dp[i][j] = dp[i-1][j-1];`
2. `p.charAt(j) == '.'`: `dp[i][j] = dp[i-1][j-1];`
3. `p.charAt(j) == '*'`:
   1. `p.charAt(j-1) != s.charAt(i)`: `dp[i][j] = dp[i][j-2]` //in this case, a* only counts as empty
   2. `p.charAt(i-1) == s.charAt(i)` or `p.charAt(i-1) == '.'`:
      * `dp[i][j] = dp[i-1][j]` //in this case, a* counts as multiple a
      * `dp[i][j] = dp[i][j-1]` // in this case, a* counts as single a
      * `dp[i][j] = dp[i][j-2]` // in this case, a* counts as empty

```java
class Solution {
  private static final char star = '*';
  private static final char dot = '.';
```

```
  public boolean isMatch(String s, String p) {
    // initialize array to 1 length greater because we set 0,0 to true for
        empty.
    boolean[][] m = new boolean[s.length() + 1][p.length() + 1];

    // if string is empty and pattern is empty
    m[0][0] = true;

    // handle condition where pattern is a* or a*b* etc.
    for (int i = 1; i < m[0].length; i++) {
      if (p.charAt(i - 1) == star) {
        m[0][i] = m[0][i - 2];
      }
    }
    // since we start at index 1, i - 1 or j - 1 refers to current
    for (int i = 1; i < m.length; i++) {
      for (int j = 1; j < m[0].length; j++) {
        char text = s.charAt(i - 1);
        char pattern = p.charAt(j - 1);
        if (pattern == dot || pattern == text) {
          // take the previous match value
          m[i][j] = m[i - 1][j - 1];
        } else if (pattern == star) {
          char prevPattern = p.charAt(j - 2);
          // handle 0 occurences of text
          m[i][j] = m[i][j - 2];
          // if char before star is dot or char before star matches text
          if (prevPattern == dot || prevPattern == text) {
            m[i][j] = m[i][j] | m[i - 1][j];
          }
        } else {
          m[i][j] = false;
        }
      }
    }
    return m[s.length()][p.length()];
  }
}
```

2.2 11. Container With Most Water

Given n non-negative integers a1, a2, ..., an , where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

The above vertical lines are represented by array [1, 8, 6, 2, 5, 4, 8, 3, 7]. In this case, the max area of water (blue section) the container can contain is 49.

Note: You may not slant the container and n is at least 2.
Example:

```
Input: [1,8,6,2,5,4,8,3,7]
Output: 49
```

```
public int maxArea(int[] height) {
  int result = 0;
  int left = 0;
  int right = height.length - 1;
  while (left < right) {
    result = Math.max(result, Math.min(height[right],
              height[left]) * (right - left));
    if (height[left] < height[right]) {
      left++;
    } else {
      right--;
    }
  }
  return result;
}
```

2.3 22. Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.
Example:

```
Input: n = 3
Output:
[
 "((()))",
 "(()())",
 "(())()",
```

```
 "()(())",
 "()()()"
]
```

```java
public void genertateParenthesis(int left,
                                 int right,
                                 String par,
                                 List<String> res,
                                 int len) {
  if (par.length() == len * 2) {
    res.add(par);
    return;
  }
  if (left < len) {
    genertateParenthesis(left + 1, right, par + "(", res, len);
  }
  if (right < left) {
    genertateParenthesis(left, right + 1, par + ")", res, len);
  }
}

public List<String> generateParenthesis(int n) {
  List<String> result = new LinkedList();
  genertateParenthesis(0, 0, "", result, n);
  return result;
}
```

2.4 33. 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 $O(logn)$.
Examples:

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

Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

```java
/**
 * This works only when there is no duplicates in a[].
 */
public class Solution {
  int search(int[] a, int start, int end, int x) {
    if (start > end) return -1;
    while (start <= end) {
      int mid = (start + end) / 2;
      if (a[mid] == x) return mid;
      if (a[mid] < a[end]) {
        if (x >= a[mid] && x <= a[end]) start = mid + 1;
        else end = mid - 1;
      } else {
        if (x >= a[start] && x <= a[mid]) end = mid - 1;
        else start = mid + 1;
```

```
      }
    }
    return -1;
  }

  public int search(int[] nums, int target) {
    if (nums.length == 0) return -1;
    return search(nums, 0, nums.length - 1, target) ;

  }
}

/**
 *  This code works with duplicates in the array.
 */
public class Solution {
  int search(int[] a, int start, int end, int x) {
    if (start > end) return -1;
    while (start <= end) {
      int mid = (start + end) / 2;
      if (a[mid] == x) return mid;
      if (a[mid] < a[end]) {
        if (x > a[mid] && x <= a[end]) start = mid + 1;
        else end = mid - 1;
        continue;
      } else if (a[mid] > a[end]) {
        if (x >= a[start] && x < a[mid]) end = mid - 1;
        else start = mid + 1;
        continue;
      } else {
        end--;
      }
    }
    return -1;
  }

  public boolean search(int[] nums, int target) {
    if (nums.length == 0) return false;
    return search(nums, 0, nums.length - 1, target) != -1;

  }
}
```

2.5 37. Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.
A sudoku solution must satisfy all of the following rules:

1. Each of the digits `1-9` must occur exactly once in each row.
2. Each of the digits `1-9` must occur exactly once in each column.
3. Each of the the digits `1-9` must occur exactly once in each of the 9 `3x3` sub-boxes of the grid.

Empty cells are indicated by the character `'.'`.
Note:
a. The given board contain only digits `1-9` and the character `'.'`.
b. You may assume that the given Sudoku puzzle will have a single unique solution.
c. The given board size is always `9x9`.
Example

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

A sudoku puzzle and its solution

```java
public class Solution {
  private boolean check(char[][] board, int i, int j, char c) {
    for (int m = 0; m < board.length; m++) {
      if (board[i][m] == c) return false;
    }

    for (int n = 0; n < board[0].length; n++) {
      if (board[n][j] == c) return false;
    }

    int row = i - i % 3, column = j - j % 3;
    for (int m = 0; m < 3; m++) {
      for (int n = 0; n < 3; n++) {
        if (board[row + m][column + n] == c) return false;
      }
    }
    return true;
  }

  private boolean solve(char[][] board, int i, int j) {
    int m = board.length;
    int n = board[0].length;
    if (i == m) return true;
    if (j == n) return solve(board, i + 1, 0);
    if (board[i][j] != '.') return solve(board, i, j + 1);
    for (char c = '1'; c <= '9'; c++) {
      if (check(board, i, j, c)) {
        board[i][j] = c;
        if (solve(board, i, j + 1)) return true;
        board[i][j] = '.';
      }
    }
    return false;
  }
  public void solveSudoku(char[][] board) {
    solve(board, 0, 0);
  }
}
```

2.6 48. Rotate Image

You are given an n x n 2D matrix representing an image.
Rotate the image by 90 degrees (clockwise).
Note:
You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.
Example 1:

```
Given input matrix =
[
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
],

rotate the input matrix in-place such that it becomes:
[
 [7, 4, 1],
 [8, 5, 2],
 [9, 6, 3]
]
```

Example 2:

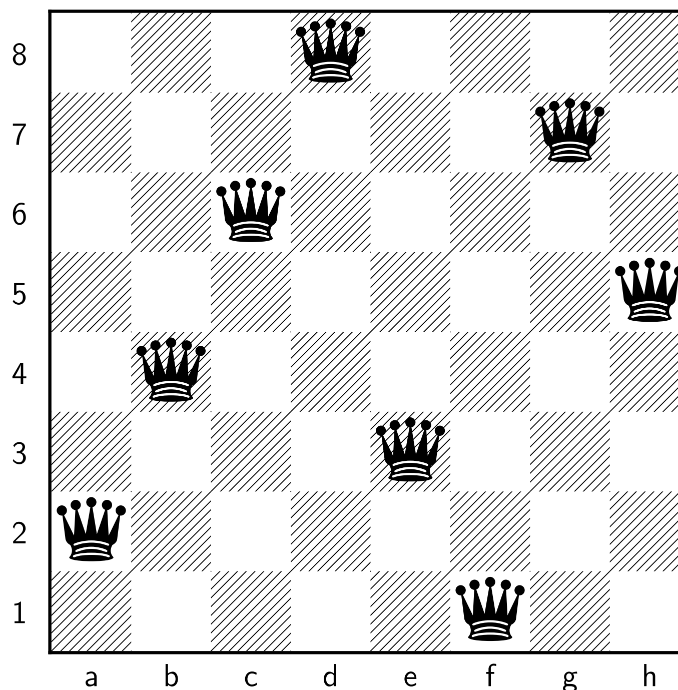
```
Given input matrix =
[
 [ 5, 1, 9, 11],
 [ 2, 4, 8, 10],
 [13, 3, 6, 7],
 [15, 14, 12, 16]
],
rotate the input matrix in-place such that it becomes:
[
 [15, 13, 2, 5],
 [14, 3, 4, 1],
 [12, 6, 8, 9],
 [16, 7, 10, 11]
]
```

```
public void rotate(int[][] matrix) {
  int n = matrix.length;
  for (int i = 0; i < n / 2; i++) {
    for (int j = i; j < n - i - 1; j++) {
      // keep top right value
      int temp = matrix[j][n - i - 1];
      // move top left to top right
      matrix[j][n - i - 1] = matrix[i][j];
      // move bottom left to top left
      matrix[i][j] = matrix[n - j - 1][i];
      // move bottom right to bottom left
      matrix[n - j - 1][i] = matrix[n - i - 1][n - j - 1];
      // move top right to bottom right
```

```
      matrix[n - i - 1][n - j - 1] = temp;
    }
  }
}
```

2.7 51. N-Queens

The n-queens puzzle is the problem of placing n queens on an `n×n` chessboard such that no two queens attack each other.



One solution to the 8-queens puzzle

Given an integer n, return all distinct solutions to the n-queens puzzle.
Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.
Example:

```
Input: 4
Output: [
 [".Q..", // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.", // Solution 2
  "Q...",
  "...Q",
```

```
 ".Q.."]
]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown
above.
```

```
public class Solution {

  public void fillResult(int[] rows, List<List<String>> res) {
    List<String> thisRes = new LinkedList();
    for (int i = 0; i < rows.length; i++) {
      char[] row = new char[rows.length];
      Arrays.fill(row, '.');
      row[rows[i]] = 'Q';
      String line = new String(row);
      thisRes.add(line);
    }
    res.add(thisRes);
  }

  public boolean valid(int[] rows, int row, int col) {
    for (int row2 = 0; row2 < row; row2++) {
      int col2 = rows[row2];
      if (col2 == col) return false;
      int distance = Math.abs(col - col2);
      if (distance == row - row2) {
        return false;
      }
    }
    return true;
  }

  public void solve(int[] rows, int row, List<List<String>> res) {
    if (row == rows.length) {
      fillResult(rows, res);
      return;
    }

    for (int i = 0; i < rows.length; i++) {
      if (valid(rows, row, i)) {
        rows[row] = i;
        solve(rows, row + 1, res);
      }
    }
  }

  public List<List<String>> solveNQueens(int n) {
    List<List<String>> res = new LinkedList();
    int[] rows = new int[n];
    solve(rows, 0, res);
    return res;
  }
}
```

2.8 74. Search a 2D Matrix

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:
a. Integers in each row are sorted from left to right.

b. The first integer of each row is greater than the last integer of the previous row.

Examples:

```
Input:
matrix = [
 [1,  3, 5, 7],
 [10, 11, 16, 20],
 [23, 30, 34, 50]
]
target = 3
Output: true

Input:
matrix = [
 [1,  3, 5, 7],
 [10, 11, 16, 20],
 [23, 30, 34, 50]
]
target = 13
Output: false
```

```
public boolean searchMatrix(int[][] matrix, int target) {
  int m = matrix.length;
  if (m < 1) return false;
  int n = matrix[0].length;
  int i = 0, j = n - 1;
  while (i < m && j >= 0) {
    if (matrix[i][j] == target) return true;
    if (matrix[i][j] < target) {
      i++;
      continue;
    } else if (matrix[i][j] > target) {
      j--;
      continue;
    }
  }
  return false;
}
```

2.9 76. Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

Note:

1. If there is no such window in S that covers all characters in T, return the empty string `""`.
2. If there is such window, you are guaranteed that there will always be only one unique minimum window in S.

Example:

```
Input: S = "ADOBECODEBANC", T = "ABC"
Output: "BANC"
```

For most substring problem, we are given a string and need to find a substring of it which satisfy some restrictions. A general way is to use a hashmap assisted with two pointers. The template [3] is given below.

```
int findSubstring(String s) {
  int[] map = new int[128];
  int counter; // check whether the substring is valid
  int begin = 0, end = 0; //two pointers, one point to tail and one  head
  int d; //the length of substring

  for () { /* initialize the hash map here */ }

  while (end < s.length()) {

    if (map[s.charAt(end++)]-- ?) {  /* modify counter here */ }

    while (/* counter condition */) {

      /* update d here if finding minimum*/

      //increase begin to make it invalid/valid again

      if (map[s.charAt(begin++)]++ ?) { /*modify counter here*/ }
    }

    /* update d here if finding maximum*/
  }
  return d;
}
```

One thing needs to be mentioned is that when asked to find maximum substring, we should update maximum after the inner while loop to guarantee that the substring is valid. On the other hand, when asked to find minimum substring, we should update minimum inside the inner while loop.
The code of solving Longest Substring with At Most Two Distinct Characters is below:

```
int lengthOfLongestSubstringTwoDistinct(String s) {
  int[] map = new int[128];
  int counter = 0, begin = 0, end = 0, d = 0;
  while (end < s.length()) {
    if (map[s.charAt(end++)]++ == 0) counter++;
    while (counter > 2) if (map[s.charAt(begin++)]-- == 1) counter--;
    d = Math.max(d, end - begin);
  }
  return d;
}
```

The code of solving Longest Substring Without Repeating Characters is below:

```
int lengthOfLongestSubstring(String s) {
  int[] map = new int[128];
  int counter = 0, begin = 0, end = 0, d = 0;
  while (end < s.length()) {
    if (map[s.charAt(end++)]++ > 0) counter++;
    while (counter > 0) {
      if (map[s.charAt(begin++)]-- > 1) counter--;
    }
    d = Math.max(d, end - begin); //while valid, update d
  }
  return d;
}
```

The code of solving Minimum Window Substring is below:

```
String minWindow(String s, String t) {
  int[] map = new int[128];
  for (char c : t.toCharArray()) map[c]++;
  int counter = t.length(), begin = 0, end = 0;
  int d = Integer.MAX_VALUE, head = 0;
  while (end < s.length()) {
    if (map[s.charAt(end++)]-- > 0) counter--;
    while (counter == 0) { //valid
      if (end - begin < d) d = end - (head = begin);
      if (map[s.charAt(begin++)]++ == 0) counter++;  //make it invalid
    }
  }
  return d == Integer.MAX_VALUE ? "" : s.substring(head, d);
}
```

2.10 98. Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).
Assume a BST is defined as follows:
a. The left subtree of a node contains only nodes with keys less than the node's key.
b. The right subtree of a node contains only nodes with keys greater than the node's key.
c. Both the left and right subtrees must also be binary search trees.
Example 1:

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

Example 2:

```
  5
 / \
 1  4
   / \
  3  6
Output: false
Explanation: The input is: [5,1,4,null,null,3,6]. The root node's value is 5 but
its right child's value is 4.
```

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 * int val;
 * TreeNode left;
 * TreeNode right;
 * TreeNode(int x) { val = x; }
 * }
 */
public boolean validateBST(TreeNode root, long min, long max) {
  if (null == root) return true;
  if (root.val <= min) return false;
  if (root.val >= max) return false;
  return validateBST(root.left, min, root.val)
```

```
        && validateBST(root.right, root.val, max);
}

public boolean isValidBST(TreeNode root) {
  return validateBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
}
```

2.11 115. Distinct Subsequences

Given a string S and a string T, count the number of distinct subsequences of S which equals 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).

Example 1:

```
Input: S = "rabbbit", T = "rabbit"
Output: 3
Explanation:
As shown below, there are 3 ways you can generate "rabbit" from S.
(The caret symbol ^ means the chosen letters)

rabbbit
^^^^ ^^
rabbbit
^^ ^^^^
rabbbit
^^^ ^^^
```

Example 2:

```
Input: S = "babgbag", T = "bag"
Output: 5
Explanation:
As shown below, there are 5 ways you can generate "bag" from S.
(The caret symbol ^ means the chosen letters)

babgbag
^^ ^
babgbag
^^    ^
babgbag
^    ^^
babgbag
  ^  ^^
babgbag
    ^^^
```

Let's first define its state `dp[i][j]` to be the number of distinct subsequences of `t[0..i-1]` in `s[0..j-1]`. Then we have the following state equations:

1. General case 1: `dp[i][j] = dp[i][j-1] if t[i-1] != s[j-1];`
2. General case 2: `dp[i][j] = dp[i][j-1] + dp[i-1][j-1] if t[i-1] == s[j-1];`

3. Boundary case 1: `dp[0][j] = 1` for all j;
4. Boundary case 2: `dp[i][0] = 0` for all positive i.

Now let's give brief explanations to the four equations above.

1. If `t[i - 1] != s[j - 1]`, the distinct subsequences will not include `s[j - 1]` and thus all the number of distinct subsequences will simply be those in `s[0..j - 2]`, which corresponds to `dp[i][j - 1]`;
2. If `t[i - 1] == s[j - 1]`, the number of distinct subsequences include two parts: those with `s[j - 1]` and those without;
3. An empty string will have exactly one subsequence in any string.
4. Non-empty string will have no subsequences in an empty string.

Putting these together, we will have the following simple codes [2].

```
int numDistinct(String s, String t) {
  int m = t.length(), n = s.length();
  int[][] dp = new int[m + 1][n + 1];
  for (int j = 0; j <= n; j++) dp[0][j] = 1;
  for (int j = 1; j <= n; j++)
    for (int i = 1; i <= m; i++)
      dp[i][j] = dp[i][j - 1] +
        (t.charAt(i - 1) == s.charAt(j - 1) ? dp[i - 1][j - 1] : 0);
  return dp[m][n];
}
```

Notice that we keep the whole `m*n` matrix simply for `dp[i - 1][j - 1]`. So we can simply store that value in a single variable and further optimize the space complexity. The final code is as follows.

```
int numDistinct(String s, String t) {
  int m = t.length(), n = s.length();
  int[] cur = new int[m + 1];
  cur[0] = 1;
  for (int j = 1; j <= n; j++) {
    int pre = 1;
    for (int i = 1; i <= m; i++) {
      int temp = cur[i];
      cur[i] = cur[i] + (t.charAt(i - 1) == s.charAt(j - 1) ? pre : 0);
      pre = temp;
    }
  }
  return cur[m];
}
```

2.12 124. 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
```

Example 2:

```
Input: [-10,9,20,null,null,15,7]

 -10
 / \
9 20
 / \
 15 7

Output: 42
```

Solution:

For every Node a, check the path sum with a as root, if sum > currentMaxSum, we can update currentMaxSum = sum. With recursion we can get all the path sum for every node.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 * int val;
 * TreeNode left;
 * TreeNode right;
 * TreeNode(int x) { val = x; }
 * }
 */
class Solution {
  int maxSum = Integer.MIN_VALUE;

  private int findMax(TreeNode root) {
    if (root == null) return 0;
    int left = findMax(root.left);
    int right = findMax(root.right);
    int sum = root.val + left + right;
    if (sum > maxSum) maxSum = sum;
    // Math.max(left, right)  because we're tracing the path, for a certain
        node,
    // only its left or right could be added to the path.
    // Math.max(Math.max(left, right) + root.val, 0),
    // if a node's path sum is negative,  we can safely ignore this node.
    return Math.max(Math.max(left, right) + root.val, 0);

  }

  public int maxPathSum(TreeNode root) {
    findMax(root);
    return maxSum;
  }
}
```

2.13 128. Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence. Your algorithm should run in $O(n)$ complexity.

Example:

```
Input: [100, 4, 200, 1, 3, 2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore
its length is 4.
```

```
public int longestConsecutive(int[] nums) {
  Set<Integer> set = new HashSet();
  for (int n : nums) set.add(n);
  int res = 0;
  for (int i : nums) {
    int count = 1;
    if (!set.contains(i - 1)) {
      int y = i + 1;
      while (set.contains(y)) y++;
      res = Math.max(res, y - i);
    }
  }
  return res;
}
```

2.14 131. Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome.
Return all possible palindrome partitioning of s.
Example:

```
Input: "aab"
Output:
[
 ["aa", "b"],
 ["a", "a", "b"]
]
```

```
class Solution {
  List<List<String>> resultLst;
  ArrayList<String> currLst;

  public List<List<String>> partition(String s) {
    resultLst = new ArrayList<>();
    currLst = new ArrayList<>();
    backTrack(s, 0);
    return resultLst;
  }

  public void backTrack(String s, int l) {
    if (currLst.size() > 0 //the initial str could be palindrome
        && l >= s.length()) {
      List<String> r = (ArrayList<String>) currLst.clone();
      resultLst.add(r);
    }
    for (int i = l; i < s.length(); i++) {
      if (isPalindrome(s, l, i)) {
        if (l == i)
          currLst.add(Character.toString(s.charAt(i)));
        else
          currLst.add(s.substring(l, i + 1));
        backTrack(s, i + 1);
        currLst.remove(currLst.size() - 1);
```

```
      }
    }
  }

  public boolean isPalindrome(String str, int l, int r) {
    if (l == r) return true;
    while (l < r) {
      if (str.charAt(l) != str.charAt(r)) return false;
      l++;
      r--;
    }
    return true;
  }
}
```

2.15 132. Palindrome Partitioning II

Given a string s, partition s such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of s.

Example:

```
Input: "aab"
Output: 1
Explanation: The palindrome partitioning ["aa","b"] could be produced using 1 cut.
```

This can be solved by two points [4]:

1. `cut[i]` is the minimum of `cut[j - 1] + 1 (j <= i)`, if `[j, i]` is palindrome.
2. If `[j, i]` is palindrome, `[j + 1, i - 1]` is palindrome, and `c[j] == c[i]`.

The 2nd point reminds us of using dp (caching).

```
public int minCut(String s) {
  char[] c = s.toCharArray();
  int n = c.length;
  int[] cut = new int[n];
  boolean[][] pal = new boolean[n][n];

  for (int i = 0; i < n; i++) {
    int min = i;
    for (int j = 0; j <= i; j++) {
      if (c[j] == c[i] && (j + 1 > i - 1 || pal[j + 1][i - 1])) {
        pal[j][i] = true;
        min = j == 0 ? 0 : Math.min(min, cut[j - 1] + 1);
      }
    }
    cut[i] = min;
  }
  return cut[n - 1];
}
```

The code below is another solution using dp:

```
class Solution {
  private void find(String s, int l, int r, int[] dp) {
    while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {
      if (l == 0) dp[r] = 0;
      else {
        dp[r] = Math.min(dp[r], dp[l - 1] + 1);
      }
      l--;
```

```
      r++;
    }
  }

  public int minCut(String s) {
    int[] dp = new int[s.length()];
    for (int i = 0; i < s.length(); i++) {
      dp[i] = i;
    }
    for (int i = 0; i < s.length(); i++) {
      find(s, i, i, dp);
      find(s, i, i + 1, dp);
    }
    return dp[s.length() - 1];

  }
}
```

2.16 134. Gas Station

There are N gas stations along a circular route, where the amount of gas at station i is `gas[i]`. You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from `station[i]` to its next `station[i+1]`. You begin the journey with an empty tank at one of the gas stations. Return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1.

Note:

a. If there exists a solution, it is guaranteed to be unique.

b. Both input arrays are non-empty and have the same length.

c. Each element in the input arrays is a non-negative integer.

Examples:

```
Input:
gas = [1,2,3,4,5]
cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 4. Your tank = 4 - 1 + 5 = 8
Travel to station 0. Your tank = 8 - 2 + 1 = 7
Travel to station 1. Your tank = 7 - 3 + 2 = 6
Travel to station 2. Your tank = 6 - 4 + 3 = 5
Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.
Therefore, return 3 as the starting index.

Input:
gas = [2,3,4]
cost = [3,4,3]
Output: -1
Explanation:
You can't start at station 0 or 1, as there is not enough gas to travel to the next
station.
```

```
Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 0. Your tank = 4 - 3 + 2 = 3
Travel to station 1. Your tank = 3 - 3 + 3 = 3
You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.
Therefore, you can't travel around the circuit once no matter where you start.
```

There are 2 ideas for this question:

1. If car starts at A and can not reach B. Any station between A and B can not reach B.(B is the first station that A can not reach.)
2. If the total number of gas is bigger than the total number of cost, there must be a solution.

```java
public int canCompleteCircuit(int[] gas, int[] cost) {
  int start = 0, total = 0, tank = 0;
  //if car fails at 'start', record the next station
  for (int i = 0; i < gas.length; i++) {
    tank += gas[i] - cost[i];
    if (tank < 0) {
      start = i + 1; //move starting position forward
      total += tank; //add the negative tank value to total
      tank = 0; //reset tank
    }
  }
  //negative total + positive tank should be 0 or more, if so we can do a
     round trip and return start
  return (total + tank < 0) ? -1 : start;
}
```

2.17 137. 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
```

The idea is using Map to count how many times a number appears [1].

```java
public int singleNumber(int[] A) {
  Map<Integer, Integer> map = new HashMap<>();
  for (int n : A) {
    int count = map.getOrDefault(n, 0);
    if (count == 2) {
      map.remove(n);
      continue;
    }
    map.put(n, count + 1);
  }
  for (int n : map.keySet()) return n;
  return -1;
}
```

2.18 140. 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:

a. The same word in the dictionary may be reused multiple times in the segmentation.

b. 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.
```

Example 3:

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

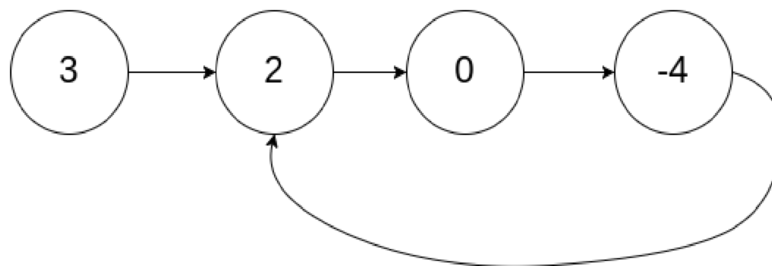
```
class Solution {
  private List<String> breakS(String s, Set<String> dic, Map<String, List<
      String>> memo) {
    if (memo.containsKey(s)) return memo.get(s);
    List<String> res = new LinkedList<>();
    if (s.length() == 0) {
      res.add("");
      return res;
    }
    for (String ds : dic) {
      if (s.startsWith(ds)) {
        String sub = s.substring(ds.length());
        List<String> subList = breakS(sub, dic, memo);
        for (String su : subList) {
          res.add(ds + (su.isEmpty() ? "" : " ") + su);
```

```
            }
        }
    }
    memo.put(s, res);
    return res;
  }

  public List<String> wordBreak(String s, List<String> wordDict) {
    List<String> res = new LinkedList();
    if (s.length() == 0) return res;
    Set<String> dic = new HashSet(wordDict);
    Map<String, List<String>> map = new HashMap<>();
    breakS(s, dic, map);
    return map.get(s);
  }
}
```

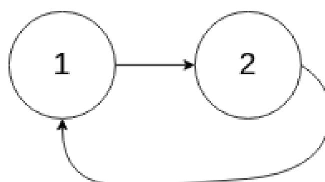
2.19 142. Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.
To represent a cycle in the given linked list, we use an integer pos which represents the position (0-indexed) in the linked list where tail connects to. If pos is -1, then there is no cycle in the linked list.
Note: Do not modify the linked list.



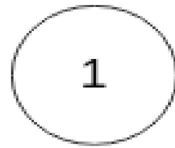
Example 1:

```
Input: head = [3, 2, 0, -4], pos = 1
Output: tail connects to node index 1
Explanation: There is a cycle in the linked list, where tail connects to the second
node.
```



Example 2:

```
Input: head = [1, 2], pos = 0
Output: tail connects to node index 0
Explanation: There is a cycle in the linked list, where tail connects to the first
node.
```



Example 3:

```
Input: head = [1], pos = -1
Output: no cycle
Explanation: There is no cycle in the linked list.
```

```
public ListNode detectCycle(ListNode head) {
  ListNode slow = head, fast = head;
  int index = 0;
  while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow == fast) {
      while (head != slow) {
        head = head.next;
        slow = slow.next;
      }
      return slow;
    }
  }
  return null;
}
```

2.20 146. 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.
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
```

```
import java.util.HashMap;
import java.util.Map;
```

```
class Node {
  int key;
  int value;
  Node pre;
  Node next;

  public Node(int key, int value) {
    this.key = key;
    this.value = value;
  }
}

class Dqueue {
  Node head = new Node(0, 0);
  Node tail = new Node(0, 0);

  public Dqueue() {
    head.next = tail;
    tail.pre = head;
    head.pre = null;
    tail.next = null;
  }

  public Node removeHead() {
    Node node = head.next;
    head.next = node.next;
    head.next.pre = head;
    return node;
  }

  public Node removeTail() {
    Node node = tail.pre;
    tail.pre = node.pre;
    node.pre.next = tail;
    return node;
  }

  public Node removeNode(Node node) {
    node.pre.next = node.next;
    node.next.pre = node.pre;
    return node;
  }

  public void addToHead(Node node) {
    node.next = this.head.next;
    node.next.pre = node;
    node.pre = head;
    head.next = node;
  }

  public void addToTail(Node node) {
    node.pre = tail.pre;
    node.pre.next = node;
    node.next = tail;
    tail.pre = node;
  }
}


public class LRUCache {
```

```
  Dqueue queue = new Dqueue();
  Map<Integer, Node> map;
  int capacity;

  public LRUCache(int capacity) {
    this.capacity = capacity;
    map = new HashMap(capacity);
  }

  public int get(int key) {
    Node node = this.map.get(key);
    if (null == node) return -1;
    this.queue.removeNode(node);
    this.queue.addToHead(node);
    return node.value;
  }

  public void put(int key, int value) {
    if (capacity == 0) {
      return;
    }
    if (-1 != this.get(key)) {
      Node node = map.get(key);
      node.value = value;
      return;
    }
    Node node = new Node(key, value);
    if (capacity == this.map.size()) {
      Node remove = this.queue.removeTail();
      map.remove(remove.key);
    }
    queue.addToHead(node);
    map.put(key, node);
  }
}

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
 */
```

2.21 169. Majority Element

Given an array of size n, find the majority element. The majority element is the element that appears more than $n/2$ times.
You may assume that the array is non-empty and the majority element always exist in the array.
Example:

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


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

```
// O(nlogn)
public int majorityElement(int[] nums) {
```

```
  Arrays.sort(nums);
  return nums[nums.length / 2];
}

// O(N) in time, O(1) space.
public class Solution {
  public static int getCandidate(int[] array) {
    int majority = 0;
    int count = 0;
    for (int n : array) {
      if (count == 0) {
        majority = n;
      }
      if (n == majority) {
        count++;
      } else {
        count--;
      }
    }
    return majority;
  }

  /**
   * If the majority do not exist, this check is necessary.
   */
  public static boolean validate(int[] array, int majority) {
    int count = 0;
    for (int n : array) {
      if (n == majority) {
        count++;
      }
    }
    return count > array.length / 2;
  }

  public static int majorityElement(int[] array) {
    int candidate = getCandidate(array);
    return validate(array, candidate) ? candidate : -1;
  }
}
```

2.22 221. 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
```

```java
/**
 * To compute M[i][j], we need to M[i][j - 1],
 * M[i - 1][j - 1] and M[i - 1][j], which are
 * the sub-problems in DP and at the top,
 * left and top-left position of M[i][j],
 * we need to iterate from left to right,
 * from top to bottom which determines how to write the 2 for loops.
 */
public int maximalSquare(char[][] matrix) {
  // corner case
  if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
    return 0;
  }

  int m = matrix.length;
  int n = matrix[0].length;

  // M[i][j] represents the length of largest square of 1's if assume matrix[
      i][j] is the bottom-right point
  int[][] M = new int[m][n];

  // initialization:
  // M[0][0]/M[0][j]/M[i][0] = matrix[0][0] - '0'

  // induction rule:
  // if matrix[i][j] = '0', then M[i][j] = 0
  // if '1', then M[i][j] = min(M[i][j - 1], M[i - 1][j - 1], M[i - 1][j]) +
      1
  int res = 0;
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      if (i == 0 || j == 0) {
        M[i][j] = matrix[i][j] - '0';
      } else {
        if (matrix[i][j] == '0') M[i][j] = 0;
        else {
          M[i][j] = Math.min(M[i][j - 1], M[i - 1][j - 1]);
          M[i][j] = Math.min(M[i][j], M[i - 1][j]) + 1;
        }
      }

      // update result
      res = Math.max(res, M[i][j] * M[i][j]);
    }
  }

  return res;
}
```

2.23 224. Basic Calculator

Implement a basic calculator to evaluate a simple expression string.
The expression string may contain open (and closing parentheses), the plus + or minus sign -, non-negative integers and empty spaces .

Note:

a. You may assume that the given expression is always valid.

b. Do not use the `eval` built-in library function.

Examples

```
Input: "1 + 1"
Output: 2

Input: " 2-1 + 2 "
Output: 3

Input: "(1+(4+5+2)-3)+(6+8)"
Output: 23
```

```
import java.util.LinkedList;


/**
 * Covert the expression to Reverse Polish Notation
 * and calculate the result on the go.
 */
class Solution {
  enum Operator {
    NO_OP('0', 0),
    ADD('+', 1),
    SUBTRACT('-', 1),
    MULTIPLY('*', 2),
    DIVIDE('/', 2),
    LEFT_BRACKET('(', 3),
    RIGHT_BRACKET(')', 3);
    char value;
    int priority;

    Operator(char c, int i) {
      this.value = c;
      this.priority = i;
    }

    static Operator createFrom(char c) {
      if (c == ADD.value) return ADD;
      if (c == SUBTRACT.value) return SUBTRACT;
      if (c == MULTIPLY.value) return MULTIPLY;
      if (c == DIVIDE.value) return DIVIDE;
      if (c == LEFT_BRACKET.value) return LEFT_BRACKET;
      if (c == RIGHT_BRACKET.value) return RIGHT_BRACKET;
      return NO_OP;
    }

    static Integer apply(Operator operator, int a, int b) {
      if (operator == ADD) return a + b;
      if (operator == SUBTRACT) return b - a;
      if (operator == MULTIPLY) return a * b;
      if (operator == DIVIDE) return b / a;
      return null;
    }
  }

  static int calculate(String s) {
    s = s.replaceAll(" ", "");
    if (s == null) return 0;
    LinkedList<Operator> operators = new LinkedList<>();
    LinkedList<Integer> result = new LinkedList<>();
    int num = 0;
    int sig = 1;
```

```
    boolean haveNumer = false;
    for (int i = 0; i < s.length(); i++) {
      char c = s.charAt(i);
      if (Character.isDigit(c)) {
        num = num * 10 + c - '0';
        haveNumer = true;
        if (i != s.length() - 1) continue;
      }
      if (c == '-') {
        if (!haveNumer) {
          sig = -1;
          continue;
        }
      }

      Operator operator = Operator.createFrom(c);
      if (haveNumer) {
        result.addFirst(sig * num);
        sig = 1;
        num = 0;
        haveNumer = false;
      }


      if (operator == Operator.RIGHT_BRACKET) {
        while (true) {
          Operator op = operators.pollFirst();
          if (op == Operator.LEFT_BRACKET) break;
          int a = result.pollFirst();
          int b = result.pollFirst();
          result.addFirst(Operator.apply(op, a, b));
        }
        continue;
      }

      if (operators.peekFirst() == Operator.LEFT_BRACKET) {
        operators.addFirst(operator);
        continue;
      }

      while (!operators.isEmpty()) {
        if (operators.peekFirst().priority < operator.priority) break;
        if (operators.peekFirst() == Operator.LEFT_BRACKET) break;
        Operator toOp = operators.pollFirst();
        int a = result.pollFirst();
        int b = result.pollFirst();
        result.addFirst(Operator.apply(toOp, a, b));
      }
      if (operator != Operator.NO_OP) operators.addFirst(operator);
    }

    while (!operators.isEmpty()) {
      Operator op = operators.pollFirst();
      int a = result.pollFirst();
      int b = result.pollFirst();
      result.addFirst(Operator.apply(op, a, b));
    }
    return result.pollFirst();
  }
```

```
  public static void main(String[] args) {
    System.out.println(calculate("6*(5+(2+3)*8+3)"));
    System.out.println(calculate("-6*(5  +(2+3)*8+3)"));
    System.out.println(calculate("10*4/   5"));
    System.out.println(calculate("1-1"));
    System.out.println(calculate("1-2"));
  }
}
```

2.24 236. Lowest Common Ancestor of a Binary Tree

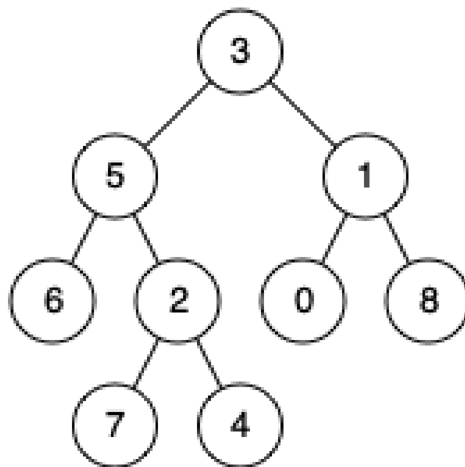
Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree. According to the definition of LCA on Wikipedia: ”The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Note:

a. All of the nodes’ values will be unique.

b. p and q are different and both values will exist in the binary tree.

Given the following binary tree: `root = [3, 5, 1, 6, 2, 0, 8, null, null, 7, 4]`



Example 1:

```
Input: root = [3, 5, 1, 6, 2, 0, 8, null, null, 7, 4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
```

Example 2:

```
Input: root = [3, 5, 1, 6, 2, 0, 8, null, null, 7, 4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself
according to the LCA definition.
```

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 * int val;
```

```
 * TreeNode left;
 * TreeNode right;
 * TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public boolean contains(TreeNode root, TreeNode p) {
    if (root == null) return false;
    if (root == p) return true;
    return contains(root.left, p) || contains(root.right, p);

  }

  public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
      {
    if (root == null || root == p || root == q) return root;
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);
    if (left != null && right != null) return root;
    return left != null ? left : right;
  }
}
```

2.25 264. 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.
Note:
a. 1 is typically treated as an ugly number.
b. n does not exceed 1690.
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.
```

```
class Solution {
  public int nthUglyNumber(int n) {
    Queue<Long> queue2 = new LinkedList();
    Queue<Long> queue3 = new LinkedList();
    Queue<Long> queue5 = new LinkedList();
    queue2.add(1l);
    long min = Integer.MAX_VALUE;

    for (int i = 0; i < n; i++) {
      long v2 = queue2.isEmpty() ? Long.MAX_VALUE : queue2.peek();
      long v3 = queue3.isEmpty() ? Long.MAX_VALUE : queue3.peek();
      long v5 = queue5.isEmpty() ? Long.MAX_VALUE : queue5.peek();
      min = Math.min(v2, Math.min(v3, v5));
      if (v2 == min) {
        queue2.remove();
        if (2 * min < Integer.MAX_VALUE)
          queue2.add(2 * min);
        if (3 * min < Integer.MAX_VALUE)
          queue3.add(3 * min);
      }

      if (v3 == min) {
```

```
        queue3.remove();
        if (3 * min < Integer.MAX_VALUE)
          queue3.add(3 * min);
      } else if (v5 == min) {
        queue5.remove();
      }
      queue5.add(5 * min);
    }
    return (int) min;
  }
}

// simpler solution based on previous code
class Solution {
  public int nthUglyNumber(int n) {
    TreeSet<Long> set = new TreeSet();
    set.add(1l);
    for (int i = 0; i < n-1; i++) {
      long min = set.pollFirst();
      set.add(2 * min);
      set.add(3 * min);
      set.add(5 * min);
    }
    return set.pollFirst().intValue();
  }
}
```

2.26 273. Integer to English Words

Convert a non-negative integer to its english words representation. Given input is guaranteed to be less than $2^{31} - 1$.

Examples

```
Input: 123
Output: "One Hundred Twenty Three"

Input: 12345
Output: "Twelve Thousand Three Hundred Forty Five"

Input: 1234567
Output: "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"

Input: 1234567891
Output: "One Billion Two Hundred Thirty Four Million Five Hundred Sixty Seven
Thousand Eight Hundred Ninety One"
```

```
class Solution {
  private static String[] bigs = new String[]{"", "Thousand", "Million", "
      Billion"};
  private static String[] tens = new String[]{"", "", "Twenty", "Thirty", "
      Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};
  private static String[] teens = new String[]{"Ten", "Eleven", "Twelve", "
      Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "
      Nineteen"};
  private static String[] digits = new String[]{"", "One", "Two", "Three", "
      Four", "Five", "Six", "Seven", "Eight", "Nine"};
```

```
  public String to100String(int num) {
    String res = "";
    if (num >= 100) {
      res = digits[num / 100] + " Hundred ";
      num = num % 100;
    }

    if (num >= 10 && num <= 19) {
      res = res + teens[num - 10] + " ";
      return res;
    } else if (num >= 20 && num <= 99) {
      res = res + tens[num / 10] + " ";
      num = num % 10;
    }
    if (num > 0 && num <= 9) {
      res = res + digits[num] + " ";
    }
    return res;
  }

  public String numberToWords(int num) {
    if (num == 0) return "Zero";
    if (num < 0) return "Negative " + numberToWords(-1 * num);
    String res = "";
    int count = 0;
    while (num > 0) {
      if (num % 1000 != 0) {
        res = to100String(num % 1000) + bigs[count] + " " + res;
      }
      count++;
      num = num / 1000;
    }
    return res.trim();
  }
}
}
```

2.27 295. Find Median from Data Stream

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.
For example,
`[2, 3, 4]`, the median is 3.
`[2, 3]`, the median is `(2 + 3) / 2 = 2.5`.
Design a data structure that supports the following two operations:
a. `void addNum(int num)` - Add a integer number from the data stream to the data structure.
b. `double findMedian()` - Return the median of all elements so far.
Example:

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

```
class MedianFinder {
  PriorityQueue<Integer> asce;
  PriorityQueue<Integer> desc;

  /**
   * initialize your data structure here.
   */
  public MedianFinder() {
    asce = new PriorityQueue<>((n1, n2) -> n1 - n2);
    desc = new PriorityQueue<>((n1, n2) -> n2 - n1);
  }

  public void addNum(int num) {
    asce.offer(num);
    desc.offer(asce.poll());
    if (desc.size() > asce.size()) {
      asce.offer(desc.poll());
    }
  }

  public double findMedian() {
    if ((asce.size() + desc.size()) % 2 == 1) {
      return asce.peek();
    } else {
      return (asce.peek() + desc.peek()) / 2.0;
    }
  }
}
```

2.28 307. Range Sum Query - Mutable

Given an integer array nums, find the sum of the elements between indices i and j (i <= j), inclusive.
The `update(i, val)` function modifies nums by updating the element at index i to val.
Note:
a. The array is only modifiable by the `update(i, val)` function.
b. You may assume the number of calls to update and sumRange function is distributed evenly.
Example:

```
Given nums = [1, 3, 5]
sumRange(0, 2) -> 9
update(1, 2)
sumRange(0, 2) -> 8
```

```
class Node {
  int start;
  int end;
  int sum;
  Node left;
  Node right;

  public Node(int start, int end) {
    this.start = start;
    this.end = end;
  }
}
```

```
class SegmentTree {
  static Node build(int start, int end) {
    if (end < start) return null;
    Node root = new Node(start, end);
    if (start == end) return root;
    int mid = (start + end) / 2;
    Node left = build(start, mid);
    Node right = build(mid + 1, end);
    root.left = left;
    root.right = right;
    return root;
  }

  static void update(Node root, int pos, int val) {
    if (root.start == root.end && root.start == pos) {
      root.sum = val;
      return;
    }
    int mid = (root.start + root.end) / 2;
    if (pos <= mid) {
      update(root.left, pos, val);
    } else {
      update(root.right, pos, val);
    }
    root.sum = root.left.sum + root.right.sum;
  }

  static int query(Node root, int start, int end) {
    if (start > root.end || end < root.start) return 0;
    if (start <= root.start && end >= root.end) return root.sum;
    return query(root.left, start, end) + query(root.right, start, end);
  }
}

public class NumArray {
  Node root = null;

  public NumArray(int[] nums) {
    root = SegmentTree.build(0, nums.length - 1);
    for (int i = 0; i < nums.length; i++) {
      SegmentTree.update(root, i, nums[i]);
    }
  }

  public void update(int i, int val) {
    SegmentTree.update(root, i, val);
  }

  public int sumRange(int i, int j) {
    return SegmentTree.query(root, i, j);
  }
}
```

2.29 315. Count of Smaller Numbers After Self

You are given an integer array nums and you have to return a new counts array. The counts array has the property where `counts[i]` is the number of smaller elements to the right of `nums[i]`.
Example:

```
Input: [5, 2, 6, 1]
Output: [2, 1, 1, 0]
Explanation:
To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.
```

The code below [5] uses SegmentTree solving this problem.

```
class Node {
  int start;
  int end;
  int count; // initial cound would be zero.
  Node left;
  Node right;

  public Node(int start, int end) {
    this.start = start;
    this.end = end;
  }
}

class SegmentTree {

  static Node build(int start, int end) {
    if (start > end) return null;
    Node root = new Node(start, end);
    if (start == end) return root;
    int mid = (start + end) / 2;
    root.left = build(start, mid);
    root.right = build(mid + 1, end);
    return root;
  }

  static void update(Node root, int pos) {
    if (root.start == root.end) {
      root.count++;
      return;
    }
    int mid = (root.start + root.end) / 2;
    if (pos <= mid) {
      update(root.left, pos);
    } else {
      update(root.right, pos);
    }
    root.count = root.left.count + root.right.count;
  }

  static int query(Node root, int pos) {
    if (root.start == root.end) return 0;
    int mid = (root.start + root.end) / 2;
    int left = 0;
    int right = 0;
    if (pos > mid) {
      right = query(root.right, pos);
      left = root.left.count;
    } else {
      left = query(root.left, pos);
    }
```

```
    return left + right;
  }
}

class Solution {
  public List<Integer> countSmaller(int[] nums) {
    int[] sortNums = Arrays.copyOf(nums, nums.length);
    Arrays.sort(sortNums);
    int[] index = new int[nums.length];
    for (int i = 0; i < nums.length; i++) {
      index[i] = Arrays.binarySearch(sortNums, nums[i]);
    }
    Integer[] count = new Integer[nums.length];
    Node root = SegmentTree.build(0, nums.length);
    for (int i = nums.length - 1; i >= 0; i--) {
      count[i] = SegmentTree.query(root, index[i]);
      SegmentTree.update(root, index[i]);
    }
    return Arrays.asList(count);
  }
}
```

2.30 322. Coin Change

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, `return -1`.
Note: You may assume that you have an infinite number of each kind of coin. **Example:**

```
Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1

Input: coins = [2], amount = 3
Output: -1
```

```
public class Solution {
  public int change(int[] coins, int amount, Map<Integer, Integer> map) {
    if (amount == 0) return 0;
    if (map.containsKey(amount)) return map.get(amount);
    int min = Integer.MAX_VALUE;
    for (int coin : coins) {
      if (coin <= amount) {
        int change = change(coins, amount - coin, map);
        if (change != -1) min = Math.min(min, change);
      }
    }
    min = min == Integer.MAX_VALUE ? -1 : 1 + min;
    map.put(amount, min);
    return map.get(amount);
  }

  public int coinChange(int[] coins, int amount) {
    return change(coins, amount, new HashMap());
  }
}

// dp
```

```
public class Solution {
  public int coinChange(int[] coins, int amount) {
    if (amount == 0) return 0;
    long[] minCounts = new long[amount + 1];
    for (int i = 1; i <= amount; ++i) {
      minCounts[i] = Integer.MAX_VALUE;
      for (int c : coins) {
        minCounts[i] = Math.min(minCounts[i], i >= c ?
                minCounts[i - c] + 1 : Integer.MAX_VALUE);
      }
    }
    return minCounts[amount] == Integer.MAX_VALUE ? -1 :
            (int) minCounts[amount];
  }
}
```

2.31 343. Integer Break

Given a positive integer n, break it into the sum of at least two positive integers and maximize the product of those integers. Return the maximum product you can get.
Note: You may assume that n is not less than 2 and not larger than 58.
Example(s)

```
Input: 2
Output: 1
Explanation: 2 = 1 + 1, 1 × 1 = 1.

Input: 10
Output: 36
Explanation: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36.
```

```
public int integerBreak(int n) {
  int[] dp = new int[n + 1];
  dp[1] = 1;
  for (int i = 2; i <= n; i++) {
    for (int j = 0; j < i; j++) {
      dp[i] = Math.max(dp[i],
              Math.max(j, dp[j]) * Math.max(i - j, dp[i - j]));
    }
  }
  return dp[n];
}
```

2.32 363. Max Sum of Rectangle No Larger Than K

Given a non-empty 2D matrix and an integer k, find the max sum of a rectangle in the matrix such that its sum is no larger than k.
Note:
a. The rectangle inside the matrix must have an area > 0.
b. What if the number of rows is much larger than the number of columns?
Example:

```
Input: matrix = [[1, 0, 1], [0, -2, 3]], k = 2
Output: 2
```

```
Explanation:
Because the sum of rectangle [[0, 1], [-2, 3]] is 2,
and 2 is the max number no larger than k (k = 2).
```

```
import java.util.TreeSet;

class Solution {
  public int maxSumSubmatrix(int[][] matrix, int k) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return
         0;
    int m = matrix.length, n = matrix[0].length;
    int res = Integer.MIN_VALUE;
    /**
     l   r      colSum
     [1,  0,  1]    [1]
     [0, -2,  3]    [-2]
     [-2, 1, -1]    [-1]
     l = 0, r = 1
     sub-array: [-2, -1] --> sub-matrix
     [0, -2]
     [-2 ,1]
     */

    for (int left = 0; left < n; left++) {
      int[] colSum = new int[m];
      for (int right = left; right < n; right++) {
        for (int i = 0; i < m; i++) {
          colSum[i] += matrix[i][right];
        }
        res = Math.max(res, largestSumNoLargerThanK(colSum, k));
      }
    }
    return res;
  }

  public int largestSumNoLargerThanK(int[] nums, int k) {
    TreeSet<Integer> set = new TreeSet();
    int sum = 0, res = Integer.MIN_VALUE;
    set.add(0);
    for (int num : nums) {
      sum += num;
      //find the smallest sum greater than or equal to (sum-k)
      Integer ceiling = set.ceiling(sum - k);
      /**
       ceiling >= sum-k, sum-ceiling <= k
       the smaller the ceiling, the larger (sum-ceiling),
       by property of TreeSet, ceiling is the smallest among all values >=
           sum-k,
       (sum-ceiling) must be the greatest among all values no larger than k
       */
      if (ceiling != null) {
        res = Math.max(res, sum - ceiling);
      }
      set.add(sum);
    }
    return res;
  }
}
```

2.33 384. Shuffle an Array

Shuffle a set of numbers without duplicates.
Example:

```
// Init an array with set 1, 2, and 3.
int[] nums = {1, 2, 3};
Solution solution = new Solution(nums);

// Shuffle the array [1, 2, 3] and return its result. Any permutation of [1, 2, 3] must
equally likely to be returned.
solution.shuffle();

// Resets the array back to its original configuration [1, 2, 3].
solution.reset();

// Returns the random shuffling of array [1, 2, 3].
solution.shuffle();
```

```
/**
 * https://leetcode.com/problems/shuffle-an-array/discuss/262867/
 */
class Solution {
  int[] original;
  Random random;
  int[] nums;

  public Solution(int[] numArray) {
    original = numArray;
    nums = numArray.clone();
    random = new Random();
  }

  /**
   * Resets the array to its original configuration and return it.
   */
  public int[] reset() {
    return original;
  }

  /**
   * Returns a random shuffling of the array.
   */
  public int[] shuffle() {
    for (int i = 0; i < nums.length; i++) {
      int temp = random.nextInt(nums.length - i);
      int tmp = nums[temp + i];
      nums[temp + i] = nums[i];
      nums[i] = tmp;
    }
    return nums;
  }
}
```

2.34 402. Remove K Digits

Given a non-negative integer num represented as a string, remove k digits from the number so that the new number is the smallest possible.

Note:

a. The length of num is less than 10002 and will be >= k.

b. The given num does not contain any leading zero.

Example 1:

```
Input: num = "1432219", k = 3
Output: "1219"
Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which is
the smallest.
```

Example 2:

```
Input: num = "10200", k = 1
Output: "200"
Explanation: Remove the leading 1 and the number is 200. Note that the output must not
contain leading zeroes.
```

Example 3:

```
Input: num = "10", k = 2
Output: "0"
Explanation: Remove all the digits from the number and it is left with nothing which
is 0.
```

A greedy method using stack, $O(n)$ time and $O(n)$ space:

```
public class Solution {
  public String removeKdigits(String num, int k) {
    int digits = num.length() - k;
    char[] stk = new char[num.length()];
    int top = 0;
    // k keeps track of how many characters we can remove
    // if the previous character in stk is larger than the current one
    // then removing it will get a smaller number
    // but we can only do so when k is larger than 0
    for (int i = 0; i < num.length(); ++i) {
      char c = num.charAt(i);
      while (top > 0 && stk[top - 1] > c && k > 0) {
        top -= 1;
        k -= 1;
      }
      stk[top++] = c;
    }
    // find the index of first non-zero digit
    int idx = 0;
    while (idx < digits && stk[idx] == '0') idx++;
    return idx == digits ? "0" : new String(stk, idx, digits - idx);
  }
}
```

2.35 403. Frog Jump

A frog is crossing a river. The river is divided into x units and at each unit there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of stones' positions (in units) in sorted ascending order, determine if the frog is able to cross the river by landing on the last stone. Initially, the frog is on the first stone and assume the first jump must be 1 unit.
If the frog's last jump was k units, then its next jump must be either k - 1, k, or k + 1 units. Note that the frog can only jump in the forward direction.
Note:
a. The number of stones is >= 2 and is < 1,100.
b. Each stone's position will be a non-negative integer $< 2^{31}$.
c. The first stone's position is always 0.
Examples:

```
[0,1,3,5,6,8,12,17]

There are a total of 8 stones.
The first stone at the 0th unit, second stone at the 1st unit,
third stone at the 3rd unit, and so on...
The last stone at the 17th unit.

Return true. The frog can jump to the last stone by jumping
1 unit to the 2nd stone, then 2 units to the 3rd stone, then
2 units to the 4th stone, then 3 units to the 6th stone,
4 units to the 7th stone, and 5 units to the 8th stone.


[0,1,2,3,4,8,9,11]

Return false. There is no way to jump to the last stone as
the gap between the 5th and 6th stone is too large.
```

```
public boolean canCross(int[] stones) {
  Map<Integer, Set<Integer>> map = new HashMap<>();
  for (int i = 0; i < stones.length; i++) {
    map.put(stones[i], new HashSet<Integer>());
  }
  map.get(0).add(0);
  for (int i = 0; i < stones.length; i++) {
    for (int k : map.get(stones[i])) {
      for (int step = k - 1; step <= k + 1; step++) {
        if (step > 0 && map.containsKey(stones[i] + step)) {
          map.get(stones[i] + step).add(step);
        }
      }
    }
  }
  return map.get(stones[stones.length - 1]).size() > 0;
}
```

2.36 435. Non-overlapping Intervals

Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Note:
a. You may assume the interval's end point is always bigger than its start point.
b. Intervals like [1, 2] and [2, 3] have borders "touching" but they don't overlap each other.
Example(s)

```
Input: [ [1,2], [2,3], [3,4], [1,3] ]
Output: 1
Explanation: [1,3] can be removed and the rest of intervals are non-overlapping.

Input: [ [1,2], [1,2], [1,2] ]
Output: 2
Explanation: You need to remove two [1, 2] to make the rest of intervals non-
overlapping.

Input: [ [1,2], [2,3] ]
Output: 0
Explanation: You don't need to remove any of the intervals since they're already
non-overlapping.
```

```
class Solution {
  public int eraseOverlapIntervals(int[][] intervals) {
    if (null == intervals || intervals.length == 0) return 0;
    int nonOverlap = 0;
    Arrays.sort(intervals, (a, b) -> a[1] - b[1]);
    //Arrays.stream(intervals).forEach(i -> System.out.println(i[0]+","+i[1])
        );
    int end = Integer.MIN_VALUE;
    for (int[] interval : intervals) {
      if (interval[0] >= end) {
        nonOverlap++;
        end = interval[1];
      }
    }
    return intervals.length - nonOverlap;
  }
}
```

2.37 452. Minimum Number of Arrows to Burst Balloons

There are a number of spherical balloons spread in two-dimensional space. For each balloon, provided input is the start and end coordinates of the horizontal diameter. Since it's horizontal, y-coordinates don't matter and hence the x-coordinates of start and end of the diameter suffice. Start is always smaller than end. There will be at most 104 balloons.
An arrow can be shot up exactly vertically from different points along the x-axis. A balloon with x[start] and x[end] bursts by an arrow shot at x if `x[start] <= x <= x[end]`. There is no limit to the number of arrows that can be shot. An arrow once shot keeps travelling up infinitely. The problem is to find the minimum number of arrows that must be shot to burst all balloons.
Example

```
Input:
[[10,16], [2,8], [1,6], [7,12]]
Output:
2
```

```
Explanation:
One way is to shoot one arrow for example at x = 6 (bursting the balloons [2,8] and
[1,6]) and another arrow at x = 11 (bursting the other two balloons).
```

```
class Solution {
  public int findMinArrowShots(int[][] points) {
    if (points == null || points.length == 0) return 0;
    Arrays.sort(points, (a, b) -> a[1] - b[1]);
    int end = points[0][1];
    int count = 1;
    for (int i = 1; i < points.length; i++) {
      if (points[i][0] > end) {
        count++;
        end = points[i][1];
      }
    }
    return count;
  }
}
```

2.38 460. LFU Cache

Design and implement a data structure for Least Frequently Used (LFU) 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 reaches its capacity, it should invalidate the least frequently used item before inserting a new item. For the purpose of this problem, when there is a tie (i.e., two or more keys that have the same frequency), the least recently used key would be evicted.
Follow up: Could you do both operations in O(1) time complexity?
Example:

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

```
import java.util.HashMap;
import java.util.LinkedHashSet;
import java.util.Map;

class Node {
  int key;
  int value;
```

```
  int count = 1;

  public Node(int key, int value) {
    this.key = key;
    this.value = value;
  }
}

public class LFUCache {
  Map<Integer, Node> map;
  Map<Integer, LinkedHashSet<Node>> countMap;
  int min = 1;
  int capacity = 0;

  public LFUCache(int capacity) {
    this.capacity = capacity;
    map = new HashMap();
    countMap = new HashMap();
    countMap.put(1, new LinkedHashSet());
  }

  public int get(int key) {
    Node node = this.map.get(key);
    if (null == node) return -1;

    countMap.get(node.count).remove(node);

    if (min == node.count && countMap.get(min).size() == 0) min += 1;
    node.count += 1;

    if (null == countMap.get(node.count)) {
      countMap.put(node.count, new LinkedHashSet());
    }

    countMap.get(node.count).add(node);

    return node.value;
  }

  public void put(int key, int value) {
    if (capacity <= 0) {
      return;
    }

    if (-1 != this.get(key)) {
      Node node = map.get(key);
      node.value = value;
      return;
    }

    Node node = new Node(key, value);
    if (capacity == this.map.size()) {
      Node remove = countMap.get(min).iterator().next();
      countMap.get(min).remove(remove);
      map.remove(remove.key);
    }
    min = 1;
    map.put(key, node);
    countMap.get(1).add(node);
```

```
  }
}
```

2.39 470. Implement Rand10() Using Rand7()

Note:
a. rand7() is predefined.
b. Each testcase has one argument: n, the number of times that rand10 is called.
Examples

```
Input: 1
Output: [7]

Input: 2
Output: [8, 4]

Input: 3
Output: [8, 1, 10]
```

```
/**
 *      0    1    2    3    4    5    6
 * 0    0    1    2    3    4    5    6
 * 1    7    8    9    10   11   12   13
 * 2    14   15   16   17   18   19   20
 * 3    21   22   23   24   25   26   27
 * 4    28   29   30   31   32   33   34
 * 5    35   36   37   38   39   40   41
 * 6    42   43   44   45   46   47   48
 *
 * We just do the mod operation in the first 40 element.
 *
 *      0    1    2    3    4    5    6
 * 0    0    1    2    3    4    5    6
 * 1    7    8    9    0    1    2    3
 * 2    4    5    6    7    8    9    0
 * 3    1    2    3    4    5    6    7
 * 4    8    9    0    1    2    3    4
 * 5    5    6    7    8    9    0    X
 * 6    X    X    X    X    X    X    X
 */
public int rand10() {
  while (true) {
    int num = (rand7() - 1) * 7 + (rand7() - 1);
    if (num < 40) {
      return num % 10 + 1;
    }
  }
}
```

2.40 480. Sliding Window Median

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.
Examples

`[2,3,4] , the median is 3`

`[2,3], the median is (2 + 3) / 2 = 2.5`
Given an array nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k(k is always valid, ie: k is always smaller than input array's size for non-empty array.) numbers in the window. Each time the sliding window moves right by one position. Your job is to output the median array for each window in the original array.
Example:

```
Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.
Window position        Median
---------------        -----
[1 3 -1] -3 5 3 6 7    1
 1 [3 -1 -3] 5 3 6 7   -1
 1 3 [-1 -3 5] 3 6 7   -1
 1 3 -1 [-3 5 3] 6 7   3
 1 3 -1 -3 [5 3 6] 7   5
 1 3 -1 -3 5 [3 6 7]   6
Therefore, return the median sliding window as [1,-1,-1,3,5,6].
```

The idea [8] is simple:
1. Use two PriorityQueue, lo and hi, (lo is in reverse order).
2. Make sure numbers in hi is always bigger than or equal to numbers in `lo`. (`hi.peek()>=lo.peek()` is always true) `hi.size()==lo.size()` when k is even, `hi.size()==lo.size()+1` when k is odd.

```java
public double[] medianSlidingWindow(int[] nums, int k) {
  double[] result = new double[nums.length - k + 1];
  int index = 0, limit = (k + 1) / 2, k1 = k - 1;
  PriorityQueue<Integer> lo = new PriorityQueue(Collections.reverseOrder());
  PriorityQueue<Integer> hi = new PriorityQueue<>();
  for (int i = 0; i < nums.length; i++) {
    hi.add(nums[i]);
    if (lo.size() > 0 && lo.peek() > hi.peek()) {
      lo.add(hi.poll());
      hi.add(lo.poll());
    }
    if (hi.size() > limit) lo.add(hi.poll());
    if (i >= k1) {
      result[index++] = hi.size() > lo.size() ? hi.peek() : (hi.peek() / 2.0
          + lo.peek() / 2.0);
      int remove = nums[i - k + 1];
      if (remove < hi.peek()) lo.remove(remove);
      else hi.remove(remove);
    }
  }
  return result;
}
```

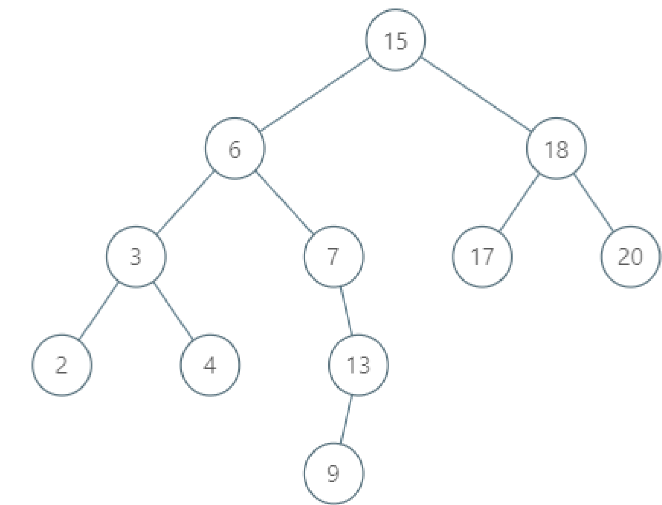
2.41 510. Inorder Successor in BST II

Given a binary search tree and a node in it, find the in-order successor of that node in the BST. The successor of a node p is the node with the smallest key greater than `p.val`.

You will have direct access to the node but not to the root of the tree. Each node will have a reference to its parent node.

Note:

a. If the given node has no in-order successor in the tree, return null.
b. It's guaranteed that the values of the tree are unique.
c. Remember that we are using the Node type instead of TreeNode type so their string representation are different.



Example:

```
Input: The tree showed in the picture.
p = 13
Output: 15
```

```
/*
// Definition for a Node.
class Node {
    public int val;
    public Node left;
    public Node right;
    public Node parent;
};
*/
class Solution {
  Node leftMost(Node x) {
    while (x.left != null) x = x.left;
    return x;
  }

  public Node inorderSuccessor(Node x) {
    if (x == null) return null;
    if (x.right != null) return leftMost(x.right);
    Node parent = x.parent;
    while (parent != null && x == parent.right) {
      x = parent;
      parent = x.parent;
    }
    return parent;
  }
}
```

2.42 581. Shortest Unsorted Continuous Subarray

Given an integer array, you need to find one continuous subarray that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order, too.
You need to find the **shortest** such subarray and output its length.
Note:
1. Then length of the input array is in range `[1, 10,000]`.
2. The input array may contain duplicates, so ascending order here means <=.
Example:

```
Input: [2, 6, 4, 8, 10, 9, 15]
Output: 5
Explanation: You need to sort [6, 4, 8, 10, 9] in ascending order to make the whole
array sorted in ascending order.
```

```
public int findUnsortedSubarray(int[] nums) {
  if (nums.length == 0) return 0;
  int n = nums.length, begin = -1, end = -2, max = nums[0], min = nums[n -
      1];
  for (int i = 1; i < n; i++) {
    max = Math.max(max, nums[i]);
    min = Math.min(min, nums[n - 1 - i]);
    if (nums[i] < max) end = i;
    if (nums[n - 1 - i] > min) begin = n - 1 - i;
  }
  return end - begin + 1;
}

// Anthoer simpler way: sort the list and check if it's
// still the same number in the list.
```

2.43 727. Minimum Window Subsequence

Given strings S and T, find the minimum (contiguous) substring W of S, so that T is a subsequence of W.
If there is no such window in S that covers all characters in T, return the empty string "". If there are multiple such minimum-length windows, return the one with the left-most starting index.
Note:
a. All the strings in the input will only contain lowercase letters.
b. The length of S will be in the range [1, 20000].
c. The length of T will be in the range [1, 100].
Example 1:

```
Input:
S = "abcdebdde", T = "bde"
Output: "bcde"
Explanation:
"bcde" is the answer because it occurs before "bdde" which has the same length.
"deb" is not a smaller window because the elements of T in the window must occur in
order.
```

There are 2 ways to solve this problem, one is DP [9] the other is using 2 pointers [10].
`dp[i][j]` stores the starting index of the substring where T has length i and S has length j.

So `dp[i][j]` would be:
1. if `T[i - 1] == S[j - 1]`, this means we could borrow the start index from `dp[i - 1][j - 1]` to make the current substring valid.
2. else, we only need to borrow the start index from `dp[i][j - 1]` which could either exist or not.
Finally, go through the last row to find the substring with min length and appears first.

```
public String minWindow(String S, String T) {
  int m = T.length(), n = S.length();
  int[][] dp = new int[m + 1][n + 1];
  for (int j = 0; j <= n; j++) {
    dp[0][j] = j + 1;
  }
  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      if (T.charAt(i - 1) == S.charAt(j - 1)) {
        dp[i][j] = dp[i - 1][j - 1];
      } else {
        dp[i][j] = dp[i][j - 1];
      }
    }
  }

  int start = 0, len = n + 1;
  for (int j = 1; j <= n; j++) {
    if (dp[m][j] != 0) {
      if (j - dp[m][j] + 1 < len) {
        start = dp[m][j] - 1;
        len = j - dp[m][j] + 1;
      }
    }
  }
  return len == n + 1 ? "" : S.substring(start, start + len);
}
```

The idea of 2 pointers is simple and based on two pointer:
1. check feasibility.
2. check optimization.

```
public String minWindow(String S, String T) {
  char[] s = S.toCharArray(), t = T.toCharArray();
  int sindex = 0, tindex = 0, slen = s.length, tlen = t.length, start = -1,
      len = slen;
  while (sindex < slen) {
    if (s[sindex] == t[tindex]) {
      if (++tindex == tlen) {
        //check feasibility from left to right of T
        int end = sindex + 1;
        //check optimization from right to left of T
        while (--tindex >= 0) {
          while (s[sindex--] != t[tindex]) ;
        }
        ++sindex;
        ++tindex;
        //record the current smallest candidate
        if (end - sindex < len) {
          len = end - sindex;
          start = sindex;
        }
      }
    }
```

```
    ++sindex;
  }
  return start == -1 ? "" : S.substring(start, start + len);
}
```

2.44 737. Sentence Similarity II

Given two sentences words1, words2 (each represented as an array of strings), and a list of similar word pairs pairs, determine if two sentences are similar.
For example, `words1 = ["great", "acting", "skills"]` and `words2 = ["fine", "drama", "talent"]` are similar, if the similar word pairs are `pairs = [["great", "good"], ["fine", "good"], ["acting","drama"], ["skills","talent"]]`.
Note that the similarity relation is transitive. For example, if "great" and "good" are similar, and "fine" and "good" are similar, then "great" and "fine" are similar.
Similarity is also symmetric. For example, "great" and "fine" being similar is the same as "fine" and "great" being similar.
Also, a word is always similar with itself. For example, the sentences `words1 = ["great"], words2 = ["great"], pairs = []` are similar, even though there are no specified similar word pairs.
Finally, sentences can only be similar if they have the same number of words. So a sentence like `words1 = ["great"]` can never be similar to `words2 = ["doubleplus","good"]`.
Note:
a. The length of words1 and words2 will not exceed 1000.
b. The length of pairs will not exceed 2000.
c. The length of each `pairs[i]` will be 2.
d. The length of each `words[i]` and `pairs[i][j]` will be in the range `[1, 20]`.

```
class Solution {

  Map<String, String> map = new HashMap();

  String findParent(String a) {
    map.putIfAbsent(a, a);
    while (a != map.get(a)) a = map.get(a);
    return map.get(a);
  }

  void union(String a, String b) {
    String pa = findParent(a);
    String pb = findParent(b);
    if (pa != pb) map.put(pa, pb);
  }

  public boolean areSentencesSimilarTwo(String[] words1, String[] words2,
      String[][] pairs) {
    if (words1.length != words2.length) return false;

    Map<String, Set<String>> map = new HashMap();
    for (String[] pair : pairs) {
      String s1 = pair[0];
      String s2 = pair[1];
      union(s1, s2);
    }

    for (int i = 0; i < words1.length; i++) {
      String s1 = words1[i];
```

```
      String s2 = words2[i];
      String pa = findParent(s1);
      String pb = findParent(s2);
      if (pa != pb) return false;
    }
    return true;
  }
}
```

2.45 768. Max Chunks To Make Sorted II

Given an array of integers (not necessarily distinct), we split the array into some number of "chunks" (partitions), and individually sort each chunk. After concatenating them, the result equals the sorted array.
What is the most number of chunks we could have made?

Note:

a. `arr` will have length in range `[1, 2000]`.
b. `arr[i]` will be an integer in range `[0, 10**8]`.

Example(s)

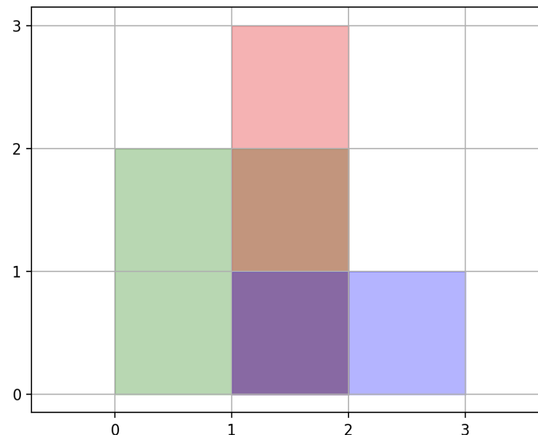
```
Input: arr = [5, 4, 3, 2, 1]
Output: 1
Explanation:
Splitting into two or more chunks will not return the required result.
For example, splitting into [5, 4], [3, 2, 1] will result in [4, 5, 1, 2, 3], which
isn't sorted.

Input: arr = [2, 1, 3, 4, 4]
Output: 4
Explanation:
We can split into two chunks, such as [2, 1], [3, 4, 4].
However, splitting into [2, 1], [3], [4], [4] is the highest number of chunks
possible.
```

```
class Solution {
  public int maxChunksToSorted(int[] arr) {
    Stack<Integer> stack = new Stack();
    for (int i : arr) {
      if (stack.isEmpty() || i >= stack.peek()) {
        stack.push(i);
      } else {
        int max = stack.pop();
        while (!stack.isEmpty() && stack.peek() > i)
          stack.pop();
        stack.push(max);
      }
    }
    return stack.size();
  }
}
```

2.46 850. Rectangle Area II

We are given a list of (axis-aligned) rectangles. Each `rectangle[i] = [x1, y1, x2, y2]`, where (x1, y1) are the coordinates of the bottom-left corner, and (x2, y2) are the coordinates of the top-right corner of the ith rectangle.
Find the total area covered by all rectangles in the plane. Since the answer may be too large, return it modulo $10^9 + 7$.



Note:
a. 1 <= rectangles.length <= 200
b. rectanges[i].length = 4
c. $0 <=$ rectangles[i][j] $<= 10^9$
d. The total area covered by all rectangles will never exceed $2^{63} - 1$ and thus will fit in a 64-bit signed integer.

Example 1:

```
Input: [[0, 0, 2, 2], [1, 0, 2, 3],[1, 0, 3, 1]]
Output: 6
Explanation: As illustrated in the picture.
```

Example 2:

```
Input: [[0, 0, 1000000000, 1000000000]]
Output: 49
Explanation: The answer is 10^18 modulo (10^9 + 7), which is (10^9)^2 = (-7)^2 = 49.
```

```
class Solution {
  public int rectangleArea(int[][] rectangles) {
    Set<Integer> xs = new HashSet();
    Set<Integer> ys = new HashSet();
    for (int[] r : rectangles) {
      xs.add(r[0]);
      xs.add(r[2]);
      ys.add(r[1]);
      ys.add(r[3]);
    }
    List<Integer> xsort = new ArrayList<>(xs);
    List<Integer> ysort = new ArrayList<>(ys);
```

```java
    Collections.sort(xsort);
    Collections.sort(ysort);
    Map<Integer, Integer> xmap = new HashMap<>();
    Map<Integer, Integer> ymap = new HashMap<>();
    for (int i = 0; i < xsort.size(); i++) {
      xmap.put(xsort.get(i), i);
    }

    for (int i = 0; i < ysort.size(); i++) {
      ymap.put(ysort.get(i), i);
    }
    boolean[][] grid = new boolean[xmap.size()][ymap.size()];
    for (int[] r : rectangles) {
      for (int x = xmap.get(r[0]); x < xmap.get(r[2]); x++) {
        for (int y = ymap.get(r[1]); y < ymap.get(r[3]); y++) {
          grid[x][y] = true;
        }

      }
    }
    long res = 0;
    for (int i = 0; i < grid.length; i++) {
      for (int j = 0; j < grid[0].length; j++) {
        if (grid[i][j]) {
          res += (long) (xsort.get(i + 1) - xsort.get(i)) * (ysort.get(j + 1)
              - ysort.get(j));
        }
      }
    }
    return (int) (res % 1000000007);
  }
}
```

2.47 851. Loud and Rich

In a group of N people (labelled $0, 1, 2, \cdots, N-1$), each person has different amounts of money, and different levels of quietness. For convenience, we'll call the person with label x, simply "person x".

We'll say that `richer[i] = [x, y]` if person x definitely has more money than person y. Note that richer may only be a subset of valid observations.

Also, we'll say `quiet[x] = q` if person x has quietness q.

Now, return answer, where `answer[x] = y` if y is the least quiet person (that is, the person y with the smallest value of `quiet[y]`), among all people who definitely have equal to or more money than person x.

Note:

a. 1 <= quiet.length = N <= 500

b. 0 <= quiet[i] < N, all quiet[i] are different.

c. 0 <= richer.length <= N * (N - 1) / 2

d. 0 <= richer[i][j] < N

e. richer[i][0] != richer[i][1]

f. richer[i]'s are all different.

g. The observations in richer are all logically consistent.

Example 1:

```
Input: richer = [[1,0],[2,1],[3,1],[3,7],[4,3],[5,3],[6,3]], quiet = [3,2,
5,4,6,1,7,0]
Output: [5,5,2,5,4,5,6,7]
Explanation:
answer[0] = 5.
Person 5 has more money than 3, which has more money than 1, which has more money than
0.
The only person who is quieter (has lower quiet[x]) is person 7, but it isn't clear if
they have more money than person 0.
answer[7] = 7.
Among all people that definitely have equal to or more money than person 7 (which
could be persons 3, 4, 5, 6, or 7), the person who is the quietest (has lower quiet[x])
is person 7.
```

```
private int dfs(int target, List<List<Integer>> tree, int[] quiet, int[] res)
    {
  if (res[target] >= 0) return res[target];
  res[target] = target;
  for (int i : tree.get(target)) {
    if (quiet[res[target]] > quiet[dfs(i, tree, quiet, res)]) res[target] =
        res[i];
  }
  return res[target];
}

public int[] loudAndRich(int[][] richer, int[] quiet) {
  ArrayList<List<Integer>> tree = new ArrayList<>();
  for (int i = 0; i < quiet.length; i++) tree.add(new LinkedList<>());
  for (int[] a : richer) {
    tree.get(a[1]).add(a[0]);
  }

  int[] res = new int[quiet.length];
  Arrays.fill(res, -1);

  for (int i = 0; i < res.length; i++) {
    dfs(i, tree, quiet, res);
  }

  return res;
```

2.48 854. K-Similar Strings

Strings A and B are K-similar (for some non-negative integer K) if we can swap the positions of two letters in A exactly K times so that the resulting string equals B.
Given two anagrams A and B, return the smallest K for which A and B are K-similar. **Note:**
a. `1 <= A.length == B.length <= 20`
b. A and B contain only lowercase letters from the set {'a', 'b', 'c', 'd', 'e', 'f'}
Example 1:

```
Input: A = "ab", B = "ba"
Output: 1
```

Example 2:

```
Input: A = "abc", B = "bca"
```

```
Output: 2
```

Example 3:

```
Input: A = "abac", B = "baca"
Output: 2
```

Example 4:

```
Input: A = "aabc", B = "abca"
Output: 2
```

```
class Solution {
  private void swap(char[] A, int i, int j) {
    char tmp = A[i];
    A[i] = A[j];
    A[j] = tmp;
  }

  private int dfs(char[] A, String B, int i, Map<String, Integer> map) {
    String s = new String(A);
    if (s.equals(B)) return 0;
    if (map.containsKey(s)) return map.get(s);
    int min = Integer.MAX_VALUE;
    while (i < A.length && A[i] == B.charAt(i)) i++;
    for (int j = i + 1; j < B.length(); j++) {
      if (A[j] == B.charAt(i)) {
        swap(A, i, j);
        int next = dfs(A, B, i + 1, map);
        if (next != Integer.MAX_VALUE) {
          min = Math.min(next + 1, min);
        }
        swap(A, i, j);
      }
    }
    map.put(s, min);
    return min;

  }

  public int kSimilarity(String A, String B) {
    return dfs(A.toCharArray(), B, 0, new HashMap());
  }
}
```

2.49 855. Exam Room

In an exam room, there are N seats in a single row, numbered $0, 1, 2, \cdots, N-1$. When a student enters the room, they must sit in the seat that maximizes the distance to the closest person. If there are multiple such seats, they sit in the seat with the lowest number. (Also, if no one is in the room, then the student sits at seat number 0.).
Return a class `ExamRoom(int N)` that exposes two functions: `ExamRoom.seat()` returning an int representing what seat the student sat in, and `ExamRoom.leave(int p)` representing that the student in seat number p now leaves the room. It is guaranteed that any calls to `ExamRoom.leave (p)` have a student sitting in seat p.

Note:

a. $1 <= N <= 10^9$

b. `ExamRoom.seat()` and `ExamRoom.leave()` will be called at most 10^4 times across all test cases.

Example 1:

Input: ["ExamRoom", "seat", "seat", "seat", "seat", "leave", "seat"], [[10], [], [], [], [], [4], []]
Output: [null, 0, 9, 4, 2, null, 5]
Explanation:
ExamRoom(10) -> null
seat() -> 0, no one is in the room, then the student sits at seat number 0.
seat() -> 9, the student sits at the last seat number 9.
seat() -> 4, the student sits at the last seat number 4.
seat() -> 2, the student sits at the last seat number 2.
leave(4) -> null
seat() -> 5, the student sits at the last seat number 5.

The idea is quite straight forward: use a list L to record the index of seats where people sit.
`seat()`:
a. find the biggest distance at the start, at the end and in the middle.
b. insert index of seat
c. return index
`leave(p)`: pop out p

Time Complexity: $O(N)$ for `seat()` and `leave()`.

```
class ExamRoom {
  TreeSet<Integer> seats;
  int n;

  public ExamRoom(int N) {
    this.n = N;
    seats = new TreeSet();
  }

  public int seat() {
    int pos = 0;

    if (seats.size() > 0) {
      int dist = seats.first();
      int prev = -1;
      for (int i : seats) {
        if (prev != -1) {
          int d = (i - prev) / 2;
          if (d > dist) {
            dist = d;
            pos = prev + d;
          }
        }
        prev = i;
      }
      if (n - 1 - seats.last() > dist) {
        pos = n - 1;
      }

    }
    seats.add(pos);
    return pos;

  }
```

```
  public void leave(int p) {
    seats.remove(p);
  }
}
```

2.50 857. Minimum Cost to Hire K Workers

There are N workers. The ith worker has a quality[i] and a minimum wage expectation wage[i]. Now we want to hire exactly K workers to form a paid group. When hiring a group of K workers, we must pay them according to the following rules:

1. Every worker in the paid group should be paid in the ratio of their quality compared to other workers in the paid group.
2. Every worker in the paid group must be paid at least their minimum wage expectation.

Return the least amount of money needed to form a paid group satisfying the above conditions.

Example 1:

```
Input: quality = [10, 20, 5], wage = [70, 50, 30], K = 2
Output: 105.00000
Explanation: We pay 70 to 0-th worker and 35 to 2-th worker.
```

Example 2:

```
Input: quality = [3, 1, 10, 10, 1], wage = [4, 8, 2, 2, 7], K = 3
Output: 30.66667
Explanation: We pay 4 to 0-th worker, 13.33333 to 2-th and 3-th workers seperately.
```

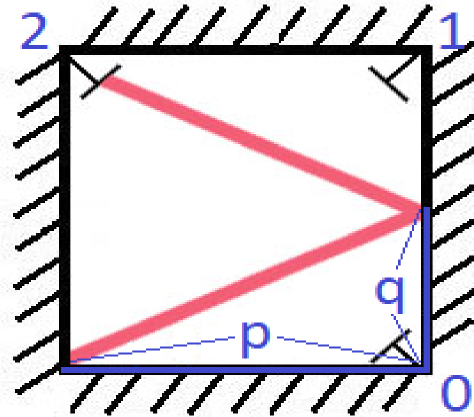
Note:

1. $1 <= K <= N <= 10000$, where N = quality.length = wage.length
2. $1 <= quality[i] <= 10000$
3. $1 <= wage[i] <= 10000$
4. Answers within 10^{-5} of the correct answer will be considered correct.

```
class Solution {
  public double mincostToHireWorkers(int[] q, int[] w, int K) {
    double[][] workers = new double[w.length][2];
    for (int i = 0; i < w.length; i++) {
      workers[i] = new double[]{((double) w[i]) / q[i],
              (double) q[i]};
    }
    Arrays.sort(workers, (a, b) ->
            Double.compare(a[0], b[0]));
    double res = Double.MAX_VALUE, qsum = 0;
    PriorityQueue<Double> pq = new PriorityQueue<>();
    for (double[] worker : workers) {
      qsum += worker[1];
      pq.add(-worker[1]);
      if (pq.size() > K) {
        qsum -= pq.poll();
      }
      if (pq.size() == K) {
        res = Math.min(res, qsum * worker[0]);
      }
    }
    return res;
  }
}
```

2.51 858. Mirror Reflection

There is a special square room with mirrors on each of the four walls. Except for the southwest corner, there are receptors on each of the remaining corners, numbered 0, 1, and 2. The square room has walls of length p, and a laser ray from the southwest corner first meets the east wall at a distance q from the 0th receptor. Return the number of the receptor that the ray meets first. (It is guaranteed that the ray will meet a receptor eventually.)



Example 1:

```
Input: p = 2, q = 1
Output: 2
Explanation: The ray meets receptor 2 the first time it gets reflected back to the
left wall.
```

a. 1 <= p <= 1000
b. 0 <= q <= p

```
class Solution {
  /**
   * 不管光线在里面怎么折射, 光线在水平方向走p,那么在垂直方向必定走q. 找最小
     公倍数。
   * |-0
   * | |
   * 2-1
   * | |
   * |-0
   * | |
   * 2-1
   * | |
   * |-0
   */
  public int mirrorReflection(int p, int q) {
    int cur = 0;
    for (int i = 1; ; i++) {

      cur += q;
      cur %= 2 * p;

      if (cur == 0) {
        return 0;
      }
      if (cur == p) {
```

```
        if (i % 2 == 1) {
          return 1;
        }
        return 2;
      }
    }
  }
}
```

2.52 873. Length of Longest Fibonacci Subsequence

A sequenceX `X_1, X_2, ..., X_n` is fibonacci-like if:

- $n >= 3$
- $X_i + X_{i+1} = X_{i+2}$ for all $i + 2 <= n$

Given a **strictly increasing** array A of positive integers forming a sequence, find the length of the longest fibonacci-like subsequence of A. If one does not exist, return 0.
(Recall that a subsequence is derived from another sequence A by deleting any number of elements (including none) from A, without changing the order of the remaining elements. For example, `[3, 5, 8]` is a subsequence of `[3, 4, 5, 6, 7, 8]`.)

Note:

a. `3 <= A.length <= 1000`

b. `1 <= A[0] < A[1] < ... < A[A.length - 1] <= 10^9`

Example 1:

```
Input: [1,2,3,4,5,6,7,8]
Output: 5
Explanation:
The longest subsequence that is fibonacci-like: [1,2,3,5,8].
```

Example 2:

```
Input: [1,3,7,11,12,14,18]
Output: 3
Explanation:
The longest subsequence that is fibonacci-like:
[1,11,12], [3,11,14] or [7,11,18].
```

```
public int lenLongestFibSubseq(int[] A) {
  Set<Integer> set = new HashSet();
  for (int i : A) {
    set.add(i);
  }
  int max = 0;
  for (int i = 0; i < A.length; i++) {
    int ai = A[i];
    for (int j = i + 1; j < A.length; j++) {
      int first = ai;
      int second = A[j];
      int len = 0;
      while (set.contains(first + second)) {
        second = first + second;
        first = second - first;
        len++;
      }
      if (len > 0) max = Math.max(len + 2, max);
    }
```

```
  }
  return max;
}
```

2.53 879. Profitable Schemes

There are G people in a gang, and a list of various crimes they could commit.
The i-th crime generates a profit[i] and requires group[i] gang members to participate.
If a gang member participates in one crime, that member can't participate in another crime.
Let's call a profitable scheme any subset of these crimes that generates at least P profit, and the total number of gang members participating in that subset of crimes is at most G.
How many schemes can be chosen? Since the answer may be very large, return it modulo `10^9 + 7`.

Note:

1. `1 <= G <= 100`
2. `0 <= P <= 100`
3. `1 <= group[i] <= 100`
4. `0 <= profit[i] <= 100`
5. `1 <= group.length = profit.length <= 100`

Example(s)

```
Input: G = 5, P = 3, group = [2, 2], profit = [2, 3]
Output: 2
Explanation:
To make a profit of at least 3, the gang could either commit crimes 0 and 1, or just
crime 1.
In total, there are 2 schemes.

Input: G = 10, P = 5, group = [2, 3, 5], profit = [6, 7, 8]
Output: 7
Explanation:
To make a profit of at least 5, the gang could commit any crimes, as long as they commit
one.
There are 7 possible schemes: (0), (1), (2), (0, 1), (0, 2), (1, 2), and (0, 1, 2).
```

```
class Solution {
  /**
   * d[k][g][v]: means:
   * the number of schemes by using 1, 2, .., k plan given g persons
   * to get more than v profits the result will be d[K][G][P].
   * due to some plan's profit is 0, we need to consider 3 cases .
   */
  public int profitableSchemes(int G, int P, int[] group, int[] profit) {
    int K = group.length;
    int V = P;
    int MOD = 1_000_000_007;
    //  given g person,  create more the v profit
    int[][][] d = new int[K + 1][G + 1][V + 1];
    for (int k = 1; k <= K; ++k) {
      for (int g = 1; g <= G; ++g) {
        int need_person = group[k - 1];
        int get_value = profit[k - 1];
        for (int v = 0; v <= V; ++v) {
```

```
          d[k][g][v] = 0;
          // case 0, only use plan[k]
          if (v <= get_value && g >= need_person) {
            d[k][g][v] += 1;
          }

          // case 1: not use plan[k]
          d[k][g][v] += (k < 1 ? 0 : d[k - 1][g][v]) % MOD;

          // case 2: use plan[k] and use plan before
          if (g > need_person) {
            d[k][g][v] += (k < 1 ? 0 : d[k - 1][g - need_person][Math.max(0,
                v - get_value)]) % MOD;
          }
          d[k][g][v] %= MOD;
        }
      }
    }
    int sum = d[K][G][P];
    return sum;
  }
}
```

2.54 947. Most Stones Removed with Same Row or Column

On a 2D plane, we place stones at some integer coordinate points. Each coordinate point may have at most one stone.

Now, a move consists of removing a stone that shares a column or row with another stone on the grid.

What is the largest possible number of moves we can make?

Note:

a. `1 <= stones.length <= 1000`

b. `0 <= stones[i][j] < 10000`

Examples:

```
Input: stones = [[0, 0], [0, 1], [1, 0],[1, 2], [2, 1], [2, 2]]
Output: 5

Input: stones = [[0, 0], [0,2], [1, 1], [2, 0], [2, 2]]
Output: 3

Input: stones = [[0,0]]
Output: 0
```

```
class Solution {
  public int removeStones(int[][] stones) {
    int[] roots = new int[20000];
    for (int i = 0; i < 20000; ++i) {
      roots[i] = i;
    }
    for (int[] stone : stones) {
      roots[findRoot(roots, stone[0])] = findRoot(roots, stone[1] + 10000);
    }
    Set<Integer> seen = new HashSet<>();
```

```
    for (int[] stone : stones) {
      seen.add(findRoot(roots, stone[0]));
    }
    return stones.length - seen.size();
  }

  private int findRoot(int[] roots, int x) {
    if (roots[x] != x) {
      roots[x] = findRoot(roots, roots[x]);
    }
    return roots[x];
  }
}
```

2.55 956. Tallest Billboard

You are installing a billboard and want it to have the largest height. The billboard will have two steel supports, one on each side. Each steel support must be an equal height.
You have a collection of rods which can be welded together. For example, if you have rods of lengths 1, 2, and 3, you can weld them together to make a support of length 6.
Return the largest possible height of your billboard installation. If you cannot support the billboard, return 0.

Note:

1. 0 <= rods.length <= 20
2. 1 <= rods[i] <= 1000
3. The sum of rods is at most 5000

Examples

```
Input: [1, 2, 3, 6]
Output: 6
Explanation: We have two disjoint subsets {1,2,3} and {6}, which have the same sum =
6.

Input: [1, 2, 3, 4, 5, 6]
Output: 10
Explanation: We have two disjoint subsets {2,3,5} and {4,6}, which have the same sum
= 10.

Input: [1, 2]
Output: 0
Explanation: The billboard cannot be supported, so we return 0.
```

Explanation

`dp[d]` means the maximum pair of sum we can get with pair difference d. For example, if have a pair of sum (a, b) with `a > b`, then `dp[a - b] = b`. If we have `dp[diff] = a`, it means we have a pair of sum `(a, a + diff)`. And this is the biggest pair with `difference = a`.
`dp` is initialized with `dp[0] = 0`;

```
Assume we have an initial state like this
------- y ------|----- d -----|
------- y ------|
```

```
case 1
If put x to tall side
------- y ------|----- d -----|----- x -----|
------- y ------|

We update dp[d + x] = max(dp[d + x], y )

case 2.1
Put x to low side and d >= x:
-------y------|----- d -----|
-------y------|--- x ---|

We update dp[d-x] = max( dp[d - x], y + x)

case 2.2
Put x to low side and d < x:
------- y ------|----- d -----|
------- y ------|------- x -------|
We update dp[x - d] = max(dp[x - d], y + d)

case 2.1 and case2.2 can merge into dp[abs(x - d)] = max(dp[abs(x - d)], y + min(d, x))
```

Time Complexity: `O(NM)`, where we have:

- `N = rod.length <= 20`
- `S = sum(rods) <= 5000`
- `M = all possible sum = min(3^N, S)`

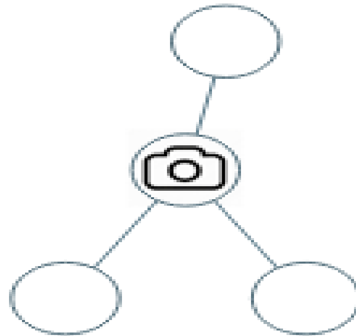
There are 3 ways to arrange a number: in the first group, in the second, not used.
The number of difference will be less than `3^N`. The only case to reach `3^N` is when `rod = [1, 3, 9, 27, 81...]`.

```java
class Solution {
  public int tallestBillboard(int[] rods) {
    Map<Integer, Integer> dp = new HashMap<>(), cur;
    dp.put(0, 0);
    for (int x : rods) {
      cur = new HashMap<>(dp);
      for (int d : cur.keySet()) {
        dp.put(d + x, Math.max(cur.get(d),
                dp.getOrDefault(x + d, 0)));
        dp.put(Math.abs(d - x), Math.max(cur.get(d) + Math.min(d, x),
                dp.getOrDefault(Math.abs(d - x), 0)));
      }
    }
    return dp.get(0);
  }
}
```

2.56 968. Binary Tree Cameras

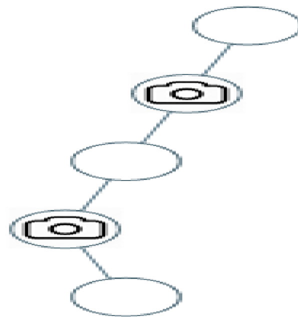
Given a binary tree, we install cameras on the nodes of the tree.
Each camera at a node can monitor **its parent, itself, and its immediate children**.

Calculate the minimum number of cameras needed to monitor all nodes of the tree.
Note:
a. The number of nodes in the given tree will be in the range [1, 1000].
b. Every node has value 0.
Example 1:



```
Input: [0, 0, null, 0, 0]
Output: 1
Explanation: One camera is enough to monitor all nodes if placed as shown.
```

Example 2:



```
Input: [0, 0, null, 0, null, 0, null, null, 0]
Output: 2
Explanation: At least two cameras are needed to monitor all nodes of the tree. The
above image shows one of the valid configurations of camera placement.
```

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 * int val;
 * TreeNode left;
 * TreeNode right;
 * TreeNode(int x) { val = x; }
 * }
 */
class Solution {
  private int NOT_MONITORED = 0;
  private int MONITORED_NOCAM = 1;
  private int MONITORED_WITHCAM = 2;
  private int cameras = 0;
```

```
  int dfs(TreeNode root) {
    if (root == null) return MONITORED_NOCAM;
    int left = dfs(root.left);
    int right = dfs(root.right);
    if (left == MONITORED_NOCAM && right == MONITORED_NOCAM) {
      return NOT_MONITORED;
    } else if (left == NOT_MONITORED || right == NOT_MONITORED) {
      cameras++;
      return MONITORED_WITHCAM;
    } else {
      return MONITORED_NOCAM;
    }
  }

  public int minCameraCover(TreeNode root) {
    if (dfs(root) == NOT_MONITORED) cameras++;
    return cameras;
  }
```

2.57 974. Subarray Sums Divisible by K

Given an array A of integers, return the number of (contiguous, non-empty) subarrays that have a sum divisible by K.

Note:

a. `1 <= A.length <= 30000`

b. `-10000 <= A[i] <= 10000`

c. `2 <= K <= 10000`

Example

```
Input: A = [4, 5, 0, -2, -3, 1], K = 5
Output: 7
Explanation: There are 7 subarrays with a sum divisible by K = 5:
[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]
```

```
/**
 * from {@link https://leetcode.com/problems/subarray-sums-divisible-by-k/
   discuss/217980/}
 * if j > i and sum[0..i] % K == sum[0..j] % K,
 * then sum[i + 1, j] is divisible by by K.
 * So for current index j, we need to find out how many indexes i (i < j)
   that have the same mod of K.
 **/

class Solution {
  public int subarraysDivByK(int[] a, int k) {
    int sum = 0;
    int[] map = new int[k];
    // sum[0] = 0
    map[0] = 1;
    int count = 0;
    for (int i : a) {
      sum = ((sum + i) % k + k) % k;
      count += map[sum];
      map[sum]++;
    }
    return count;
```

```
  }
}
```

2.58 975. Odd Even Jump

You are given an integer array A. From some starting index, you can make a series of jumps. The (1st, 3rd, 5th, ...) jumps in the series are called odd numbered jumps, and the (2nd, 4th, 6th, ...) jumps in the series are called even numbered jumps.
You may from index i jump forward to index j (with i < j) in the following way:

- During odd numbered jumps (ie. jumps 1, 3, 5, ...), you jump to the index j such that `A[i] <= A[j]` and `A[j]` is the smallest possible value. If there are multiple such indexes j, you can only jump to the smallest such index j.
- During even numbered jumps (ie. jumps 2, 4, 6, ...), you jump to the index j such that `A[i] >= A[j]` and `A[j]` is the largest possible value. If there are multiple such indexes j, you can only jump to the smallest such index j.
- It may be the case that for some index i, there are no legal jumps.

A starting index is good if, starting from that index, you can reach the end of the array (index `A.length - 1`) by jumping some number of times (possibly 0 or more than once.)
Return the number of good starting indexes.

Note:

a. `1 <= A.length <= 20000`
b. `0 <= A[i] < 100000`

Example

```
Input: [10, 13, 12, 14, 15]
Output: 2
Explanation:
From starting index i = 0, we can jump to i = 2 (since A[2] is the smallest among A[1],
A[2], A[3], A[4] that is greater or equal to A[0]), then we can't jump any more.
From starting index i = 1 and i = 2, we can jump to i = 3, then we can't jump any more.
From starting index i = 3, we can jump to i = 4, so we've reached the end.
From starting index i = 4, we've reached the end already.
In total, there are 2 different starting indexes (i = 3, i = 4) where we can reach the
end with some number of jumps.

Input: [2, 3, 1, 1, 4]
Output: 3
Explanation:
From starting index i = 0, we make jumps to i = 1, i = 2, i = 3:

During our 1st jump (odd numbered), we first jump to i = 1 because A[1] is the smallest
value in (A[1], A[2], A[3], A[4]) that is greater than or equal to A[0].

During our 2nd jump (even numbered), we jump from i = 1 to i = 2 because A[2] is the
largest value in (A[2], A[3], A[4]) that is less than or equal to A[1]. A[3] is also
the largest value, but 2 is a smaller index, so we can only jump to i = 2 and not i = 3.
```

```
During our 3rd jump (odd numbered), we jump from i = 2 to i = 3 because A[3] is the
smallest value in (A[3], A[4]) that is greater than or equal to A[2].

We can't jump from i = 3 to i = 4, so the starting index i = 0 is not good.

In a similar manner, we can deduce that:
From starting index i = 1, we jump to i = 4, so we reach the end.
From starting index i = 2, we jump to i = 3, and then we can't jump anymore.
From starting index i = 3, we jump to i = 4, so we reach the end.
From starting index i = 4, we are already at the end.
In total, there are 3 different starting indexes (i = 1, i = 3, i = 4) where we can reach
the end with some number of jumps.

Input: [5, 1, 3, 4, 2]
Output: 3
Explanation:
We can reach the end from starting indexes 1, 2, and 4.
```

We [12] need to jump higher and lower alternately to the end.
Take `[5, 1, 3, 4, 2]` as example.
If we start at 2,
we can jump either higher first or lower first to the end,
because we are already at the end.
$\text{higher}(2) = \text{true}$
$\text{lower}(2) = \text{true}$
If we start at 4,
we can't jump higher, $\text{higher}(4) = \text{false}$
we can jump lower to 2, $\text{lower}(4) = \text{higher}(2) = \text{true}$
If we start at 3,
we can jump higher to 4, $\text{higher}(3) = \text{lower}(4) = \text{true}$
we can jump lower to 2, $\text{lower}(3) = \text{higher}(2) = \text{true}$
If we start at 1,
we can jump higher to 2, $\text{higher}(1) = \text{lower}(2) = \text{true}$
we can't jump lower, $\text{lower}(1) = \text{false}$
If we start at 5,
we can't jump higher, $\text{higher}(5) = \text{false}$
we can jump lower to 4, $\text{lower}(5) = \text{higher}(4) = \text{false}$
So we know that for a certain index n, we could reach to the end if $\text{higher}(n) = \text{true}$, because we could only do odd numbered jumps from the first step.

```java
public int oddEvenJumps(int[] a) {
  int n = a.length;
  boolean[] higher = new boolean[n];
  boolean[] lower = new boolean[n];
  higher[n - 1] = lower[n - 1] = true;
  TreeMap<Integer, Integer> map = new TreeMap();
  map.put(a[n - 1], n - 1);
  int res = 1;
  for (int i = n - 2; n >= 0; i--) {
    Map.Entry<Integer, Integer> hi = map.ceilingEntry(a[i]);
```

```
    Map.Entry<Integer, Integer> lo = map.floorEntry(a[i]);
    if (null != hi) higher[i] = lower[hi.getValue()];
    if (null != lo) lower[i] = higher[lo.getValue()];
    // we can only do odd numbered jumps for the start point
    if (higher[i]) res++;
    map.put(a[i], i);
  }
  return res;
}
```

2.59 984. String Without AAA or BBB

Given two integers A and B, return any string S such that:
a. S has length A + B and contains exactly A 'a' letters, and exactly B 'b' letters;
b. The substring 'aaa' does not occur in S;
c. The substring 'bbb' does not occur in S.
Note:
a. 0 <= A <= 100
b. 0 <= B <= 100
c. It is guaranteed such an S exists for the given A and B.
Example 1:

```
Input: A = 1, B = 2
Output: "abb"
Explanation: "abb", "bab" and "bba" are all correct answers.
```

Example 2:

```
Input: A = 4, B = 1
Output: "aabaa"
```

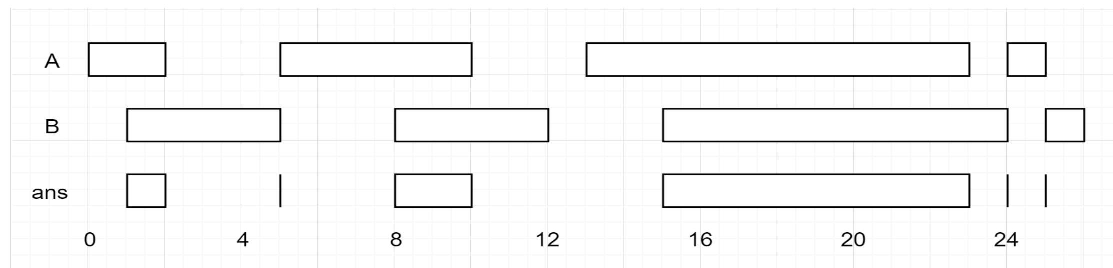
```
public String strWithout3a3b(int A, int B) {
  StringBuilder res = new StringBuilder(A + B);
  char a = 'a', b = 'b';
  int i = A, j = B;
  if (B > A) { a = 'b'; b = 'a'; i = B; j = A; }
  while (i-- > 0) {
    res.append(a);
    if (i > j) { res.append(a); --i; }
    if (j-- > 0) res.append(b);
  }
  return res.toString();
}
```

2.60 986. Interval List Intersections

Given two lists of closed intervals, each list of intervals is pairwise disjoint and in sorted order. Return the intersection of these two interval lists.
(Formally, a closed interval [a, b] (with a <= b) denotes the set of real numbers x with a <= x <= b. The intersection of two closed intervals is a set of real numbers that is either empty, or can be represented as a closed interval. For example, the intersection of [1, 3] and [2, 4] is [2, 3].)
Note:
a. `0 <= A.length < 1000`
b. `0 <= B.length < 1000`

c. 0 <= A[i].start, A[i].end, B[i].start, B[i].end < 10^9

Example:



Input: A = [[0, 2], [5, 10], [13, 23], [24, 25]], B = [[1, 5], [8, 12], [15, 24], [25, 26]]
Output: [[1, 2], [5, 5], [8, 10], [15, 23], [24, 24], [25, 25]]
Reminder: The inputs and the desired output are lists of Interval objects, and not arrays or lists.

Since the 2 Interval arrays are sorted, this problem could be simply solved by using 2 pointers [6].

```
/**
 * Definition for an interval.
 * public class Interval {
 * int start;
 * int end;
 * Interval() { start = 0; end = 0; }
 * Interval(int s, int e) { start = s; end = e; }
 * }
 */
class Solution {

  Interval intersection(Interval a, Interval b) {
    Interval res = new Interval();
    res.start = Math.max(a.start, b.start);

    res.end = Math.min(a.end, b.end);
    return res;
  }

  public Interval[] intervalIntersection(Interval[] A, Interval[] B) {
    List<Interval> res = new ArrayList();
    int i = 0;
    int j = 0;
    while (i < A.length && j < B.length) {
      if (A[i].end < B[j].start) i++;
      else if (A[i].start > B[j].end) j++;
      else {
        res.add(intersection(A[i], B[j]));
        if (A[i].end > B[j].end) j++;
        else i++;
      }
    }
    return res.toArray(new Interval[0]);
  }
}
```

2.61 992. Subarrays with K Different Integers

Given an array A of positive integers, call a (contiguous, not necessarily distinct) subarray of A good if the number of different integers in that subarray is exactly K.
(For example, [1, 2, 3, 1, 2] has 3 different integers: 1, 2, and 3.)
Return the number of good subarrays of A.
Note:
a. `1 <= A.length <= 20000`
b. `1 <= A[i] <= A.length`
c. `1 <= K <= A.length`
Example 1:

```
Input: A = [1, 2, 1, 2, 3], K = 2
Output: 7
Explanation: Subarrays formed with exactly 2 different integers: [1, 2], [2, 1], [1,
2], [2, 3], [1, 2, 1], [2, 1, 2], [1, 2, 1, 2].
```

Example 2:

```
Input: A = [1, 2, 1, 3, 4], K = 3
Output: 3
Explanation: Subarrays formed with exactly 3 different integers: [1, 2, 1, 3], [2,
1, 3], [1, 3, 4].
```

```
public int subarraysWithKDistinct(int[] A, int K) {
  return atMostK(A, K) - atMostK(A, K - 1);
}

int atMostK(int[] A, int K) {
  int i = 0, res = 0;
  Map<Integer, Integer> count = new HashMap<>();
  for (int j = 0; j < A.length; ++j) {
    if (count.getOrDefault(A[j], 0) == 0) K--;
    count.put(A[j], count.getOrDefault(A[j], 0) + 1);
    while (K < 0) {
      count.put(A[i], count.get(A[i]) - 1);
      if (count.get(A[i]) == 0) K++;
      i++;
    }
    res += j - i + 1;
  }
  return res;
}
```

2.62 1000. Minimum Cost to Merge Stones

There are N piles of stones arranged in a row. The ith pile has `stones[i]` stones.
A *move* consists of merging **exactly K consecutive** piles into one pile, and the cost of this move is equal to the total number of stones in these K piles. Find the minimum cost to merge all piles of stones into one pile. If it is impossible, return -1.
Example 1:

```
Input: stones = [3, 2, 4, 1], K = 2
Output: 20
Explanation:
```

```
We start with [3, 2, 4, 1].
We merge [3, 2] for a cost of 5, and we are left with [5, 4, 1].
We merge [4, 1] for a cost of 5, and we are left with [5, 5].
We merge [5, 5] for a cost of 10, and we are left with [10].
The total cost was 20, and this is the minimum possible.
```

Example 2:

```
Input: stones = [3, 2, 4, 1], K = 3
Output: -1
Explanation: After any merge operation, there are 2 piles left, and we can't merge
anymore. So the task is impossible.
```

Example 3:

```
Input: stones = [3, 5, 1, 2, 6], K = 3
Output: 25
Explanation:
We start with [3, 5, 1, 2, 6].
We merge [5, 1, 2] for a cost of 8, and we are left with [3, 8, 6].
We merge [3, 8, 6] for a cost of 17, and we are left with [17].
The total cost was 25, and this is the minimum possible.
```

Note:

a. 1 <= stones.length <= 30
b. 2 <= K <= 30
c. 1 <= stones[i] <= 100

Solution:

```java
class Solution {
  int max = Integer.MAX_VALUE;

  int dp(int[] sum, int[][][] dp, int start, int end, int piles, int k) {
    if (dp[start][end][piles] > 0) return dp[start][end][piles];
    if (end - start + 1 < piles) return max;
    if (start == end) {
      return (piles == 1) ? 0 : Integer.MAX_VALUE;
    }

    if (piles == 1) {
      int minCost = dp(sum, dp, start, end, k, k);
      if (minCost != max) {
        dp[start][end][piles] = sum[end + 1] - sum[start] + minCost;
      } else {
        dp[start][end][piles] = max;
      }
      return dp[start][end][piles];
    }
    int minCost = max;
    for (int t = start; t <= end - 1; t++) {
      int leftCost = dp(sum, dp, start, t, piles - 1, k);
      if (leftCost == max) continue;
      int rightCost = dp(sum, dp, t + 1, end, 1, k);
      if (rightCost == max) continue;
      minCost = Math.min(leftCost + rightCost, minCost);
    }
    dp[start][end][piles] = minCost;
    return minCost;
  }
```

```
  public int mergeStones(int[] stones, int K) {
    int n = stones.length;
    if ((n - K) % (K - 1) != 0) return -1;
    int[][][] dp = new int[n + 1][n + 1][K + 1];
    int[] sum = new int[n + 1];
    for (int i = 0; i < n; i++) {
      sum[i + 1] = sum[i] + stones[i];
    }
    int res = dp(sum, dp, 0, n - 1, 1, K);
    return res == max ? -1 : res;
  }
}
```

2.63 1022. Smallest Integer Divisible by K

Given a positive integer K($1 <= K <= 10^5$), you need find the smallest positive integer N such that N is divisible by K, and N only contains the digit 1.
Return the length of N. If there is no such N, return -1.
Example 1:

```
Input: 1
Output: 1
Explanation: The smallest answer is N = 1, which has length 1.
```

Example 2:

```
Input: 2
Output: -1
Explanation: There is no such positive integer N divisible by 2.
```

Example 3:

```
Input: 3
Output: 3
Explanation: The smallest answer is N = 111, which has length 3.
```

Intuition:
If N exist, $N <= K$, just do a brute force check, also $(A * B + C)\%K = (A\%K * B\%K + C\%K)\%K$
Also if $K\%2 == 0$, return -1, because 111....11 can't be even.
Also if $K\%5 == 0$, return -1, because 111....11 can't end with 0 or 5.

```
public int smallestRepunitDivByK(int K) {
  if (K % 2 == 0 || K % 5 == 0) return -1;
    int r = 0;
    for (int N = 1; N <= K; ++N) {
      r = (r * 10 + 1) % K;
      if (r == 0) return N;
    }
    return -1;
}
```

2.64 1024. Video Stitching

You are given a series of video clips from a sporting event that lasted T seconds. These video clips can be overlapping with each other and have varied lengths.

Each video clip `clips[i]` is an interval: it starts at time `clips[i][0]` and ends at time `clips[i][1]`. We can cut these clips into segments freely: for example, a clip `[0, 7]` can be cut into segments `[0, 1] + [1, 3] + [3, 7]`.
Return the minimum number of clips needed so that we can cut the clips into segments that cover the entire sporting event (`[0, T]`). If the task is impossible, return `-1`.

Note:

a. `1 <= clips.length <= 100`
b. `0 <= clips[i][0], clips[i][1] <= 100`
c. `0 <= T <= 100`

Example 1:

```
Input: clips = [[0, 2], [4, 6], [8, 10], [1, 9], [1, 5], [5, 9]], T = 10
Output: 3
Explanation:
We take the clips [0,2], [8,10], [1,9]; a total of 3 clips.
Then, we can reconstruct the sporting event as follows:
We cut [1, 9] into segments [1, 2] + [2, 8] + [8, 9].
Now we have segments [0, 2] + [2, 8] + [8, 10] which cover the sporting event [0, 10].
```

Example 2:

```
Input: clips = [[0, 1], [1, 2]], T = 5
Output: -1
Explanation:
We can't cover [0, 5] with only [0, 1] and [0, 2].
```

Example 3:

```
Input: clips = [[0, 1], [6, 8], [0, 2], [5, 6], [0, 4], [0, 3], [6, 7], [1, 3], [4, 7],
[1, 4], [2, 5], [2, 6], [3, 4], [4, 5], [5, 7], [6, 9]], T = 9
Output: 3
Explanation:
We can take clips [0, 4], [4, 7], and [6, 9].
```

Example 4:

```
Input: clips = [[0, 4], [2, 8]], T = 5
Output: 2
Explanation:
Notice you can have extra video after the event ends.
```

```java
public int videoStitching(int[][] clips, int T) {
  int[] dic = new int[T + 1];
  Arrays.fill(dic, -1);
  for (int[] clip : clips) {
    for (int i = clip[0]; i <= Math.min(T, clip[1]); i++) {
      dic[i] = Math.max(dic[i], clip[1]);
    }
  }
  int max = -1;
  for (int i : dic) {
    if (i == -1) return -1;
  }
  int ans = 1;
  max = dic[0];
  while (max < T) {
```

```
    max = dic[max];
    ans++;
  }
  return ans;
}
```

2.65 1025. Divisor Game

Alice and Bob take turns playing a game, with Alice starting first.
Initially, there is a number N($1 <= N <= 1000$) on the chalkboard. On each player's turn, that player makes a move consisting of:
a. Choosing any x with $0 < x < N$ and N % x == 0.
b. Replacing the number N on the chalkboard with N - x.
Also, if a player cannot make a move, they lose the game.
Return True if and only if Alice wins the game, assuming both players play optimally.
Example 1:

```
Input: 2
Output: true
Explanation: Alice chooses 1, and Bob has no more moves.
```

Example 2:

```
Input: 3
Output: false
Explanation: Alice chooses 1, Bob chooses 1, and Alice has no more moves.
```

```
public boolean divisorGame(int N) {
  boolean[] dp = new boolean[N + 1];
  dp[0] = false;
  dp[1] = false;
  for (int i = 2; i <= N; i++) {
    for (int j = 1; j < i; j++) {
      if (i % j == 0) {
        if (dp[i - j] == false) {
          dp[i] = true;
          break;
        }
      }
    }
  }
  return dp[N];
}
```

2.66 1027. Longest Arithmetic Sequence

Given an array `A` of integers, return the **length** of the longest arithmetic subsequence in `A`.
Recall that a subsequence of `A` is a list `A[i_1], A[i_2], ..., A[i_k]` with `0 <= i_1 < i_2 < ... < i_k <= A.length - 1`, and that a sequence B is arithmetic if `B[i+1] - B[i]` are all the same value (`for 0 <= i < B.length - 1`).
Notes:
a. $2 <= A.length <= 2000$
b. $0 <= A[i] <= 10000$
Example 1:

```
Input: [3, 6, 9, 12]
Output: 4
Explanation:
The whole array is an arithmetic sequence with steps of length = 3.
```

Example 2:

```
Input: [9, 4, 7, 2, 10]
Output: 3
Explanation:
The longest arithmetic subsequence is [4, 7, 10].
```

Example 3:

```
Input: [20, 1, 15, 3, 10, 5, 8]
Output: 4
Explanation:
The longest arithmetic subsequence is [20, 15, 10, 5].
```

```
public int longestArithSeqLength(int[] A) {
  int max = 0;
  Map<Integer, Map<Integer, Integer>> dp = new HashMap();
  for (int i = 0; i < A.length; i++) {
    dp.put(i, new HashMap());
  }
  for (int i = 0; i < A.length; i++) {
    for (int j = 0; j < i; j++) {
      int diff = A[i] - A[j];
      Map<Integer, Integer> map = dp.get(i);
      map.put(diff, dp.get(j).getOrDefault(diff, 0) + 1);
      max = Math.max(max, dp.get(i).get(diff));
    }
  }
  return max + 1;
}
```

2.67 1029. 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.

Note

a. `1 <= costs.length <= 100`

b. It is guaranteed that costs.length is even.

c. `1 <= costs[i][0], costs[i][1] <= 1000`

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.
```

```java
public int twoCitySchedCost(int[][] costs) {
  int res = 0;
  int n = costs.length;
  Arrays.sort(costs, (a, b) -> (b[1] - b[0]) - (a[1] - a[0]));
  for (int i = 0; i < n / 2; i++) {
    res += costs[i][0] + costs[n - i - 1][1];
  }
  return res;
}
```

2.68 1049. Last Stone Weight II

We have a collection of rocks, each rock has a positive integer weight.
Each turn, we choose any two rocks 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 smallest possible weight of this stone (the weight is 0 if there are no stones left.)

Note:

a. `1 <= stones.length <= 30`

b. `1 <= stones[i] <= 100`

Example

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

Intuition

Same problem as divid all numbers into two groups in a set, what is the minimum difference between the sum of the two groups. It is a classic knapsack problem.

```java
class Solution {
  public int lastStoneWeightII(int[] stones) {
    int sum = IntStream.of(stones).sum();
    int n = stones.length;
    // dp[i][j] = true means the elements from 0 to i
    // could achieve sum j if we could ignore some elements.
    boolean[][] dp = new boolean[n + 1][sum + 1];
    for (int i = 0; i <= n; i++) {
      dp[i][0] = true;
    }
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= sum; j++) {
        dp[i][j] = dp[i - 1][j];
```

```
        if (j - stones[i - 1] >= 0) {
          dp[i][j] |= dp[i - 1][j - stones[i - 1]];
        }
      }
    }
    int max = 0;
    for (int i = sum / 2; i >= 0; i--) {
      if (dp[n][i]) {
        max = i;
        break;
      }
    }
    return sum - 2 * max;
  }
}
```

3 Cracking the Code Interview

3.1 BST Sequences

A binary search tree was created by traversing through an array from left to right and inserting each element. Given a binary search tree with **distinct** elements, print all possible arrays that could have led to this tree.
Example 1:

```
Input:
  2
 / \
 1  3
Output:
[[2, 1, 3], [2, 3, 1]]
```

```
public static void weaveLists(LinkedList<Integer> first,
                              LinkedList<Integer> second,
                              ArrayList<LinkedList<Integer>> results,
                              LinkedList<Integer> prefix) {
  /*
   * One list is empty. Add the remainder to
   * [a cloned] prefix and store result.
   */
  if (first.size() == 0 || second.size() == 0) {
    LinkedList<Integer> result = (LinkedList<Integer>) prefix.clone();
    result.addAll(first);
    result.addAll(second);
    results.add(result);
    return;
  }

  /*
   * Recurse with head of first added to the prefix. Removing the
   * head will damage first, so we'll need to put it back where we
   * found it afterwards.
   */
  int headFirst = first.removeFirst();
  prefix.addLast(headFirst);
  weaveLists(first, second, results, prefix);
  prefix.removeLast();
```

```
  first.addFirst(headFirst);

  /* Do the same thing with second, damaging and then restoring
   * the list.
   */
  int headSecond = second.removeFirst();
  prefix.addLast(headSecond);
  weaveLists(first, second, results, prefix);
  prefix.removeLast();
  second.addFirst(headSecond);
}

public static ArrayList<LinkedList<Integer>> allSequences(TreeNode node) {
  ArrayList<LinkedList<Integer>> result = new ArrayList<LinkedList<Integer
      >>();

  if (node == null) {
    result.add(new LinkedList<Integer>());
    return result;
  }

  LinkedList<Integer> prefix = new LinkedList<Integer>();
  prefix.add(node.data);

  /* Recurse on left and right subtrees. */
  ArrayList<LinkedList<Integer>> leftSeq = allSequences(node.left);
  ArrayList<LinkedList<Integer>> rightSeq = allSequences(node.right);

  /* Weave together each list from the left and right sides. */
  for (LinkedList<Integer> left : leftSeq) {
    for (LinkedList<Integer> right : rightSeq) {
      ArrayList<LinkedList<Integer>> weaved = new ArrayList<LinkedList<
          Integer>>();
      weaveLists(left, right, weaved, prefix);
      result.addAll(weaved);
    }
  }
  return result;
}
```

3.2 Insertion

You are give 2 32-bit numbers, N and M, and two bit positions, i and j. Write a method to insert M into N such that M starts a bit j and ends at bit i. You can assume that the bits j through i have enough space to fit all of M. That is, if $M = 10011$, you can assume that there are at least 5 bits between j and i. You would not, for example, have $j = 3$ and $i = 2$, because M could not fully fit between 3 and bit 2.

Example

```
Input: N = 10000000000, M = 10011, i = 2, j = 6
Output: N = 10001001100
```

```
int updateBits(int n, int m, int i, int j) {
  // Validation
  if (i > j || i < 0 || j >= 32) {
    return 0;
  }
```

```
  /* Create a mask to clear bits i through j in n
   * EXAMPLE: i = 2, j = 4. Result should be 11100011.
   * (Using 8 bits for this example.  This is obviously not actually 8 bits.)
   */
  int allOnes = ~0; // allOnes = 11111111

  // 1s until position j, then 0s. left = 11100000
  int left = j < 31 ? (allOnes << (j + 1)) : 0;
  // 1s after position i.  right = 00000011
  int right = ((1 << i) - 1);

  // All 1s, except for 0s between i and j. mask = 11100011
  int mask = left | right;
  /* Clear i through j, then put m in there */
  // Clear bits j through i.
  int n_cleared = n & mask;
  // Move m into correct position.
  int m_shifted = m << i;

  /* OR them, and we're done! */
  return n_cleared | m_shifted;
}
```

3.3 Count number of bits to be flipped to convert A to B

Example

```
Input : a = 10, b = 20
Output : 4
Binary representation of a is 00001010
Binary representation of b is 00010100
We need to flip highlighted four bits in a to make it b.
```

The idea is count the 1s in `A ^ B`.

```
static int countSetBits(int n) {
  int count = 0;
  while (n != 0) {
    count += n & 1;
    n >>= 1;
  }
  return count;
}

static int flippCount(int a, int b) {
  return countSetBits(a ^ b);
}
```

3.4 Magic Index

A magic line in an array `A[0...n-1]` is defined to be an index such that `A[i] = i`. Given a sorted array of distinct integers, write a method to find a magic index, if one exists, in array A.
FOLLOW UP: What if the values are not distinct.
Example

```
int[] A = { -1, 0, 1, 2, 4, 10 };
Magic index or fixed point is : 4
```

```java
/*
 * This solution doest work when array has duplicates.
 */
static int magicIndex(int[] array, int start, int end) {
  if (end < start) {
    return -1;
  }
  int mid = (start + end) / 2;
  if (array[mid] == mid) {
    return mid;
  } else if (array[mid] > mid) {
    return magicIndex(array, start, mid - 1);
  } else {
    return magicIndex(array, mid + 1, end);
  }
}

/**
 * The array could have duplicated elements.
 */
public static int magicIndex(int[] array, int start, int end) {
  if (end < start) {
    return -1;
  }
  int midIndex = (start + end) / 2;
  int midValue = array[midIndex];
  if (midValue == midIndex) {
    return midIndex;
  }
  /* Search left */
  int leftIndex = Math.min(midIndex - 1, midValue);
  int left = magicIndex(array, start, leftIndex);
  if (left >= 0) {
    return left;
  }

  /* Search right */
  int rightIndex = Math.max(midIndex + 1, midValue);
  int right = magicIndex(array, rightIndex, end);

  return right;
}
```

3.5 Boolean Parenthesization Problem

Given a boolean expression with following symbols:

Symbols

'T' ---> true

'F' ---> false

And following operators filled between symbols:

Operators

& ---> boolean AND

| ---> boolean OR

^ ---> boolean XOR

Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.

Let the input be in form of two arrays one contains the symbols (T and F) in order and other contains operators (&, | and ^).

Example

```
Input: symbol[]  = {T, T, F, T}
   operator[] = {|, &, ^}
Output: 4
The given expression is "T | T & F ^ T", it evaluates true in 4 ways:
((T|T)&(F^T)), (T|(T&(F^T))),
(((T|T)&F)^T) and (T|((T&F)^T)).
```

```
class Solution {

  // Returns count of all possible
  // parenthesizations that lead to
  // result true for a boolean
  // expression with symbols like true
  // and false and operators like &, |
  // and ^ filled between symbols
  static int countParenth(char symb[],
                          char oper[],
                          int n) {
    int F[][] = new int[n][n];
    int T[][] = new int[n][n];

    // Fill diaginal entries first
    // All diagonal entries in T[i][i]
    // are 1 if symbol[i] is T (true).
    // Similarly, all F[i][i] entries
    // are 1 if symbol[i] is F (False)
    for (int i = 0; i < n; i++) {
      F[i][i] = (symb[i] == 'F') ? 1 : 0;
      T[i][i] = (symb[i] == 'T') ? 1 : 0;
    }

    // Now fill T[i][i+1], T[i][i+2],
    // T[i][i+3]... in order And F[i][i+1],
    // F[i][i+2], F[i][i+3]... in order
    for (int gap = 1; gap < n; ++gap) {
      for (int i = 0, j = gap; j < n; ++i, ++j) {
        T[i][j] = F[i][j] = 0;
        for (int g = 0; g < gap; g++) {
          // Find place of parenthesization
          // using current value of gap
          int k = i + g;

          // Store Total[i][k] and Total[k+1][j]
          int tik = T[i][k] + F[i][k];
          int tkj = T[k + 1][j] + F[k + 1][j];

          // Follow the recursive formulas
          // according to the current operator
          if (oper[k] == '&') {
            T[i][j] += T[i][k] * T[k + 1][j];
            F[i][j] += (tik * tkj - T[i][k] * T[k + 1][j]);
          }
          if (oper[k] == '|') {
            F[i][j] += F[i][k] * F[k + 1][j];
            T[i][j] += (tik * tkj - F[i][k] * F[k + 1][j]);
```

```
          }
          if (oper[k] == '^') {
            T[i][j] += F[i][k] * T[k + 1][j] +
                    T[i][k] * F[k + 1][j];
            F[i][j] += T[i][k] * T[k + 1][j] +
                    F[i][k] * F[k + 1][j];
          }
        }
      }
    }
    return T[0][n - 1];
  }
}
```

3.6 Find Duplicates[1]

You have an array of N numbers, where N is at most 32,000. The array may have duplicates entries and you do not know what N is. With only 4 Kilobytes of memory available, how would print all duplicates elements in the array?

Examples

```
Input : arr[] = {1, 5, 1, 10, 12, 10}
Output : 1 10
1 and 10 appear more than once in given
array.

Input : arr[] = {50, 40, 50}
Output : 50
```

```
public class Solution {
  static class BitSet {
    int[] bitset;

    public BitSet(int size) {
      bitset = new int[(size >> 5) + 1]; // divide by 32
    }

    boolean get(int pos) {
      int wordNumber = (pos >> 5); // divide by 32
      int bitNumber = (pos & 0x1F); // mod 32
      return (bitset[wordNumber] & (1 << bitNumber)) != 0;
    }

    void set(int pos) {
      int wordNumber = (pos >> 5); // divide by 32
      int bitNumber = (pos & 0x1F); // mod 32
      bitset[wordNumber] |= 1 << bitNumber;
    }
  }

  public static void checkDuplicates(int[] array) {
    BitSet bs = new BitSet(32000);
    for (int i = 0; i < array.length; i++) {
      int num = array[i];
      int num0 = num - 1; // bitset starts at 0, numbers start at 1
      if (bs.get(num0)) {
```

[1] This problem is a good example of how to implement bitset.

```
        System.out.println(num);
      } else {
        bs.set(num0);
      }
    }
  }
}
```

3.7 Rank of An Eelement in A Stream

Given a stream of integers, lookup the rank of a given integer x. Rank of an integer in stream is "Total number of elements less than or equal to x (not including x)".
If element is not found in stream or is smallest in stream, return -1.
Examples

```
Input : arr[] = {10, 20, 15, 3, 4, 4, 1}, x = 4;
Output : Rank of 4 in stream is: 3
There are total three elements less than
or equal to x (and not including x)

Input : arr[] = {5, 1, 14, 4, 15, 9, 7, 20, 11}, x = 20;
Output : Rank of 20 in stream is: 8
```

```
class Solution {

  static class Node {
    int data;
    Node left, right;
    int leftSize;
  }

  static Node newNode(int data) {
    Node temp = new Node();
    temp.data = data;
    temp.left = null;
    temp.right = null;
    temp.leftSize = 0;
    return temp;
  }

  // Inserting a new Node.
  static Node insert(Node root, int data) {
    if (root == null)
      return newNode(data);

    // Updating size of left subtree.
    if (data <= root.data) {
      root.left = insert(root.left, data);
      root.leftSize++;
    } else
      root.right = insert(root.right, data);

    return root;
  }

  // Function to get Rank of a Node x.
  static int getRank(Node root, int x) {
```

```
    // Step 1.
    if (root.data == x)
      return root.leftSize;

    // Step 2.
    if (x < root.data) {
      if (root.left == null)
        return -1;
      else
        return getRank(root.left, x);
    }

    // Step 3.
    else {
      if (root.right == null)
        return -1;
      else {
        int rightSize = getRank(root.right, x);
        return root.leftSize + 1 + rightSize;
      }
    }
  }

  // Driver code
  public static void main(String[] args) {
    int arr[] = {5, 1, 4, 4, 5, 9, 7, 13, 3};
    int n = arr.length;
    int x = 4;

    Node root = null;
    for (int i = 0; i < n; i++)
      root = insert(root, arr[i]);

    System.out.println("Rank of " + x + " in stream is : " + getRank(root, x)
        );

    x = 13;
    System.out.println("Rank of " + x + " in stream is : " + getRank(root, x)
        );

  }
}
```

3.8 Diving Board

You are building a diving board by placing a bunch of planks of wood end-to-end. There are two types of planks: **shorter** and **longer**. You must use exaclty K planks of wood. Write a method to generate all possible lengths for the diving board.

```
public static Set<Integer> allLengths(int k, int shorter, int longer) {
  Set<Integer> lengths = new HashSet();
  for (int nShorter = 0; nShorter <= k; nShorter++) {
    int nLonger = k - nShorter;
    int length = nShorter * shorter + nLonger * longer;
    lengths.add(length);
  }
  return lengths;
}
```

3.9 Sum Swap

Given two arrays of integers, find a pair of values (one value from each array) that you can swap to give the two arrays the same sum.

Example

```
Input : A[] = {4, 1, 2, 1, 1, 2} B[] = {3, 6, 3, 3}
Output : {1, 3}
Sum of elements in A[] = 11
Sum of elements in B[] = 15
To get same sum from both arrays, we
can swap following values:
1 from A[] and 3 from B[]
```

We are looking for two values, a and b, such that:

```
sumA - a + b = sumB - b + a
2a - 2b = sumA - sumB
a - b = (sumA - sumB) / 2
```

Therefore, we are looking for two values that have a specific target difference: `(sumA - sumB)/2`.

```
class Solution {
  // Function to calculate sum of elements of array
  static int getSum(int X[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
      sum += X[i];
    return sum;
  }

  // Function to calculate : a - b = (sumA - sumB) / 2
  static int getTarget(int A[], int n, int B[], int m) {
    // Calculation of sums from both arrays
    int sum1 = getSum(A, n);
    int sum2 = getSum(B, m);

    // because that the target must be an integer
    if ((sum1 - sum2) % 2 != 0)
      return 0;
    return ((sum1 - sum2) / 2);
  }

  // Function to prints elements to be swapped
  static void findSwapValues(int A[], int n, int B[], int m) {
    int target = getTarget(A, n, B, m);
    if (target == 0)
      return;

    // Look for val1 and val2, such that
    // val1 - val2 = (sumA - sumB) / 2
    int val1 = 0, val2 = 0;
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        if (A[i] - B[j] == target) {
          val1 = A[i];
          val2 = B[j];
        }
      }
```

```
    }
    System.out.println(val1 + " " + val2);
  }
}
```

3.10 Number of 2s

Count the number of 2s as digit in all numbers from 0 to n.

Example

```
Input : 22
Output : 6
Explanation:
Total 2s that appear as digit from 0 to 22 are
(2, 12, 20, 21, 22);

Input : 100
Output : 20
Explanation:
Total 2s comes between 0 to 100 are
(2, 12, 20, 21, 22..29, 32, 42, 52, 62, 72, 82, 92);
```

```
class Solution {

  // Counts the number of 2s
  // in a number at d-th digit
  static int count2sinRangeAtDigit(int number, int d) {
    int powerOf10 = (int) Math.pow(10, d);
    int nextPowerOf10 = powerOf10 * 10;
    int right = number % powerOf10;

    int roundDown = number - number % nextPowerOf10;
    int roundup = roundDown + nextPowerOf10;

    int digit = (number / powerOf10) % 10;

    // if the digit in spot digit is
    if (digit < 2) {
      return roundDown / 10;
    }

    if (digit == 2) {
      return roundDown / 10 + right + 1;
    }
    return roundup / 10;
  }

  // Counts the number of '2' digits between 0 and n
  static int numberOf2sinRange(int number) {
    // Convert integer to String
    // to find its length
    String convert;
    convert = String.valueOf(number);
    String s = convert;
    int len = s.length();

    // Traverse every digit and
```

```
    // count for every digit
    int count = 0;
    for (int digit = 0; digit < len; digit++) {
      count += count2sinRangeAtDigit(number, digit);
    }

    return count;
  }
}
```

3.11 The Masseuse

A popular masseuse receives a sequence of back-to-back appointment requests and is debating which ones to accept. She needs a 15-minute break between appointments and therefore she cannot accept any adjacent requests. Given a sequence of back-to-back appointment requests (all multiples of 15 minutes, none overlap, and none can be moved), find the optimal(highest total booked minutes) set the masseuse can honor. Return the number of minutes.

Example

```
Input: {30, 15, 60, 75, 45, 15, 15, 45}
Output: 180 minutes({30, 60, 45, 45})
```

```
public class Solution {
  // O(2^n)
  public static int maxMinutes(int[] massages) {
    return maxMinutes(massages, 0);
  }

  public static int maxMinutes(int[] massages, int index) {
    if (index >= massages.length) { // Out of bounds
      return 0;
    }

    /* Best with this reservation. */
    int bestWith = massages[index] + maxMinutes(massages, index + 2);

    /* Best without this reservation. */
    int bestWithout = maxMinutes(massages, index + 1);

    /* Return best of this subarray, starting from index. */
    return Math.max(bestWith, bestWithout);
  }

  // O(n) in time and O(n) in space.
  public static int maxMinutes(int[] massages) {
    int[] memo = new int[massages.length];
    return maxMinutes(massages, 0, memo);
  }

  public static int maxMinutes(int[] massages, int index, int[] memo) {
    if (index >= massages.length) {
      return 0;
    }
    if (memo[index] == 0) {
      int bestWith = massages[index] + maxMinutes(massages, index + 2, memo);
      int bestWithout = maxMinutes(massages, index + 1, memo);
      memo[index] = Math.max(bestWith, bestWithout);
    }
```

```
    return memo[index];
  }

}
```

References

[1] Java Solutions for Single Number II

[2] Distinct Subsequences

[3] 10-line Template That Can Solve Most 'substring' Problems

[4] Easy Java solution for Palindrome Partitioning II

[5] Java Segment Tree Solution

[6] Java Two Pointers Solution to Interval List Intersections

[7] Sliding Window Solution to K Different Integers

[8] Short and Simple Solution to Sliding Window Median Using Two PriorityQueue

[9] DP Solution to Minimum Window Subsequence

[10] Java 2 Pointer Solution to Minimum Window Subsequence

[11] Both Iterative and Recursive Solutions with Explanations

[12] DP idea, Using TreeMap

[13] A Concise Template for "Overlapping Interval Problem"