Algorithm & Data Structure

Sai Ma

Oct 2015

1 LintCode

1.1 Construct Binary Tree from Pre-order and In-order Traversal

From the pre-order array, we know that first element is the root. We can find the root in in-order array. Then we can identify the left and right sub-trees of the root from in-order array. Using the length of left sub-tree, we can identify left and right sub-trees in pre-order array. Recursively, we can build up the tree. It based on a definition that in the in-order traversal, the left node is located its left, and right node located its right.

```
public TreeNode buildTree(int[] preorder, int[] inorder) {
        // write your code here

        int preStart = 0;
        int preEnd = preorder.length - 1;
        int inStart = 0;
        int inEnd = inorder.length - 1;

        return construct(preorder, preStart, preEnd, inorder, inStart, inEnd);
}

public TreeNode construct(int[] preorder, int preStart,
    int preEnd, int[] inorder, int inStart, int inEnd){
        if (preStart > preEnd || inStart > inEnd){
                return null;
        }

        int val = preorder[preStart];
        TreeNode p = new TreeNode(val);

        //find parent element index from inorder
        int k = 0;
        for (int i = 0; i < inorder.length; i++){
                if (val == inorder[i]){
                        k = i;
                        break;
                }
        }
        p.left = construct(preorder, preStart + 1, preStart
                + (k - inStart), inorder, inStart, k - 1);
        p.right = construct(preorder, preStart + (k - inStart) + 1, preEnd,
           inorder, k + 1 , inEnd);
        return p;
}
```

1.2 Search a 2D Matrix II

In this problem, we should use a pointer to move through matrix based its definition. However, we should also consider of situation that not stuck in first line, not raise error at edge.

```
public int searchMatrix(int[][] matrix, int target) {
        // write your code here

        int returnNumber = 0;
        if (matrix.length == 0) {
                return returnNumber;
        }
        int rowIndex = 0;
        int columnIndex = 0;
        while (columnIndex < matrix[rowIndex].length) {
                if (matrix[rowIndex][columnIndex] == target) {
                        returnNumber++;
                        rowIndex++;
                        break;
                } else if (matrix[rowIndex][columnIndex] < target) {
                        columnIndex++;
                } else {
                        rowIndex++;
                        break;
                }
        }
        if (columnIndex == matrix[0].length) {
                columnIndex--;
        }

        while (rowIndex < matrix.length) {
                if (matrix[rowIndex][columnIndex] == target) {
                        returnNumber++;
                        rowIndex++;
                        columnIndex--;
                } else if (matrix[rowIndex][columnIndex] > target) {
                        columnIndex--;
                } else {
                        rowIndex++;
                }
                if (columnIndex == matrix[0].length) {
                        rowIndex++;
                        columnIndex--;
                } else if (columnIndex == -1) {
                        rowIndex++;
                        columnIndex++;
                }
                //System.out.println(columnIndex);
        }

        return returnNumber;
}
```

1.3 Climbing Stairs

You are climbing a stair case. It takes n steps to reach to the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

```
public int climbStairs(int n) {
        // write your code here
        if (n < 1) {
                return 1;
        }
        int past = 1;
        int last = 1;
        int now = 1;

        for (int i = 2; i < n + 1; i++) {
                now = past + last;
                last = past;
                past = now;
        }
        return now;
}
```

1.4 Find Minimum in Rotated Sorted Array II

In this problem, the main task is find its original order based on three points, which are 'start', 'mid' and 'end'. Once the mid greater than end, which means left part of mid index is ordered, then the pivot located at right part, and we change start index to mid plus 1. If mid less than end, which means that right part is ordered well, then pivot must located at left part. Then, we change our end to mid index. if they are equal, we just reduce one step of end index to reduce range. Finally, the minimum number stopped at start index.

```
public int findMin(int[] num) {
        // write your code here

        int start = 0;
        int end = num.length - 1;

        while (start < end) {
                int mid = (start + end) / 2;
                if (num[mid] > num[end]) {
                        start = mid + 1;
                } else if (num[mid] < num[end]) {
                        end = mid;
                } else {
                        end--;
                }
        }
        return num[start];
}
```

Partition Array by Odd and Even The main task is that we should define that the left and right already have the odd and even number which are ordered well. Then, we use *while* loop to find first number pair which are not fulfill our assumption, and then exchange them.

```
public void partitionArray(int[] nums) {
        // write your code here;
    if (nums == null || nums.length == 0) {
        return;
    }
    int start = 0, end = nums.length - 1;
    while (start < end) {
        while (start < end && nums[start] % 2 != 0) {
                start++; // find all odd number which at left
        }
        while (start < end && nums[end] % 2 == 0) {
                end--; // find all even number which at right
        }
        if (start < end) { // exchange them
                int tmp = nums[start];
                nums[start] = nums[end];
                nums[end] = tmp;
                start++;
                end--;
        }
        }
}
```

Valid Soduku In this problem, in order to speed up the process on calculate the number in char matrix is unique, we use the char minus '1' to get the distance in ASCII code, and use it as index of a list, then if it already exits this number, it boolean result should be true, and then return false;

```
public boolean isValidSudoku(char[][] board) {
        if (board == null || board.length != 9 || board[0].length != 9)
                return false;
        // check each column
        for (int i = 0; i < 9; i++) {
                boolean[] m = new boolean[9];
                for (int j = 0; j < 9; j++) {
                        if (board[i][j] != '.') {
                                if (m[(int) (board[i][j] - '1')]) {
                                        return false;
                                }
                                m[(int) (board[i][j] - '1')] = true;
                        }
                }
        }

        //check each row
        for (int j = 0; j < 9; j++) {
                boolean[] m = new boolean[9];
                for (int i = 0; i < 9; i++) {
                        if (board[i][j] != '.') {
                                if (m[(int) (board[i][j] - '1')]) {
                                        return false;
                                }
                                m[(int) (board[i][j] - '1')] = true;
```

```
                        }
                }
        }

        //check each 3*3 matrix
        for (int block = 0; block < 9; block++) {
                boolean[] m = new boolean[9];
                for (int i = block / 3 * 3; i < block / 3 * 3 + 3; i++) {
                        for (int j = block % 3 * 3; j < block % 3 * 3 + 3;
                            j++) {
                                if (board[i][j] != '.') {
                                        if (m[(int) (board[i][j] - '1')]) {
                                                return false;
                                        }
                                        m[(int) (board[i][j] - '1')] = true;
                                }
                        }
                }
        }

        return true;
}
```

1.5 Subtree

It is the perfectly implement on recursion.

```
public boolean isSubtree(TreeNode T1, TreeNode T2) {
        // write your code here
        if(T2 == null)
                return true;
        else if(T1 == null)
                return false;
        else return isSameTree(T1, T2) || isSubtree(T1.left, T2) ||
            isSubtree(T1.right, T2);
    }

public boolean isSameTree(TreeNode T1, TreeNode T2) {
        if(T1 == null && T2 == null)
                return true;
        if(T1 == null || T2 == null)
                return false;
        if(T1.val != T2.val)
                return false;
        return isSameTree(T1.left,T2.left) && isSameTree(T1.right, T2.right);
}
```

1.6 Remove Nth Node From End of List

Perform *Quick* and *Slow* pointer to find the deleted node from end of list. The quick and slow pointer aims to construct the distance between two pointers, and then apply this distance to finish our task.

```
public ListNode removeNthFromEnd(ListNode head, int n) {
        if(head == null) {
```

```
                return null;
        }
        ListNode fast = head;
        ListNode slow = head;

        for(int i=0; i<n; i++){
                fast = fast.next;
        }
        //if remove the first node
        if (fast == null){
                head = head.next;
                return head;
        }
        while (fast.next != null){
                fast = fast.next;
                slow = slow.next;
        }
        slow.next = slow.next.next;
        return head;
}
```

1.7 Lower & Upper Bound Method

By this method, it is easy to get the lower and upper bound of a target number based in a ordered list. Moreover, to some problem, the result would be *searched* by this method.

```
public static int lowerBound(int[] nums, int target) {
        if (nums == null || nums.length == 0) {
                return -1;
        }
        int lb = -1, ub = nums.length;
        while (lb + 1 < ub) {
                int mid = lb + (ub - lb) / 2;
                if (nums[mid] < target) {
                        lb = mid;
                } else  {
                        ub = mid;
                }
        }
        return lb + 1;
}

public static int upperBound(int[] nums, int target) {
        if (nums == null || nums.length == 0) {
                return -1;
        }
        int lb = -1, ub = nums.length;
        while (lb + 1 < ub) {
                int mid = lb + (ub - lb) / 2;
                if (nums[mid] > target) {
                        ub = mid;
                } else {
                        lb = mid;
```

```
                }
        }
        return ub - 1;
}
```

1.8 Convert Binary Search Tree to Doubly Linked List

In this method, we should perform traversal method to traversal all nodes in the tree, and then construct the double-linked list.

```
public DoublyListNode bstToDoublyList(TreeNode root) {
        // Write your code here
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode curt = root;
        DoublyListNode prev = null;
        DoublyListNode head = null;
        while (curt != null || !stack.empty()) {
                while (curt != null) {
                        stack.add(curt);
                        curt = curt.left;
                }
                curt = stack.peek();
                stack.pop();
                DoublyListNode curtListNode = new DoublyListNode(curt.val);
                if (head == null) {
                        head = curtListNode;
                        prev = curtListNode;
                } else {
                        curtListNode.prev = prev;
                        prev.next = curtListNode;
                        prev = curtListNode;
                }
                        curt = curt.right;
        }
        return head;
}
```

1.9 Maximum Subarray

Given an array of integers, find a contiguous sub-array which has the largest sum. In this problem, the main point is how to skip tiny minus, and achieve highest goal. Then, we use one variable named *maxEnd*, to make the tiny minus will not effect on the current highest goal, and make it possible to get more scare in the following indexes.

```
public int maxSubArray(int[] nums) {
        // write your code
        int maxHere = nums[0], maxEnd = nums[0];
        for (int i = 1; i < nums.length; i++) {
                maxEnd = Math.max(nums[i], maxEnd + nums[i]);
                maxHere = Math.max(maxHere, maxEnd);
        }
        return maxHere;
}
```

1.10 Kth Largest Element

The easiest solution is to choose a random pivot, which yields almost certain linear time. Deterministically, one can use median-of-3 pivot strategy (as in the quicksort), which yields linear performance on partially sorted data, as is common in the real world. However, contrived sequences can still cause worst-case complexity; David Musser describes a "median-of-3 killer" sequence that allows an attack against that strategy, which was one motivation for his introselect algorithm.

To sum up, we should sort this array firstly, but only use the pivot to divided the array to two sub-array. If the pivot is our target ordered index, we return, or reduce size of sort.

```
public int kthLargestElement(int k, ArrayList<Integer> numbers) {
        // write your code here
        if (k < 1 || numbers == null) {
                return 0;
        }
        return getKth(numbers, numbers.size() - k + 1, 0, numbers.size() - 1);
}
public int getKth(ArrayList<Integer> nums, int k, int start, int end) {
        int pivot = nums.get(end);
        int left = start;
        int right = end;
        while (true) {
                while (nums.get(left) < pivot && left < right) {
                        left++;
                }
                while (nums.get(right) >= pivot && right > left) {
                        right--;
                }
                if (left == right) {
                        break;
                }
                swap(nums, left, right);
        }
        swap(nums, left, end);
        if (k == left + 1) {
                return pivot;
        } else if (k < left + 1) {
                return getKth(nums, k, start, left - 1);
        } else {
                return getKth(nums, k, left + 1, end);
        }
}
public void swap(ArrayList<Integer> nums, int n1, int n2) {
        int temp = nums.get(n1);
        nums.set(n1, nums.get(n2));
        nums.set(n2, temp);
}
```

1.11 Ugly Number

First of all, we use a priority queue to make all ugly number in the queue is ordered well. Then, we can easily to find that all ugly number is the multiply result of our factor numbers 3 5 and 7. However, we should avoid the situation that two same ugly number in the queue, for example $15 = 3 \times 5$ and $15 = 5 \times 3$. Therefore, we define a *while* loop to delete the duplicate ugly number.

```
public long kthPrimeNumber(int k) {
```

```
        // write your code here
        if (k == 1) {
                return 1;
        }
        PriorityQueue<Long> q = new PriorityQueue();
        q.add(1l);
        for (long i = 1; i < k + 1; i++) {
                long temp = q.poll();
                while (! q.isEmpty() && q.peek() == temp) {
                        temp = q.poll();
                }
                q.add(temp * 3);
                q.add(temp * 5);
                q.add(temp * 7);
        }
        return q.poll();
}
```

1.12 Digit Counts

In this problem, it has three main principle:

- when current digit is less than target number, the repeat possible is $higherdigital \times base$
- when current digit is greater than target number, the repeat possible is $(higherdigital + 1) \times base$
- when they are equal, we add $higherdigital \times base + lower + 1$

```
public int digitCounts(int k, int n) {
        int count = 0;
        int base = 1;
        while (n / base >= 1) {
                int curBit = n % (base*10) / base;
                int higher = n / (base*10);
                int lower = n % base;
                if (curBit < k) {
                        count += higher * base;
                }
                else if (curBit == k) {
                        count += higher * base + lower + 1;
                }
                else {
                        count += (higher + 1) * base;
                }
                base *= 10;
        }
        return count;
}
```

1.13 Majority Number II

Given an array of integers, the majority number is the number that occurs more than $\frac{1}{3}$ of the size of the array. Therefore, in this problem, we can update 3 candidate numbers, and use their count to compute which one is majority number.

```
public int majorityNumber(ArrayList<Integer> nums) {
        // write your code
        int candidate1 = 0;
        int candidate2 = 0;
        int count1 = 0;
        int count2 = 0;
        for (int elem : nums) {
                if (count1 == 0) {
                        candidate1 = elem;
                }
                if (count2 == 0 && elem != candidate1) {
                        candidate2 = elem;
                }
                if (candidate1 == elem) {
                        count1++;
                }
                if (candidate2 == elem) {
                        count2++;
                }
                if (candidate1 != elem && candidate2 != elem) {
                        count1--;
                        count2--;
                }
        }
        count1 = 0;
        count2 = 0;
        for (int elem : nums) {
                if (elem == candidate1) count1++;
                else if (elem == candidate2) count2++;
        }
        return count1>count2? candidate1 : candidate2;
}
```

1.14 Maximum Sub-array II

In this problem, we should get the max value from left to right, and right to left. Then, we add two elements form two lists, and get the max values.

```
public int maxTwoSubArrays(ArrayList<Integer> nums) {
        if (nums == null)
                return 0;
        int len = nums.size(), currSum = 0;
        int[] left = new int[len];
        for (int i = 0; i < len - 1; i++) {
                int sum = currSum + nums.get(i);
                if (i == 0)
                        left[i + 1] = sum;
                else
                        left[i + 1] = sum > left[i]? sum: left[i];
                currSum = sum <= 0? 0: sum;
        }
        currSum = 0;
        int max = Integer.MIN_VALUE;
        for (int i = len - 1; i > 0; i--) {
```

```
                int sum = currSum + nums.get(i);
                if (sum + left[i] > max)
                        max = sum + left[i];
                currSum = sum <= 0? 0: sum;
        }
        return max;
}
```

1.15 Maximum Sub-array III

Standard dynamic programming method, which construct a huge matrix, which saves all cost in sub-array maximum values. Then, we iteration sum first line, then move to move other cells. http://hehejun.blogspot.com.au/2015/01/lintcodemaximum-subarray-iii.html

```
public int maxSubArray(ArrayList<Integer> nums, int k) {
        // write your code
        int len = nums.size();
        int[][] f = new int[k+1][len];
        for (int i = 1; i < k+1; i++) {
                int sum = 0;
                for (int j = 0; j < i; j++) {
                        sum += nums.get(j);
                }
                f[i][i-1] = sum;
        }
        for (int i = 1; i < len; i++) {
        f[1][i] = Math.max(f[1][i-1]+nums.get(i), nums.get(i));
        }
        for (int i = 2; i < k+1; i++) {
                for (int n = i;  n< len; n++) {
                        int curMax = f[i][n-1] + nums.get(n);
                        for (int j = i-2; j < n; j++) {
                                if ((f[i-1][j] + nums.get(n)) > curMax) {
                                        curMax = f[i-1][j] + nums.get(n);
                                }
                        }
                        f[i][n] = curMax;
                }
        }
        int res = Integer.MIN_VALUE;
        for (int i = k-1; i < len; i++){
                if (f[k][i] > res) {
                        res = f[k][i];
                }
        }
        return res;
}
```

1.16 Maximum Depth of Binary Tree

It is typical recursion method problem, and we just want to know which branch can get deeper, and it is easy to write method.

```
public int maxDepth(TreeNode root) {
        // write your code here
```

```
        if (root == null) {
                return 0;
        }
        return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
}
```

1.17 Single Number

Use the definition that XOR operation, and it is easy to get unique number from a list.

```
public int singleNumber(int[] A) {
        if (A.length == 0) {
                return 0;
        }
        int n = A[0];
        for(int i = 1; i < A.length; i++) {
                n = n \^ A[i];
        }
        return n;
}
```

1.18 Rotate String

In the problem, to hardest part is that we should solve problem in-place. Therefore, we should perform reverse method to avoid use extra memory.

```
public void rotateString(char[] str, int offset) {
        // write your code here
        if (str != null && str.length != 0) {
                int n = str.length;
                offset = offset % n;
                int s = 0;
                int e = n - 1;
                reverse(str, n - offset, n - 1);
                reverse(str, 0, n - offset - 1);
                reverse(str, s, e);
        }
}
public void reverse(char[] str, int s, int e) {
        for (int i = s, j = e; i < j; i++, j--) {
                char temp = str[i];
                str[i] = str[j];
                str[j] = temp;
        }
}
```

1.19 2 Sum

In this method, we use a HashMap to store index and number information. Moreover, we also use target - currentNumber to get the pair of numbers which sum result is target number. After we meet, we add them into rst, and return it.

```
public int[] twoSum(int[] numbers, int target) {
        // write your code here
        if (numbers == null || numbers.length == 0) {
```

```
                return null;
        }
        int[] rst = new int[2];
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int i = 0; i < numbers.length; i++) {
                if (map.containsKey(target - numbers[i])) {
                        rst[0] = map.get(target - numbers[i]) + 1;
                        rst[1] = i + 1;
                } else {
                         map.put(numbers[i], i);
                }
        }
        return rst;
}
```

1.20 Reimplement Plus Operation

In this problem, we use move and XOR operation to compute the sum result of two 32 bits int numbers.

```
public int aplusb(int a, int b) {
        // write your code here, try to do it without arithmetic operators.
        int i = 0;
        int res = 0;
        int carry = 0;
        for (; i<32; i++) {
                int aa = (a >> i) & 1;
                int bb = (b >> i) & 1;
                res |= (aa ^ bb ^ carry) << i;
                if (aa == 1 && bb == 1 || ((aa ==1 || bb == 1) && carry ==
                    1)) {
                        carry = 1;
                }
                else carry = 0;
        }
        return res;
}
```

1.21 Remove Linked List Elements

The main part is that how to express the head of list node. We use a dummy to finish this job.

```
public ListNode removeElements(ListNode head, int val) {
        // Write your code here
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode curr = dummy;
        while (curr.next != null) {
                if (curr.next.val == val) {
                        curr.next = curr.next.next;
                } else {
                        curr = curr.next;
                }
        }
        return dummy.next;
}
```

1.22 Unique Solution to a Single Number

Given an array of integers, every element appears k times except for one. Find that single one which appears l times. We need a array $x[i]$ with size k for saving the bits appears i times. For every input number a, generate the new counter by $x[j] = (x[j-1]\&a)|(x[j]\&\ a)$. Except $x[0] = (x[k]\&a)|(x[0]\&\ a)$.

```
public int singleNumber(int[] A, int k, int l) {
        if (A == null) return 0;
        int t;
        int[] x = new int[k];
        x[0] = ~0;
        for (int i = 0; i < A.length; i++) {
                t = x[k-1];
                for (int j = k-1; j > 0; j--) {
                        x[j] = (x[j-1] & A[i]) | (x[j] & ~A[i]);
                }
                x[0] = (t & A[i]) | (x[0] & ~A[i]);
        }
        return x[l];
}
```

1.23 Find two Single Numbers

The two numbers that appear only once must differ at some bit, this is how we can distinguish between them. Otherwise, they will be one of the duplicate numbers. Let's say the at the ith bit, the two desired numbers differ from each other. which means one number has bit i equaling: 0, the other number has bit i equaling 1. Thus, all the numbers can be partitioned into two groups according to their bits at location i. the first group consists of all numbers whose bits at i is 0. the second group consists of all numbers whose bits at i is 1. Notice that, if a duplicate number has bit i as 0, then, two copies of it will belong to the first group. Similarly, if a duplicate number has bit i as 1, then, two copies of it will belong to the second group. by XoRing all numbers in the first group, we can get the first number. by XoRing all numbers in the second group, we can get the second number.

```
public List<Integer> singleNumberIII(int[] A) {
        // write your code here
        int xor = 0;
        for (int a : A) {
                xor = xor ^ a;
        }
        //xor = re1 ^ re2
        //as re1 != re2 thus xor != 0
        int mask = 1;
        while ((mask & xor) == 0)
                mask = mask << 1;
                // find the rightmost bit at position P such that xor@P == 1,
                // we know that re1@P != re2@P. Set the mask = 1(@P)000...
        int xor1 = 0, xor2 = 0;
        //divide the array elements into two paritions
        // according to A[i]@P == 0 / 1
        for (int a : A) {
                if ((mask & a) == 0)
                        //Let's suppose re1@P == 0,
                        // then xor1 = [xor each (A[i]@P == 1)] = re1
                        xor1 = xor1 ^ a;
                else
```

```
                        xor2 = xor2 ^ a;
                        //As re2@P == 0, then [xor each (A[i]@P == 0)] = re2
        }
        List<Integer> result = new ArrayList<Integer>();
        result.add(xor1);
        result.add(xor2);
        return result;
}
```

1.24 Lowest Common Ancestor

Given the root and two nodes in a Binary Tree. Find the lowest common ancestor(LCA) of the two nodes. The lowest common ancestor is the node with largest depth which is the ancestor of both nodes. The main point is that we should find these two nodes in the tree, and make sure they exists in the tree. Moreover, if a node which left and right nodes are our target nodes, then we can say this node is our target nodes.

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode A, TreeNode B) {
        // write your code here
        if (root == null) {
                return root;
        }
        if (root == A || root == B) {
                return root;
        } else {
                TreeNode l = lowestCommonAncestor(root.left, A, B);
                TreeNode r = lowestCommonAncestor(root.right, A, B);
                if (l != null && r != null) {
                        return root;
                } else if (l != null) {
                        return l;
                } else {
                        return r;
                }
        }
}
```

1.25 String to Integer(atoi)

In this method, we should consider of following items.

- first discards as many whitespace characters as necessary until the first non-whitespace character is found
- starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, , and interprets them as a numerical value
- The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function
- If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed enditemize

```
public int atoi(String str) {
        // write your code here
```

```
        if (str == null) {
                return 0;
        }
        str = str.trim();
        if (str.length() == 0) {
                return 0;
        }
        int sign = 1;
        int index = 0;
        if (str.charAt(index) == '+') {
                index++;
        } else if (str.charAt(index) == '-') {
                sign = -1;
                index++;
        }
        long num = 0;
        for (; index < str.length(); index++) {
                if (str.charAt(index) < '0' || str.charAt(index) > '9') {
                        break;
                }
                num = num * 10 + (str.charAt(index) - '0');
                if (num > Integer.MAX_VALUE) {
                        break;
                }
        }
        if (num * sign >= Integer.MAX_VALUE) {
                return Integer.MAX_VALUE;
        }
        if (num * sign <= Integer.MIN_VALUE) {
                return Integer.MIN_VALUE;
        }
        return (int)num * sign;
}
```

1.26 Product of Array Exclude Itself

In this problem, we use the multiply results from left to right, and right to left, then sum them to get our result result.

```
public ArrayList<Long> productExcludeItself(ArrayList<Integer> A) {
        // write your code
        ArrayList<Long> res = new ArrayList<Long>();
        if (A == null || A.size() == 0) {
                return res;
        }
        if (A.size() == 1) {
                res.add(1l);
                return res;
        }
        long[] lProduct = new long[A.size()];
        long[] rProduct = new long[A.size()];
        lProduct[0] = 1;
        for (int i=1; i<A.size(); i++) {
                lProduct[i] = lProduct[i-1] * A.get(i-1);
```

```
        }
        rProduct[A.size()-1] = 1;
        for (int j=A.size()-2; j>=0; j--) {
                rProduct[j] = rProduct[j+1]*A.get(j+1);
        }
        for (int k=0; k<A.size(); k++) {
                res.add(lProduct[k] * rProduct[k]);
        }
        return res;
}
```

1.27 Binary Tree Serialization

In this method, we have two sub-methods, which include serialize & deserialize. The serialize method is easy to understand and implement by perform pre-order traversal, and add traversal string into a total string. For the deserialize method, we should divide total string into several tokens, and if is "#", means empty node, then construct a node by current vale, then get its left node and right nodes.

```
public String serialize(TreeNode root) {
        // write your code here
        String result = "";
        if (root == null) {
                result = "# ";
                return result;
        }
        result += root.val + " ";
        result += serialize(root.left);
        result += serialize(root.right);
        return result;
}

public TreeNode deserialize(String data) {
        // write your code here
        TreeNode result = null;
        if (data == null || data.length() == 0) {
                return result;
        }
        StringTokenizer st = new StringTokenizer(data, " ");
        return deserialize(st);
}
private TreeNode deserialize(StringTokenizer st) {
        if (!st.hasMoreTokens()) {
                return null;
        }
        String val = st.nextToken();
        if (val.equals("#")) {
                return null;
        }
        TreeNode root = new TreeNode(Integer.parseInt(val));
        root.left = deserialize(st);
        root.right = deserialize(st);
        return root;
}
```

1.28 Wildcard Matching

```
public boolean isMatch(String s, String p) {
        int i = 0;
        int j = 0;
        int star = -1;
        int mark = -1;
        while (i < s.length()) {
                if (j < p.length()
                        && (p.charAt(j) == '?' || p.charAt(j) ==
                            s.charAt(i))) {
                        ++i;
                        ++j;
                } else if (j < p.length() && p.charAt(j) == '*') {
                        star = j++;
                        mark = i;
                } else if (star != -1) {
                        j = star + 1;
                        i = ++mark;
                } else {
                        return false;
                }
        }
        while (j < p.length() && p.charAt(j) == '*') {
                ++j;
        }
        return j == p.length();
}
```

1.29 Linked List Cycle II

The main part is that we should make sure that it exists a cycle in the linked list. Then, if we can find next node, we make quick pointer move twice, or return null to get there is no cycle in the linked list.

```
public ListNode detectCycle(ListNode head) {
        // write your code here
        if (head == null) {
                return head;
        }
        ListNode slowNode = head;
        ListNode quickNode = head;
        while (slowNode != null && quickNode != null) {
                slowNode = slowNode.next;
                quickNode = quickNode.next;
                if (quickNode != null) {
                        quickNode = quickNode.next;
                }
                if (quickNode == slowNode) {
                        break;
                }
        }
        if (quickNode == null) {
                return null;
        }
        slowNode = head;
        while (slowNode != quickNode) {
```

```
                slowNode = slowNode.next;
                quickNode = quickNode.next;
        }
        return quickNode;
}
```

1.30 Data Stream Median

In this problem, we will apply priority queue to sort stream, and return the median when it needs. The main point is using two queue, and when facing a new number, we make a decision on which queue should be add, then when we face the size of two queues are not same, we change one elements in the larger queue to another.

```
public int[] medianII(int[] nums) {
        // write your code here
        if(nums==null) {
                return null;
        }
        int[] res = new int[nums.length];
        //left half and the median are stored in the maxHeap. median ==
           maxHeap.peek()
        //right half of the median are stored in the minHeap.
        PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>();
           //default min heap
        PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(11
           /*default capacity*/, new Comparator<Integer>() {
        @Override
        public int compare(Integer x, Integer y) {
                return y-x;
        }
        });
        res[0] = nums[0];
        maxHeap.add(nums[0]);
        for(int i=1; i<nums.length; i++) {
                int x = maxHeap.peek();
                if(nums[i] <= x) {
                        maxHeap.add(nums[i]);
                } else {
                        minHeap.add(nums[i]);
                }
                //maxHeap.size() == minHeap.size() || maxHeap.size() ==
                   minHeap.size() + 1;
                if(maxHeap.size() > minHeap.size()+1 ) {
                        minHeap.add(maxHeap.poll());
                } else if(maxHeap.size() < minHeap.size()) {
                        maxHeap.add(minHeap.poll());
                }
                res[i] = maxHeap.peek();
        }
        return res;
}
```

1.31 Sliding Window Maximum

Given an array of n integer with duplicate number, and a moving window(size k), move the window at each iteration from the start of the array, find the maximum number inside the window at each moving. In the problem, we use a *deque*, and update them to get the maximum values during the sliding process.

```
public ArrayList<Integer> maxSlidingWindow(int[] nums, int k) {
        // write your code here
        ArrayList<Integer> res = new ArrayList<Integer>();
        if (nums == null) {
                return null;
        }
        if (nums.length == 0) {
                return res;
        }
        Deque<Integer> deque = new LinkedList<Integer>();
        for(int i = 0;i < nums.length; i++){
                if(i >= k && deque.getLast() == nums[i - k]) {
                        deque.removeLast();
                }
                while(!deque.isEmpty() && deque.getFirst() < nums[i]) {
                        deque.removeFirst();
                }
                deque.addFirst(nums[i]);
                if(i + 1 >= k) {
                        res.add(deque.getLast());
                }
        }
        return res;
}
```

2 US Giants

2.1 String to Char

In these problem, I use a map to save the relationship between char and its frequency, then compare this map with next string content.

```
public boolean anagram(String s, String t) {
        // write your code here
        boolean flag = false;
        if (s == null || t == null || s.length() != t.length()) {
                return flag;
        }
        Map<Character, Integer> map = new HashMap<Character, Integer>();
        for (int i = 0; i < s.length(); i++) {
                char c = s.charAt(i);
                if (map.containsKey(c)) {
                        map.put(c, map.get(c) + 1);
                } else {
                        map.put(c, 1);
                }
        }
        for (int i = 0; i < t.length(); i++) {
                char c = t.charAt(i);
```

```
                if (map.containsKey(c)) {
                        if (map.get(c) - 1 < 0) {
                                return false;
                        }
                        map.put(c, map.get(c) - 1);
                } else {
                        return false;
                }
        }
        for (int value : map.values()) {
                if (value != 0) {
                        return false;
                }
        }
        return true;
}
```

2.2 KMP Algorithm

There is a simple method, which can reduce the size of first loop.

```
public int strStr(String source, String target) {
        if (source == null || target == null) {
                return -1;
        }
        int i, j;
        for (i = 0; i < source.length() - target.length() + 1; i++) {
                for (j = 0; j < target.length(); j++) {
                        if (source.charAt(i + j) != target.charAt(j)) {
                                break;
            }
        }
                if (j == target.length()) {
                        return i;
                }
        }
        return -1;
}
```

We can also apply KMP algorithm.

```
public int strStr(String haystack, String needle) {
        if(haystack == null || needle == null)
                return 0;
        int h = haystack.length();
        int n = needle.length();

        if (n > h)
                return -1;
        if (n == 0)
                return 0;

        int[] next = getNext(needle);
        int i = 0;

        while (i <= h - n) {
```

```
                int success = 1;
                for (int j = 0; j < n; j++) {
                        if (needle.charAt(0) != haystack.charAt(i)) {
                                success = 0;
                                i++;
                                break;
                        } else if (needle.charAt(j) != haystack.charAt(i +
                            j)) {
                                success = 0;
                                i = i + j - next[j - 1];
                                break;
                        }
                }
                if (success == 1)
                        return i;
        }

        return -1;
}

//calculate KMP array
public int[] getNext(String needle) {
        int[] next = new int[needle.length()];
        next[0] = 0;

        for (int i = 1; i < needle.length(); i++) {
                int index = next[i - 1];
                while (index > 0 && needle.charAt(index) != needle.charAt(i))
                    {
                        index = next[index - 1];
                }

                if (needle.charAt(index) == needle.charAt(i)) {
                        next[i] = next[i - 1] + 1;
                } else {
                        next[i] = 0;
                }
        }

        return next;
}
```

2.3 Anagrams

This kind of problem is focus on the sorted result of a string. We perform char list sort of the given string, then use the sorted string as unique key to located the anagrams of string.

```
public List<String> anagrams(String[] strs) {
        // write your code here
        List<String> res = new ArrayList<String>();
        Map<String, List<String>> dict = new HashMap<String, List<String>>();
        for (String s : strs) {
                char[] ss = s.toCharArray();
                Arrays.sort(ss);
```

```
                String tempString = new String(ss);
                if (! dict.containsKey(tempString)) {
                        dict.put(tempString, new ArrayList<String>());
                }
                dict.get(tempString).add(s);
        }
        for (Map.Entry<String, List<String>> entry : dict.entrySet()) {
                if (entry.getValue().size() > 1) {
                        res.addAll(entry.getValue());
                }
        }
        return res;
}
```

2.4 Longest Common Subsequence

Given two strings, find the longest common subsequence (LCS). Your code should return the length of LCS. For "ABCD" and "EDCA", the LCS is "A" (or "D", "C"), return 1. For "ABCD" and "EACB", the LCS is "AC", return 2. In this problem, we use dynamic programming method to solve.

```
public int longestCommonSubsequence(String A, String B) {
        // write your code here
        int M = A.length();
        int N = B.length();
        int res = 0;

        // opt[i][j] = length of LCS of x[i..M] and y[j..N]
        int[][] opt = new int[M + 1][N + 1];

        // compute length of LCS and all subproblems via dynamic programming
        for (int i = M - 1; i >= 0; i--) {
                for (int j = N - 1; j >= 0; j--) {
                        if (A.charAt(i) == B.charAt(j)) {
                                opt[i][j] = opt[i + 1][j + 1] + 1;
                        }
                        else {
                                opt[i][j] = Math.max(opt[i + 1][j], opt[i][j
                                    + 1]);
                        }
                        res = Math.max(res, opt[i][j]);
                }
        }
        return res;
}
```

2.5 Longest Common Sub-string

Given two strings, find the longest common substring. Return the length of it. The only different of LCS, is that in the opt matrix, it construct from 1 to end, and when two strings have the same char, then assign its location to left-top plus 1. FOr the LCS, it assign right-down plus 1 when facing a same char. Moreover, int the LCSs, it assigns other indexes to 0. But for the LCS, it assigns the max values of right and down values. The similar things is that they both return the max values in the matrix as their target numbers.

```
public int longestCommonSubstring(String A, String B) {
        // write your code here
```

```
        if (A == null || B == null || A.length() == 0 || B.length() == 0) {
                return 0;
        }
        int[][] opt = new int[A.length() + 1][B.length() + 1];
        int result = 0;
        for (int i = 1; i < A.length() + 1; i++) {
                for (int j = 1; j < B.length() + 1; j++) {
                        if (i == 0 && j == 0) {
                                opt[i][j] = 0;
                        } else if (A.charAt(i - 1) == B.charAt(j - 1)) {
                                opt[i][j] = opt[i - 1][j - 1] + 1;
                                result = Math.max(result, opt[i][j]);
                        } else {
                                opt[i][j] = 0;
                        }
                }
        }
        return result;
}
```

2.6 Longest Common Prefix

In this problem, it is easy to solve by compare the prefix based on chars in first string. When facing the different, we return the current result.

```
public String longestCommonPrefix(String[] strs) {
        // write your code here
        if (strs == null || strs.length == 0) {
                return "";
        }
        for (int i = 0; i < strs[0].length(); i++) {
                char c = strs[0].charAt(i);
                for (int j = 1; j < strs.length; j++) {
                        if (i >= strs[j].length() || strs[j].charAt(i) != c) {
                                return strs[0].substring(0, i);
                        }
                }
        }
        return strs[0];
}
```

2.7 Edit Distance

In the problem, it assumes that each string can be change to another change by perform length of string operations. Therefore, in the initialize operation of dynamic programming matrix, we increase the values of first column and first row. After that, we loop this matrix, when facing a same char between two words, we assign this cell with the minimum value of its left-top, left, and top (which means there is no need the add a new operation). If they are not same, we also get the minimum vales of its left-top, left and top, but plus 1 to this cell.

```
public int minDistance(String word1, String word2) {
        // write your code here
        int len1 = 0, len2 = 0;
        if (word1 != null && word2 != null) {
                len1 = word1.length();
```

```
                len2 = word2.length();
        }
        if (word1 == null || word2 == null) {
                return Math.max(len1, len2);
        }
        int[][] dp = new int[len1 + 1][len2 + 1];
        for (int i = 0; i <= len1; i++) {
                dp[i][0] = i;
        }
        for (int i = 0; i <= len2; i++) {
                dp[0][i] = i;
        }
        for (int i = 1; i <= len1; i++) {
                for (int j = 1; j <= len2; j++) {
                        if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
                                dp[i][j] = Math.min(dp[i - 1][j - 1], 1 +
                                    dp[i - 1][j]);
                                dp[i][j] = Math.min(dp[i][j], 1 + dp[i][j -
                                    1]);
                        } else {
                                dp[i][j] = Math.min(dp[i - 1][j - 1], dp[i -
                                    1][j]);
                                dp[i][j] = 1 + Math.min(dp[i][j], dp[i][j -
                                    1]);
                        }
                }
        }
        return dp[len1][len2];
}
```

2.8 Valid Palindrome

The simple solution is based on the filter action on whether this char is a alpha or a number based on its char value is between 'a' and 'z', and '0' and '9'.

```
public boolean isPalindrome(String s) {
        // Write your code here
        if (s == null || s.isEmpty()) return true;
        int l = 0, r = s.length() - 1;
        while (l < r) {
                // find left alphanumeric character
                if (!Character.isLetterOrDigit(s.charAt(l))) {
                        l++;
                        continue;
                }
                // find right alphanumeric character
                if (!Character.isLetterOrDigit(s.charAt(r))) {
                        r--;
                continue;
                }
                // case insensitive compare
                if (Character.toLowerCase(s.charAt(l)) ==
                    Character.toLowerCase(s.charAt(r))) {
                        l++;
```

```
                r--;
            } else {
                return false;
            }
        }
        return true;
}
```

2.9 Remove Element

In this problem, consider of that there is no need to worry about the order of new list, then we just change the first value which is equal to elem from the last value of list. This can be assumed as a kind of partition. Then, the return int will be used to divided original list into two parts, and first part is list which remove all elements.

```
public int removeElement(int[] A, int elem) {
        // write your code here
        if (A == null ||A.length == 0) {
            return 0;
        }
        int n = A.length;
        for (int i = 0; i < n; i++) {
            if (A[i] == elem) {
                A[i] = A[n - 1];
                --i; // we should consider of change one is still not
                    elem
                --n;
            }
        }
        return n;
}
```

2.10 Subarray Zero Sum

In this situation, we should consider of three kinds of condition which sum is zero. The current value is zero, the sum value (from 0 to current index) is zero, and values which in the map have the same values.

```
public ArrayList<Integer> subarraySum(int[] nums) {
        // write your code here
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (nums == null || nums.length == 0) {
            return result;
        }
        // Creates an empty hashMap hM
        HashMap<Integer, Integer> hM = new HashMap<Integer, Integer>();
        int sum = 0;
        for (int i = 0; i < nums.length; i++)
        {
            sum += nums[i];
            if (nums[i] == 0) {
                result.add(i);
                result.add(i);
                return result;
            } else if (sum == 0) {
                result.add(0);
```

```
                        result.add(i);
                        return result;
                } else if (hM.get(sum) != null) {
                        result.add(hM.get(sum) + 1);
                        result.add(i);
                        return result;
                }

                // Add sum to hash map
                hM.put(sum, i);
        }
        return result;
}
```

2.11 Remove Duplicates from Sorted Array

Using a index to loop the array, and only left the unique values in the loop.

```
public int removeDuplicates(int[] nums) {
        // write your code here
        if (nums == null || nums.length == 0) {
                return 0;
        }
        if (nums.length == 1) {
                return 1;
        }
        int newIndex = 0;
        for (int i = 0; i < nums.length; i++) {
                if (nums[i] != nums[newIndex]) {
                        newIndex++;
                        nums[newIndex] = nums[i];
                }
        }
        return newIndex + 1;
}
```

2.12 First Missing Positive

This is not similar to find first missing number because the numbers in this list are not all positive. Moreover, it also has the problem on memory usage. Then, we perform bucket sort, and find the first value which not fulfil the equation that $value = index + 1$. In this process, we use a definition that the sorted value should located at index that value minus one. Then, if we meet a value which are not in its suitable location, we change it to its suitable location (assume the list have enough space).

```
public int firstMissingPositive(int[] A) {
        // write your code here
        for (int i = 0; i < A.length; i++) {
                while (A[i] > 0 && A[i] <= A.length && (A[i] != i + 1) &&
                    (A[i] != A[A[i] - 1])) {
                        int temp = A[A[i] - 1];
                        A[A[i] - 1] = A[i];
                        A[i] = temp;
                }
        }
        for (int i = 0; i < A.length; i++) {
```

```
                if (A[i] != i + 1) {
                        return i + 1;
                }
        }
        return A.length + 1;
}
```

2.13 3 Sum Closest

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. In this kind of situation, once you want to overcome the n^3 time complexity on 3 values, you have to control the two pointers by your self to save time. Then, we loop one index i, and control other indexes j and k by ourselves.

```
public int threeSumClosest(int[] numbers ,int target) {
        // write your code here
        if (numbers == null) {
                return 0;
        }
        if (numbers.length < 3) {
                int sum = 0;
                for (int num : numbers) {
                        sum += num;
                }
                return sum;
        }
        Arrays.sort(numbers);
        int result = numbers[0] + numbers[1] + numbers[2];
        int n = numbers.length;
        int temp = 0;
        for (int i = 0; i < n - 2; i++) {
                int j = i + 1, k = n - 1;
                while (j < k) {
                temp = numbers[i] + numbers[j] + numbers[k];
                        if (Math.abs(target - result) > Math.abs(target -
                            temp)) {
                                result = temp;
                        }
                        if (result == target) {
                                return result;
                        }
                        if (temp > target) {
                                k--;
                        } else {
                                j++;
                        }
                }
        }
        return result;
}
```

2.14 3 Sum

In this problem, we should overcome the values in the list which have same values. Then, we use a *while* loop to move all value which have same value with its neighbour.

```
public ArrayList<ArrayList<Integer>> threeSum(int[] numbers) {
        // write your code here
        ArrayList<ArrayList<Integer>> resultList = new
            ArrayList<ArrayList<Integer>>();
        if (numbers == null || numbers.length < 3) {
                return resultList;
        }
        Arrays.sort(numbers);
        for (int i = 0; i < numbers.length; i++) {
                if (i > 0 && numbers[i] == numbers[i - 1]) {
                continue;
                }
                int j = i + 1;
                int k = numbers.length - 1;
                while (j < k) {
                        int ans = numbers[i] + numbers[j] + numbers[k];
                        if (ans == 0) {
                                ArrayList<Integer> temp = new
                                    ArrayList<Integer>();
                                temp.add(numbers[i]);
                                temp.add(numbers[j]);
                                temp.add(numbers[k]);
                                resultList.add(temp);
                                j++;
                                while (j < numbers.length && numbers[j] ==
                                    numbers[j - 1]) {
                                        j++;
                                }
                                k--;
                                while (k >= 0 && numbers[k] == numbers[k +
                                    1]) {
                                        k--;
                                }
                        } else if (ans > 0) {
                                k--;
                        } else {
                                j++;
                        }
                }
        }
        return resultList;
}
```

2.15 Partition Array

The partition process (from quicksort algorithm), it applies two pointers, and exchange the values of two values. However, in the full version of quick sort, it use pivot as the target value, but this problem, we only use k as pivot and divided array into two parts.

```
public int partitionArray(int[] nums, int k) {
```

```
        //write your code here
        if (nums == null) {
                return 0;
        }
        int left = 0, right = nums.length - 1;
        while (left <= right) {
                while (left <= right && nums[left] < k) {
                        left++;
                }
                while (left <= right && nums[right] >= k) {
                        right--;
                }
                // find first pair which need to be replaced
                if (left <= right) {
                        int temp = nums[left];
                        nums[left] = nums[right];
                        nums[right] = temp;
                        left++;
                        right--;
                }
        }
        return left;
}
```

2.16 Sub-array Sum Closet

In this problem, it aims to get the minimum closet value to zero. Which means that once we find a new sum result which is less than before, we should clean our result list, and insert this new pair. Moreover, in order to save time and space complexity, we use a instance named pair which save the end index value and sum result from zero index. Then, we sort this pair list, and get the minimum gap between two sum values, and get the closet sum of different sub-array.

```
class Pair {
        int sum;
        int index;
        public Pair(int s, int i) {
                sum = s;
                index = i;
        }
}
public ArrayList<Integer> subarraySumClosest(int[] nums) {
        // write your code here
        ArrayList<Integer> res = new ArrayList<Integer> ();
        if (nums == null || nums.length == 0) {
                return res;
        }
        int len = nums.length;
        if(len == 1) {
                res.add(0);
                res.add(0);
                return res;
        }
        Pair[] sums = new Pair[len+1];
        int prev = 0;
```

```
        sums[0] = new Pair(0, 0);
        for (int i = 1; i <= len; i++) {
                sums[i] = new Pair(prev + nums[i-1], i);
                prev = sums[i].sum;
        }
        Arrays.sort(sums, new Comparator<Pair>() {
                public int compare(Pair a, Pair b) {
                        return a.sum - b.sum;
                }
        });
        int ans = Integer.MAX_VALUE;
        for (int i = 1; i <= len; i++) {
                if (ans > sums[i].sum - sums[i-1].sum) {
                        ans = sums[i].sum - sums[i-1].sum;
                        res.clear();
                        int[] temp = new int[]{sums[i].index - 1, sums[i -
                            1].index - 1};
                        Arrays.sort(temp);
                        res.add(temp[0] + 1);
                        res.add(temp[1]);
                }
        }
        return res;
}
```

2.17 Sqrt - by Newtonś Method

```
public int sqrt(int x) {
        // write your code here
        if (x == 0) {
                return 0;
        }
        double last = 0;
        double res = 1;
        while (res != last) {
                last = res;
                res = (res + x / res) / 2;
        }
        return (int)res;
}
```

2.18 Search Insert Position

The classic binary search process. Note: the mid is $start + (end - start) \div 2$. And the start starts from -1 and end starts from length of list. And our target index is start plus 1.

```
public int searchInsert(int[] A, int target) {
        if (A == null || A.length == 0) {
                return 0;
        }
        int start = -1, end = A.length;
        while (start + 1 < end) {
                int mid = start + (end - start) / 2;
                if (A[mid] == target) {
                        return mid; // no duplicates
```

```
                } else if (A[mid] < target) {
                        start = mid;
                } else {
                        end = mid;
                }
        }
    return start + 1;
}
```

2.19 Wood Cut

We **guess** a result as mid, and use this mid to compute it fulfil the condition or not. If fulfil, we add mid, else we reduce.

```
public int woodCut(int[] L, int k) {
        // write your code here
        if (L == null || L.length == 0) {
                return 0;
        }
        int end = Integer.MAX_VALUE;
        int start = 0;
        while (start + 1 < end) {
                int mid = start + (end - start) / 2;
                if (C(L, k, mid)) {
                        start = mid;
                } else {
                        end = mid;
                }
        }
        return start;
}
private boolean C(int[] L, int k, int x) {
        int sum = 0;
        for (int l : L) {
                sum += l / x;
        }
        if (sum >= k) {
                return true;
        } else {
                return false;
        }
}
```

2.20 Find Peak Element II

Write a method to test it is fulfil the peak condition. If is not, we move to a greater postion to find the peak.

```
private boolean isOK(int[][] A, int i, int j) {
        if (i < 1 || i >= A.length - 1 || j < 1 || j >= A[0].length - 1) {
                return false;
        }
        return A[i][j] > A[i - 1][j] && A[i][j] > A[i][j - 1] &&
            A[i][j] > A[i][j + 1] && A[i][j] > A[i + 1][j];
}
```

```
public List<Integer> findPeakII(int[][] A) {
        // write your code here
        List<Integer> res = new ArrayList<Integer>();
        if (A == null) {
                return res;
        }
        int i = 1, j = 1;
        while (i < A.length - 1 && j < A[0].length - 1) {
                if (isOK(A, i, j)) {
                        res.add(i);
                        res.add(j);
                        return res;
                } else {
                        if (A[i + 1][j] > A[i][j + 1]) {
                                i++;
                        } else {
                                j++;
                        }
                }
        }
        return res;
}
```

2.21 Search in Rotated Sorted Array

In this problem, we should draw a picture to analysis the location of our target values. If it exist duplicate numbers, we just add start number to move the location (when mid number equals to start number).

```
public int search(int[] A, int target) {
        // write your code here
        if (A == null || A.length == 0) {
                return -1;
        }
        int start = 0;
        int end = A.length - 1;
        while (start + 1 < end) {
                int mid = start + (end - start) / 2;
                if (A[mid] == target) {
                        return mid;
                }
                if (A[mid] > A[start]) {
                        if (A[start] <= target && target <= A[mid]) {
                                end = mid;
                        } else {
                                start = mid;
                        }
                } else {
                        if (A[mid] <= target && target <= A[end]) {
                                start = mid;
                        } else {
                                end = mid;
                        }
                }
        }
```

```
        if (A[start] == target) {
                return start;
        } else if (A[end] == target) {
                return end;
        }
        return -1;
}
```

2.22 Unique Path

If we want to get the unique path from top-left to bottom-right of a matrix. We just make first column and first row equal to 1. And assign dp[i][j] equals to dp[i][j - 1] + dp[i - 1][j]; If we set obstacles into matrix, when we face a obstacles, we set the current location is 0.

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        // write your code here
        if (obstacleGrid.length < 1 || obstacleGrid[0].length < 1) {
                return 0;
        }
        int m = obstacleGrid.length;
        int n = obstacleGrid[0].length;
        int[][] dp = new int[m][n];
        for (int j = 0; j < n; j++) {
                dp[0][j] = 0;
        }
        for (int j = 0; j < n; j++) {
                if (obstacleGrid[0][j] != 1) {
                        dp[0][j] = 1;
                } else {
                        break;
                }
        }
        for (int i = 0; i < m; i++) {
                dp[i][0] = 0;
        }
        for (int i = 0; i < m; i++) {
                if (obstacleGrid[i][0] != 1) {
                        dp[i][0] = 1;
                } else {
                        break;
                }
        }
        for (int i = 1; i < m; i++) {
                for (int j = 1; j < n; j++) {
                        if (obstacleGrid[i][j] == 1) {
                                dp[i][j] = 0;
                        } else {
                                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
                        }
                }
        }
        return dp[m - 1][n - 1];
}
```

2.23 Trialling Zeros

Based on the definition of factorial, it is easy to notice that quantity of 5 of the input number can decide how many 0 at the trialling (one pair of 2 and 5 can contribute to a 10, but the number of 2 is much more - each even number have a 2). Then, we just need to compute how many 5 in the given number.

```
public long trailingZeros(long n) {
        // write your code here
        if (n < 0) {
                return -1l;
        }
        long count = 0;
        for (; n > 0; n /= 5) {
                count += (n / 5);
        }
        return count;
}
```

2.24 Update Bits

First of all, construct a mask code, which have index i to index j are 0, and others are 1. After get mask code, we move m with i digits, and use mask code to and original number n, then sum them to get result.

```
public int updateBits(int n, int m, int i, int j) {
        // write your code here
        int mask;
        if (j < 31) {
                mask = ~((1 << (j + 1)) - (1 << i));
        } else {
                mask = (1 << i) - 1;
        }
        return (m << i) + (mask & n);
}
```

2.25 Unique Binary Search Tree I & II

In this problem, they are math problem, which former is the the *Catalan* number result, and the latter one is the process on how to construct the binary tree.

```
public int numTrees(int n) {
        // write your code here
        if (n < 0) {
        return -1;
        }
        int[] count = new int[n + 1];
        count[0] = 1;
        for (int i = 1; i < n + 1; ++i) {
                for (int j = 0; j < i; ++j) {
                        count[i] += count[j] * count[i - j - 1];
                }
        }
        return count[n];
}

public List<TreeNode> generateTrees(int n) {
        // write your code here
```

```
        return generateTrees(1, n);
}

public List<TreeNode> generateTrees(int start, int end) {
        List<TreeNode> list = new LinkedList<>();
        if (start > end) {
                list.add(null);
                return list;
        }
        for (int i = start; i <= end; i++) {
                List<TreeNode> lefts = generateTrees(start, i - 1);
                List<TreeNode> rights = generateTrees(i + 1, end);
                for (TreeNode left : lefts) {
                        for (TreeNode right : rights) {
                                TreeNode node = new TreeNode(i);
                                node.left = left;
                                node.right = right;
                                list.add(node);
                        }
                }
        }
        return list;
}
```

2.26 Binary Representation

The main problem is how the compute represent to float number. For the part less than 1, we use it to multiply 2, and once its result is greater than 1, we set 1 in the binary (then minus 1), else is 0.

```
private String parseInteger(String str) {
        int n = Integer.parseInt(str);
        if (str.equals("") || str.equals("0")) {
                return "0";
        }
        String binary = "";
        while (n != 0) {
                binary = Integer.toString(n % 2) + binary;
                n = n / 2;
        }
        return binary;
}
private String parseFloat(String str) {
        double d = Double.parseDouble("0." + str);
        String binary = "";
        HashSet<Double> set = new HashSet<Double>();
        while (d > 0) {
                if (binary.length() > 32 || set.contains(d)) {
                        return "ERROR";
                }
                set.add(d);
                d = d * 2;
                if (d >= 1) {
                        binary = binary + "1";
                        d = d - 1;
```

```
                } else {
                        binary = binary + "0";
                }
        }
        return binary;
}
public String binaryRepresentation(String n) {
        if (n.indexOf('.') == -1) {
                return parseInteger(n);
        }
        String[] params = n.split("\\.");
        String flt = parseFloat(params[1]);
        if (flt == "ERROR") {
                return flt;
        }
        if (flt.equals("0") || flt.equals("")) {
                return parseInteger(params[0]);
        }
        return parseInteger(params[0]) + "." + flt;
}
```

2.27 Gas Station

In this problem, it has a main point that it may not exist the solution, which means that no matter where we start our car, the total gas is less than total cost. Then, we use a variables named *total* to make a decision on available solution. Moreover, it the current sum is greater than 0, then the start index is index plus one (because last time cause error, and assign this index).

```
public int canCompleteCircuit(int[] gas, int[] cost) {
        // write your code here
        if (gas == null || cost == null || gas.length == 0 || cost.length ==
            0) {
                return -1;
        }
        int sum = 0;
        int total = 0;
        int index = -1;
        for (int i = 0; i < gas.length; i++) {
                sum += gas[i] - cost[i];
                total += gas[i] - cost[i];
                if (sum < 0) {
                        index = i;
                        sum = 0;
                }
        }
        if (total < 0) {
                return -1;
        } else {
                return index + 1;
        }
}
```

2.28 Largest Number

The key process is compare two strings. We override a comparator of two strings. After that, we filter 0 in the return strings.

```
public String largestNumber(int[] num) {
        // write your code here
        if (num == null || num.length == 0) {
                return "";
        }
        String[] strs = new String[num.length];
        for (int i = 0; i < num.length; i++) {
                strs[i] = Integer.toString(num[i]);
        }
        Arrays.sort(strs, new NumbersComparator());
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < strs.length; i++) {
                sb.append(strs[i]);
        }
        String result = sb.toString();
        int index = 0;
        while (index < result.length() && result.charAt(index) == '0') {
                index++;
        }
        if (index == result.length()) {
                return "0";
        }
        return result.substring(index);
}

class NumbersComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        return (s2 + s1).compareTo(s1 + s2);
    }
}
```

2.29 Delete Digits

We use a stack to sort the string, and get the smallest digits of total length minus k. Then, we use these sorted well stack to construct a number.

```
public String DeleteDigits(String A, int k) {
        // write your code here
        Stack<Integer> st = new Stack<Integer>();
        int popCount = 0;
        StringBuffer res = new StringBuffer();
        for (int i = 0; i < A.length(); i++) { // get smallest values in A
                int num = (int)(A.charAt(i) - '0'); // get value
                if (st.isEmpty()) {
                        st.push(num);
                } else if (num >= st.peek()) {
                        st.push(num); // sort in the stack
                } else {
                        if (popCount < k) {
```

```
                                st.pop();
                                i--;
                                popCount++;
                        } else {
                                st.push(num);
                        }
                }
        }
        while (popCount < k) {
                st.pop();
                popCount++;
        }
        while (!st.isEmpty()) {
                res.insert(0, st.pop());
        }
        while (res.length() > 1 && res.charAt(0) == '0') {
                res.deleteCharAt(0); // filter 0 at start
        }
        return res.toString();
}
```

2.30 Jump Game I & II

In the jump game, we should consider that whether the current index is smaller than farthest available index, and if this current index available range plus its location is greater than farthest available index, which means that we can move to this index to get farther locations. Therefore, we move to this index. When we achieve the farthest index, if it is greater than end of list, we achieve the end of list.

```
public boolean canJump(int[] A) {
        // wirte your code here
        if (A == null || A.length == 0) {
                return false;
        }
        int farest = A[0];
        for (int i = 0; i < A.length; i++) {
                if (i <= farest && A[i] + i > farest) {
                        farest = i + A[i];
                }
        }
        return farest >= A.length - 1;
}
```

In the jump game II, we use a range to represent to available area.

```
public int jump(int[] A) {
        // write your code here
        if (A == null || A.length == 0) {
                return 0;
        }
        int start = 0, end = 0, jumps = 0;
        while (end < A.length - 1) {
            int farthest = end;
            for (int i = start; i <= end; i++) {
                if (i + A[i] >= farthest) {
                    farthest = i + A[i];
                }
```

```
            }
            if (end < farthest) {
                jumps++;
                start = end + 1; // represent to a range
                end = farthest;
            } else {
                return -1;
            }
        }
        return jumps;
}
```

2.31 Next & Previous Permutation

In the problem on *Permutation*, the main process is to find the value which is greater or less than current digit, and then swap them. But this time, it is not enough, we should reverse the values from least values because we should get the maximum value following this digital. In the other word, we find $a[k] < a[k+1]$, then find $a[k] < a[l]$, and swap k and l, and reverse $k+1$ to end. For the previous part, it change to $a[k] > a[k+1]$ and $a[k] > a[l]$.

```
public int[] nextPermutation(int[] nums) {
        // write your code here
        if (nums == null || nums.length < 2) {
                return nums;
        }
        int len = nums.length;
        for (int i = len - 2; i >= 0; i--) {
                if (nums[i + 1] > nums[i]) {
                        int j;
                        for (j = len - 1; j > i - 1; j--) {
                                if (nums[j] > nums[i]) {
                                break;
                }
                        }
                        swap(nums, i, j);
                        reverse(nums, i + 1, len - 1);
                        return nums;
                }
        }
        reverse(nums, 0, len - 1);
        return nums;
}
```

Note: the *lexicographically* order which make the second index search from start of the list, and swap them, and reverse the list after first searched index.

```
public void nextPermutation(int[] nums) {
        // write your code here
        if (nums == null || nums.length < 2) {
                return;
        }
        int k = -1, l = 0;
        for (int i = 0; i < nums.length - 1; ++i) {
                if (nums[i] < nums[i + 1]) {
                        k = i;
                }
```

```
        }
        if (k >= 0) {
                for (int i = 0; i < nums.length; ++i) {
                        if (nums[i] > nums[k]) {
                                l = i;
                        }
                }
                swap(nums, k, l);
        }
        reverse(nums, k + 1, nums.length - 1);
}
```

2.32 Permutation II

Given a list of numbers with duplicate number in it. Find all unique permutations. Which means that when we face same number during construction process, we skip this step.

```
public void helper(ArrayList<ArrayList<Integer>> result, ArrayList<Integer>
    list, int[] visited, ArrayList<Integer> num) {
        if(list.size() == num.size()) {
                result.add(new ArrayList<Integer>(list));
                return;
        }
        for(int i = 0; i < num.size(); i++) {
                if (visited[i] == 1 || (i != 0 && num.get(i) == num.get(i -
                    1) && visited[i - 1] == 0)){
                continue;
                }
                visited[i] = 1;
                list.add(num.get(i));
                helper(result, list, visited, num);
                list.remove(list.size() - 1);
                visited[i] = 0;
        }
}

public ArrayList<ArrayList<Integer>> permuteUnique(ArrayList<Integer> nums) {
        // write your code here
        ArrayList<ArrayList<Integer>> result = new
            ArrayList<ArrayList<Integer>>();
        if(nums == null || nums.size() == 0)
                return result;
        ArrayList<Integer> list = new ArrayList<Integer>();
        int[] visited = new int[nums.size()];
        Collections.sort(nums);
        helper(result, list, visited, nums);
        return result;
}
```

2.33 Permutation Sequence

Given n and k, return the number of k permutation sequence. We should simulation the process on how construct permutation.

```
public String getPermutation(int n, int k) {
        if (n <= 0 && k <= 0) {
                return "";
        }
        int fact = 1;
        List<Integer> nums = new ArrayList<Integer>();
        for (int i = 1; i <= n; i++) {
                fact *= i;
                nums.add(i);
        }
        StringBuilder sb = new StringBuilder();
        for (int i = n; i >= 1; i--) {
                fact /= i;
                int rank = (k - 1) / fact;
                k = (k - 1) % fact + 1;
                sb.append(nums.get(rank));
                nums.remove(rank);
        }
        return sb.toString();
}
```

2.34 Convert Sorted List to Binary Search Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

```
private ListNode current;
private int getListLength(ListNode head) {
        int size = 0;
        while (head != null) {
                size++;
                head = head.next;
        }
        return size;
}
public TreeNode sortedListToBST(ListNode head) {
        int size;
        current = head;
        size = getListLength(head);
        return sortedListToBSTHelper(size);
}

public TreeNode sortedListToBSTHelper(int size) {
        if (size <= 0) {
                return null;
        }
        TreeNode left = sortedListToBSTHelper(size / 2);
        TreeNode root = new TreeNode(current.val);
        current = current.next;
        TreeNode right = sortedListToBSTHelper(size - 1 - size / 2);
        root.left = left;
        root.right = right;
        return root;
}
```

2.35 Reorder List

In this problem, there are three templates on linked list.

```
ListNode slow = head, fast = head.next;
while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        }
ListNode rHead = slow.next;
slow.next = null;
```

```
ListNode prev = null;
while (rHead != null) {
        ListNode temp = rHead.next;
        rHead.next = prev;
        prev = rHead;
        rHead = temp;
}
```

```
rHead = prev;
ListNode lHead = head;
while (lHead != null && rHead != null) {
        ListNode temp1 = lHead.next;
        lHead.next = rHead;
        rHead = rHead.next;
        lHead.next.next = temp1;
        lHead = temp1;
}
```

2.36 Add Two Numbers

You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1ś digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list. Then, we use a variable carry to compute the sum results from less digit values to high digit values.

```
public ListNode addLists(ListNode l1, ListNode l2) {
        // write your code here
        int carry = 0;
        ListNode newHead = new ListNode(0);
        ListNode p1 = l1, p2 = l2, p3 = newHead;
        while (p1 != null || p2 != null) {
                if (p1 != null) {
                        carry += p1.val;
                        p1 = p1.next;
                }
                if (p2 != null) {
                        carry += p2.val;
```

```
                p2 = p2.next;
            }
            p3.next = new ListNode(carry % 10);
            p3 = p3.next;
            carry /= 10;
        }
        if (carry == 1) {
            p3.next = new ListNode(1);
        }
        return newHead.next;
}
```

2.37 Insert Node in a Binary Search Tree

```
public TreeNode insertNode(TreeNode root, TreeNode node) {
        // write your code here
        if (root == null) {
            return node;
        }
        if (root.val > node.val) {
            root.left = insertNode(root.left, node);
        } else {
            root.right = insertNode(root.right, node);
        }
        return root;
}
```

2.38 Validate Binary Search Tree

We use recursion based on definition of binary search tree, which left subtree of contains nodes that less than node key, and right subtree only have nodes which greater than node value.

```
public boolean isValidBST(TreeNode root) {
        // write your code here
        if (root == null) {
            return true;
        }
        return helper(root, Long.MIN_VALUE, Long.MAX_VALUE);
}
public boolean helper(TreeNode root, long lower, long upper) {
        if (root == null) {
            return true;
        }
        if (root.val >= upper || root.val <= lower) {
            return false;
        }
        boolean isLeftValidBST = helper(root.left, lower, root.val);
        boolean isRightValidBST = helper(root.right, root.val, upper);
        return isLeftValidBST && isRightValidBST;
}
```

2.39 Balanced Binary Tree

We set when facing error, we return -1 to upper layer. And when the upper layer get -1, and we will know that there is a error.

```
public boolean isBalanced(TreeNode root) {
        // write your code here
        if (root == null) {
                return true;
        }
        return helper(root) != -1;
}
private int helper(TreeNode root) {
        if (root == null) {
                return 0;
        }
        int leftDepth = helper(root.left);
        int rightDepth = helper(root.right);
        if (leftDepth == -1 || rightDepth == -1 ||
                Math.abs(leftDepth - rightDepth) > 1) {
                return -1;
        }
        return 1 + Math.max(leftDepth, rightDepth);
}
```

2.40 Remove Node in Binary Search Tree

There are four kinds of situations that delete this node.

- root node is the target node: find the right most node if its left child and replace the root node
- target node has no child leaf: set its parent left/right child to null
- target node has only one child: replace target node with that child
- target node has both children: find the right most leaf of its left children and replace the target node; remember to set its parent child to null and replace children of this node to the target node children (another corner case: check if its child is itself)

```
public TreeNode removeNode(TreeNode root, int value) {
        if (root == null)
                return null;
        if (value < root.val)
                root.left = removeNode(root.left, value);
        else if (value > root.val)
                root.right = removeNode(root.right, value);
        else {
                if (root.left == null)
                        return root.right;
                if (root.right == null)
                        return root.left;
                TreeNode x = root;
                root = findMin(root.right);
                root.right = deleteMin(x.right);
                root.left = x.left;
```

```
        }
        return root;
}

public TreeNode removeNode(TreeNode root, int value) {
        // write your code here
        TreeNode dummy = new TreeNode(0);
        dummy.left = root;
        TreeNode parent = findNode(dummy, root, value);
        TreeNode node;
        if (parent.left != null && parent.left.val == value) {
                node = parent.left;
        } else if (parent.right != null && parent.right.val == value) {
                node = parent.right;
        } else {
                return dummy.left;
        }
        deleteNode(parent, node);
        return dummy.left;
}
private TreeNode findNode(TreeNode parent, TreeNode node, int value) {
        if (node == null) {
                return parent;
        }
        if (node.val == value) {
                return parent;
        }
        if (value < node.val) {
                return findNode(node, node.left, value);
        } else {
                return findNode(node, node.right, value);
        }
}
private void deleteNode(TreeNode parent, TreeNode node) {
        if (node.right == null) {
                if (parent.left == node) {
                        parent.left = node.left;
                } else {
                        parent.right = node.left;
                }
        } else {
                TreeNode temp = node.right;
                TreeNode father = node;
                while (temp.left != null) {
                        father = temp;
                        temp = temp.left;
                }
                if (father.left == temp) {
                        father.left = temp.right;
                } else {
                        father.right = temp.right;
                }
                if (parent.left == node) {
                        parent.left = temp;
```

```
                } else {
                        parent.right = temp;
                }
                temp.left = node.left;
                temp.right = node.right;
        }
}
```

2.41 Sub-Set I & II: Template

Given a set of distinct integers, nums, return all possible subsets. We perform DFS method to solve it, and construct a template.

```
private void dfs(int[] nums, int pos, List<Integer> list,
        List<List<Integer>> ret) {
        // add temp result first
        ret.add(new ArrayList<Integer>(list));
        for (int i = pos; i < nums.length; i++) {
                list.add(nums[i]);
                dfs(nums, i + 1, list, ret);
                list.remove(list.size() - 1);
        }
}
public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new
            ArrayList<List<Integer>>();
        if (nums == null || nums.length == 0) {
                return result;
        }
        Arrays.sort(nums);
        dfs(nums, 0, list, result);
        return result;
}
```

When we want to remove duplicate elements, we can add one line to filter.

```
public ArrayList<ArrayList<Integer>> subsetsWithDup(ArrayList<Integer> S) {
        // write your code here
        ArrayList<ArrayList<Integer>> result = new
            ArrayList<ArrayList<Integer>>();
        ArrayList<Integer> list = new ArrayList<Integer>();
        if (S == null || S.size() == 0) {
                return result;
        }
        Collections.sort(S);
        dfs(result, list, S, 0);
        return result;
}
private void dfs(ArrayList<ArrayList<Integer>> result, ArrayList<Integer>
    list, ArrayList<Integer> S, int pos) {
        result.add(new ArrayList<Integer>(list));
        for (int i = pos; i < S.size(); i++) {
                if (i != pos && S.get(i) == S.get(i - 1)) {
                        continue;
                }
```

```
                list.add(S.get(i));
                dfs(result, list, S, i + 1);
                list.remove(list.size() - 1);
        }
}
```

2.42 Permutations: Template

Given a list of numbers, return all possible permutations, in this place, we use the sub-set template.

```
private void dfs(int[] nums, List<Integer> list, List<List<Integer>> result) {
        if (list.size() == nums.length) {
                result.add(new ArrayList<Integer>(list));
                return;
        }
        for (int i = 0; i < nums.length; i++) {
                if (list.contains(nums[i])) continue;
                list.add(nums[i]);
                dfs(nums, list, result);
                list.remove(list.size() - 1);
        }
}
public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        if (nums == null || nums.length == 0) return result;
        List<Integer> list = new ArrayList<Integer>();
        dfs(nums, list, result);
        return result;
}
```

2.43 Combinations: Template

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.

```
public List<List<Integer>> combine(int n, int k) {
        // write your code her
        assert(n >= 1 && n >= k && k >= 1);
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        List<Integer> list = new ArrayList<Integer>();
    dfs(n, k, 1, list, result);
    return result;
}
private void dfs(int n, int k, int pos, List<Integer> list,
   List<List<Integer>> result) {
        if (list.size() == k) {
                result.add(new ArrayList<Integer>(list));
                return;
        }
        for (int i = pos; i <= n; i++) {
                list.add(i);
                dfs(n, k, i + 1, list, result);
                list.remove(list.size() - 1);
        }
```

```
}
```

Moreover, we can use this template to solve the combination sum problem.

```
private void helper(int[] candidates, int pos, int gap, List<Integer> list,
    List<List<Integer>> result) {
        if (gap == 0) {
                result.add(new ArrayList<Integer>(list));
                return;
        }
        for (int i = pos; i < candidates.length; i++) {
                if (gap < candidates[i]) {
                        return;
                }
                list.add(candidates[i]);
                helper(candidates, i, gap - candidates[i], list, result);
                list.remove(list.size() - 1);
        }
}
public List<List<Integer>> combinationSum(int[] candidates, int target) {
        // write your code here
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        List<Integer> list = new ArrayList<Integer>();
        if (candidates == null) {
                return result;
        }
        Arrays.sort(candidates);
        helper(candidates, 0, target, list, result);
        return result;
}
```

If we want to make sure that not get the same values in the result list, we should add one line to statement.

```
if (i != pos && candidates[i] == candidates[i - 1]) {
        continue;
}
helper(candidates, i + 1, gap - candidates[i], list, result);
```

2.44 Topological Sorting

In this method, we should use a hash map to save the indegree information of each node, and perform dfs to finish the task.

```
public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode>
    graph) {
        ArrayList<DirectedGraphNode> result = new
            ArrayList<DirectedGraphNode>();
        HashMap<DirectedGraphNode, Integer> map = new HashMap();
        for (DirectedGraphNode node : graph) {
                for (DirectedGraphNode neighbor : node.neighbors) {
                        if (map.containsKey(neighbor)) {
                                map.put(neighbor, map.get(neighbor) + 1);
                        } else {
                                map.put(neighbor, 1);
                        }
                }
```

```
        }
        Queue<DirectedGraphNode> q = new LinkedList<DirectedGraphNode>();
        for (DirectedGraphNode node : graph) {
                if (!map.containsKey(node)) {
                        q.offer(node);
                        result.add(node);
                }
        }
        while (!q.isEmpty()) {
                DirectedGraphNode node = q.poll();
                for (DirectedGraphNode n : node.neighbors) {
                        map.put(n, map.get(n) - 1);
                        if (map.get(n) == 0) {
                                result.add(n);
                                q.offer(n);
                        }
                }
        }
        return result;
}
```

2.45 Word Ladder I & II

Given two words (start and end), and a dictionary, find the length of shortest transformation sequence from start to end. We apply BFS to solve. It contains following steps:

- construct a queue, and a hash to save passed node word
- use a method named *getNextWords* to get the neighbour word list based on current word.
- it we passed this word, ignore it.

```
private Set<String> getNextWords(String curr, Set<String> dict) {
        Set<String> nextWords = new HashSet<String>();
        for (int i = 0; i < curr.length(); i++) {
                char[] chars = curr.toCharArray();
                for (char c = 'a'; c <= 'z'; c++) {
                        chars[i] = c;
                        String temp = new String(chars);
                        if (dict.contains(temp)) {
                                nextWords.add(temp);
                        }
                }
        }
        return nextWords;
}
public int ladderLength(String start, String end, Set<String> dict) {
        // write your code here
        if (start == null && end == null) {
                return 0;
        }
        if (start.length() == 0 && end.length() == 0) {
                return 0;
        }
```

```
        assert(start.length() == end.length());
        if (dict == null || dict.size() == 0) {
                return 0;
        }
        int ladderLen = 1;
        dict.add(end);
        Queue<String> q = new LinkedList<String>();
        Set<String> hash = new HashSet<String>();
        q.offer(start);
        hash.add(start);
        while (! q.isEmpty()) {
                ladderLen++;
                int qLen = q.size();
                for (int i = 0; i < qLen; i++) {
                        String strTemp = q.poll();
                        for (String nextWord : getNextWords(strTemp, dict)) {
                                if (nextWord.equals(end)) {
                                        return ladderLen;
                                }
                                if (hash.contains(nextWord)) {
                                        continue;
                                }
                                q.offer(nextWord);
                                hash.add(nextWord);
                        }
                }
        }
        return 0;
}
```

If we would like to get all results, we should perform this method to dfs to get all solutions.

```
// DFS: output all paths with the shortest distance.
private void dfs(String cur, String end, Set<String> dict, HashMap<String,
    ArrayList<String>> nodeNeighbors, HashMap<String, Integer> distance,
    ArrayList<String> solution, List<List<String>> res) {
                solution.add(cur);
        if (end.equals(cur)) {
                res.add(new ArrayList<String>(solution));
        } else {
                for (String next : nodeNeighbors.get(cur)) {
                        if (distance.get(next) == distance.get(cur) + 1) {
                                dfs(next, end, dict, nodeNeighbors, distance,
                                    solution, res);
                        }
                }
        }
        solution.remove(solution.size() - 1);
}
```

2.46 N-Queues I & II Problem

In this problem, it can solve the n queues problems in the chessboard.

```
ArrayList<ArrayList<String>> solveNQueens(int n) {
```

```
        // write your code here
        ArrayList<ArrayList<String>> rst = new ArrayList<ArrayList<String>>();
        if (n <= 0) {
                return rst;
        }
        search(n, new ArrayList<Integer>(), rst);
        return rst;
}
private ArrayList<String> createBoard(ArrayList<Integer> cols) {
        ArrayList<String> solution = new ArrayList<String>();
        for (int i = 0; i < cols.size(); i++) {
                StringBuffer sb = new StringBuffer();
                for (int j = 0; j < cols.size(); j++){
                        if (j == cols.get(i)) {
                                sb.append( "Q");
                        } else {
                                sb.append( ".");
                        }
                }
                solution.add(sb.toString());
        }
        return solution;
}
private boolean isValid(ArrayList<Integer> cols, int col) {
        int row = cols.size();
        for (int i = 0; i < row; i++) {
                if (cols.get(i) == col) {
                        return false;
                }
                if (i - cols.get(i) == row - col) {
                        return false;
                }
                if (i + cols.get(i) == row + col) {
                        return false;
                }
        }
        return true;
}
void search(int n, ArrayList<Integer> cols, ArrayList<ArrayList<String>> rst)
   {
        if (cols.size() == n) {
                rst.add(createBoard(cols));
                return;
        }
        for (int i = 0; i < n; i++) {
                if (!isValid(cols, i)) {
                        continue;
        }
        cols.add(i);
        search(n, cols, rst);
        cols.remove(cols.size() - 1);
        }
}
```

When we are asked to get the number of available solutions to the N-queens problems based no given number, we just construct a global variable, and use it to update and get solution number.

```
int res;
public int totalNQueens(int n) {
        res = 0;
        if(n <= 0)
                return res;
        int [] columnVal = new int[n];
        DFS_helper(n, 0, columnVal);
        return res;
}
public void DFS_helper(int nQueens, int row, int[] columnVal){
        if(row == nQueens){
                res += 1;
        } else {
                for(int i = 0; i < nQueens; i++){
                columnVal[row] = i; // (row,columnVal[row)==>(row,i)
                if(isValid(row,columnVal))
                        DFS_helper(nQueens, row+1, columnVal);
                }
        }
}
public boolean isValid(int row, int [] columnVal){
        for(int i = 0; i < row; i++){
                if(columnVal[row] == columnVal[i]
                        || Math.abs(columnVal[row] - columnVal[i]) == row - i)
                return false;
        }
        return true;
}
```

2.47 Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below. We traversal the triangle from bottom to top, and make the value which located at index i and j only decided by the sum of values in $i+1$, j and $i+1$, $j+1$.

```
public int minimumTotal(int[][] triangle) {
        // write your code here
        if (triangle == null || triangle.length == 0 || triangle[0].length ==
            0) {
                return 0;
        }
        for (int i = triangle.length - 2; i >= 0; i--) {
                for (int j = 0; j < triangle[i].length; j++) {
                        int value = Math.min(triangle[i + 1][j]  +
                            triangle[i][j], triangle[i + 1][j + 1] +
                            triangle[i][j]);
                        triangle[i][j] = value;
                }
        }
        return triangle[0][0];
}
```

2.48 Maximum Product Sub-array

The product result is special because its result can change from minimum to maximum only by multiply a negative number. Therefore, we update two list, which save minimum product and maximum product numbers. Once we meet a new positivity number, we compare two list themselves, but if we meet a negative number, we update the list based on different list.

```
public int maxProduct(int[] nums) {
        // write your code here
        int[] max = new int[nums.length];
        int[] min = new int[nums.length];
        min[0] = max[0] = nums[0];
        int result = nums[0];
        for (int i = 1; i < nums.length; i++) {
                min[i] = max[i] = nums[i];
                if (nums[i] > 0) {
                        max[i] = Math.max(max[i], max[i - 1] * nums[i]);
                        min[i] = Math.min(min[i], min[i - 1] * nums[i]);
                } else if (nums[i] < 0) {
                        max[i] = Math.max(max[i], min[i - 1] * nums[i]);
                        min[i] = Math.min(max[i], max[i - 1] * nums[i]);
                }
        result = Math.max(result, max[i]);
        }
        return result;
}
```

2.49 Distinct Subsequence

Given a string S and a string T, count the number of distinct subsequences of T in S. 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.

```
public int numDistinct(String S, String T) {
        if (S == null || T == null) return 0;
        if (S.length() < T.length()) return 0;
        if (T.length() == 0) return 1;
        int[][] f = new int[S.length() + 1][T.length() + 1];
        for (int i = 0; i < S.length(); i++) {
                f[i][0] = 1;
                for (int j = 0; j < T.length(); j++) {
                        if (S.charAt(i) == T.charAt(j)) {
                                f[i + 1][j + 1] = f[i][j + 1] + f[i][j];
                        } else {
                                f[i + 1][j + 1] = f[i][j + 1];
                        }
                }
        }
        return f[S.length()][T.length()];
}
```

2.50 Interleaving String

Given three strings, and determine whether third string is formed by the first two strings. We construct a hug matrix with boolean values, and determine the finial result. The key equation is that:

$$f[i1][i2] = (s1[i1-1] == s3[i1+i2-1] \&\& f[i1-1][i2]) || (s2[i2-1] == s3[i1+i2-1] \&\& f[i1][i2-1]) \quad (1)$$

```
public boolean isInterleave(String s1, String s2, String s3) {
        // write your code here
        int len1 = (s1 == null) ? 0 : s1.length();
        int len2 = (s2 == null) ? 0 : s2.length();
        int len3 = (s3 == null) ? 0 : s3.length();
        if (len3 != len1 + len2) return false;
        boolean [][] f = new boolean[1 + len1][1 + len2];
        f[0][0] = true;
        // s1[i1 - 1] == s3[i1 + i2 - 1] && f[i1 - 1][i2]
        for (int i = 1; i <= len1; i++) {
                f[i][0] = s1.charAt(i - 1) == s3.charAt(i - 1) && f[i - 1][0];
        }
        // s2[i2 - 1] == s3[i1 + i2 - 1] && f[i1][i2 - 1]
        for (int i = 1; i <= len2; i++) {
                f[0][i] = s2.charAt(i - 1) == s3.charAt(i - 1) && f[0][i - 1];
        }
        // i1 >= 1, i2 >= 1
        for (int i1 = 1; i1 <= len1; i1++) {
                for (int i2 = 1; i2 <= len2; i2++) {
                        boolean case1 = s1.charAt(i1 - 1) == s3.charAt(i1 +
                            i2 - 1) && f[i1 - 1][i2];
                        boolean case2 = s2.charAt(i2 - 1) == s3.charAt(i1 +
                            i2 - 1) && f[i1][i2 - 1];
                        f[i1][i2] = case1 || case2;
                }
        }
        return f[len1][len2];
}
```

2.51 Word Break

```
public boolean wordBreak(String s, Set<String> dict) {
        // write your code here
        if (s == null || s.length() == 0) return true;
        if (dict == null || dict.isEmpty()) return false;
        int max_word_len = 0;
        for (String word : dict) {
                max_word_len = Math.max(max_word_len, word.length());
        }
        boolean[] can_break = new boolean[s.length() + 1];
        can_break[0] = true;
        for (int i = 1; i <= s.length(); i++) {
                for (int j = i - 1; j >= 0; j--) {
                        // optimize for too long interval
                        if (i - j > max_word_len) break;
                                String word = s.substring(j, i);
```

```
                        if (can_break[j] && dict.contains(word)) {
                                can_break[i] = true;
                                break;
                        }
                }
        }
        return can_break[s.length()];
}
```

2.52 Backpack I & II: Template

Given n items with size A_j, an integer m denotes the size of a backpack. How full you can fill this backpack?

```
public int backPack(int m, int[] A) {
        // write your code here
        if (A == null || A.length == 0) return 0;
        final int M = m;
        final int N = A.length;
        int[][] bp = new int[N + 1][M + 1];
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < M + 1; j++) {
                        if (A[i] > j) {
                                bp[i + 1][j] = bp[i][j];
                        } else {
                                bp[i + 1][j] = Math.max(bp[i][j], bp[i][j -
                                    A[i]] + A[i]);
                        }
                }
        }
        return bp[N][M];
}
```

If we want to get the maximum values in the bag, we should only add the value from value list to the dp matrix.

```
bp[i + 1][j] = Math.max(bp[i][j], bp[i][j - A[i]] + V[i]);
```

2.53 Minimum Adjustment Cost

Given an integer array, adjust each integers so that the difference of every adjacent integers are not greater than a given number target. It is a kind of backpack problem, and we should emulate all kinds of possible situation to find global solution (because it assumes that all number in the array is positive and not greater than 100, we set second dimensional size is 100).

```
public int MinAdjustmentCost(ArrayList<Integer> A, int target) {
        // write your code here
        if (A == null || A.size() == 0) {
                return 0;
        }
        int[][] res = new int[A.size() + 1][100];
        for (int j = 0; j <= 99; j++) {
                res[0][j] = 0;
        }
        for (int i = 1; i <= A.size(); i++) {
                for (int j = 0; j <= 99; j++) {
```

```
                        res[i][j] = Integer.MAX_VALUE;
                        int lowerRange = Math.max(0, j - target);
                        int upperRange = Math.min(99, j + target);
                        for (int p = lowerRange; p <= upperRange; p++) {
                                res[i][j] = Math.min(res[i][j], res[i-1][p] +
                                    Math.abs(j - A.get(i - 1)));
                        }
                }
        }
        int result = Integer.MAX_VALUE;
        for (int j = 0; j <= 99; j++) {
        result = Math.min(result, res[A.size()][j]);
        }
        return result;
}
```

2.54 Longest Increasing Subsequence

Given a sequence of integers, find the longest increasing subsequence (LIS). In order to solve this problem by dynamic programming method, we should know that the dp list save the information that the number in the current index have the LIS value. In order to finish that, we should compare current number and its previous numbers. In order to speed up this process, we use a dp list to save the previous LIS values.

```
public int longestIncreasingSubsequence(int[] nums) {
        // write your code here
        if (nums == null || nums.length == 0) {
                return 0;
        }
        int []f = new int[nums.length];
        int max = 0;
        for (int i = 0; i < nums.length; i++) {
                f[i] = 1;
                for (int j = 0; j < i; j++) {
                        if (nums[j] <= nums[i]) {
                                f[i] = Math.max(f[i], f[j] + 1);
                        }
                }
                if (f[i] > max) {
                        max = f[i];
                }
        }
        return max;
}
```

2.55 Maximum Sub-array Difference

In this problem, the dp list used to save the current previous and post index min and max values, then use these max and min value list to get the maximum difference.

```
public int maxDiffSubArrays(ArrayList<Integer> nums) {
        // write your code
        int max = Integer.MIN_VALUE;
        if (nums.size() <2) {
```

```
                return 0;
        }
        int n = nums.size();
        int[] leftMax = new int[n], leftMin = new int[n];
        int localMax = 0; int localMin = 0;
        for (int i = 0; i < n; i++) {
                int num = nums.get(i);
                localMax = Math.max(num + localMax, num);
                localMin = Math.min(num + localMin, num);
                if (i == 0) {
                        leftMax[i] = localMax;
                        leftMin[i] = localMin;
                } else {
                        leftMax[i] = Math.max(localMax, leftMax[i-1]);
                        leftMin[i] = Math.min(localMin, leftMin[i-1]);
                }
        }
        localMax = 0; localMin = 0;
        int lastMax = 0, lastMin = 0;
        for (int i = n-1; i > 0; i--) {
                int num = nums.get(i);
                localMax = Math.max(num + localMax, num);
                localMin = Math.min(num + localMin, num);
                if (i == n-1) {
                        lastMax = localMax;
                        lastMin = localMin;
                } else {
                        lastMax = Math.max(localMax, lastMax);
                        lastMin = Math.min(localMin, lastMin);
                }
                max = Math.max(Math.max(Math.abs(leftMax[i-1]-lastMin),
                Math.abs(lastMax - leftMin[i-1])), max);
        }
        return max;
}
```

2.56 Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence. We apply the hash set, and save all number information into it. After that, once we face a number, we delete it from set, then we search for the number less than it by one, and greater than it by one. If we find these numbers, we also remove them, and return the longest number.

```
public int longestConsecutive(int[] num) {
        // write you code here
        if (num == null || num.length == 0) {
                return 0;
        }
        Set<Integer> set = new HashSet<Integer>();
        for (int n : num) {
                set.add(n);
        }
        int lcs = 0;
```

```
        for (int n : num) {
                int i = n, count = 1;
                set.remove(n);
                while (set.contains(--i)) {
                        count++;
                        set.remove(i);
                }
                i = n;
                while (set.contains(++i)) {
                        count++;
                        set.remove(i);
                }
                lcs = Math.max(lcs, count);
        }
        return lcs;
}
```

2.57 Merge k Sorted Lists

Based on this problem, if we want to speed up merge several times, we have to apply divide and merge. Then, for the divide method, we divide a the lists into left part and right, and then perform merge them.

```
public ListNode mergeKLists(List<ListNode> lists) {
        // write your code here
        if (lists == null || lists.size() == 0) {
                return null;
        }
        return divide(lists, 0, lists.size() - 1);
}
private ListNode divide(List<ListNode> lists, int start, int end) {
        if (start == end) {
                return lists.get(start);
        } else if (start + 1 == end) {
                return merge2Lists(lists.get(start), lists.get(end));
        }
        ListNode left = divide(lists, start, start + (end - start) / 2);
        ListNode right = divide(lists, start + (end - start) / 2 + 1, end);
        return merge2Lists(left, right);
}
private ListNode merge2Lists(ListNode left, ListNode right) {
        ListNode dummy = new ListNode(0);
        ListNode last = dummy;
        while (left != null && right != null) {
                if (left.val < right.val) {
                        last.next = left;
                        left = left.next;
                } else {
                        last.next = right;
                        right = right.next;
                }
                last = last.next;
        }
        last.next = (left != null) ? left : right;
```

```
        return dummy.next;
}
```

2.58 Word Search I & II

In this method, we search the word based on given dict. In order to finish this task, we maintain a visited matrix to perform dfs search. In order to achieve the goal, we should use recursion to test result.

```
public boolean exist(char[][] board, String word) {
        // write your code here
        if (board == null || board.length == 0 || board[0].length == 0)
            return false;
        if (word == null || word.length() == 0) return false;
        boolean[][] visited = new boolean[board.length][board[0].length];
        for (int i = 0; i < board.length; i++) {
                for (int j = 0; j < board[0].length; j++) {
                        if (dfs(board, word, visited, i, j, 0)) {
                                return true;
                        }
                }
        }
        return false;
}
public boolean dfs(char[][] board, String word, boolean[][] visited, int row,
    int col, int wi) {
        // out of index
        if (row < 0 || row > board.length - 1 ||
                col < 0 || col > board[0].length - 1) {
                return false;
        }
        if (!visited[row][col] && board[row][col] == word.charAt(wi)) {
                // return instantly
                if (wi == word.length() - 1) return true;
                // traverse unvisited row and col
                visited[row][col] = true;
                boolean down = dfs(board, word, visited, row + 1, col, wi +
                    1);
                boolean right = dfs(board, word, visited, row, col + 1, wi +
                    1);
                boolean up = dfs(board, word, visited, row - 1, col, wi + 1);
                boolean left = dfs(board, word, visited, row, col - 1, wi +
                    1);
                // reset with false if none of above is true
                visited[row][col] = up || down || left || right;
                return up || down || left || right;
        }
        return false;
}
```

For the words, it is easy to solve by perform a loop to the words to solve part II problem.

2.59 Largest Rectangle in Histogram

The main point is that we should know that when we face the new less area, we should compute the current maximum area of rectangle in histogram.

```
public int largestRectangleArea(int[] height) {
        // write your code here
        if (height == null || height.length == 0) {
                return 0;
        }
        Stack<Integer> stack = new Stack<Integer>();
        int max = 0;
        int i = 0;
        while (i < height.length) {
                if (stack.isEmpty() || height[i] >= height[stack.peek()]) {
                        stack.push(i);
                        i++;
                } else {
                        int p = stack.pop();
                        int h = height[p];
                        // consider of that we pop one, then we add 1 to the
                           width
                        int w = stack.isEmpty() ? i : i - (stack.peek() + 1);
                        max = Math.max(max, w * h);
                }
        }
        while (!stack.isEmpty()) {
                int p = stack.pop();
                int h = height[p];
                int w = stack.isEmpty() ? i : i - (stack.peek() + 1);
                max = Math.max(max, w * h);
        }
        return max;
}
```

3 LintCode Remains

3.1 Max Tree (Cartesian Tree)

Given an integer array with no duplicates. A max tree building on this array is defined as follow:

- The root is the maximum number in the array
- The left subtree and right subtree are the max trees of the subarray divided by the root number.

This is the Cartesian tree.

```
public TreeNode maxTree(int[] A) {
        // write your code here
        int len = A.length;
        TreeNode[] stk = new TreeNode[len];
        for (int i = 0; i < len; ++i)
                stk[i] = new TreeNode(0);
        int cnt = 0;
```

```
        for (int i = 0; i < len; ++i) {
                TreeNode tmp = new TreeNode(A[i]);
                while (cnt > 0 && A[i] > stk[cnt - 1].val) {
                        tmp.left = stk[cnt-1];
                        cnt --;
                }
                if (cnt > 0) {
                        stk[cnt - 1].right = tmp;
                }
                stk[cnt++] = tmp;
    }
    return stk[0];
}
```

3.2 Add and Search Word

In this problem, we construct a global map to save the relationship between character and instance TrieNode. In the TrieNode, we also assign a variable named *isLeaf* to define this character can be a end of a word or not. After we add new word, we will search this word by compare the dfsSearch method can go to the end, and the end character is one of end of previous words.

```
class TrieNode{
    char c;
    HashMap<Character, TrieNode> children = new HashMap<Character,
       TrieNode>();
    boolean isLeaf;

    public TrieNode() {}

    public TrieNode(char c){
        this.c = c;
    }
}

private TrieNode root;

public WordDictionary () {
        root = new TrieNode();
}

// Adds a word into the data structure.
public void addWord(String word) {
        HashMap<Character, TrieNode> children = root.children;
        for(int i = 0; i < word.length(); i++){
                char c = word.charAt(i);
                TrieNode t = null;
                if (children.containsKey(c)) {
                        t = children.get(c);
                } else {
                        t = new TrieNode(c);
                        children.put(c,t);
                }
                children = t.children;
```

```
                if (i == word.length() - 1) {
                        t.isLeaf = true;
                }
        }
}

public boolean search(String word) {
        return dfsSearch(root.children, word, 0);
}
public boolean dfsSearch(HashMap<Character, TrieNode> children, String word,
    int start) {
        if(start == word.length()){
                if(children.size() == 0)
                        return true;
                else
                        return false;
    }
        char c = word.charAt(start);
        if (children.containsKey(c)) {
                if(start == word.length() - 1 && children.get(c).isLeaf){
                        return true;
                }
                return dfsSearch(children.get(c).children, word, start + 1);
        } else if (c == '.') {
                boolean result = false;
                for(Map.Entry<Character, TrieNode> child :
                    children.entrySet()){
                        if(start == word.length() - 1 &&
                            child.getValue().isLeaf){
                                return true;
                        }
                        //if any path is true, set result to be true;
                        if (dfsSearch(child.getValue().children, word, start
                            + 1)) {
                                result = true;
                        }
                }
                return result;
        } else {
                return false;
        }
}
```

There is another similar problem is that find the prefix words in the data structure. In this time, there is no need to determine the end of a word by the isLeaf variable on search prefix.

3.3 Palindrome Partitioning I & II

In this problem, we should construct all kinds of sub-string in the given word, and make a decision it is a palindrome or not. If it is, then add it into result list. We can solve it by dfs method.

```
public ArrayList<ArrayList<String>> partition(String s) {
        ArrayList<ArrayList<String>> result = new
            ArrayList<ArrayList<String>>();
```

```
        if (s == null || s.length() == 0) {
                return result;
        }
        ArrayList<String> partition = new ArrayList<String>();
        addPalindrome(s, 0, partition, result);
        return result;
}
private void addPalindrome(String s, int start, ArrayList<String> partition,
                ArrayList<ArrayList<String>> result) {
        //stop condition
        if (start == s.length()) {
                ArrayList<String> temp = new ArrayList<String>(partition);
                result.add(temp);
                return;
        }
        for (int i = start + 1; i <= s.length(); i++) {
                String str = s.substring(start, i);
                if (isPalindrome(str)) {
                        partition.add(str);
                        addPalindrome(s, i, partition, result);
                        partition.remove(partition.size() - 1);
                }
        }
}
private boolean isPalindrome(String str) {
        int left = 0;
        int right = str.length() - 1;

        while (left < right) {
                if (str.charAt(left) != str.charAt(right)) {
                        return false;
                }

                left++;
                right--;
        }
        return true;
}
```

We we face the cutting questions, we set the maximum of cut value is the size of word (if we divided this word into each character, it must be palindrome).

```
public int minCut(String s) {
        // write your code here
        int n = s.length();
        boolean dp[][] = new boolean[n][n];
        int cut[] = new int[n];
        for (int j = 0; j < n; j++) {
                cut[j] = j; //set maximum # of cut
                for (int i = 0; i <= j; i++) {
                        if (s.charAt(i) == s.charAt(j) && (j - i <= 1 || dp[i
                            + 1][j - 1])) {
                                dp[i][j] = true;
                                // if need to cut, add 1 to the previous
                                    cut[i - 1]
```

```
                                if (i > 0){
                                        cut[j] = Math.min(cut[j], cut[i - 1]
                                            + 1);
                                }else{
                                // if [0...j] is palindrome, no need to cut
                                        cut[j] = 0;
                                }
                        }
                }
        }
        return cut[n - 1];
}
```

3.4 Palindrome Linked List

We first apply quick and slow pointer to find the middle of a list, and reverse the second list. Then we compare with each element values one by one until face error.

```
public boolean isPalindrome(ListNode head) {
        // Write your code here
        if (head == null || head.next == null) {
                return true;
        }
        ListNode fast = head;
        ListNode slow = head;
        while (fast.next != null && fast.next.next != null) {
                fast = fast.next.next;
                slow = slow.next;
        }
        ListNode secondHead = slow.next;
        slow.next = null;
        ListNode p1 = secondHead;
        ListNode p2 = p1.next;
        while (p1 != null && p2 != null) {
                ListNode temp = p2.next;
                p2.next = p1;
                p1 = p2;
                p2 = temp;
        }
        secondHead.next = null;
        ListNode p = (p2 == null ? p1 : p2);
        ListNode q = head;
        while (p != null) {
                if (p.val != q.val) {
                        return false;
                }
                p = p.next;
                q = q.next;
        }
        return true;
}
```

3.5 Reverse Nodes in k-Group

First of all, divided list to several k-groups, and reverse each group.

```
public ListNode reverseKGroup(ListNode head, int k) {
        // Write your code here
        if (head ==null || k == 1){
                return head;
        }
        ListNode dummy =new ListNode (0);
        dummy.next = head;
        ListNode pre=dummy;
        int i = 0;
        while (head != null){
                i++;
                if (i % k == 0) {
                        pre = reverse(pre, head.next);
                        head = pre.next;
                } else {
                        head = head.next;
                }
        }
        return dummy.next;
}
public ListNode reverse(ListNode pre, ListNode next){
        ListNode last = pre.next;
        ListNode cur = last.next;
        while (cur != next){
                last.next = cur.next;
                cur.next = pre.next;
                pre.next = cur;
                cur = last.next;
        }
        return last;
}
```

3.6 Max Points on a Line

Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.

```
public int maxPoints(Point[] points) {
        if(points == null || points.length == 0) return 0;
        HashMap<Double, Integer> result = new HashMap<Double, Integer>();
        int max=0;
        for (int i = 0; i < points.length; i++) {
                int duplicate = 1;
                int vertical = 0;
                for (int j=i+1; j<points.length; j++) {
                        //handle duplicates and vertical
                        if (points[i].x == points[j].x) {
                                if (points[i].y == points[j].y) {
                                        duplicate++;
                                } else {
                                        vertical++;
```

```
                        }
                    } else {
                        double slope = points[j].y == points[i].y ?
                            0.0 : (1.0 * (points[j].y - points[i].y))
                        / (points[j].x - points[i].x);
                        if (result.get(slope) != null) {
                            result.put(slope, result.get(slope) +
                                1);
                        } else {
                            result.put(slope, 1);
                        }
                    }
                }
                for (Integer count: result.values()) {
                    if(count+duplicate > max){
                        max = count + duplicate;
                    }
                }
                max = Math.max(vertical + duplicate, max);
                result.clear();
            }
            return max;
}
```

3.7 Number of Islands

This problem requires us to search to next locations of each pixel, and find the final solution.

```
static int[] dx = {1, 0, 0, -1};
static int[] dy = {0, 1, -1, 0};
private int n, m;

private void removeIsland(boolean[][] grid, int x, int y) {
        grid[x][y] = false;
        for (int i = 0; i < 4; i++) {
            int nextX = x + dx[i];
            int nextY = y + dy[i];
            if (nextX >= 0 && nextX < n && nextY >= 0 && nextY < m) {
                if (grid[nextX][nextY]) {
                    removeIsland(grid, nextX, nextY);
                }
            }
        }
}
public int numIslands(boolean[][] grid) {
        n = grid.length;
        if (n == 0) {
            return 0;
        }
        m = grid[0].length;
        if (m == 0) {
            return 0;
        }
```

```
        int count = 0;
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                        if (grid[i][j]) {
                                removeIsland(grid, i, j);
                                count++;
                        }
                }
        }
        return count;
}
```

3.8 Maximal Square

Given a 2D binary matrix filled with 0 and 1, find the largest square containing all 1 and return its area. The main point is that we assign the dp matrix with the minimum of surrounding values.

```
public int maxSquare(int[][] matrix) {
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0)
            return 0;
        int n = matrix.length;
        int m = matrix[0].length;
        int[][] d = new int[n][m];
        int max = 0;
        for(int  i = 0; i < n; i++) {
                if(matrix[i][0] == '1') {
                        d[i][0] = 1;
                        max = 1;
                }
        }
        for(int j = 0; j < m; j++) {
                if(matrix[0][j] == '1') {
                        d[0][j] = 1;
                        max = 1;
                }
        }
        for(int i = 1; i < n; i++) {
                for(int j = 1; j < m; j++) {
                        if(matrix[i][j] == '0') d[i][j] = 0;
                        else {
                                d[i][j] = Math.min(Math.min(d[i - 1][j],
                                    d[i][j - 1]),
                                d[i - 1][j - 1] ) + 1;
                                max = Math.max(max, d[i][j]);
                        }
                }
        }
        return max * max;
}
```

3.9 Find the Weak Connected Component in the Directed Graph

In this problem, we define a class named UnionFind, and use to to speed up process on find node.

```
class UnionFind{
        HashMap<Integer, Integer> father = new HashMap<Integer, Integer>();
        UnionFind(HashSet<Integer> hashSet){
                for(Integer now : hashSet) {
                        father.put(now, now);
                }
        }
        int find(int x){
        int parent = father.get(x);
        while(parent != father.get(parent)) {
                parent = father.get(parent);
        }
        return parent;
        }
        int compressed_find(int x){
        int parent = father.get(x);
        while(parent != father.get(parent)) {
                parent = father.get(parent);
        }
        int temp = -1;
        int fa = father.get(x);
        while(fa != father.get(fa)) {
                temp = father.get(fa);
                father.put(fa, parent) ;
                fa = temp;
        }
        return parent;
        }

        void union(int x, int y){
        int fa_x = find(x);
        int fa_y = find(y);
        if(fa_x != fa_y)
                father.put(fa_x, fa_y);
        }
}

List<List<Integer>> print(HashSet<Integer> hashSet, UnionFind uf, int n) {
        List<List <Integer>> ans = new ArrayList<List<Integer>>();
        HashMap<Integer, List<Integer>> hashMap
                = new HashMap<Integer, List<Integer>>();
        for(int i : hashSet){
                int fa = uf.find(i);
                if(!hashMap.containsKey(fa)) {
                        hashMap.put(fa, new ArrayList<Integer>() );
                }
                List<Integer> now = hashMap.get(fa);
                now.add(i);
                hashMap.put(fa, now);
        }
        for(List<Integer> now : hashMap.values()) {
```

```
                Collections.sort(now);
        ans.add(now);
        }
        return ans;
}

public List<List<Integer>> connectedSet2(ArrayList<DirectedGraphNode> nodes){
        // Write your code here
        HashSet<Integer> hashSet = new HashSet<Integer>();
        for(DirectedGraphNode now : nodes){
                hashSet.add(now.label);
                for(DirectedGraphNode neighbour : now.neighbors) {
                        hashSet.add(neighbour.label);
                }
        }
        UnionFind uf = new UnionFind(hashSet);
        for(DirectedGraphNode now : nodes){
                for(DirectedGraphNode neighbour : now.neighbors) {
                        int fnow = uf.find(now.label);
                        int fneighbour = uf.find(neighbour.label);
                        if(fnow != fneighbour) {
                                uf.union(now.label, neighbour.label);
                        }
                }
        }
        return print(hashSet, uf, nodes.size());
}
```

3.10 Copy Books

In this problem, we apply binary search to find the accurate solution.

```
private boolean search(int mid, int[] pages, int k) {
        int people = 0;
        int sum = 0;
        for (int i = 0; i < pages.length;) {
                if (sum + pages[i] <= mid) {
                        sum += pages[i];
                        i++;
                } else if (pages[i] <= mid) {
                        sum = 0;
                        people++;
                } else {
                        return false;
                }
        }
        if (sum != 0) {
                people++; // also need one to finish
        }
        return people <= k;
}

public int copyBooks(int[] pages, int k) {
```

```
        // write your code here
        if (pages == null || pages.length == 0) {
                return 0;
        }
        int sum = 0;
        for (Integer page : pages) {
                sum += page;
        }
        int min = sum / k;
        while (min <= sum) {
                int time = 0;
                int mid = min + (sum - min) / 2;
                if (search(mid, pages, k)) {
                        sum = mid - 1;
                } else {
                        min = mid + 1;
                }
        }
        return min;
}
```

3.11 Post Office Problem

This problem apply a special definition in the dynamic programming problem. http://blog.csdn.net/find_my_dream/article/details/4931222

```
int[][] init(int []A) {
        int n = A.length;
        int [][]dis = new int [n + 1][n + 1];
        for(int i = 1; i <= n; i++) {
                for(int j = i+1 ;j <= n;++j) {
                        int mid = (i+j) /2;
                        for(int k = i;k <= j;++k) {
                                dis[i][j] += Math.abs(A[k - 1] - A[mid - 1]);
                        }
        }
        }
        return dis;
}

public int postOffice(int[] A, int k) {
        // Write your code here
        int n = A.length;
        Arrays.sort(A);
        int [][]dis = init(A);
        int [][]dp = new int[n + 1][k + 1];
        if(n == 0 || k >= A.length)
                return 0;
        int ans = Integer.MAX_VALUE;
        for(int i = 0;i <= n;++i)  {
                dp[i][1] = dis[1][i];
        }
        for(int nk = 2; nk <= k; nk++) {
```

```
                for(int i = nk; i <= n; i++) {
                        dp[i][nk] = Integer.MAX_VALUE;
                        for(int j = 0; j < i; j++) {
                                if(dp[i][nk] == Integer.MAX_VALUE ||
                                   dp[i][nk] > dp[j][nk - 1] + dis[j + 1][i])
                                   {
                                        dp[i][nk] = dp[j][nk - 1] + dis[j +
                                           1][i];
                                }
                        }
                }
        }
        return dp[n][k];
}
```

3.12 Reverse Integer

In this problem, we should consider of that the result is over range of given data structure.

```
public int reverseInteger(int n) {
        long result = 0;
        while (n != 0) {
                result = n % 10 + 10 * result;
                n /= 10;
        }
        if (result < Integer.MIN_VALUE || result > Integer.MAX_VALUE) {
                return 0;
        }
        return (int)result;
}
```

3.13 Scramble String

we apply the dynamic programming to solve this problem.

```
int len = s1.length();
        if (len != s2.length()) {
                return false;
        }
        boolean[][][] canScramble = new boolean[len][len][len+1];   // i,j
           with sublength
        for(int i=0; i<len; i++) {
                for(int j=0; j<len; j++) {
                        canScramble[i][j][1] = s1.charAt(i) == s2.charAt(j);
                              // substring start from i with length 1,
                           compared with substring start from j with length 1
                }
        }
        for(int sublen=2; sublen<=len; sublen++) {
                // end_pos = x+(sublen-1) <= len-1, so x <= len-sublen
                for(int i=0; i<=len-sublen; i++) {
                        for(int j=0; j<=len-sublen; j++) {
```

```
                                for(int p=1; p<sublen; p++) {   // split
                                    position
                                        canScramble[i][j][sublen] |=
                                            (canScramble[i][j][p] &&
                                            canScramble[i+p][j+p][sublen-p]) ||
(canScramble[i][j+sublen-p][p] && canScramble[i+p][j][sublen-p]);
                                }
                            }
                        }
                }
                return canScramble[0][0][len];
}
```

3.14 Sort Colors

In the three color flag problem, we use a index to loop the list, once we meet the zero, put it at the left, meet the two, put it at right, meet one, continue.

```
public void sortColors(int[] nums) {
        // write your code here
        if (nums == null || nums.length < 2) {
                return;
        }
        int left = 0;
        int right = nums.length - 1;
        int i = 0;
        while (i <= right) {
                if (nums[i] == 0) {
                        swap(nums, left, i);
                        left++;
                        i++;
                } else if (nums[i] == 1) {
                        i++;
                } else {
                        swap(nums, right, i);
                        right--;
                }
        }
}
```

3.15 Coins in a Line I & II & III

When we want to know who will take the last coin, we just make sure the reminder of the given coins number.

```
return n % 3 != 0;
```

When we want to get the maximum values of coins of each players, we perform dynamic programming. From any index of the array, we should know:

- the maximum value the first player will get if moving from it.
- whether it should cover the next index in the move.

Moreover, for any index i, we can know the gap between the two players if the first player moves from it. Suppose the first player moves from i and it wins f[i](the gap). If the second player moves from it, we should consider f[i] as a loss of the first player. We simulate the process on decision of the first player, it take the next two (once sum greater than max income of next third) or next one (next greater than max income of next second).

```
public boolean firstWillWin(int[] values) {
        // write your code here
        int size = values.length;
        if (size <= 2) {
                return true;
        }
        int[] f = new int[size]; // max income list
        for (int i = 0; i < size; i++) {
                f[i] = 0;
        }
        f[size - 1] = values[size - 1];
        f[size - 2] = values[size - 1] + values[size - 2];
        for (int i = size - 3; i >= 0; i--) {
                f[i] = Math.max(values[i] - f[i + 1], values[i] + values[i +
                    1] - f[i + 2]);
                // when it take one, give up max income at second index
        }
        return f[0] > 0;
}
```

When we can change the select coin from head or tail.

```
public boolean firstWillWin(int[] values) {
        int len = values.length;
        if (len <= 1) {
                return true;
        }
        int[][] store = new int[len][len];
        int[][] sum = new int[len][len];
        for (int i = 0; i < len; i++) {
                for (int j = i; j < len; j++) {
                        sum[i][j] = i == j ? values[j] : sum[i][j-1] +
                            values[j];
                }
        }
        for (int i = len - 1; i >= 0; i--) {
                for (int j = i; j < len; j++) {
                        if (i == j) {
                                store[i][j] = values[i];
                        } else {
                                int cur = Math.min(store[i+1][j],
                                    store[i][j-1]);
                                store[i][j] = sum[i][j] - cur;
                        }
                }
        }
        return store[0][len - 1] > sum[0][len-1] - store[0][len - 1];
}
```

3.16 Median

```
public int median(int[] nums) {
        if (nums == null) return -1;
        return helper(nums, 0, nums.length - 1, (nums.length + 1) / 2);
}
// l: lower, u: upper, m: median
private int helper(int[] nums, int l, int u, int size) {
        if (l >= u) return nums[u];
                int m = l;
        for (int i = l + 1; i <= u; i++) {
            if (nums[i] < nums[l]) {
                m++;
                int temp = nums[m];
                nums[m] = nums[i];
                nums[i] = temp;
            }
        }
        // swap between array[m] and array[l]
        // put pivot in the mid
        int temp = nums[m];
        nums[m] = nums[l];
        nums[l] = temp;

        if (m - l + 1 == size) {
            return nums[m];
        } else if (m - l + 1 > size) {
            return helper(nums, l, m - 1, size);
        } else {
            return helper(nums, m + 1, u, size - (m - l + 1));
        }
}
```

3.17 Segment Tree Build II

The special data structure, which is a binary tree and each nodes in the tree has two attributes start and end denote and segment or interval.

```
public SegmentTreeNode build(int[] A) {
        if (A == null || A.length == 0) {
                return null;
        }
        return helper(A, 0, A.length - 1);
}
private SegmentTreeNode helper(int[] A, int start, int end) {
        if (start > end) {
                return null;
        }
        SegmentTreeNode node = new SegmentTreeNode(start, end);
        if (start == end) {
                node.max = A[start];
                return node;
        } else {
                node.left = helper(A, start, (start + end) / 2);
                node.right = helper(A, (start + end) / 2 + 1, end);
```

```
                if (node.right != null) {
                        node.max = Math.max(node.left.max, node.right.max);
                } else {
                        node.max = node.left.max;
                }
        }
        return node;
}
```

3.18 Sub-matrix Sum

Given an integer matrix, find a sub-matrix where the sum of numbers is zero. Your code should return the coordinate of the left-up and right-down number. First of all, construct the sum matrix of this problem.

```
public int[][] submatrixSum(int[][] matrix) {
        // Write your code here
        int[][] result = new int[2][2];
        int M = matrix.length;
        if (M == 0) return result;
        int N = matrix[0].length;
        if (N == 0) return result;
        // pre-compute: sum[i][j] = sum of submatrix [(0, 0), (i, j)]
        int[][] sum = new int[M+1][N+1];
        for (int j=0; j<=N; ++j) sum[0][j] = 0;
        for (int i=1; i<=M; ++i) sum[i][0] = 0;
        for (int i=0; i<M; ++i) {
                for (int j=0; j<N; ++j)
                        sum[i+1][j+1] = matrix[i][j] + sum[i+1][j] +
                            sum[i][j+1] - sum[i][j];
        }
        for (int l=0; l<M; ++l) {
                for (int h=l+1; h<=M; ++h) {
                        Map<Integer, Integer> map = new HashMap<Integer,
                            Integer>();
                        for (int j=0; j<=N; ++j) {
                                int diff = sum[h][j] - sum[l][j];
                                if (map.containsKey(diff)) {
                                int k = map.get(diff);
                                result[0][0] = l;   result[0][1] = k;
                                result[1][0] = h-1; result[1][1] = j-1;
                                        return result;
                                } else {
                                        map.put(diff, j);
                                }
                        }
                }
        }
        return result;
}
```

3.19 Evaluate Reverse Polish Notation

In this problem, the main point is change the operators to a string, and use its index to make a decision to perform which kinds of operator action. We also use a stack to store all number information, and it similar to the simulation of calculator.

```
public int evalRPN(String[] tokens) {
        // Write your code here
        String operators = "+-*/";
        Stack<String> stack = new Stack<String>();
        for (String token : tokens) {
                if (! operators.contains(token)) {
                        stack.push(token); // it is a number
                } else {
                        int a = Integer.valueOf(stack.pop());
                        int b = Integer.valueOf(stack.pop());
                        int index = operators.indexOf(token);
                        switch(index) {
                                case 0:
                                        stack.push(String.valueOf(a + b));
                                        break;
                                case 1:
                                stack.push(String.valueOf(b - a));
                                        break;
                                case 2:
                                        stack.push(String.valueOf(a * b));
                                        break;
                                case 3:
                                        stack.push(String.valueOf(b / a));
                                        break;
                        }
                }
        }
        return Integer.valueOf(stack.pop());
}
```

3.20 Simplify Path

To understand the definition of absolute path for a Unix-style path.

```
public String simplifyPath(String path) {
        // Write your code here
        String result = "/";
        String[] stubs = path.split("/+");
        ArrayList<String> paths = new ArrayList<String>();
        for (String s : stubs){
                if(s.equals("..")){
                        if(paths.size() > 0){
                                paths.remove(paths.size() - 1);
                        }
                }
                else if (!s.equals(".") && !s.equals("")){
                        paths.add(s);
                }
```

```
        }
        for (String s : paths){
                result += s + "/";
        }
        if (result.length() > 1)
                result = result.substring(0, result.length() - 1);
        return result;
}
```

3.21 Count and Say

```
public String countAndSay(int n) {
        // Write your code here
        if (n <= 0) {
                return null;
        }
        String s = "1";
        for (int i = 1; i < n; i++) {
                int count = 1;
                StringBuilder sb = new StringBuilder();
                int sLen = s.length();
                for (int j = 0; j < sLen; j++) {
                        if (j < sLen - 1 && s.charAt(j) == s.charAt(j + 1)) {
                                count++;
                        } else {
                                sb.append(count + "" + s.charAt(j));
                                count = 1;
                        }
                }
                s = sb.toString();
        }
        return s;
}
```

3.22 Kth Smallest Number in Sorted Matrix

Find the kth smallest number in at row and column sorted matrix.

```
private void set(int row,int col,boolean [][] visited, int [][]matrix,
    PriorityQueue<Node> heap) {
        int m = visited.length;
        int n = visited[0].length;
        if (row < 0 || row >= m || col < 0 || col >= n || visited[row][col])
            return;
        heap.offer(new Node(row,col,matrix[row][col]));
        visited[row][col] = true;
}
public int kthSmallest(int[][] matrix, int k) {
        Comparator<Node> mycompare = new Comparator<Node>(){
                public int compare(Node n1,Node n2){
                        return n1.value - n2.value;
                }
        };
```

```
        boolean [][] visited = new boolean[matrix.length][matrix[0].length];
        PriorityQueue<Node> heap = new PriorityQueue<Node>(k, mycompare);
        heap.offer(new Node(0, 0, matrix[0][0]));
        int selected = 0;
        visited[0][0] = true;
        for(int i = 0;i < k; i++){
                Node node = heap.poll();
                selected = node.value;
                int row = node.row;
                int col = node.col;
                set(row + 1, col, visited, matrix, heap); // down
                set(row, col + 1, visited, matrix, heap); // right
        }
        return selected;
}
```

3.23 Maximum Gap: Bucket Sort Template

This is the template on Bucket Sort method.

```
public int maximumGap(int[] nums) {
        if (nums.length < 2) {
                return 0;
        }
        int max = nums[0], min = nums[0];
        for (int i = 1; i < nums.length; i++) {
                max = Math.max(max, nums[i]);
                min = Math.min(min, nums[i]);
        }
        int bucketLen = (max - min) / (nums.length - 1) + 1, bucketNo = (max
           - min) / bucketLen, result = 0;
        Map<Integer, List<Integer>> buckets = new HashMap<Integer,
           List<Integer>>();
        for (int i = 0; i < nums.length; i++) {
                int bucketNumber = (nums[i] - min) / bucketLen;
                if (buckets.containsKey(bucketNumber)) {
                        List<Integer> list = buckets.get(bucketNumber);
                        list.set(0, Math.min(list.get(0), nums[i]));
                        list.set(1, Math.max(list.get(1), nums[i]));
                } else {
                        List<Integer> list = new ArrayList<Integer>();
                        list.add(nums[i]);
                        list.add(nums[i]);
                        buckets.put(bucketNumber, list);
                }
        }
        int prev = min;
        for (int i = 0; i < bucketNo; i++) {
                if (buckets.containsKey(i)) {
                        result = Math.max(result, buckets.get(i).get(0) -
                           prev);
                        prev = buckets.get(i).get(1);
                }
```

```
        }
        result = Math.max(result, max - prev);
        return result;
}
```

3.24 Sub-array Sum II

Given an integer array, find a subarray where the sum of numbers is between two given interval. Your code should return the number of possible answer. (The element in the array should be positive).

```
public int subarraySumII(int[] A, int start, int end) {
        // Write your code here
        int len = A.length;
        for (int i = 1; i < len; ++i) {
                A[i] += A[i - 1];
        }
        Arrays.sort(A);
        int cnt = 0;
        for (int i = 0; i < len; ++i) {
                if (A[i] >= start && A[i] <= end) {
                        cnt++;
                }
                int l = A[i] - end;
                int r = A[i] - start;
                cnt += find(A, len, r + 1) - find(A, len, l);
        }
        return cnt;
}
int find(int[] A, int len, int value) {
        if (A[len - 1] < value) {
                return len;
        }
        int l = 0, r = len - 1, ans = 0;
        while (l <= r) {
                int mid = (l + r) / 2;
                if (value <= A[mid]) {
                        ans = mid;
                        r = mid - 1;
                } else {
                        l = mid + 1;
                }
    }
        return ans;
}
```

3.25 Roman to Integer & Integer to Roman

When we want to convert Roman to integer, we could use the definition that when right number is less than current one, we plus the current number, else, we minus current number.

```
public int romanToInt(String s) {
        // Write your code here
        if (s == null || s.length() == 0) {
```

```
                return 0;
        }
        Map<Character, Integer> map = new HashMap<Character, Integer>();
        map.put('I', 1);
        map.put('V', 5);
        map.put('X', 10);
        map.put('L', 50);
        map.put('C', 100);
        map.put('D', 500);
        map.put('M', 1000);
        int length = s.length();
        int result = map.get(s.charAt(length - 1));
        for (int i = length - 2; i >= 0; i--) {
                if (map.get(s.charAt(i + 1)) <= map.get(s.charAt(i))) {
                        result += map.get(s.charAt(i));
                } else {
                        result -= map.get(s.charAt(i));
                }
        }
        return result;
}
```

When we want to go back, it can perform as its definition. In order to simply the process, w also define the number 90 direct to *CD*.

```
public String intToRoman(int n) {
        // Write your code here
        String str = "";
        String [] symbol =
           {"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
        int [] value = {1000,900,500,400,100,90,50,40,10,9,5,4,1};
        for(int i = 0; n != 0; i++){
                while(n >= value[i]){
                        n -= value[i];
                        str += symbol[i];
                }
        }
        return str;
}
```

3.26 Valid Number

When we meet a string, when should skip the empty spaces between and after string. Then, we skip the signal of string (if it has). Then, once we meet a character is digit, we set number flag is true. Once we meet a dot, we set dot flag is true. Once we meet a e, if current is not a number or not a exp, it return false, or we set exp flag is true, and number flag is false. If we meet a plus or minus operation, if the character before it not the e, it also false.

```
public boolean isNumber(String s) {
        // Write your code here
        int len = s.length();
        int i = 0, e = len - 1;
        // skip empty space between and after content
        while (i <= e && Character.isWhitespace(s.charAt(i))) {
                i++;
```

```
        }
        while (e >= i && Character.isWhitespace(s.charAt(e))) {
                e--;
        }
        if (i <= e && (s.charAt(i) == '+' || s.charAt(i) == '-')) {
                i++;
        }
        boolean num = false;
        boolean dot = false;
        boolean exp = false;
        while (i <= e) {
                char c = s.charAt(i);
                if (Character.isDigit(c)) {
                        num = true;
                } else if (c == '.') {
                        dot = true;
                } else if (c == 'e') {
                        if (exp || num == false) {
                                return false;
                        }
                        exp = true;
                        num = false;
                } else if (c == '+' || c == '-') {
                        if (s.charAt(i - 1) != 'e') {
                                return false;
                        }
                } else {
                        return false;
                }
                i++;
        }
        return num;
}
```

3.27 Minimum Size Sub-array Sum

Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the sum greater or equal to s. Then, we can use two pointers to perform add or minus number to sum. Once the current sum is smaller than target number, we add one from new number at right. When current sum is greater than target, we minus one number from old number at left. Moreover, our task is to find the minimum size of subarray which sum is equal to target number, we should also save the minimum length by use right to minus left.

```
public int minimumSize(int[] nums, int s) {
        // write your code here
        if (nums.length ==  0) {
                return -1;
        }
        int left = 0, right = 0, sum = 0, len = nums.length, res = len + 1;
        while (right < len) {
                while (sum < s && right < len) {
                        sum += nums[right++];
                }
```

```
                while (sum >= s) {
                        res = Math.min(res, right - left);
                        sum -= nums[left++];
                }
        }
        return res == len + 1 ? -1 : res;
}
```

3.28 Longest Increasing Continuous Subsequence II

```
public int longestIncreasingContinuousSubsequenceII(int[][] A) {
        int res = 0;
        if (A == null || A.length == 0 || A[0].length == 0) {
                return res;
        }
        int[][] store = new int[A.length][A[0].length];
        for (int i = 0; i < A.length; i++) {
                for (int j = 0; j < A[0].length; j++) {
                        if (store[i][j] == 0) {
                                res = Math.max(res, dfs(A, store, i, j));
                        }
                }
        }
        return res;
}
private int dfs(int[][] a, int[][] store, int i, int j) {
        if (store[i][j] != 0) {
                return store[i][j];
        }
        int left = 0, right = 0, up = 0, down = 0;
        if (j + 1 < a[0].length && a[i][j + 1] > a[i][j]) {
                right = dfs(a, store, i, j + 1);
        }
        if (j > 0 && a[i][j - 1] > a[i][j]) {
                left = dfs(a, store, i, j - 1);
        }
        if (i + 1 < a.length && a[i + 1][j] > a[i][j]) {
                down = dfs(a, store, i + 1, j);
        }
        if (i > 0 && a[i - 1][j] > a[i][j]) {
                up = dfs(a, store, i - 1, j);
        }
        store[i][j] = Math.max(Math.max(up, down), Math.max(left, right)) + 1;
        return store[i][j];
}
```

3.29 Longest Palindromic Sub-string

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

```
public String longestPalindrome(String s) {
        int n = s.length();
```

```
        String res = null;
        boolean[][] dp = new boolean[n][n];
        for (int i = n - 1; i >= 0; i--) {
                for (int j = i; j < n; j++) {
                        dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i < 3
                            || dp[i + 1][j - 1]);
                        if (dp[i][j] && (res == null || j - i + 1 >
                            res.length())) {
                                res = s.substring(i, j + 1);
                        }
                }
        }
        return res;
}
```

We should also have a try on reverse the String, and perform KMP algorithm to see the complexity.

3.30 Merge Intervals

First of all, we sort the intervals at their start. Then, we update the last interval, when we meet overlapping, we set the last end to maximum of current end or last end.

```
private class IntervalComparator implements Comparator<Interval> {
        public int compare(Interval a, Interval b) {
                return a.start - b.start;
        }
}
public List<Interval> merge(List<Interval> intervals) {
        if (intervals == null || intervals.size() <= 1) {
                return intervals;
        }
        Collections.sort(intervals, new IntervalComparator());
        ArrayList<Interval> result = new ArrayList<Interval>();
        Interval last = intervals.get(0);
        for (int i = 1; i < intervals.size(); i++) {
                Interval curt = intervals.get(i);
                if (curt.start <= last.end) {
                        last.end = Math.max(last.end, curt.end);
                } else {
                        result.add(last);
                        last = curt;
                }
        }
        result.add(last);
        return result;
}
```

3.31 House Robber

When we meet a house, we have to decision on robber its previous two or three houses to achieve maximum values.

```
public long houseRobber(int[] A) {
        // write your code here
```

```
        int n = A.length;
        long[] res = new long[A.length];
        long ans = 0;
        if(n==0) {
                return 0;
        }
        if(n >= 1) {
                res[0] = A[0];
        }
        if(n >= 2) {
                res[1] = Math.max(A[0], A[1]);
        }
        if(n >= 3) {
                res[2] = Math.max(A[0] + A[2], A[1]);
        }
        if(n > 2) {
                for (int i = 3; i < n; i++) {
                        res[i] = Math.max(res[i - 3], res[i - 2]) + A[i];
                }
        }
        for (int i =0 ; i < n; i++) {
                ans = Math.max(ans, res[i]);
        }
        return ans;
}
```

3.32 Candy

There are N children standing in a line. Each child is assigned a rating value. You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

In this task, we use *arrays fill* to fill new list with all number 1.

```
public int candy(int[] ratings) {
        if(ratings == null || ratings.length == 0) {
                return 0;
        }
        int[] count = new int[ratings.length];
        Arrays.fill(count, 1);
        int sum = 0;
        for(int i = 1; i < ratings.length; i++) {
                if(ratings[i] > ratings[i - 1]) {
                        count[i] = count[i - 1] + 1;
                }
        }
        for (int i = ratings.length - 1; i >= 1; i--) {
                sum += count[i];
                if (ratings[i - 1] > ratings[i] && count[i - 1] <= count[i]) {
                        count[i - 1] = count[i] + 1;
                }
        }
```

```
        sum += count[0];
        return sum;
}
```

3.33 Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

```
public int maxPathSum(TreeNode root) {
        // write your code here
        int[] max = {Integer.MIN_VALUE};
        rec(root, max);
        return max[0];
}
public static int rec(TreeNode root, int[] max) {
        if (root == null) {
                return 0;
        }
        int leftSubtreeMaxSum = rec(root.left, max);
        int rightSubtreeMaxSum = rec(root.right, max);
        int arch = leftSubtreeMaxSum + root.val + rightSubtreeMaxSum;
        int maxPathAcrossRootToParent = Math.max(root.val,
           Math.max(leftSubtreeMaxSum, rightSubtreeMaxSum) + root.val);
        max[0] = Math.max(max[0], Math.max(arch, maxPathAcrossRootToParent));
        return maxPathAcrossRootToParent;
}
```

3.34 k Sum I & II

For the k Sum problem.

```
public int kSum(int A[], int k, int target) {
        if (A == null || A.length == 0) {
                return 0;
        }
        int[][][] f = new int[A.length + 1][k + 1][target + 1];
        //only the slots with k == 0 && t == 0 will be initialized as 1. All
           others need to be 0
        //when A[i] == t so we could derive f[i][k][t] from f[i - 1][k - 1][0]
        for (int i = 0; i < A.length + 1; i++) {
                f[i][0][0] = 1;
        }
        for (int i = 1; i < A.length + 1; i++) { //the first i elements in A[]
                for (int n = 1; n <= k && n <= i; n++) { //pick n items from
                   first i elements
                        for (int t = 1; t <= target; t++) { //sum from 1 to
                           target
                                f[i][n][t] = 0;
                                //case 1: the # of solutions with A[i - 1]
                                if (A[i - 1] <= t) {
                                //pick n - 1 items from first i - 1 elements
                                   with the sum = t - A[i - 1] plus this item
                                   A[i-1]
```

```
                                        f[i][n][t] = f[i - 1][n - 1][t - A[i
                                            - 1]];
                                }
                                //case 2: the # of solution w/o A[i - 1]
                                f[i][n][t] += f[i - 1][n][t]; //directly
                                    inherit from i - 1
                        }
                }
        }
        return f[A.length][k][target];
}
```

Once we want to get the actual solutions.

```
public ArrayList<ArrayList<Integer>> kSumII(int A[], int k, int target) {
        ArrayList<ArrayList<Integer>> result = new
            ArrayList<ArrayList<Integer>>();
        if (A == null || A.length == 0) {
                return result;
        }
        helper(A, k, target, result, 0, new ArrayList<Integer>());
        return result;
}
private void helper(int A[], int k, int target, ArrayList<ArrayList<Integer>>
    result, int curIndex, ArrayList<Integer> curList) {
        if (curList.size() == k) {
                if (target == 0) {
                        ArrayList<Integer> temp = new
                            ArrayList<Integer>(curList);
                        result.add(temp);
                }
                return;
        }
        for (int i = curIndex; i < A.length; i++) {
                curList.add(A[i]);
                helper(A, k, target - A[i], result, i + 1, curList);
                curList.remove(curList.size() - 1);
        }
}
```

3.35 Median of two Sorted Arrays

There are two sorted arrays A and B of size m and n respectively. Find the median of the two sorted arrays. We can solve this problem by find the kth number in a sorted array.

```
public double findMedianSortedArrays(int[] A, int[] B) {
        if ((A == null || A.length == 0) && (B == null || B.length == 0)) {
                return -1.0;
        }
        int lenA = (A == null) ? 0 : A.length;
        int lenB = (B == null) ? 0 : B.length;
        int len = lenA + lenB;
        // return median for even and odd cases
        if (len % 2 == 0) {
```

```
                return (findKth(A, 0, B, 0, len/2) + findKth(A, 0, B, 0,
                    len/2 + 1)) / 2.0;
        } else {
                return findKth(A, 0, B, 0, len/2 + 1);
        }
}
private int findKth(int[] A, int indexA, int[] B, int indexB, int k) {
        int lenA = (A == null) ? 0 : A.length;
        if (indexA > lenA - 1) {
                return B[indexB + k - 1];
        }
        int lenB = (B == null) ? 0 : B.length;
        if (indexB > lenB - 1) {
                return A[indexA + k - 1];
        }
        // avoid infilite loop if k == 1
        if (k == 1) return Math.min(A[indexA], B[indexB]);
        int keyA = Integer.MAX_VALUE, keyB = Integer.MAX_VALUE;
        if (indexA + k/2 - 1 < lenA) keyA = A[indexA + k/2 - 1];
        if (indexB + k/2 - 1 < lenB) keyB = B[indexB + k/2 - 1];
        if (keyA > keyB) {
                return findKth(A, indexA, B, indexB + k/2, k - k/2);
        } else {
                return findKth(A, indexA + k/2, B, indexB, k - k/2);
        }
}
```

3.36 Hash Function

In this task, we apply mod function to simply the process.

```
public int hashCode(char[] key,int HASH_SIZE) {
        // write your code here
        if (key == null || key.length == 0) {
                return -1;
        }
        long hashSum = 0;
        for (int i = 0; i < key.length; i++) {
                hashSum = 33 * hashSum + key[i];
                hashSum %= HASH_SIZE;
        }
        return (int)hashSum;
}
```

3.37 LRU Cache

Apply the memory algorithm.

```
private class Node{
        Node prev;
        Node next;
        int key;
        int value;
```

```
        public Node(int key, int value) {
                this.key = key;
                this.value = value;
                this.prev = null;
                this.next = null;
        }
}

private int capacity;
private HashMap<Integer, Node> hs = new HashMap<Integer, Node>();
private Node head = new Node(-1, -1);
private Node tail = new Node(-1, -1);

// @param capacity, an integer
public Solution(int capacity) {
        this.capacity = capacity;
        tail.prev = head;
        head.next = tail;
}
// @return an integer
public int get(int key) {
        if( !hs.containsKey(key)) {
                return -1;
        }
        // remove current
        Node current = hs.get(key);
        current.prev.next = current.next;
        current.next.prev = current.prev;
        // move current to tail
        move_to_tail(current);
        return hs.get(key).value;
}
// @param key, an integer
// @param value, an integer
// @return nothing
public void set(int key, int value) {
        if( get(key) != -1) {
                hs.get(key).value = value;
                return;
        }
        if (hs.size() == capacity) {
                hs.remove(head.next.key);
                head.next = head.next.next;
                head.next.prev = head;
        }
        Node insert = new Node(key, value);
        hs.put(key, insert);
        move_to_tail(insert);
}
private void move_to_tail(Node current) {
        current.prev = tail.prev;
        tail.prev = current;
        current.prev.next = current;
```

```
        current.next = tail;
}
```

3.38 Gray Code

It based on move and or operations.

```
public ArrayList<Integer> grayCode(int n) {
        // Write your code here
        if (n < 0) {
                return null;
        }
        ArrayList<Integer> currGray = new ArrayList<Integer>();
        currGray.add(0);
        for (int i = 0; i < n; i++) {
                int msb = 1 << i;
                for (int j = currGray.size() - 1; j >= 0; j--) {
                        currGray.add(msb | currGray.get(j));
                }
        }
        return currGray;
}
```

3.39 Divide Two Integers

```
public int divide(int dividend, int divisor) {
        // Write your code here
        if (divisor == 0) {
                return dividend >= 0 ? Integer.MAX_VALUE : Integer.MIN_VALUE;
        }
        if (dividend == 0) {
                return 0;
        }
        if (dividend == Integer.MIN_VALUE && divisor == -1) {
                return Integer.MAX_VALUE;
        }
        boolean isNegative = (dividend < 0 && divisor > 0) ||
                                                (dividend > 0 && divisor < 0);
        long a = Math.abs((long)dividend);
        long b = Math.abs((long)divisor);
        int result = 0;
        while(a >= b){
                int shift = 0;
                while(a >= (b << shift)){
                        shift++;
                }
                a -= b << (shift - 1);
                result += 1 << (shift - 1);
        }
        return isNegative ? -result : result;
}
```

3.40 Longest Sub-string Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters. In this task, when we meet a repeat character, we should use another index to delete all characters before this repeat character, not delete all characters.

```
public int lengthOfLongestSubstring(String s) {
        // write your code here
        if (s == null || s.length() == 0) {
                return 0;
        }
        HashSet<Character> set = new HashSet<Character>();
        int leftBound = 0, max = 0;
        for (int i = 0; i < s.length(); i++) {
                if (set.contains(s.charAt(i))) {
                        // delete all character before repeat char
                        while (leftBound < i && s.charAt(leftBound) !=
                            s.charAt(i)) {
                                set.remove(s.charAt(leftBound));
                                leftBound++;
                        }
                        leftBound++;
                } else {
                        set.add(s.charAt(i));
                        max = Math.max(max, i - leftBound + 1);
                }
        }
        return max;
}
```

3.41 The Smallest Difference

We sort these two array firstly, and use two pointers to move and compare to get minimum difference.

```
public int smallestDifference(int[] A, int[] B) {
        // write your code here
        if (A == null || B == null || A.length == 0 || B.length == 0) {
                return 0;
        }
        Arrays.sort(A);
        Arrays.sort(B);
        int aIndex = 0;
        int bIndex = 0;
        int diff = Integer.MAX_VALUE;
        while (aIndex < A.length && bIndex < B.length) {
                if (A[aIndex] <= B[bIndex]) {
                        diff = Math.min(diff, B[bIndex] - A[aIndex++]);
        } else {
                        diff = Math.min(diff, A[aIndex] - B[bIndex++]);
                }
                if (diff == 0) {
                        return diff;
                }
```

```
        }
        if (bIndex != B.length - 1) {
                while (bIndex < B.length) {
                        diff = Math.min(diff, Math.abs(A[aIndex - 1] -
                            B[bIndex++]));
                }
        } else {
                while (aIndex < A.length) {
                        diff = Math.min(diff, Math.abs(A[aIndex++] - B[bIndex
                            - 1]));
                }
        }
        return diff;
}
```

3.42 Longest Sub-string with At Most K Distinct Characters

```
public int lengthOfLongestSubstringKDistinct(String s, int k) {
        if (s == null || s.length() == 0 || k <= 0) {
                return 0;
        }
        int start = 0;
        int res = 1;
        Map<Character, Integer> map = new HashMap<Character, Integer>();
        map.put(s.charAt(0), 1);
        for (int end = 1; end < s.length(); end++) {
                char ch = s.charAt(end);
                if (map.containsKey(ch)) {
                        map.put(ch, map.get(ch) + 1);
                } else {
                        if (map.size() == k) {
                                res = Math.max(res, end - start);
                                //full map, need to remove the first
                                    character in ths substring
                                for (int index = start; index < end; index++)
                                    {
                                map.put(s.charAt(index),
                                    map.get(s.charAt(index))-1);
                                     start++;
                                     if (map.get(s.charAt(index)) == 0) {
                                     //have removed all occurance of a char
                                             map.remove(s.charAt(index));
                                             break;
                                     }
                                }
                        }
                        map.put(ch, 1);
                }
        }
        res = Math.max(res, s.length() - start);
        return res;
}
```

3.43 Trapping Rain Water

We find highest wall, and compute trapping water at left and right.

```
public int trapRainWater(int[] heights) {
        int sum = 0;
        int max = -1;
        int maxIndex = -1;
        int prev;
        // find the highest bar
        for (int i = 0; i < heights.length; i++) {
                if (max < heights[i]) {
                        max = heights[i];
                        maxIndex = i;
                }
        }
        // process all bars left to the highest bar
        prev = 0;
        for (int i = 0; i < maxIndex; i++) {
                if (heights[i] > prev) {
                        sum += (heights[i] - prev) * (maxIndex - i);
                        prev = heights[i];
                }
                sum -= heights[i];
        }
        // process all bars right to the highest bar
        prev = 0;
        for (int i = heights.length - 1; i > maxIndex; i--) {
                if (heights[i] > prev) {
                        sum += (heights[i] - prev) * (i - maxIndex);
                        prev = heights[i];
                }
                sum -= heights[i];
        }
        return sum;
}
```

3.44 Sort Colors II

We apply bucket sorting algorithm.

```
public void sortColors2(int[] colors, int k) {
        // write your code here
        if (colors == null) {
                return;
        }
        int len = colors.length;
        for (int i = 0; i < len; i++) {
                // Means need to deal with A[i]
                while (colors[i] > 0) {
                        int num = colors[i];
                        if (colors[num - 1] > 0) {
                                // 1. There is a number in the bucket,
```

```
                                // Store the number in the bucket in position
                                    i;
                                colors[i] = colors[num - 1];
                                colors[num - 1] = -1;
                        } else if (colors[num - 1] <= 0) {
                                // 2. Bucket is using or the bucket is empty.
                                colors[num - 1]--;
                                // delete the A[i];
                                colors[i] = 0;
                        }
                }
        }
        int index = len - 1;
        for (int i = k - 1; i >= 0; i--) {
                int cnt = -colors[i];
                // Empty number.
                if (cnt == 0) {
                        continue;
                }
                while (cnt > 0) {
                        colors[index--] = i + 1;
                        cnt--;
                }
        }
}
```

3.45 Interleaving Positive and Negative Numbers

```
public void rerange(int[] A) {
        // Check the input parameter.
        if (A == null || A.length <= 2) {
                return;
        }
        int len = A.length;
        int cntPositive = 0;
        // store the positive numbers index.
        int i1 = 0;
        for (int i2 = 0; i2 < len; i2++) {
                if (A[i2] > 0) {
                        cntPositive++;
                        // Put all the positive numbers at in the left part.
                        swap(A, i1++, i2);
                }
        }
        // If positive numbers are more than negative numbers,
        // Put the positive numbers at first.
        int posPointer = 1;
        int negPointer = 0;
        if (cntPositive > A.length / 2) {
                // Have more Positive numbers;
                posPointer = 0;
                negPointer = 1;
```

```
                // Reverse the array.
                int left = 0;
                int right = len -1;
                while (right >= cntPositive) {
                        swap(A, left, right);
                        left++;
                        right--;
                }
        }
        // Reorder the negative and the positive numbers.
        while (true) {
                // Should move if it is in the range.
                while (posPointer < len && A[posPointer] > 0) {
                        posPointer += 2;
                }
                // Should move if it is in the range.
                while (negPointer < len && A[negPointer] < 0) {
                        negPointer += 2;
                }
                if (posPointer >= len || negPointer >= len) {
                        break;
                }
                swap(A, posPointer, negPointer);
        }
}
```

3.46 Best Time to Buy and Sell Stock I to IV

For the part one, if we only perform one purchase, then it is easy to know that we can get current maximum and current minimum until end.

```
public int maxProfit(int[] prices) {
        // write your code here
        if (prices == null || prices.length <= 1) {
                return 0;
        }
        int profit = 0;
        int curPriceMin = Integer.MAX_VALUE;
        for (int price : prices) {
                profit = Math.max(profit, price - curPriceMin);
                curPriceMin = Math.min(curPriceMin, price);
        }
        return profit;
}
```

For part two, we can buy as much times as possible as we want, then we gen get all profit between two peak and gap.

```
public int maxProfit(int[] prices) {
        // write your code here
        if (prices == null || prices.length <= 1) {
                return 0;
        }
        int profit = 0;
        for (int i = 1; i < prices.length; i++) {
                int diff = prices[i] - prices[i - 1];
```

```
                if (diff > 0) {
                        profit += diff;
                }
        }
        return profit;
}
```

For part three, if we can purchase two stack but not same time, we can divide the total process into two time periods, and find a time period, which first maximum profit happen before it, and second maximum profit happen after it.

```
public int maxProfit(int[] prices) {
        // write your code here
        if (prices == null || prices.length <= 1) return 0;
        // get profit in the front of prices
        int[] profitFront = new int[prices.length];
        profitFront[0] = 0;
        for (int i = 1, valley = prices[0]; i < prices.length; i++) {
                profitFront[i] = Math.max(profitFront[i - 1], prices[i] -
                    valley);
                valley = Math.min(valley, prices[i]);
        }
        // get profit in the back of prices, (i, n)
        int[] profitBack = new int[prices.length];
        profitBack[prices.length - 1] = 0;
        for (int i = prices.length - 2, peak = prices[prices.length - 1]; i
            >= 0; i--) {
                profitBack[i] = Math.max(profitBack[i + 1], peak - prices[i]);
                peak = Math.max(peak, prices[i]);
        }
        // add the profit front and back
        int profit = 0;
        for (int i = 0; i < prices.length; i++) {
                profit = Math.max(profit, profitFront[i] + profitBack[i]);
        }
        return profit;
}
```

For the part four, if the quantity of transaction is big enough, it can change to part two problem. Or we will apply dynamic programming to solve it.

```
public int maxProfit(int k, int[] prices) {
        // write your code here
        if (k == 0) {
                return 0;
        }
        if (k >= prices.length / 2) {
                int profit = 0;
                for (int i = 1; i < prices.length; i++) {
                        if (prices[i] > prices[i - 1]) {
                        profit += prices[i] - prices[i - 1];
                        }
                }
                return profit;
        }
        int n = prices.length;
        int[][] mustsell = new int[n + 1][n + 1];
```

```
        int[][] globalbest = new int[n + 1][n + 1];
        mustsell[0][0] = globalbest[0][0] = 0;
        for (int i = 1; i <= k; i++) {
                mustsell[0][i] = globalbest[0][i] = 0;
        }
        for (int i = 1; i < n; i++) {
                int gainorlose = prices[i] - prices[i - 1];
                mustsell[i][0] = 0;
                for (int j = 1; j <= k; j++) {
                        mustsell[i][j] = Math.max(globalbest[(i - 1)][j - 1]
                           + gainorlose,       mustsell[(i - 1)][j] +
                           gainorlose);
                globalbest[i][j] = Math.max(globalbest[(i - 1)][j],
                   mustsell[i][j]);
                }
        }
        return globalbest[(n - 1)][k];
}
```

3.47 Binary Search Tree Iterator

```
private Stack<TreeNode> stack = new Stack<>();
private TreeNode curt;
//@param root: The root of binary tree.
public BSTIterator(TreeNode root) {
        // write your code here
        curt = root;
}
//@return: True if there has next node, or false
public boolean hasNext() {
        // write your code here
        return (curt != null || !stack.isEmpty()); // important to judge curt
           != null
}
//@return: return next node
public TreeNode next() {
        // write your code here
        while (curt != null) {
                stack.push(curt);
                curt = curt.left;
        }
        curt = stack.pop();
        TreeNode node = curt;
        curt = curt.right;
        return node;
}
```

3.48 Expression Evaluation

We use a stack to save operators.

```
public boolean isOp(String s) {
```

```
        return s.equals("+") || s.equals("-") || s.equals("*") ||
            s.equals("/");
}
public int compute(int left, String op, int right) {
        if (op.equals("+")) {
                return left + right;
        } else if (op.equals("-")) {
                return left - right;
        } else if (op.equals("*")) {
                return left * right;
        } else {
                return left / right;
        }
}
public int evaluateExpression(String[] expression) {
        // write your code here
        int len = expression.length;
        if (len == 0) {
                return 0;
        }
        Stack<Integer> vals = new Stack<Integer>();
        int[] index = {0};
        return helper(expression, index, vals);
}
public int helper(String[] exp, int[] index, Stack<Integer> vals) {
        if (index[0] == exp.length) {
                return 0;
        }
        Stack<String> operators = new Stack<String>();
        // alwasy start from 1st element after "("
        int i = index[0];
        for (; i < exp.length; i++) {
                if (exp[i].equals(")")) {
                        // done with current recursion
                        break;
                } else if (isOp(exp[i])) {
                        operators.push(exp[i]);
                } else {
                        int val = 0;
                        if (exp[i].equals("(")){
                                int[] tmp = {i + 1};
                                val = helper(exp, tmp, vals);
                                i = tmp[0];
                        } else {
                                val = Integer.parseInt(exp[i]);
                        }
                        if (!operators.isEmpty() &&
                            (operators.peek().equals("*") ||
                            operators.peek().equals("/"))) {
                                int right = val;
                        int left = vals.pop();
                        String op = operators.pop();
                        int res = compute(left, op, right);
                        vals.push(res);
```

```
                        } else {
                                // treat "-" as sign rather than operator,
                                    since "-" operator is buggy and hard to
                                    handle.
                                if (!operators.isEmpty() &&
                                    operators.peek().equals("-")) {
                                        val = -val;
                                }
                                vals.push(val);
                        }
                }
        }
        // sum all remaining values
        while(!operators.isEmpty()) {
                int right = vals.pop();
                int left = vals.pop();
                operators.pop();
                // int res = compute(left, "+", right);
                vals.push(left + right);
        }
        index[0] = i;
        return vals.isEmpty() ? 0 : vals.pop();
}
```

3.49 Regular Expression Matching

```
public boolean isMatch(String s, String p) {
        // base case
        if (p.length() == 0) {
                return s.length() == 0;
        }
        // special case
        if (p.length() == 1) {
                // if the length of s is 0, return false
                if (s.length() < 1) {
                        return false;
                }
                //if the first does not match, return false
                else if ((p.charAt(0) != s.charAt(0)) && (p.charAt(0) !=
                    '.')) {
                        return false;
                }
                // otherwise, compare the rest of the string of s and p.
                else {
                        return isMatch(s.substring(1), p.substring(1));
                }
        }
        // case 1: when the second char of p is not '*'
        if (p.charAt(1) != '*') {
                if (s.length() < 1) {
                        return false;
                }
```

```
                if ((p.charAt(0) != s.charAt(0)) && (p.charAt(0) != '.')) {
                        return false;
                } else {
                        return isMatch(s.substring(1), p.substring(1));
                }
        }
        // case 2: when the second char of p is '*', complex case.
        else {
                //case 2.1: a char & '*' can stand for 0 element
                if (isMatch(s, p.substring(2))) {
                        return true;
                }
                //case 2.2: a char & '*' can stand for 1 or more preceding
                    element,
                //so try every sub string
                int i = 0;
                while (i<s.length() && (s.charAt(i)==p.charAt(0) ||
                    p.charAt(0)=='.')){
                        if (isMatch(s.substring(i + 1), p.substring(2))) {
                                return true;
                        }
                        i++;
                }
                return false;
        }
}
```

3.50 Count of Smaller Number before itself

We apply segment tree to speed up the process on location smaller values.

```
class SegmentTreeNode {
        int start, end;
        int count;
        SegmentTreeNode left;
        SegmentTreeNode right;
        public SegmentTreeNode(int start, int end) {
                this.start = start;
                this.end = end;
                this.count = 0;
        }
}
public ArrayList<Integer> countOfSmallerNumberII(int[] A) {
        ArrayList<Integer> result = new ArrayList<>();
        if (A == null || A.length == 0) {
                return result;
        }
        SegmentTreeNode node = buildSegmentTree(0, 10000);
        for (int i = 0; i < A.length; i++) {
                if (A[i] == 0) {
                        result.add(0);
                } else {
                        result.add(query(node, 0, A[i] - 1));
                }
                update(node, A[i]);
```

```
        }
        return result;
}
public SegmentTreeNode buildSegmentTree(int start, int end) {
        SegmentTreeNode node = new SegmentTreeNode(start, end);
        if (start < end) {
                int mid = start + (end - start) / 2;
                node.left = buildSegmentTree(start, mid);
                node.right = buildSegmentTree(mid + 1, end);
                node.count = node.left.count + node.right.count;
        }
        return node;
}
public int query(SegmentTreeNode node, int start, int end) {
        if (node.start == start && node.end == end) {
                return node.count;
        }
        int leftCount = 0, rightCount = 0;
        int mid = node.start + (node.end - node.start) / 2;
        if (start <= mid) {
                if (end <= mid) {
                        leftCount = query(node.left, start, end);
                } else {
                        leftCount = query(node.left, start, mid);
                }
        }
        if (end > mid) {
                if (start > mid) {
                        rightCount = query(node.right, start, end);
                } else {
                        rightCount = query(node.right, mid + 1, end);
                }
        }
        return leftCount + rightCount;
}
public void update(SegmentTreeNode node, int index) {
        if (node.start == index && node.end == index) {
                node.count = node.count + 1;
        } else {
                int mid = node.start + (node.end - node.start) / 2;
                if (node.start <= index && index <= mid) {
                        update(node.left, index);
                } else if (mid < index && index <= node.end) {
                        update(node.right, index);
                }
                node.count = node.left.count + node.right.count;
                //or node.count = node.count+1;
        }
}
```

3.51 Expression Tree Build

```
class TreeNode {
```

```
        public int val;
        public String s;
        public ExpressionTreeNode root;
        public TreeNode(int val, String ss) {
                this.val = val;
                this.root = new ExpressionTreeNode(ss);
        }
}

int get(String a, Integer base) {
        if (a.equals("+") || a.equals("-"))
                return 1 + base;
        if (a.equals("*") || a.equals("/"))
                return 2 + base;
        return Integer.MAX_VALUE;
}

public ExpressionTreeNode build(String[] expression) {
        // write your code here
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode root = null;
        int val = 0;
        Integer base = 0;
        for (int i = 0; i <= expression.length; i++) {
                if(i != expression.length) {
                        if (expression[i].equals("(")) {
                                base += 10;
                                continue;
                        }
                        if (expression[i].equals(")")) {
                                base -= 10;
                                continue;
                        }
                        val = get(expression[i], base);
                }
                TreeNode right = i == expression.length ? new TreeNode(
                        Integer.MIN_VALUE, "") : new TreeNode(val,
                        expression[i]);
                while (!stack.isEmpty()) {
                        if (right.val <= stack.peek().val) {
                                TreeNode nodeNow = stack.pop();
                                if (stack.isEmpty()) {
                                        right.root.left = nodeNow.root;
                                } else {
                                        TreeNode left = stack.peek();
                                        if (left.val < right.val) {
                                                right.root.left =
                                                   nodeNow.root;
                                        } else {
                                                left.root.right =
                                                   nodeNow.root;
                                        }
                                }
                        } else {
```

```
                                break;
                        }
                }
                stack.push(right);
        }
        return stack.peek().root.left;
}
```

3.52 Convert Expression to (Reverse) Polish Notation

In this kind of problem, the key point is how to deal with (and). For the normal order, it is:

```
public ArrayList<String> convertToPN(String[] expression) {
        ArrayList<String> list = new ArrayList<String>();
        Stack<String> stack = new Stack<String>();
        for (int cur = expression.length - 1; cur >= 0; cur--) {
                String str = expression[cur];
                if (isOp(str)) {
                        if (str.equals(")")) {
                                stack.push(str);
                        } else if (str.equals("(")) {
                                while (!stack.isEmpty()) {
                                        String inPar = stack.pop();
                                        if (inPar.equals(")")) {
                                                break;
                                        }
                                        list.add(inPar);
                                }
                        } else {
                                while (!stack.isEmpty() && order(str) <
                                    order(stack.peek())) {
                                        list.add(stack.pop());
                                }
                                stack.push(str);
                        }
                } else {
                        list.add(str);
                }
        }
        while (!stack.isEmpty()) {
                list.add(stack.pop());
        }
        ArrayList<String> result = new ArrayList<String>();
        for (int i = list.size() - 1; i >= 0; i--) {
                result.add(list.get(i));
        }
        return result;
}
boolean isOp(String str) {
        if (str.equals("+") || str.equals("-") || str.equals("*") ||
            str.equals("/") || str.equals("(") || str.equals(")")) {
                return true;
        } else {
                return false;
        }
```

```
}
int order(String a) {
        if (a.equals("*") || a.equals("/")) {
                return 2;
        } else if (a.equals("+") || a.equals("-")) {
                return 1;
        } else {
                return 0;
        }
}
```

For the reverse order, it is:

```
public ArrayList<String> convertToRPN(String[] expression) {
        ArrayList<String> list = new ArrayList<String>();
        Stack<String> stack = new Stack<String>();
        for (int i = 0; i < expression.length; i++) {
                String str = expression[i];
                if (isOp(str)) {
                        if (str.equals("(")) {
                                stack.push(str);
                        } else if (str.equals(")")) {
                                while (!stack.isEmpty()) {
                                        String p = stack.pop();
                                        if (p.equals("(")) {
                                                break;
                                        }
                                        list.add(p);
                                }
                        } else {
                                while (!stack.isEmpty() && order(str) <=
                                    order(stack.peek())) {
                                        list.add(stack.pop());
                                }
                                stack.push(str);
                        }
                } else {
                        list.add(str);
                }
        }
        while (!stack.isEmpty()) {
                list.add(stack.pop());
        }
        return list;
}
boolean isOp(String str) {
        if (str.equals("+") || str.equals("-") || str.equals("*") ||
            str.equals("/") || str.equals("(") || str.equals(")")) {
                return true;
        } else {
                return false;
        }
}
int order(String a) {
        if (a.equals("*") || a.equals("/")) {
```

```
                return 2;
        } else if (a.equals("+") || a.equals("-")) {
                return 1;
        } else {
                return 0;
        }
}
```

3.53 Building Outline I & II

http://blog.csdn.net/earthma/article/details/45575343l

Permutation Index The key point to compute how many permutation had already be constructed based on current situation.

```
public long permutationIndex(int[] A) {
        if (A == null || A.length == 0) return 0L;
        long index = 1, fact = 1;
        for (int i = A.length - 1; i >= 0; i--) {
                // get rank in every iteration
                int rank = 0;
                for (int j = i + 1; j < A.length; j++) {
                        if (A[i] > A[j]) rank++;
                }
                index += rank * fact;
                fact *= (A.length - i);
        }
        return index;
}
```

If we add repeat number.

```
public long permutationIndexII(int[] A) {
        if (A == null || A.length == 0) return 0L;
        Map<Integer, Integer> hashmap = new HashMap<Integer, Integer>();
        long index = 1, fact = 1, multiFact = 1;
        for (int i = A.length - 1; i >= 0; i--) {
                // collect its repeated times and update multiFact
                if (hashmap.containsKey(A[i])) {
                        hashmap.put(A[i], hashmap.get(A[i]) + 1);
                        multiFact *= hashmap.get(A[i]);
                } else {
                        hashmap.put(A[i], 1);
                }
                // get rank every turns
                int rank = 0;
                for (int j = i + 1; j < A.length; j++) {
                        if (A[i] > A[j]) rank++;
                }
                // do divide by multiFact
                index += rank * fact / multiFact;
                fact *= (A.length - i);
        }
```

```
        return index;
}
```

3.54 Heapify

This task is the process change a list to a heap.

```
public void heapify(int[] A) {
        // write your code here
        for (int i = A.length / 2; i >= 0; i--) {
                minHeap(A, i);
        }
}
private void minHeap(int[] A, int i) {
        int len = A.length;
        while (i < len) {
                int minIndex = i;
                if (i * 2 + 1 < len && A[i * 2 + 1] < A[minIndex]) {
                        minIndex = i * 2 + 1;
                }
                if (i * 2 + 2 < len && A[i * 2 + 2] < A[minIndex]) {
                        minIndex = i * 2 + 2;
                }
                if (i == minIndex) {
                        break;
                }
                int temp = A[i];
                A[i] = A[minIndex];
                A[minIndex] = temp;
                i = minIndex;
        }
}
```

3.55 Segment Tree Query

Because it nodes already have the information on range, and we can use this definition to search node quickly.

```
public int query(SegmentTreeNode root, int start, int end) {
        if(start > end || root==null) {
                return 0;
        }
        if(start <= root.start && root.end <= end) {
                return root.count;
        }
        int mid = (root.start + root.end) / 2;
        int leftsum = 0, rightsum = 0;
        if(start <= mid) {
                if( mid < end) {
                        leftsum =  query(root.left, start, mid);
                } else {
                        leftsum =  query(root.left, start, end);
                }
        }
        if(mid < end) {
                if (start <= mid) {
```

```
                        rightsum = query(root.right, mid + 1, end);
                } else {
                        rightsum = query(root.right, start, end);
                }
        }
        return leftsum + rightsum;
}
```