Table of Contents

Introduction

Notebook for Interview Question Preparation includes Algorithm, Data Structure, Object-Oriented Programming and System Design. Everything for Technical Interview. Most of Questions come from Leetcode / Lintcode / Cracking Coding Interview / Nine Chapter Online Course / GeeksForGeeks / HackerRank.

Technical Interviews are always full of challenges. Sometime it depends on how well you learn about IT knowledge. Sometimes it is just about the lucky and destiny. People got a good offer doesn't mean they are better than us. Maybe you learn a lot but just don't know that question. But learning is never harmful. Don't be upset! Keep learning, record every thing you learned, show and help other people who need it. That is main purpose of this notebook.

Visit my blog: xmruibi.github.io

Algorithm Part:

1. Categories of algorithm questions
   - Pointer
   - Sort and Search
   - From recursion to Dynamic Programming
   - Mathematics in CS
   - Big Data
2. The analysis and summary of programming problems, and most of the programming problems come from Leetcode, Lintcode and GeeksForGeeks.

Data Structure Part:

1. Conclusion of popular data structure
2. The analysis and summary on some data structure questions collected from Leetcode, Lintcode and GeeksForGeeks.

Oject-Oriented Programming Part:

1. Principle of Object Oriented Programming
2. Conclusion of popular design patterns
3. The analysis and summary on typical design patterns
4. Classical Object-Oriented Programming questions:
    - Poker Game
    - Chess Game
    - Parking Lot
    - Online Book System
    - Chat Server

Scalable System Design Part:

1. System Design Procedure
2. Typical System Design questions:

Algorithm

Algorithm is most interesting part in computer science. It measure a person how smart and how deep in learning computer science.

I concluded some parts or categories of algorithm questions, but all of these are intertwined together. In one algorithm question, it can combine multiple knowledge points with several categories.

As I know, the algorithm should contains two things. The data, which is the source of information, is the stuff what the algorithm manipulates; Another is strategy, the core of algorithm, which is the thinking, the soul. Here I mainly focus on concludes the strategy that commonly used in algorithm interview questions.

Strategy - Thinking in Algorithms

1. Pointers

Base on Data Structure

- In Array
    - Interleaving
    - Partition
    - Squeeze Rule
    - with Binary Search
- In String
    - Palindrome
    - Sub-string Windows
- In Linked List
    - Walker and Runner Node
    - Dummy Node

Base on Purpose

- Remove Duplicate
- Make Partition
- Do swap
- Find element

2. Sort and Search

Sort

- Selection Sort
- Insertion Sort
- Bubble Sort
- Merge Sort
- Quick Sort
- Count Sort
- Radix Sort

Search

- Binary Search
- Ternary Search
- Breath First Search
- Depth First Search
- Union Find

3. From Recrusion to Dynamic Programming

Recursion

Divide and Conquer

- In array
- In tree
- In linked list

Backtracking

- Combination
- Permutation
- N Queen

Dynamic Programming

- Matrix DP
- One Sequence DP
- Two Sequence DP
- Backpack

4. Computer Science cannot live without Math

Math

- Greatest Common Division
- Least Common Multiple
- Elementary Arithmetic Operation with Bitwise Idea
- Binary, octonary and tenary
- Power and square root (Divide and Conquer)

Bitwise Manipulation

- OR's Magic
- Left or Right Shifting

5. Big Data

Top K Problem

- Kth largest / smallest element in dataset
- Find median in data stream
- Top K frequent element

Cache Algorithm

- Least Recently Used
- Least Frequently Used

Bit Set

External Sort

Bloom Fliter

Array

Array is most fundamental container to hold multiple elements. It the skeletal part of Array List. In java, array can contain both primitive element or object.

Attribute

- Index
- Value

Routines

The array can be applied to every algorithm. Th Basic question for testing typically array knowledge are following:

- Union or Intersection of Array
- Based on Two Pointers
  - Interleaving
  - Partition
  - Squeeze Rule
- Element Count
  - Count Inversions in an array
  - Count nearest large element
  - Applied with Segment Tree
- Sort
  - Most of Sort algorithm
- Search
  - Binary Search

You can it is almost every where.

Given an array `nums` of integers and an int `k`, partition the array (i.e move the elements in "nums") such that:

- All elements < k are moved to the left
- All elements >= k are moved to the right

Return the partitioning index, i.e the first index i nums[i] >= k.

Solution:

- Typical index rotate two pointer problem, looks like the idea of quick sort .
- Set an index `pivot` for marking the real position of element less than `k` during pass the orginal array.
- Once the current passing index `i` hits the element less than `k`, we do the swap with `pivot` and `i`.
- Make sure the `pivot` should add 1 after swap since the marked position is increase for next one.
- However in new `pivot` position we not sure the element's value, so... we need to check in next procedure.
- Make sure the `i` position decrease 1, because we just did a swap and we need to check the new `i` is less than 'k'

```
public class Solution {
    /**
     *@param nums: The integer array you should partition
     *@param k: As description
     *return: The index after partition
     */
    public int partitionArray(int[] nums, int k) {
        if(nums == null || nums.length == 0)
            return 0;

        int pivot = 0;
        for(int i = 0; i < nums.length; i++) {
            if(i > pivot && nums[i] < k) {
                int tmp = nums[pivot];
                nums[pivot++] = nums[i];
                nums[i--] = tmp;
            }
        }
        // this is just for corner case when the last element st
ill less than k
        if(nums[nums.length - 1] < k)
            return nums.length;
        return pivot;
    }
}
```

Given two array of integers(the first array is array A, the second array is array B), now we are going to find a element in array A which is A[i], and another element in array B which is B[j], so that the difference between A[i] and B[j] (|A[i] - B[j]|) is as small as possible, return their smallest difference.

Example

For example, given array A = `[3,6,7,4]`, B = `[2,8,9,3]`, return `0`

Challenge

O(n log n) time

Solution

- Do sort on one of array
- One pass on another array and do binary search on the sorted array.
- Search the target value from passing array and get the minimum difference on sorted array
- Update the global minimum difference each time.

```
public class Solution {
    /**
     * @param A, B: Two integer arrays.
     * @return: Their smallest difference.
     */
    public int smallestDifference(int[] A, int[] B) {
        if(A == null || B == null || A.length == 0 || B.length =
= 0)
            return 0;
        Arrays.sort(B);
        int mindiff = Integer.MAX_VALUE;
        for(int i = 0; i < A.length; i++) {
            int curdiff = binarySearch(B, A[i]);
            mindiff = Math.min(mindiff, curdiff);
        }
        return mindiff;
    }


    private int binarySearch(int[] A, int value) {
        if(A == null || A.length == 0)
            return 0;
        int l = 0, r = A.length - 1;
        while(l + 1 < r) {
            int m = l + ((r - l) >> 1);
            if(value > A[m])
                l = m;
            else
                r = m;
        }
        return Math.min(Math.abs(A[l] - value), Math.abs(A[r] -
value));
    }
}
```

Given an integer array, find a subarray where the sum of numbers is zero. Your code should return the index of the first number and the index of the last number.

Solution

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the fir
st number
     *          and the index of the last number
     */
    public ArrayList<Integer> subarraySum(int[] nums) {
        // write your code here
        ArrayList<Integer> res = new ArrayList<Integer>();
        if(nums == null || nums.length == 0)
            return res;
        HashMap<Integer, Integer> map = new HashMap<Integer>();
        int sum = 0;
        map.put(0, 0);
        for(int i = 0; i < nums.length; i++) {
            if(map.containsKey(sum+nums[i])) {
                res.add(map.get(sum+nums[i]));
                res.add(i);
                return res;
            }
            map.put(sum+=nums[i], i+1);
        }
        return res;
    }
}
```

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.

Example

Given `[1,2,3,4]` and interval = `[1,3]`, return `4`. The possible answers are:

```
[0, 0]
[0, 1]
[1, 1]
[2, 2]
```

Solution

```
public int subarraySumII(int[] A, int start, int end) {
    int res = 0;
        for(int i = 1; i < A.length; i++)
            A[i] += A[i - 1];

        Arrays.sort(A);
        for(int i = 0; i < A.length; i++) {
            if(A[i] >= start && A[i] <= end)
                res++;
            // start <= A[i] - A[j] <= end
            // so the max bound and min bound of A[j] are follow
ing:
            int max = A[i] - start;
            int min = A[i] - end;
            // max + 1 make sure the right bound of max value an
d also index problem
            int range = findInsPos(A, max + 1) - findInsPos(A, m
in);
            res += range;
        }
        return res;
}
private int findInsPos(int[] A, int value) {
        int l = 0, r = A.length - 1;

        while(l < r - 1) {
            int m = l + ((r - l) >>1);
            if(A[m] < value)
                l = m;
            else
                r = m;
        }
        if(A[l] >= value)
            return l;
        else
            return r;
}
```

Longest Common Sub-sequence

Given two strings, find the longest common subsequence (LCS).

Your code should return the length of LCS.

Example

For `ABCD` and `EDCA`, the LCS is `A` (or `D`, `C`), return 1.

For `ABCD` and `EACB`, the LCS is `AC`, return 2.

Solution

```
public class Solution {
    /**
     * @param A, B: Two strings.
     * @return: The length of longest common subsequence of A an
d B.
     */
    public int longestCommonSubsequence(String A, String B) {
        if(A == null || B == null)
            return 0;

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

Longest Increasing Continuous Subsequence

Give you an integer array (index from 0 to n-1, where n is the size of this array)， find the longest increasing continuous subsequence in this array. (The definition of the longest increasing continuous subsequence here can be from right to left or from left to right)

Example

For `[5, 4, 2, 1, 3]` , the LICS is `[5, 4, 2, 1]` , return `4` .

For `[5, 1, 2, 3, 4]` , the LICS is `[1, 2, 3, 4]` , return `4` .

Note

O(n) time and O(1) extra space.

Solution

- This is O(1) space dynamic programming. Just maintain one local max and one global max variable.
- The default value of local maximum variable is 2.
- The condition for growing the local maximum is by `(A[i] - A[i-1])*(A[i-1] - A[i-2]) > 0` , which means `[i-2] < [i-1] < [i]` or `[i-2] > [i-1] > [i]` .

```
public class Solution {
    /**
     * @param A an array of Integer
     * @return  an integer
     */
    public int longestIncreasingContinuousSubsequence(int[] A) {
        if( A == null )
            return 0;
        if( A.length <= 1)
            return A.length;

        int max = 2;
        int cur = 2;
        for(int i = 2; i < A.length; i++) {
            if((A[i] - A[i-1])*(A[i-1] - A[i-2]) > 0){
                cur ++;
            }else
                cur = 2;
            max = Math.max(max, cur);
        }
        return max;
    }
}
```

Give you an integer matrix (with row size n, column size m)，find the longest increasing continuous subsequence in this matrix. (The definition of the longest increasing continuous subsequence here can start at any row or column and go up/down/right/left any direction).

Example

Given a matrix:

```
[
  [1 ,2 ,3 ,4 ,5],
  [16,17,24,23,6],
  [15,18,25,22,7],
  [14,19,20,21,8],
  [13,12,11,10,9]
]
```

return `25`

Challenge

O(nm) time and memory.

Solution:

- This is great question with DFS, Dynamic Problem and Subsequence idea.
- The idea is also simple. Recursively search by DFS while we can do some memorized stuff.
- Each time we figure out the maximum length with reversed increasing sequence from each element in matrix.
- Note that the searching is by decreasing way.

```
public class Solution {
    /**
     * @param A an integer matrix
     * @return  an integer
     */
    // memorized the local maximum length
    int[][] memo;

    boolean[] visited;
    int n ,m;

    // stepping way for dfs
    int[] dx = {1,-1,0,0};
    int[] dy = {0,0,1,-1};

    public int longestIncreasingContinuousSubsequenceII(int[][]
A) {
        if(A.length == 0)
            return 0;
        n = A.length;
        m  = A[0].length;

        memo = new int[n][m];
        visited = new boolean[n*m];

        int res = 0;
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < m; j++) {
                memo[i][j] = helper(i, j, A);
                res = Math.max(res, memo[i][j]);
            }
        }
        return res;
    }

    private int helper(int x, int y, int[][] A) {
        // once it touched the visited element, return that value

        if(visited[x * m + y])
            return memo[x][y];

        int res = 1;
        for(int i = 0; i < 4; i++) {
            int nx = x + dx[i];
```

```
                int ny = y + dy[i];
                if(0<= nx && nx < n && 0<= ny && ny < m ) {
                    // this is tricky point, we search by decreasing
                    if( A[x][y] > A[nx][ny])
                        res = Math.max(res,  helper(nx, ny, A) + 1);
                }
            }
            visited[x * m + y] = true;
            memo[x][y] = res;
            return res;
        }
    }
```

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

Example

Given 4 points: `(1,2)` , `(3,6)` , `(0,0)` , `(1,3)` .

The maximum number is `3` .

Solution

- Use one point as a baseline. (`Point pa`)
- Iterate other points (`Point pb`) (index greater than `pa`): `j = i + 1` and
- Use Hash Map to record the ratio and count
- Note:
  - Ratio is `double radio = (double)(pa.y - pb.y) / (double)(pa.x - pb.x)` ;
  - When pa.x == pb.x && pa.y == pb.y, consider two points are the same, also need to count the same point.
  - When only `pa.x == pb.x` , that means the ratio is infinity as `(double)Integer.MAX_VALUE` ;
  - When only `pa.y == pb.y` , that means the ratio is zero;
- Iterate Hash Map and get the local max with updating the global max;

```
public class Solution {
    /**
     * @param points an array of point
     * @return an integer
     */
    public int maxPoints(Point[] points) {
        if(points == null || points.length == 0)
            return 0;


        int maxLine = 0;
        for(int i = 0; i < points.length; i++) {
            Map<Double, Integer> map = new HashMap<>();
            Point pa = points[i];
            int same = 0;
            for(int j = i + 1; j < points.length; j++) {
                    Point pb = points[j];
                    int cnt = 0;
                    if(pa.x == pb.x && pa.y == pb.y)
                        same ++;
                    else if(pa.x == pb.x) {
                        map.put((double)Integer.MAX_VALUE, map.c
ontainsKey((double)Integer.MAX_VALUE)?map.get((double)Integer.MA
X_VALUE) + 1 : 2);
                    }else if(pa.y == pb.y)
                        map.put((double)0, map.containsKey((doub
le)0)?map.get((double)0) + 1 : 2);
                    else{
                        double radio = (double)(pa.y - pb.y) / (
double)(pa.x - pb.x);
                        map.put(radio, map.containsKey(radio)?ma
p.get(radio) + 1 : 2);
                    }
            }
            int localMax = 1;
            for (Integer value : map.values())
                localMax = Math.max(localMax, value);
            localMax += same;
            maxLine = Math.max(maxLine, localMax);
        }

        return maxLine;
    }
}
```

Write an efficient algorithm that searches for a value in an m x n matrix.

This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

Solution:

- First binary search with checking the first element in each row so that we can find the row may contain the target number;
- So we search the lowbound of target number. However, we also can do if the element is just equals to target number then directly return for reducing time.
- Once we get the target row, then we do the second binary search in this row to find the target number.
- All in all, two binary search to find the target!

```
public class Solution {
    /**
     * @param matrix, a list of lists of integers
     * @param target, an integer
     * @return a boolean, indicate whether matrix contains targe
t
     */
    public boolean searchMatrix(int[][] matrix, int target) {
        // write your code here
        if(matrix == null || matrix.length == 0)
            return false;

        int up = 0, down = matrix.length - 1;
        while(up < down - 1) {
            int mrow = up + ((down - up) >> 1);
            if(matrix[mrow][0] == target)
                return true;
            if(matrix[mrow][0] < target)
                up = mrow;
            else
                down = mrow;
        }
        int curRow;
        if(matrix[up][0] > target)
            return false;
        else if(matrix[up][0] <= target && matrix[down][0] > tar
get)
            curRow = up;
        else
            curRow = down;

        int l = 0, r = matrix[curRow].length - 1;
        while(l < r - 1) {
            int m = l + ((r - l) >> 1);
            if(matrix[curRow][m] == target)
                return true;
            else if(matrix[curRow][m] < target)
                l = m;
            else
                r = m;
        }

        if(matrix[curRow][l] == target || matrix[curRow][r] == t
arget)
            return true;
        else
            return false;
    }
}
```

Write an efficient algorithm that searches for a value in an m x n matrix, return the occurrence of it.

This matrix has the following properties:

- Integers in each row are sorted from left to right.
- Integers in each column are sorted from up to bottom.
- No duplicate integers in each row or column.

Solution:

- Typical Matrix Search Problem, using a condition for driven coordinate moving.
- Here the value and target comparasion is the driven condition.
- Since the sorted matrix, we can start from right top element.
- Because on the diagonal from right top to left down, all the left elements are less than the right elments.
- So we have three type of running condition and set the x, y coordinate differently.

```
public class Solution {
    /**
     * @param matrix: A list of lists of integers
     * @param: A number you want to search in the matrix
     * @return: An integer indicate the occurrence of target in
the given matrix
     */

    public int searchMatrix(int[][] matrix, int target) {
        // write your code here
        if(matrix == null || matrix.length == 0)
            return 0;
        int rightTop = matrix[0][matrix[0].length - 1];
        int x = 0, y = matrix[0].length - 1;
        int occ = 0;
        while(x < matrix.length && y >= 0) {
            int cur = matrix[x][y];
            if(cur == target) {
                occ ++;
                x++; y--;
            }else if(cur < target)
                x++;
            else
                y--;
        }
        return occ;
    }
}
```

Given a m x n matrix, if an element is 0, set its entire row and column to 0. Do it in place.

Solution:

- Use the first row (up row) and the first col (left col) to record the position info of zeroes in matrix;
- But we also need to set two boolean value to check if there is zero in first row and first col;
- Then go through the matrix again, when [i][0] is marked zero or [0][j] is marked zero set current position as zero! This is important!;
- Finally, go back to check two boolean value, and set that row or col as zero if boolean value is true;

```
public class Solution {
    /**
     * @param matrix: A list of lists of integers
     * @return: Void
     */
    public void setZeroes(int[][] matrix) {

        if(matrix == null || matrix.length == 0)
            return;

        boolean row = false;
        boolean col = false;

        for(int i = 0; i < matrix.length; i++)
            if(matrix[i][0] == 0)
                col = true;

        for(int j = 0; j < matrix[0].length; j++)
            if(matrix[0][j] == 0)
                row = true;

        for(int i = 1; i < matrix.length; i++)
            for(int j = 1; j < matrix[i].length; j++)
                if(matrix[i][j] == 0) {
                    matrix[i][0] = 0;
                    matrix[0][j] = 0;
                }

        for(int i = 1; i < matrix.length; i++){
            for(int j = 1; j < matrix[0].length; j++){
                if(matrix[i][0] == 0 || matrix[0][j] == 0)
                    matrix[i][j] = 0;
            }
        }

        if(row)
            for(int j = 0; j < matrix[0].length; j++)
                    matrix[0][j] = 0;

        if(col)
            for(int i = 0; i < matrix.length; i++)
                    matrix[i][0] = 0;
    }
}
```

You are given an n x n 2D matrix representing an image. Rotate the image by 90 degrees (clockwise).

Solution:

- Headache Implement question!
- Very carefully to treat index.
- Only calculate the 1/4 of index in matrix!

```
public class Solution {
    /**
     * @param matrix: A list of lists of integers
     * @return: Void
     */
    public void rotate(int[][] matrix) {

        int n = matrix.length;

        // One of i or j need to consider boundry!
        for(int i = 0; i < ( n >> 1); i ++) {
        // that's why j < (n+1) / 2, that is the boundry!
            for(int j = 0; j < ( n+1 >> 1); j++) {
                int tmp = matrix[i][j];
                matrix[i][j] = matrix[n - j - 1][i];
                matrix[n - j - 1][i] = matrix[n - i - 1][n - j -
 1];
                matrix[n - i - 1][n - j - 1] = matrix[j][n - i -
 1];
                matrix[j][n - i - 1] = tmp;
            }
        }
    }
}
```

Given an integer matrix, find a submatrix where the sum of numbers is zero. Your code should return the coordinate of the left-up and right-down number.

Example

Given matrix

```
[
  [1 ,5 ,7],
  [3 ,7 ,-8],
  [4 ,-8 ,9],
]
```

return `[(1,1), (2,2)]`

Challenge

O(n3) time.

String

Routines

Two Pointers

String Compress

String Serialization

Given a string s, find the length of the longest substring T that contains at most k distinct characters.

Example

For example, Given s = `eceba` , k = 3,

T is `eceb` which its length is 4.

Challenge

O(n), n is the size of the string s.

Solution:

- Set two pointers as a windows for input string.
- Use a hashmap to store the characters in a window (substring) in string.
- However, we notice that we use hashmap for count the character appearance times.
- Remove the character as a key only if the count for this key is zero.
- Update the max window size each time.

```
public class Solution {
    /**
     * @param s : A string
     * @return : The length of the longest substring
     *           that contains at most k distinct characters.
     */
    public int lengthOfLongestSubstringKDistinct(String s, int k
) {
        if(s == null)
            return 0;

        HashMap<Character, Integer> dict = new HashMap<>();

        int prev = 0;
        int maxLen = 0;
        for(int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if(dict.containsKey(c)) {
                dict.put(c, dict.get(c) + 1);
            }else {
                dict.put(c, 1);
                while(dict.size() > k) {
                    char prevChar = s.charAt(prev++);
                    if(dict.get(prevChar) > 1)
                        dict.put(prevChar, dict.get(prevChar) -
1);
                    else
                        dict.remove(prevChar);
                }
            }
            maxLen = Math.max(maxLen, i - prev + 1);
        }

        return maxLen;
    }
}
```

At the first I made a mistake by using hashset. However, like the previous mentioned, we need to count the character appearance. Why? Since there is possible when the character on index `prev` has another one in this window. But when you simply remove this element, there is still one inside this window. Tha makes the mistake happened!

> Say, in `acdab`, pointer `prev` is on first `a`, once we do `set.remove(a)`, the `size()` become 3, but in fact, it is still 4.

```
    // This is the wrong code!!!
    public int lengthOfLongestSubstringKDistinct(String s, int k
) {
        // write your code here
        Set<Character> disc = new HashSet<>();

        int maxLen = 0;
        int prev = 0;
        for(int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            disc.add(c);
            while(disc.size() > k){
                disc.remove(s.charAt(prev++));
            }
            maxLen = Math.max(maxLen, i - prev + 1);
        }
        return maxLen;
    }
```

Given a string, find the length of the longest substring without repeating characters.

Example

For example, the longest substring without repeating letters for `abcabcbb` is `abc`, which the length is 3.

For `bbbbb` the longest substring is `b`, with the length of 1.

Challenge

O(n) time

Solution:

- Set two pointers as a windows for input string.
- Very simple idea, to use a hashset to store the characters in a window (substring) in string.
- While the repeat detect, move forward the previous pointer.
- Update the max window size each time.

```
public class Solution {
    /**
     * @param s: a string
     * @return: an integer
     */
    public int lengthOfLongestSubstring(String s) {
        Set<Character> disc = new HashSet<>();

        int maxLen = 0;
        int prev = 0;
        for(int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);

            while(disc.contains(c)){
                disc.remove(s.charAt(prev++));
            }
            maxLen = Math.max(maxLen, i - prev + 1);
            disc.add(c);
        }
        return maxLen;
    }
}
```

Given a string source and a string target, find the minimum window in source which will contain all the characters in target.

Example

source = "ADOBECODEBANC" target = "ABC" Minimum window is "BANC".

Note

If there is no such window in source that covers all characters in target, return the emtpy string "".

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in source.

Challenge

Can you do it in time complexity O(n) ?

Clarification

The characters in minimum window doesn't need to has the same order in target.

Solution:

```
    public String minWindow(String source, String target) {
        // preload for target checking
        if(source == null || source.length() == 0 || target == n
ull || target.length() == 0)
            return "";


        int tarLen = target.length();
        HashMap<Character, Integer> dict = new HashMap<>();
        for(char c : target.toCharArray())
            dict.put(c, dict.containsKey(c)? dict.get(c) + 1 : 1
);

        int hitCount = 0; // record current window hits how many
 characters in target
        int prevIdx = 0; // record the left bound of current win
dow
        int minWindow = source.length() + 1; // initial the mini
mum window length
        int start = 0;
        for(int i = 0; i < source.length(); i++) {
            char cur = source.charAt(i);
            // if current char is not in dict, continue
            if(!dict.containsKey(cur))
                continue;

            dict.put(cur, dict.get(cur) - 1);
            if(dict.get(cur) >= 0)
                hitCount++;

            // check the windows has amount of this char more th
an it in target string
            // loop until the amount back to normal, but always
reduce the prev index char
            while(hitCount == tarLen) {
                if( minWindow > i - prevIdx + 1) {
                    start = prevIdx;
                    minWindow = i - prevIdx + 1;
                }
                char prevChar = source.charAt(prevIdx);
                if(dict.containsKey(prevChar)) {
                    dict.put(prevChar, dict.get(prevChar)+1);
                    if(dict.get(prevChar) > 0)
                        hitCount--;
                }
                prevIdx++;
            }
        }
        //
        if(minWindow > source.length())
            return "";
        return source.substring(start, start + minWindow);
    }
```

Word Pattern

Given a pattern and a string str, find if str follows the same pattern.

Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty word in str.

Examples:

pattern = `abba` , str = `dog cat cat dog` should return true. pattern = `abba` , str = `dog cat cat fish` should return false. pattern = `aaaa` , str = `dog cat cat dog` should return false. pattern = `abba` , str = `dog dog dog dog` should return false.

Notes:

You may assume pattern contains only lowercase letters, and str contains lowercase letters separated by a single space.

Think

- Firstly, you need to convert `str` into string array with spliting by space.
- Check two stuffs are equal length, if not? directly return false!
- Setup a hashmap for storing element both pattern and strs. However, the key are different types: character, string. Why we do not just use string? Consider about this case: pattern - `abba` , str - `a a b a` . Pattern and Word has the same kind of element. It cannot easily to distinguish.
- The value in hashmap should be the
- There is one more tricky thing: why we compare the `map.put()` return value? Here is a segment in HashMap API.

```
      if (e != null) { // existing mapping for key
         V oldValue = e.value;
         if (!onlyIfAbsent || oldValue == null)
             e.value = value;
             afterNodeAccess(e);
             return oldValue;
     }
```

- That means the previous index(value) of character(key) will be returned as index reference. So we return the previous index and also update the index for next index reference!
- So if the both returned index doesn't matched, it should return false.

Solution

```
public class Solution {
    public boolean wordPattern(String pattern, String str) {
        String[] strs = str.split(" ");
        if(strs.length != pattern.length())
            return false;
        Map<Object, Integer> map = new HashMap<>();
        for(int i = 0; i < strs.length; i++) {
            if(!Objects.equals(map.put(pattern.charAt(i), i), ma
p.put(strs[i], i)))
                return false;
        }
        return true;
    }
}
```

Problem II

Given a `pattern` and a string `str`, find if `str` follows the same pattern. Here follow means a full match, such that there is a bijection between a letter in `pattern` and a non-empty substring in `str`.

Examples

pattern = `"abab"` , str = `"redblueredblue"` should return `true` .

pattern = `"aaaa"` , str = `"asdasdasdasd"` should return `true` .

pattern = `"aabb"` , str = `"xyzabcxzyabc"` should return `false` .

Notes

You may assume both pattern and str contains only lowercase letters.

Think

- Got all combinations and check any combination matched the pattern

Solution

```
public boolean wordPatternMatch(String pattern, String str) {
        // get all combinations
        List<String> combinations = new ArrayList<>();
        backtracking(combinations, str, "", 0);

        // check any matched case
        boolean res = false;
        for (String s : combinations) {
            res |= wordPattern(pattern, s);
        }
        return res;
    }

    private void backtracking(List<String> combinations, String
str,
            String cur, int index) {
        if (index >= str.length()) {
            combinations.add(new String(cur));
            return;
        }

        for (int i = index; i < str.length(); i++) {
            String origin = cur;
            backtracking(combinations, str, cur + (cur.length()
== 0 ? "" : " ")
                    + str.substring(index, i + 1), i + 1);
            cur = origin;
        }
    }
```

Longest Palindromic Substring

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.

Example

Given the string = `"abcdzdcab"`, return `"cdzdc"`.

Challenge

O(n2) time is acceptable. Can you do it in O(n) time.

Think

- One pass on each position in string characters, assume the axis of symmetry on each character or between two character:

```
    |                        |
c a b a c f     or     c   a   b   a   c   f
```

- On each axis of symmetry we get the left and right pointers and make them move toward left or right

```
left right
   \ /
c a b a c f

...or...

    l r
    | |
c a b a c f
```

Solution

```
public class Solution {
    /**
     * @param s input string
     * @return the longest palindromic substring
     */
    public String longestPalindrome(String s) {
        if(s == null || s.length() == 0)
            return "";

        String longest = "";
        int maxLength = 0;
        for(int i = 0; i < s.length()*2 - 1; i++){
            int l = i / 2;
            int r = i / 2;
            if(i % 2 != 0)
                r++;
            String cur = palindrome(s, l , r);
            if(cur.length() > maxLength) {
                longest = cur;
                maxLength = cur.length();
            }
        }
        return longest;
    }

    private String palindrome(String s, int l, int r) {
        while(l>=0 && r < s.length() && s.charAt(l) == s.charAt(
r)) {
            l--; r++;
        }
        return s.substring(l + 1, r);
    }
}
```

Analysis

One pass from 0 - ; inside the loop the palindromic substring search also costs a loop from l - r; So the total is O(n^2);

Shortest Word Distance

Problem I

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

Example

Assume that words = `["practice", "makes", "perfect", "coding", "makes"]` . Given word1 = `“coding”` , word2 = `“practice”` , return `3` . Given word1 = `"makes"` , word2 = `"coding"` , return `1` .

Note

You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

Think

- The problem can be solved by one-pass of the array.
- Iterate through the array, use two pointers pointing to the index of the word1 and word2, maintain the minimum distance.

Solution

```
     public int shortestDistance(String[] words, String word1, S
tring word2) {
         if(words == null)
            throw new IllegalArgumentException("Invalid Input!")
;
         int w1Idx = -1, w2Idx = -1;
         int min = Integer.MAX_VALUE;
         for(int i = 0; i < words.length; i++) {
             if(words[i].equals(word1))
                 w1Idx = i;
             else if(words[i].equals(word2))
                 w2Idx = i;
             if(w1Idx!=-1 && w2Idx!=-1){
                 min = Math.min(min, Math.abs(w2Idx-w1Idx));
             }
         }
         return min;
     }
```

Problem II

This is a follow up of Problem I.The only difference is now you are given the list of words and your method will be called repeatedly many times with different parameters. How would you optimize it? Design a class which receives a list of words in the constructor, and implements a method that takes two words word1 and word2 and return the shortest distance between these two words in the list.

Example,

Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`. Given word1 = `"coding"`, word2 = `"practice"`, return `3`. Given word1 = `"makes"`, word2 = `"coding"`, return `1`.

Think

- Since the calls are from different words, we have to save the index for each word. So HashMap is a good choice.

- Save the word and its indexes as key and value in constructor.

Solution

```java
public class ShortestWordDistance {
    // here is thread safe, since only constructor contains writ
ing method
    private final HashMap<String, List<Integer>> map;

    public ShortestWordDistance(String[] words) {
        map = new HashMap<>();
        for (int i = 0; i < words.length; i++) {
            List<Integer> list = new ArrayList<>();
            if (map.containsKey(words[i]))
                list = map.get(words[i]);
            list.add(i);
            map.put(words[i], list);
        }
    }

    public int shortestDistanceII(String word1, String word2) {
        int min = Integer.MAX_VALUE;
        if (!map.containsKey(word1) || !map.containsKey(word2))
            return min;
        for (int i : map.get(word1)) {
            for (int j : map.get(word2)) {
                min = Math.min(min, Math.abs(i - j));
            }
        }
        return min;
    }
}
```

Problem III

This is a follow up of Problem I. The only difference is **now word1 could be the same as word2**. Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list. word1 and word2 may be the same and they represent two individual words in the list.

Example,

Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`. Given word1 = `"makes"`, word2 = `"coding"`, return `1`. Given word1 = `"makes"`, word2 = `"makes"`, return `3`.

Think

- Most code should remain the same as the Problem I. But need to deal with the situation that word1 and word2 are the same.
- `w1Idx` always record the index when `word[i].equals(word1)` but `w2Idx` should be assigned as the value from `w1Idx` when `word1 == word2`

Solution

```
    public int shortestDistance(String[] words, String word1, S
tring word2) {
        if(words == null)
            throw new IllegalArgumentException("Invalid Input!")
;
        int w1Idx = -1, w2Idx = -1;
        int min = Integer.MAX_VALUE;
        for(int i = 0; i < words.length; i++) {
            if(words[i].equals(word1))
                w1Idx = i;
            else if(words[i].equals(word2)) // else if to avoid
 w2Idx be recorded whrn word1==word2
                w2Idx = i;
            if(w1Idx!=-1 && w2Idx!=-1 && w1Idx != w2Idx){
                min = Math.min(min, Math.abs(w2Idx-w1Idx));
            }
            if(word2.equals(word1))
               w2Idx = w1Idx; // update previous index record
        }
        return min;
    }
```

Group Shifted Strings

Given a string, we can “shift” each of its letter to its successive letter, for example: “abc” -> “bcd”. We can keep “shifting” which forms the sequence:

"abc" -> "bcd" -> ... -> "xyz" Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.

Example,

given: ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"], Return:

```
[
  ["abc","bcd","xyz"],
  ["az","ba"],
  ["acef"],
  ["a","z"]
]
```

Note

For the return value, each inner list’s elements must follow the lexicographic order.

Think

- There is a regular pattern: if two words are shifted, the distance between two adjacent characters are the same.

```
    2   3         2   3
    /\ /\         /\ /\
   a  c  f - >  e  g  j
```

- So all we need to do is figure out the distance between each character in each word and make these distance as a code to represent the word.

Solution

```
public class GroupShiftedString {
    public List<List<String>> groupStrings(String[] strings) {
        List<List<String>> res = new ArrayList<>();
        HashMap<String, List<String>> map = new HashMap<>();
        for (String str : strings) {
            String code = getCode(str);
            List<String> val;
            if (!map.containsKey(code)) {
                val = new ArrayList<>();
            } else {
                val = map.get(code);
            }
            val.add(str);
            map.put(code, val);
        }

        for (Map.Entry<String, List<String>> entry : map.entrySe
t()) {
            List<String> val = entry.getValue();
            Collections.sort(val);
            res.add(val);
        }
        return res;
    }

    private String getCode(String s) {
        StringBuilder sb = new StringBuilder();
        sb.append("#");
        for (int i = 1; i < s.length(); i++) {
            int tmp = ((s.charAt(i) - s.charAt(i - 1)) + 26) % 26
;
            sb.append(tmp).append("#");
        }
        return sb.toString();
    }
}
```

Encode String

Given a String like `"ABBCCCC"` , encode it to `A2B4C` . Avoid to make the encode string larger than original string. Do it in-place! The repeat count should less than 10.

Think

- Pass the char array by reversed order so that it's easier to modify the char to count number in-place
- Index passing should from `len - 2` to `-1` , this is tricky part to avoid the case when index is zero cannot be count
- The in-place modification index come from the `len - 1` , rewrite the result char by this index
- For the integer convert to character, it may need to use `Character.forDigit(count, 10)`
- For those rest char less than rewrite index, set them as null character `"\u0000"`

Solution

```
class Solution {
  public static void main(String[] args) {
    String str = "ABC";
    inplaceEncode(str.toCharArray());
  }

  public static void inplaceEncode(char[] chars){
    int idx = chars.length - 1;
    int count = 1;
    for(int i = chars.length - 2; i >= -1; i--) {
      if(i>=0 && chars[i] == chars[i+1])
        count++;
      else {
        char cur = chars[i+1];
        if(count > 1){
          chars[idx--] = cur;
          chars[idx--] = Character.forDigit(count, 10);
        }else
          chars[idx--] = cur;
        count = 1;
      }
    }

    // place the rest char position as null char
    while(idx>=0)
      chars[idx--] = '\u0000';
  }
}
```

Encode and Decode Strings

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:

```
public String encode(String[] strs) {
  // ... your code
  return encoded_string;
}
```

Machine 2 (receiver) has the function:

```
public String[] decode(String s) {
  //... your code
  return strs;
}
```

So Machine 1 does:

```
String encoded_string = encode(strs);
```

and Machine 2 does:

```
String[] strs2 = decode(encoded_string);
```

`strs2` in Machine 2 should be the same as `strs` in Machine 1.

Implement the encode and decode methods.

Note

The string may contain any possible characters out of 256 valid ascii characters. Your algorithm should be generalized enough to work on any possible characters.

Do not use class **member / global / static variables** to store states. Your encode and decode algorithms should be **stateless**.

Do not rely on any library method such as eval or serialize methods. You should implement your own encode/decode algorithm.

Think

- In theoretic way, there should be nothing can do separation for Strings
- So just make the encode as adding the length of word and "#" before the word
- Decode function should be carefully designed

Solution

```java
public class EncodeDecodeString {
    public String encode(List<String> strs) {
        StringBuilder sb = new StringBuilder();
        for (String str : strs) {
            sb.append(str.length());
            sb.append("#");
            sb.append(str);
        }

        return sb.toString();
    }

    public List<String> decode(String str) {
        List<String> list = new ArrayList<>();
        int strlen = 0;
        for (int i = 0; i < str.length(); i++) {
            char cur = str.charAt(i);
            if (cur == '#' && strlen > 0) {
                StringBuilder sb = new StringBuilder();
                while (strlen > 0 && i < str.length()) {
                    sb.append(str.charAt(++i));
                    strlen--;
                }
                list.add(sb.toString());
            } else
                strlen = strlen * 10 + (cur - '0');
        }
        return list;
    }
}
```

Palindrome Permutation

Problem I

Given a string, determine if a permutation of the string could form a palindrome.

Example

`"code"` -> `false` , `"aab"` -> `true` , `"carerac"` -> `true` .

Think

The problem can be easily solved by count the frequency of each character using a hash map. The only thing need to take special care is consider the length of the string to be even or odd.

- If the length is even. Each character should appear exactly times of 2, e.g. 2, 4, 6, etc..
- If the length is odd. One and only one character could appear odd times.

Solution

```
    public boolean canPermutePalindrome(String s) {
        if (s == null || s.length() == 0)
            return true;
        HashMap<Character, Integer> map = new HashMap<>();
        for (char c : s.toCharArray())
            map.put(c, map.containsKey(c) ? map.get(c) + 1 : 1);

        int tolerent = 0;
        for (Map.Entry<Character, Integer> entry : map.entrySet(
)) {
            if (entry.getValue() % 2 != 0) {
                tolerent++;
            }
        }
        if (s.length() % 2 != 0)
            return tolerent == 1;
        else
            return tolerent == 0;
    }
```

Problem II

Given a string s, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.

Example

Given s = `"aabb"` , return `["abba", "baab"]` . Given s = `"abc"` , return `[]` .

Think

- Similar with the last problem, check if the input String can form any valid palindrome
- Address the case when the length is odd
  - Record the character with odd frequency

  - Initialize the generation String with the Odd character
- Backtracking to generate the symmetry characters on the generation String

Solution

```
    public static List<String> generatePalindromes(String s) {
        HashSet<String> res = new HashSet<>();
        if (s == null || s.length() == 0)
            return new ArrayList<String>(res);
        HashMap<Character, Integer> map = new HashMap<>();
        for (char c : s.toCharArray())
            map.put(c, map.containsKey(c) ? map.get(c) + 1 : 1);

        // check if it is odd length, increase the tolerance whe
n it is odd length
        int tolerent = 0;
        if (s.length() % 2 != 0)
            tolerent++;

        // record the odd item to set as the base of generate St
ring
        char odd = '\u0000';
        for (Map.Entry<Character, Integer> entry : map.entrySet(
)) {
            if (entry.getValue() % 2 != 0) {
                if (tolerent > 0) {
                    tolerent--;
                    odd = entry.getKey(); // set it
                } else
                    return new ArrayList<String>(res);
            }
        }
        // set the base String when the odd case
        String cur = "";
        if (odd != '\u0000') {
            map.put(odd, map.get(odd) - 1);
            if (map.get(odd) == 0)
                map.remove(odd);
            cur = "" + odd;
        }

        // generate the palindrome
        helper(res, map, cur, s);
        return new ArrayList<String>(res);
    }

    private static void helper(Set<String> res,
            HashMap<Character, Integer> map, String cur, String
origin) {
        if (map.size() == 0) {
            res.add(new String(cur));
            return;
        }

        for (int i = 0; i < origin.length(); i++) {
            char c = origin.charAt(i);
            if (!map.containsKey(c))
                continue;
            cur = (c + cur + c);
            map.put(c, map.get(c) - 2);
            if (map.get(c) == 0)
                map.remove(c);
            helper(res, map, cur, origin);
            cur = cur.substring(1, cur.length() - 1);
            map.put(c, map.containsKey(c) ? map.get(c) + 2 : 2);
        }
    }
```

Sort

Sorting is ordering a list of objects. We can distinguish two types of sorting. If the number of objects is small enough to fits into the main memory, sorting is called internal sorting. If the number of objects is so large that some of them reside on external storage during the sort, it is called external sorting. In this chapter we consider the following internal sorting algorithms

By Complexity

- Time Complexity: $O(N^2)$
  - Bubble Sort
  - Selection Sort
  - Insertion Sort (min: $O(N)$)
- Time Complexity: $O(Nlog_2N)$
  - Quick Sort (Space Complexity: $O(Nlog_2N)$)
  - Merge Sort (Space Complexity: $O(N)$)
  - Heap Sort
- Time Complexity: $O(N)$ && Space Complexity: $O(N)$
  - Bucket sort

By Stable

- Stable
  - Insertion sort
  - Merge Sort
- Not Stable
  - Bubble Sort
  - Selection Sort
  - Quick Sort
  - Merge Sort
  - Heap Sort

Elementary Sort

These are most basic sort methods with $O(N^2)$ time complexity.

Bubble Sort

Think

- Swap when the two adjacent elements is not in order
- Do a while loop until swap not happened in previous element order check

Solution

```
public void bubbleSort(int[] arr){
    boolean swap;
    do{
        swap = false;
        for(int i = 0; i < arr.length - 1; i++){
            if(arr[i] > arr[i + 1]){
                int tmp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = tmp;
                swap = true;
            }
        }
    }while(swap);
}
```

Insertion Sort

Think

- Get the target value from head of array

- Find its suitable position from 0 to its position

Solution

```
public void insertionSort(int[] arr){
    for(int i = 1; i < arr.length; i++){
        for(int j = 0; j < i; j++) {
            if(arr[j] > arr[i]) {
                int tmp = arr[j];
                arr[j] = arr[i];
                arr[i] = tmp;
            }
        }
    }
}
```

Selection Sort

Think

- Get the target value from head of array
- Find the minimum element on the right of target, swap them if minimum less then target

Solution

```
public void selectionSort(int[] arr){
    for(int i = 0; i < arr.length - 1; i++){
        int min = i;
        for(int j = i + 1; j < arr.length; j++) {
            if(arr[min] > arr[j])
                min = j;
        }
        if(i!=min) {
            int tmp = arr[min];
            arr[min] = arr[i];
            arr[i] = tmp;
        }
    }
}
```

Advanced Sort

Merge Sort

Think

- Divide and Conquer
- Recursion

Solution

```
public void mergeSort(int[] arr, int left, int right){
    if(left >= right)
        return;
    int mid = left + ((right - left) >> 1);
    mergeSort(arr, left, mid - 1);
    mergeSort(arr, mid, right);
    merge(arr, left, mid, right);
}

private void merge(int[] arr, int left, int mid, int right){
    int[] newarr = new int[right - left + 1];

    for(int i = left; i <= right; i++)
        newarr[i - left] = arr[i];

    int l = 0;
    int r = mid - left + 1;
    for(int i = left; i <= right; i++){
        if(l > mid - left)
            arr[i] = newarr[r++];
        else if(r > right - left)
            arr[i] = newarr[l++];
        else
            arr[i] = newarr[l] < newarr[r]? newarr[l++] : newarr
[r++];
    }
}
```

Quick Sort

Think

- Set pivot (the rear of array or) and consider it as a standard
- Pass the array and make the element less than pivot on the pivot's left and the element larger than pivot on the pivot's right;

Solution

```
public void quickSort(int[] arr, int l, int r){
    if(l > r)
        return;

    int pivot = pivot(arr, l, r);
    quickSort(arr, l, pivot - 1);
    quickSort(arr, pivot + 1, r);
}


private int pivot(int[] arr, int l, int r){
    int pivot = arr[r];
    int idx = l;
    for(int i = l; i < r; i++){
        if(arr[i] < pivot){
            int tmp = arr[idx];
            arr[idx++] = arr[i];
            arr[i] = tmp;
        }
    }
    arr[r] = arr[idx];
    arr[idx] = pivot;
    return idx;
}
```

Linear Sort

There are sorting algorithms that run faster than $O(nlogn)$ time but they require special assumptions about the input sequence to be sort. Examples of sorting algorithms that run in linear time $O(n)$, are counting sort, radix sort and bucket sort. Counting sort and radix sort assume that the input consists of integers in a small range. Whereas, bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval.

Since we already know that the best comparison-based sorting can to is $(nlogn)$. It is not difficult to figure out that linear-time sorting algorithms use operations other than comparisons to determine the sorted order.

Bucket Sort

Think

- Setup `n` buckets (`n` is the range of elements value)
- Put elements in each bucket
- Space consuming

Solution

```
public void bucketSort(int[] arr){
    // step 1: find range
    int min = Integer.MAX_VALUE;
    int max = Integer.MIN_VALUE;
    for(int i = 0; i < arr.length; i++) {
        min = Math.min(min, arr[i]);
        max = Math.max(max, arr[i]);
    }

    // step 2: setup bucket
    int[] bucket = new int[max - min + 1];
    for(int i = 0; i < arr.length; i++){
        bucket[arr[i] - min]++;
    }

    // step 3: output the sort
    int idx = 0;
    for(int i = 0; i < bucket.length; i++){
        while(bucket[i] > 0) {
            arr[idx++] = min + i;
            bucket[i]--;
        }
    }
}
```

Radix Sort

Think

Solution

Sort Color

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

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

Have you met this question in a real interview? Yes

Example

Given [1, 0, 1, 2], return [0, 1, 1, 2].

Note

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

Challenge

A rather straight forward solution is a two-pass algorithm using counting sort. First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

Think

- Index mark O's first position from head and 2's first position from the rear.
- Exchange the two values.

Solution

```
public void sortColors(int[] nums) {
    if(nums == null || nums.length == 0)
        return;
    int zero = 0;
    int two = nums.length - 1;
    for(int i = 0; i < nums.length; i++){
        if(i > zero && nums[i] == 0){
            int tmp = nums[zero];
            nums[zero++] = nums[i];
            nums[i--] = tmp;
        }else if(i < two && nums[i] == 2){
            int tmp = nums[two];
            nums[two--] = nums[i];
            nums[i--] = tmp;
        }
    }
}
```

Sort Color II

Given an array of n objects with k different colors (numbered from 1 to k), sort them so that objects of the same color are adjacent, with the colors in the order 1, 2, ... k.

Have you met this question in a real interview? Yes

Example

Given colors=[3, 2, 2, 1, 4], k=4, your code should sort colors in-place to [1, 2, 2, 3, 4].

Note

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

Challenge

A rather straight forward solution is a two-pass algorithm using counting sort. That will cost O(k) extra memory. Can you do it without using extra memory?

Think #1

- Bucket Sort but space complexity with $O(k)$

Solution #1

```
    /**
     * @param colors: A list of integer
     * @param k: An integer
     */
    public void sortColors2(int[] colors, int k) {
        if(colors == null || colors.length == 0)
            return;

        int[] bucket = new int[k];
        for(int i : colors)
            bucket[i - 1] ++;
        int idx = 0, bidx = 0;
        while(bidx < bucket.length) {
            while(bucket[bidx] > 0) {
                colors[idx++] = bidx+1;
                bucket[bidx]--;
            }
            bidx++;
        }
    }
```

Think #2

- Complex bucket sort with in-place counting
- Get a value and find its index by `value - 1`
- If the target index has another value, exchange and set target index as `-1`
- If target index is counter, make it minus 1, e.g. `-2` and set original index as `0`
- Steps like following:

```
  3   2   2   1   4

  2   2  -1   1   4

  2  -1  -1   1   4

  0  -2  -1   1   4

 -1  -2  -1   0   4

 -1  -2  -1  -1   0
```

- Get back the result by counter value from rear to head

Solution #2

```
    /**
     * @param colors: A list of integer
     * @param k: An integer
     * @return: nothing
     */
    public void sortColors2(int[] colors, int k) {
        if(colors == null || colors.length == 0)
            return;

        for(int i = 0; i < colors.length; i++){
            if(colors[i] <= 0)
                continue;
            else{
                int idx = colors[i] - 1;
                if(colors[idx] > 0){
                    colors[i--] = colors[idx];
                    colors[idx] = -1;
                }else{
                    colors[i] = 0;
                    colors[idx]--;
                }
            }
        }

        int idx = colors.length - 1;
        for(int i = k - 1; i >= 0; i--){
            int cnt = -colors[i];
            while(cnt-- > 0 && idx >= 0) {
                colors[idx--] = (i+1);
            }
        }
    }
```

Largest Number

Given a list of non negative integers, arrange them such that they form the largest number.

For example, given `[3, 30, 34, 5, 9]`, the largest formed number is `9534330`.

Note

The result may be very large, so you need to return a string instead of an integer.

Think

- Sort the elements in array by the rule of pair combination:
  - a, b -> `compare(ab, ba)`;
- One pass the sorted array and make the largest number

Solution

```
public String largestNumber(int[] nums) {
    if(num==null || num.length==0)
        return "";

    // customized sort method only can sort the object
    // so here it should change to a String array
    String[] Snum = new String[num.length];
    for(int i=0;i<num.length;i++)
        Snum[i] = num[i]+"";

    Comparator<String> comp = new Comparator<String>(){
        @Override
        public int compare(String str1, String str2){
            String s1 = str1+str2;
            String s2 = str2+str1;
            return s1.compareTo(s2);
        }
    };

    Arrays.sort(Snum,comp);
    if(Snum[Snum.length-1].charAt(0)=='0')
        return "0";

    StringBuilder sb = new StringBuilder();

    for(String s: Snum)
        sb.insert(0, s);

    return sb.toString();
}
```

Wiggle Sort

Given an array, and re-arrange it to wiggle style in one pass.

Example

- [1] `A0 >= A1 <= A2 >= A3 .... .... An`
- [2] `A0 <= A1 >= A2 <= A3 .... .... An`

Think

Base case. The first two elements A_0, A_1 satisfy the rules, and A_0 is in its desired position.

Suppose A_0, ... A_k satisfy the rules, and A_0, ... A_{k-1} are in their desired positions. We want to show that when we consider the pair A_k and A_{k+1}, the rules are not violated, and the new k-th element will be in its desired position. Without loss of generality, assume that the k-th element should be higher than both of its neighbors. Two cases:

1) A_k > A_{k+1}. We are good in this case. A_k is its desired position, and no rules are violated so far.

2) A_k < A_{k+1}. We swap A_k and A_{k+1}. Note that this does not violate A_{k-1}, since A_{k-1} <A_k< A_{k+1}. And the new k-th element (previous A_{k+1}) satisfies the rules, and is in its desired position.

So throughout the process, we do not violate any rules. The algorithm is correct.

Solution

```
public void wiggleSort(int[] arr, boolean swither){
    int idx = 0;
    while(idx < arr.length - 1){
        if((switcher && arr[idx] < arr[idx + 1])||(!switcher &&
arr[idx] > arr[idx + 1])){
            int tmp = arr[idx];
            arr[idx] = arr[idx + 1];
            arr[idx+1] = tmp;
            switcher ^= true;
        }
      idx ++;
    }
}
```

Sort Letters by Case

Given a string which contains only letters. Sort it by lower case first and upper case second.

Example

For "abAcD", a reasonable answer is "acbAD"

Note

It's not necessary to keep the original order of lower-case letters and upper case letters.

Challenge

Do it in one-pass and in-place.

Solution

```
    /**
     *@param chars: The letter array you should sort by Case
     *@return: void
     */
    public void sortLetters(char[] chars) {
        if(chars == null || chars.length == 0)
            return;
        int lower = 0;
        for(int i = 0; i < chars.length; i++) {
            if(chars[i] - 'a' >= 0 && chars[i] - 'a' <= 26) {
                if(i > lower) {
                    char tmp = chars[i];
                    chars[i--] = chars[lower];
                    chars[lower] = tmp;
                }
                lower++;
            }
        }
    }
```

Given an unsorted array, find out the most closest two elements in this array.

Solution:

- Sort is important here! You must think about sort first. Since other may may cost
- Then the gap between adjacent elements are the

```
public ArrayList<int[]> closestNumber(int[] arr){
    ArrayList<int[]> res = new ArrayList<>();
    if(arr == null || arr.length == 0)
        return res;
    Arrays.sort(arr);

    int mindiff = Integer.MAX_VALUE;
    for(int i = 1; i < arr.length; i++) {
         int curdiff = arr[i] - arr[i - 1];
         if(curdiff >= mindiff) {
             if(curdiff > mindiff)
                 res.clear();
             int[] cres = new int[]{arr[i - 1], arr[i]};
             res.add(cres);
         }
    }
    return res;
}
```

3Sum Smaller

Given an array of n integers nums and a target, find the number of index triplets i, j, k with 0 <= i < j < k < n that satisfy the condition `nums[i]` + `nums[j]` + `nums[k]` < `target` .

Example

given nums = `[-2, 0, 1, 3]` , and target = `2` . Return 2. Because there are two triplets which sums are less than `2` :

```
[-2, 0, 1]
[-2, 0, 3]
```

Think

- Set iterate from rear.
- One tricky pattern:
    - When `nums[l] + nums[r] +nums[i] < target` , all for combinations like: `nums[l] + nums[r-1] +nums[i] < target` , `nums[l] + nums[r-2] +nums[i] < target` , ..., `nums[l] + nums[l+1] +nums[i] < target` are workable.
    - So result counter should directly add the `r - l`
- All in all, two cases:
    - `nums[l] + nums[r] +nums[i] < target` , add the cnt and move the `l`
    - `nums[l] + nums[r] +nums[i] >= target` , just move the `r`
- It is not very similar like binary search but still has kinda idea inside.

Solution

```
public static int threeSumSmaller(int[] nums, int target) {
        if (nums == null || nums.length < 3)
            return 0;
        Arrays.sort(nums);
        int resCnt = 0;
        for (int i = nums.length - 1; i > 1; i--) {
            int one = nums[i];
            int l = 0, r = i - 1;
            while (l < r) {
                if(one + nums[l] + nums[r] < target) {
                    resCnt += (r - l);
                    l++;
                }else
                    r--;
            }
        }
        return resCnt;
    }
```

Count Smaller Number

Give you an integer array (index from 0 to n-1, where n is the size of this array, value from 0 to 10000) and an query list. For each query, give you an integer, return the number of element in the array that are smaller that the given integer.

Example

For array `[1,2,7,8,5]`, and queries `[1,8,5]`, return `[0,4,2]`

Challenge

Could you use three ways to do it.

- Just loop
- Sort and binary search

Solution

1. Solution by Loop with O(n^2)

```
    /** O(n^2) Loop implement
      * @param A: An integer array
      * @return: The number of element in the array that
      *           are smaller that the given integer
      */
     public ArrayList<Integer> countOfSmallerNumber(int[] A, int[
] queries) {
         ArrayList<Integer> res = new ArrayList<>();
         if(A == null || queries == null)
             return res;

         for(int i = 0; i < queries.length; i++) {
             int cnt = 0;
             for(int j = 0; j < A.length; j++) {
                 if(A[j] < queries[i])
                     cnt++;
             }
             res.add(cnt);
         }
         return res;
     }
```

2. Solution by Sort and Binary search with O(nlogn)

```
    /** O(nlogn) Sort and Binary search
     * @param A: An integer array
     * @return: The number of element in the array that
     *          are smaller that the given integer
     */
    public ArrayList<Integer> countOfSmallerNumber(int[] A, int[
] queries) {
        ArrayList<Integer> res = new ArrayList<>();
        if(queries == null)
            return res;

        Arrays.sort(A);
        for(int i = 0; i < queries.length; i++) {
            int cnt = binarySearch(A, queries[i]);
            res.add(cnt);
        }
        return res;
    }

    private int binarySearch(int[] A, int value) {
        if(A == null || A.length == 0)
            return 0;
        int l = 0, r = A.length - 1;
        while(l + 1 < r) {
            int m = l + ((r - l) >> 1);
            if(value > A[m])
                l = m;
            else
                r = m;
        }
        if(A[l] < value)
            return l + 1;
        else
            return l;
    }
```

Triangle Count

Given an array of integers, how many three numbers can be found in the array, so that we can build an triangle whose three edges length is the three numbers that we find?

Example

Given array S = [3,4,6,7], return 3. They are:

```
[3,4,6]
[3,6,7]
[4,6,7]
```

Given array S = [4,4,4,4], return 4. They are:

```
[4(1),4(2),4(3)]
[4(1),4(2),4(4)]
[4(1),4(3),4(4)]
[4(2),4(3),4(4)]
```

Think

- Sort
- Binary Search
- But how to define driven condition? (Tricky Part)
  - As we know, triangle is made by `i + j > k` ;
  - So we capture the largest one `[k]` (passing from `length - 1` to `2`)
  - Get `left = 0` and `right = k - 1`
  - If `[left] + [right] > [k]` , that means in the segment, `[left]` can be valued between `[left]` and `[right-1]` , all of that can make valid triangle. So `count += right - left` !
  - If `[left] + [right] <= [k]` , just make the `left` increase to detect any valid possibility.

Solution

```java
public class Solution {
    /**
     * @param S: A list of integers
     * @return: An integer
     */
    public int triangleCount(int S[]) {
        if(S == null || S.length == 0)
            return 0;
        int cnt = 0;
        Arrays.sort(S);
        for(int i = S.length - 1; i >= 2; i--) {
            int cur = S[i];
            int l = 0, r = i - 1;
            while(l < r) {
                if(S[l] + S[r] > S[i]) {
                    cnt += (r - l); // keypoint!
                    r--;
                }else
                    l++;
            }
        }
        return cnt;
    }
}
```

Recursion

How to understand a recursion?

Recursion is a function to achieve the result with calling itself. For example, when doing the factorial, . Sometimes we can image it as a tree, a decision tree.

Space

Recursion is consuming the stack space in JVM or other programming stack. So we care how deep the recursion functions are going on and we need to set a smart terminate condition for recursion.

Time

Pruning! Pruning is important when doing a complex recursion function. We can avoid some unnecessary recursion by terminate it with smart condition.

Print Numbers by Recursion

Print numbers from 1 to the largest number with `N` digits by recursion.

Example

Given `N = 1`, return `[1,2,3,4,5,6,7,8,9]`.

Given `N = 2`, return `[1,2,3,4,5,6,7,8,9,10,11,12,...,99]`.

Note

It's pretty easy to do recursion like:

```
recursion(i) {
    if i > largest number:
        return
    results.add(i)
    recursion(i + 1)
}
```

However this cost a lot of recursion memory as the recursion depth maybe very large ($10^n - 1$). Can you do it in another way to recursive with at most N depth?

Challenge

Do it in recursion, not for-loop.

Think

- Think from bottom to top.
- Build the result list from number with one digits to N digits.
- Since we considering with digits as its deep, we have to set a loop to add the number in list on the new-generated base number (1 - 9 with following digits):

```
when zero digit, none;
when one digit, new-generated base number is 1, add 1,2,...9;
when two digit, new-generated base number is 10, add 10,20,...90
;
```

- Each time when having the new-generated base number, we need to pass through the original result list to fill the rest of number with beginning as new-generated base number.

```
when one digit, new-generated base number is 1, add 1,2,...9, bu
t original result list has nothing. so just add itself;
when two digit, new-generated base number is 10, when add 10, go
 back the original result list with "1, 2, 3,..., 9", add them a
s 11, 12, 13, ..., 19, when add 20, go back the original result
list with "1, 2, 3,..., 9", add them as 21, 22, 23, ..., 29;
so the same as for 30,..., 90, 100, ..., 900, ...
```

Solution

```java
public class Solution {
    /**
     * @param n: An integer.
     * return : An array storing 1 to the largest number with n
digits.
     */
    public List<Integer> numbersByRecursion(int n) {
        List<Integer> res = new ArrayList<>();
        if(n >= 0)
            add(res, n);
        return res;
    }

    private int add(List<Integer> res, int n){
        if(n == 0)
            return 1;

        int cur = add(res, n - 1);
        int size = res.size();
        for(int i = 1; i <= 9; i ++) {
            int digit = i * cur;
            res.add(digit);
            for(int j = 0; j < size; j++) {
                res.add(digit + res.get(j));
            }
        }
        return cur * 10;
    }
}
```

Complexity Analysis

Time: $O(10^n - 1)$

Space: $O(n)$

Permutation Index

Given a permutation which contains no repeated number, find its index in all the permutations of these numbers, which are ordered in lexicographical order. The index begins at 1.

Example

Given `[1,2,4]`, return `1`.

Thinking

Illustrating by manually getting the index of {2, 4, 3, 1}. Since this is a 4-element set, we know there are 4! permutations (4! = 4*321). If the set only had 3 elements, we would have 321 permutations. If the set only had 2 elements, we would have 2!=2*1 permutations; and so on.

ASIDE: The decimal system of counting is a positional system. A 3-element decimal number, for instance, has the following three positional weights: hundred, ten, unit. Hence, we know the value of the number 472 because we understand: 4*hundred* + 7ten + 2*unit.

If we treat our 4-element set as a positional system, then we get the following positional weights: 3!, 2!, 1!, 0. So that the index of {2, 4, 3, 1} is: x*3!+y*2!+z*1!+w*0. Presently it suffices to find the values of x,y,z to calculate the index (we ignore w because it is paired with 0). x,y,z are counters: the number of succeeding elements less than the element being considered. For example, in {2, 4, 3, 1}, there are two succeeding elements less than 4 (namely 3 and 1). For 2 it's 1 (1); for 4 it's 2 (3 and 1); for 3 it's 1 (1); for 1 it's 0. Now we can calculate the index of {2, 4, 3, 1} as: x=1, y=2, z=1: `x*3!+y*2!+z*1!+w*0 = 1*3! + 2*2! + 1*1! = 6 + 4 + 1 = 11`.

> The key is counting from low digit(right) to higher digit(left), and checking how many digits are less than current digits. Then using that count as the coefficient with positional weight.

Solution

```
public class Solution {
    /**
     * @param A an integer array
     * @return a long integer
     */
    public long permutationIndex(int[] A) {

        if(A == null || A.length == 0)
            return 0L;

        int pos = 2;
        long factor = 1;
        long index = 1;
        for(int i = A.length - 2; i >= 0; i--) {
            int cnt = 0;
            for(int j = i + 1; j < A.length; j++) {
                if(A[i] > A[j])
                    cnt++;
            }
            index += (cnt*factor);
            factor *= pos;
            pos++;
        }

        return index;
    }
}
```

Complexity Analysis

Two loop: `i range 0 -> length - 2` and `j range i + 1 -> length - 1` , So it is `O(n^2)` ; Constant Space with some integer variable, Space: `O(1)` ;

Permutation Index II

Given a permutation which may contain repeated numbers, find its index in all the permutations of these numbers, which are ordered in lexicographical order. The index begins at 1.

Example

Given the permutation `[1, 4, 2, 2]` , return `3` .

Think

The key is counting from low digit(right) to higher digit(left), and checking how many digits are less than current digits. Then using that count as the coefficient with positional weight. However, there are duplicates occured. So that means we can use a hash map to do the count. But the positional system should be modified. The multiple of the factorial of the duplicates occurence should be divided by original position system. That means the `entry.value` need to to the factorial and multiply those factors. Why? For example, n numbers with 2 duplicates, like `2,4,3,3` , when `4`

Solution

```
public class Solution {
    /**
     * @param A an integer array
     * @return a long integer
     */
    public long permutationIndexII(int[] A) {
        if(A == null || A.length == 0)
            return 0L;

        int pos = 2;
        long factor = 1;
        long index = 1;
        for(int i = A.length - 2; i >= 0; i--) {
            HashMap<Integer, Integer> map = new HashMap<>();
            int cnt = 0;
            // count itself
            map.put(A[i], 1);
            for(int j = i + 1; j < A.length; j++) {
                // count all occurence on following element in A
rray
                map.put(A[j], map.containsKey(A[j])?map.get(A[j]
)+1:1);
                if(A[i] > A[j])
                    cnt++;
            }
            index += (cnt*factor)/factorialMultiple(map);
            factor *= pos;
            pos++;
        }

        return index;
    }

    private int factorialMultiple(HashMap<Integer, Integer> map)
{
        int res = 1;
        for(int value : map.values()) {
            // do the factor on occurence
            int factor = 1;
            for(int i = 1; i <= value; i++)
                factor*= i;
            // get the multiple of occurence factor
            res *= factor;
        }
        return res;
    }
}
```

Permutation Sequence

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

Example

For n = 3, all permutations are listed as follows:

```
"123"
"132"
"213"
"231"
"312"
"321"
```

If k = `4`, the fourth permutation is "`231`"

Note

n will be between 1 and 9 inclusive.

Challenge

O(n*k) in time complexity is easy, can you do it in O(n^2) or less?

Think

a1,a2,a3.....an的permutation 如果确定了a1,那么剩下的permutation就有(n-1)!种 所以 a1 = k / (n-1)! k2 = k % (n-1)! a2 = k2 / (n-2)!

要注意的是

- 得到的应该是剩下选择数字的index,而不是value,所以要建一个存储可用数字的list
- 在用完一个数字后要将它从list中删去

- array是0-based index, 那么K也应该减去1变为0-based的 (`k--`)

Solution

```
class Solution {
    /**
      * @param n: n
      * @param k: the kth permutation
      * @return: return the k-th permutation
      */
    public String getPermutation(int n, int k) {

        int[] factors = new int[n + 1];
        List<Integer> list = new ArrayList<>();
        factors[0] = 1;
        for(int i = 1; i <= n; i++) {
            factors[i] = i * factors[i-1];
            list.add(i);
        }

        StringBuilder sb = new StringBuilder();
        // for index alignment： e.g: k == 12, k / 6 = 2,
        // however, it should still belong the number start for
list.get(1); So here need to make a alignment
        k--;
        while(n > 1) {
            int index = k / factors[n-1];
            sb.append(list.remove(index));
            k %= factors[n-1];
            n--;
        }
        sb.append(list.get(0));
        return sb.toString();
    }
}
```

Next Permutation

Given a list of integers, which denote a permutation.

Find the next permutation in ascending order.

Example

For `[1,3,2,3]` , the next permutation is `[1,3,3,2]` For `[4,3,2,1]` , the next permutation is `[1,2,3,4]`

Note

The list may contains duplicate integers.

Think

字典序算法：

- 从后往前寻找索引满足 a[k] < a[k + 1], 如果此条件不满足，则说明已遍历到最后一个。
- 如 k == 0, 说明是字典序最后一位， 因此直接倒置整个数组即可.
- 从后往前遍历，找到第一个比a[k]大的数a[l], 即a[k] < a[l].
- 交换a[k]与a[l].
- 反转k + 1 ~ n之间的元素.

Solution

```
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: return nothing (void), do not return anything, m
odify nums in-place instead
     */
    public int[] nextPermutation(int[] nums) {
        if(nums == null || nums.length == 1)
            return nums;

        // step1: find nums[i] < nums[i + 1]
        int i = 0;
        for (i = nums.length - 2; i >= 0; i--) {
            if (nums[i] < nums[i + 1]) {
                break;
            } else if (i == 0) {
                // reverse nums if reach maximum
                reverse(nums, 0, nums.length - 1);
                return nums;
            }
        }
        // step2: find nums[i] < nums[j]
        int j = 0;
        for (j = nums.length - 1; j > i; j--) {
            if (nums[i] < nums[j]) {
                break;
            }
        }
        // step3: swap betwenn nums[i] and nums[j]
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;

        // step4: reverse between [i + 1, n - 1]
        reverse(nums, i + 1, nums.length - 1);

        return nums;
    }

    private void reverse(int[] nums, int l, int r) {
        while(l < r) {
            int tmp = nums[l];
            nums[l++] = nums[r];
            nums[r--] = tmp;
        }
    }
}
```

Complexity Analysis

At most, two pass o(2*n) -> O(n); Constant Space Complexity: O(1)

Previous Permutation

Given a list of integers, which denote a permutation.

Find the previous permutation in ascending order.

Example

For `[1,3,2,3]` , the previous permutation is `[1,2,3,3]`

For `[1,2,3,4]` , the previous permutation is [`4,3,2,1]`

Note

The list may contains duplicate integers.

Think

- step 1: find last nums[k] > nums[k + 1];
- step 2: find last nums[i] < nums[k];
- step 3: swap i, j;
- step 4: reverse num after i;

Solution

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A list of integers that's previous permuation
     */
    public ArrayList<Integer> previousPermuation(ArrayList<Integ
er> nums) {
        if(nums == null || nums.size() <= 1)
            return nums;
        // step 1: find last nums[k] > nums[k + 1]
        int i = nums.size() - 2;
        for (; i >= 0; i--) {
            if (nums.get(i) > nums.get(i + 1)) {
                break;
            }
            if(i <= 0) {
                reverse(nums, 0, nums.size() - 1);
                return nums;
            }
        }

        // step 2: find last nums[i] > nums[k]
        int j = nums.size() - 1;
        for (; j > i; j--) {
            if (nums.get(i) > nums.get(j)) {
                break;
            }
        }

        // step 3: swap i, j
        Collections.swap(nums, i, j);

        // step 4: reverse num after i
        reverse(nums, i + 1, nums.size() - 1);

        return nums;
    }

    private void reverse(ArrayList<Integer> nums,  int start, int
  end) {
        for (int i = start, j = end; i < j; i++, j--) {
            Collections.swap(nums, i, j);
        }
    }
}
```

K Sum II

Given n unique integers, number k (1<=k<=n) and target. Find all possible k integers where their sum is target.

Example

Given [1,2,3,4], k=2, target=5, [1,4] and [2,3] are possible solutions.

Think

Unlike the k Sum I, here we need to get the each solution which can achieve the `target` within `k` values. Since the solution should be shown in the result, the dynamic programming cannot be used. Thus, the backtracking should be the only way.

Solution

```
public class Solution {
    /**
     * @param A: an integer array.
     * @param k: a positive integer (k <= length(A))
     * @param target: a integer
     * @return a list of lists of integer
     */
    public ArrayList<ArrayList<Integer>> kSumII(int A[], int k,
int target) {
        ArrayList<ArrayList<Integer>> res = new ArrayList<>();
        helper(res, new ArrayList<Integer>(), A , k, target, 0);
        return res;
    }

    private void helper(ArrayList<ArrayList<Integer>> res, Array
List<Integer> cur, int[] A, int k, int target, int idx) {
        if(target < 0 || k < 0)
            return;

        if(target == 0 && k == 0) {
            res.add(new ArrayList<>(cur));
            return;
        }

        for(int i = idx; i < A.length; i++) {
            cur.add(A[i]);
            helper(res, cur, A, k - 1, target - A[i], i + 1);
            cur.remove(cur.size() - 1);
        }
    }
}
```

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

Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.

Example

There exist two distinct solutions to the 4-queens puzzle:

```
[

    [".Q..", // Solution 1

     "...Q",

     "Q...",

     "..Q."],

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

     "Q...",

     "...Q",

     ".Q.."]

]
```

Think

- To save the space, use one dimension integer array to represent board: index -> num, value -> col;

```
  . . Q . .
  . . . . Q
  Q . . . .  ----->>>> {2, 4, 0, 3, 1}
  . . . Q .
  . Q . . .
```

- Backtracking each possible position of queen and check if that is reasonable

Solution

```
class Solution {
    /**
     * Get all distinct N-Queen solutions
     * @param n: The number of queens
     * @return: All distinct solutions
     * For example, A string '...Q' shows a queen on forth posit
ion
     */
    ArrayList<ArrayList<String>> solveNQueens(int n) {
        ArrayList<ArrayList<String>> res = new ArrayList<>();
        int[] board = new int[n];
        recursion(res, board, 0);
        return res;
    }


    private void recursion(ArrayList<ArrayList<String>> res, int
[] board, int row) {
        if(row == board.length) {
            // encode the board from 1-d array to string list
            ArrayList<String> cur = new ArrayList<>();
            for(int i = 0 ; i < board.length; i++){
                StringBuilder sb = new StringBuilder();
                for(int j = 0 ; j < board.length; j++) {
                    if(j == board[i])
                        sb.append("Q");
                    else
                        sb.append(".");
                }
                cur.add(sb.toString());
            }
            res.add(cur);
            return;
        }
        // recursion
        for(int i = 0; i < board.length; i++) {
            board[row] = i;
            if(isSafe(board, row))
                recursion(res, board, row+1);
        }

    }

    /**
    * Check the current board is safe.
    */
    private boolean isSafe(int[] board, int row) {
        for(int i = 0; i < row; i++) {
            // difference on col value shouldn't equal to the di
fference on row value;
            if((board[i]==board[row]) || (Math.abs(i - row) == M
ath.abs(board[i] - board[row])))
                return false;
        }
        return true;
    }
};
```

N Queen II

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.

Solution

```
class Solution {
    /**
     * Calculate the total number of distinct N-Queen solutions.
     * @param n: The number of queens.
     * @return: The total number of distinct solutions.
     */
    private int solutions = 0;
    public int totalNQueens(int n) {
        //write your code here
        int[] board = new int[n];
        recursion(board, 0);
        return solutions;
    }

    private void recursion(int[] board, int row) {
        // when valid solution found
        if(row == board.length) {
            solutions++;
            return;
        }
        // recursion
        for(int i = 0; i < board.length; i++) {
            board[row] = i;
            if(isSafe(board, row))
                recursion(board, row+1);
        }

    }

    private boolean isSafe(int[] board, int row) {
        for(int i = 0; i < row; i++) {
            if((board[i]==board[row]) || (Math.abs(i - row) == M
ath.abs(board[i] - board[row])))
                return false;
        }
        return true;
    }
};
```

Subset

Given a set of distinct integers, return all possible subsets.

Example

If S = `[1,2,3]` , a solution is:

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

Note

Elements in a subset must be in non-descending order.

For Question I: The solution set must not contain duplicate subsets. For Question II: The solution set may contain duplicate subsets.

Solution

```
// Question I :
public class Solution {

    /**
    * Recursive method, backtracking with a boolean array
    **/
    public ArrayList<ArrayList<Integer>> subsets(int[] num) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<Arr
ayList<Integer>>();
        if(num == null || num.length == 0)
            return result;
        ArrayList<Integer> list = new ArrayList<Integer>();
        Arrays.sort(num);
        subsetsHelper(result, list, num, 0);
        return result;
    }

    private void subsetsHelper(ArrayList<ArrayList<Integer>> res
ult, ArrayList<Integer> list, int[] num, int pos) {
        result.add(new ArrayList<Integer>(list));
        for (int i = pos; i < num.length; i++) {
            list.add(num[i]);
            subsetsHelper(result, list, num, i + 1);
            list.remove(list.size() - 1);
        }
    }

    /**
    * Iterate method, a loop and retreve the value in original l
ist and generate new value with insert it
    **/
    public ArrayList<ArrayList<Integer>> subsets(int[] num) {
        ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayL
ist<Integer>>();
        if(num == null || num.length == 0)
            return res;
        ArrayList<Integer> list = new ArrayList<Integer>();
        Arrays.sort(num);
        res.add(list);
        for(int i = 0; i < num.length; i++) {
            int size = res.size();
            for(int j = 0; j < size; j++) {
                list = new ArrayList<>(res.get(j));
                list.add(num[i]);
                res.add(list);
            }
        }
        return res;
    }
}


// Question II:
public class Solution {
    public ArrayList<ArrayList<Integer>> subsetsWithDup(ArrayLis
t<Integer> num) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<Arr
ayList<Integer>>();
        if(num == null || num.size() == 0) {
            return result;
        }
        ArrayList<Integer> list = new ArrayList<Integer>();
        Collections.sort(num);
        subsetsHelper(result, list, num, 0);

        return result;
    }


    private void subsetsHelper(ArrayList<ArrayList<Integer>> res
ult,
        ArrayList<Integer> list, ArrayList<Integer> num, int pos)
 {

        result.add(new ArrayList<Integer>(list));

        for (int i = pos; i < num.size(); i++) {
            if(i > pos && num.get(i) == num.get(i-1))
                continue;
            list.add(num.get(i));
            subsetsHelper(result, list, num, i + 1);
            list.remove(list.size() - 1);
        }
    }
}
```

Phone Number

Given a digit string, return all possible letter combinations that the number could represent.

Example

Given `23`

Return `["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]`

Note

Although the above answer is in lexicographical order, your answer could be in any order you want.

Solution

```
public class Solution {
    /**
     * @param digits A digital string
     * @return all posible letter combinations
     */
    public ArrayList<String> letterCombinations(String digits) {
        ArrayList<String> res = new ArrayList<>();
        if(digits == null ||digits.length() == 0)
            return res;
        HashMap<Character, String> dict = new HashMap<>();
        dict.put('2',"abc");dict.put('3',"def");dict.put('4',"gh
i");dict.put('5',"jkl");
        dict.put('6',"mno");dict.put('7',"pqrs");dict.put('8',"t
uv");dict.put('9',"wxyz");

        helper(dict, res, "", digits, 0);
        return res;
    }

    private void helper(HashMap<Character, String> dict, ArrayLi
st<String> res, String str, String digits, int idx) {
        if(idx == digits.length()) {
            res.add(new String(str));
            return;
        }
        String letters = dict.get(digits.charAt(idx));
        for(int i = 0; i < letters.length(); i++) {
            str += letters.charAt(i);
            helper(dict, res, str, digits, idx + 1);
            str = str.substring(0, idx);
        }
    }
}
```

Boggle Game

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

Think

- DFS on character board to do backtracking.
- Searching the character for 8 directions.
- Mark the visited position by a boolean board (Takes O(n^2) space)

Solution

```
public class Solution {

    public List<String> findWords(HashSet<String> dict, char[][]
 board) {
        List<String> res = new ArrayList<>();
        boolean[][] visited = new boolean[board.length][board[0]
.length];
        for(int i = 0; i < board.length; i++) {
            for(int j = 0; j < board[i].length; j++) {
                findUtil(res, dict, board, visited, "", i, j);
            }
        }
        return res;
    }

    private void findUtil(List<String> res, HashSet<String> dict
, char[][] board, boolean[][] visited, String cur, int x, int y)
{
        visited[x][y] = true;
        cur += board[x][y];
        if(dict.contains(cur)) {
            res.add(cur);
            dict.remvoe(cur);
            return;
        }
        // 8 - direction
        int[] xs = {1,1,1,0,0,-1,-1,-1};
        int[] ys = {1,-1,0,1,-1,0,1,-1};
        for(int i = 0; i < 8; i++) {
            int nx = xs[i] + x;
            int ny = ys[i] + y;
            if(nx >= 0 && nx < board.length && ny >= 0 && ny < b
oard[nx].length && !visited[nx][ny])
                findUtil(res, dict, board, visited, cur, nx, ny)
;
        }
        visited[x][y] = false;
        cur = cur.substring(0, cur.length() - 1);
    }

}
```

Scramble Number Pair Calculator

Let a pair of distinct positive integers, (i, j), be considered "scrambled" if you can obtain j by reordering the digits of i. For example, (12345, 25341) is a scrambled pair, but (12345, 67890) is not.

Given integers A and B with the same number of digits and no leading zeroes, how many distinct scrambled pairs (i, j) are there that satisfy: A <= i < j <= B?

Example

For instance, if we let A = 10 and B = 99, the answer is 36:

```
(12,21), (13,31), (14,41), (15,51), (16,61), (17,71), (18,81), (
19,91), (23,32), (24,42), (25,52), (26,62), (27,72), (28,82), (2
9,92), (34,43), (35,53), (36,63), (37,73), (38,83), (39,93), (45
,54), (46,64), (47,74), (48,84), (49,94), (56,65), (57,75), (58,
85), (59,95), (67,76), (68,86), (69,96), (78,87), (79,97), (89,9
8)
```

Think and Solution

```
public class Test {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Input min range: ");
        int min = scanner.nextInt();
        System.out.println("Input max range: ");
        int max = scanner.nextInt();
        System.out.print(scrambleNumberCalculator(min, max));
        Map<Object,Object>  map = new HashMap();

    }
    /**
     * Main function to do the scramble number pair calculation
     * @param minRange  : minimum of range
     * @param maxRange : maximum of range
     * @return
     */
    public  static long scrambleNumberCalculator(int min, int ma
x) {
        // the total scramble number set to avoid duplicate calc
ulate
        Set<Integer> pairs = new HashSet<>();

        // the result of pairs count;
        long cnt = 0;
        int[] range = new int[]{min, max};
        for (int i = range[0]; i <= range[1]; i++) {
            if (pairs.contains(i))
                continue;
            Set<Integer> res = new HashSet<>();
            List<Integer> list = convertDigitsList(i);
            permutation(res, list, range, 0, list.size());

            cnt += combinationPair(res.size());
            // res size equal to one means there is no pair for
this number
            // for saving pairs space I don't add the number wit
h no scramble pair.
            if(res.size() > 1)
                pairs.addAll(res);
        }
        // System.out.println(pairs);
        return cnt;
    }

    /**
     * The function to check current number has how many scrambl
e number and store it in a set,
     * recursion without return value but values are recorded in
 permutation set
     *
     * @param res: the permutation result storage as a set
     * @param digits: the current number digits for permutation
     * @param range: the range of output val
     * @param cur: current permutation value
     * @param idx: current permutation index for digits list
     */
    private static void permutation(Set<Integer> res, List<Integ
er> digits, int[] range,
            int cur, int idx) {
        if (idx == 0) {
            if (cur >= range[0] && cur <= range[1])
                res.add(cur);
            return;
        }
        for (int i = 0; i < digits.size(); i++) {
            // avoid the zero leading number
            if (cur == 0 && digits.get(i) == 0)
                continue;
            cur = cur * 10 + digits.get(i);
            digits.remove(i);
            permutation(res, digits, range, cur, idx - 1);
            digits.add(i, cur % 10);
            cur /= 10;
        }
    }

    /**
     * The function to count the pair amount in permutation set
by just using the size of current scramble number set
     * @param num: consider the large input I use long type here
     * @return the amount of pair in current permutation set
     */
    private static long combinationPair(long num) {
        long pairCnt = 0;
        for (int i = 0; i < num - 1; i++)
            for (int j = i + 1; j < num; j++)
                pairCnt++;
        return pairCnt;
    }
    /**
     * The function to convert a number into a list with digits
make the permutation more convenient
     * @param num
     * @return
     */
    private static List<Integer> convertDigitsList(int num) {
        List<Integer> res = new LinkedList<>();
        while (num > 0) {
            res.add(0, num % 10);
            num /= 10;
        }
        return res;
    }
}
```

Strobogrammatic Number

Problem I

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to determine if a number is strobogrammatic. The number is represented as a string.

For example, the numbers `"69"` , `"88"` , and `"818"` are all strobogrammatic.

Solution

```
public class Strobogrammatic {
    public boolean isStrobogrammatic(String num) {
        if (num == null || num.length() == 0)
            return true;
        int l = 0, r = num.length() - 1;
        while (l < r) {
            if (isEqual(num.charAt(l), num.charAt(r))) {
                l++;
                r--;
            } else
                return false;
        }
        return true;
    }

    private boolean isEqual(char l, char r) {
        if ((l == '9' && r == '6') || (l == '6' && r == '9')
                || (l == '1' && r == '1') || (l == '8' && r == '
8')
                || (l == '0' && r == '0'))
            return true;
        else
            return false;
    }

    // Use HashMap
    public boolean isStrobogrammatic(String num) {
        if(num == null || num.length() == 0) {
            return true;
        }

        Map<Character, Character> map = new HashMap<>();
        map.put('0', '0');
        map.put('1', '1');
        map.put('8', '8');
        map.put('6', '9');
        map.put('9', '6');

        int lo = 0;
        int hi = num.length() - 1;

        while (lo <= hi) {
            char c1 = num.charAt(lo);
            char c2 = num.charAt(hi);

            if (!map.containsKey(c1) || map.get(c1) != c2) {
                return false;
            }

            lo++;
            hi--;
        }

        return true;
    }
}
```

Problem II

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down). Find all strobogrammatic numbers that are of length = n.

Example,

Given n = 2, return `["11","69","88","96"]`.

Think

- Typical backtracking to generate something question

Solution

```
    public static List<String> findStrobogrammatic(int n) {
        Map<Character, Character> map = new HashMap<>();
        map.put('0', '0');
        map.put('1', '1');
        map.put('8', '8');
        map.put('6', '9');
        map.put('9', '6');
        List<String> res = new ArrayList<>();
        // two cases: even or odd
        if(n%2==0)
                generate(res, map, "", n);
        else{
            // the central digit can be any number
            for(int i = 0; i <= 9; i++)
                generate(res, map, ""+i, n);
        }
        return res;
    }

    private static void generate(List<String> res, Map<Character
, Character> map, String cur, int n) {
        if(cur.length() == n) {
            res.add(new String(cur));
            return;
        }

        for(Map.Entry<Character, Character> entry : map.entrySet
()) {
            String origin = new String(cur);
            cur = entry.getKey() + cur + entry.getValue();
            generate(res, map, cur, n);
            cur = origin;
        }
    }
```

Problem III

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to count the total strobogrammatic numbers that exist in the range of low <= num <= high.

Example

Given low = "50", high = "100", return 3. Because 69, 88, and 96 are three strobogrammatic numbers.

Note

Because the range might be a large number, the low and high numbers are represented as string.

Solution

```
public static int strobogrammaticInRange(String low, String high)
 {
        Map<Character, Character> map = new HashMap<>();
        map.put('0', '0');
        map.put('1', '1');
        map.put('8', '8');
        map.put('6', '9');
        map.put('9', '6');
        int[] cnt = new int[1];
        for (int n = low.length(); n <= high.length(); n++) {
            if (n % 2 == 0)
                generateII(cnt, map, "", n, low, high);
            else {
                for (int i = 0; i <= 9; i++)
                    generateII(cnt, map, "" + i, n, low, high);
            }
        }
        return cnt[0];
    }
    private static void generateII(int[] cnt,
            Map<Character, Character> map, String cur, int n, St
ring low,
            String high) {
        if (cur.length() == n) {
            if (cur.charAt(0) != '0' && compare(low, cur) < 0
                    && compare(cur, high) < 0)
                cnt[0]++;
            return;
        }

        for (Map.Entry<Character, Character> entry : map.entrySe
t()) {
            String origin = new String(cur);
            cur = entry.getKey() + cur + entry.getValue();
            generateII(cnt, map, cur, n, low, high);
            cur = origin;
        }
    }

    private static int compare(String s1, String s2) {
        return Integer.compare(Integer.parseInt(s1), Integer.par
seInt(s2));
    }
```

Factor Combinations

Numbers can be regarded as product of its factors. For example,

```
8 = 2 x 2 x 2;
  = 2 x 4.
```

Write a function that takes an integer n and return all possible combinations of its factors.

Note

Each combination's factors must be sorted ascending, for example: The factors of `2` and `6` is `[2, 6]`, not `[6, 2]`.

You may assume that n is always positive.

Factors should be greater than 1 and less than n.

Examples

input: `1`

output: `[]`

input: `37`

output: `[]`

input: `12`

output: `[ [2, 6], [2, 2, 3], [3, 4] ]`

input: `32`

output: `[ [2, 16], [2, 2, 8], [2, 2, 2, 4], [2, 2, 2, 2, 2], [2, 4, 4], [4, 8] ]`

Think

- For input value `n`, it has possible factors start from `2` to `Sqrt(n)`
- For every factor, we also calculate its factors, like: `16 -> 2, 8 -> 2, 2, 4 -> 2, 2, 2, 2, 2`
- Build helper function, the only difference between main recursion and helper recursion function is, in helper, we have to consider about the input value is one of factor which should also include in result list

Solution

```
    public List<List<Integer>> getFactors(int n) {
        // use hashset to avoid replicate
        Set<List<Integer>> res = new HashSet<>();
        int end = (int) Math.sqrt(n);
        for (int i = 2; i <= end; i++) {
            if (n % i != 0)
                continue;
            List<List<Integer>> prev = helper(n / i);
            for (List<Integer> each : prev) {
                each.add(i);
                // make sure the elements are sorted
                Collections.sort(each);
                res.add(each);
            }
        }
        return  new ArrayList<>(res);
    }

    private List<List<Integer>> helper(int n) {
        List<List<Integer>> res = new ArrayList<>();
        // add it self which is also a factor
        List<Integer> list = new ArrayList<>();
        list.add(n);
        res.add(list);

        int end = (int) Math.sqrt(n);
        for (int i = 2; i <= end; i++) {
            if (n % i != 0)
                continue;
            List<List<Integer>> prev = helper(n / i);
            for (List<Integer> each : prev) {
                each.add(i);
                res.add(each);
            }
        }
        return res;
    }
```

Dynamic Programming

When should we think about using dynamic programming?

Satisfied both two following situation.

1. When if following one of condition is true

- Find maximum value or minimum value.
- Existed or not, by `boolean` value.
- Count all possible solution (return just count).

2. Cannot rely on sort or swap operation.

3. Avoid greedy algorithm

What's the steps when using dynamic programming?

Status (State)

How to represent memory by data structure?

Function

How to pass through the memorial data structure?

Initialization

How to start the memorizing procedure and think about the boundary conditions?

Answer

Where is answer? Is it the last element in memorial data? Or it need to record any maximum or minimum value?

Routines

Matrix DP (2-D memorized)

Status: `f[i][j]` means to get the value by the start point to the position `[i][j]`

Function: different step approaches need to consider carefully

Initialization: in most case it doesn't needed

Answer: `f[last_i][last_j]` or recording max value or min value (depend on question)

Example: Triangle, Unique Path, Maximal Square...

Sequence DP (1-D memorized)

Status: `f[i]` means first 'i' position/number/alphabet

Function: `f[i]` = `f[j]` ... j is the position before i

Initialization: `f[0]` should be initialized with a certain value

Answer: `f[length-1]` or `f[length]`

Example: Partition Palindrome, Word Break, Longest Increasing Subsequence...

Two Sequence (2-D memorized)

Most of time, we setup 2-D memorized array has one plus original length, e.g. `memo[len+1][len+1]` . Why? Because we have to calculate the case of zero, and this also lead to the index of what we got should minus one, e.g. `s.charAt(i-1)` .

Status: `f[i][j]` means the first `[i]` number/alphabet and first `[j]` number/alphabet

Function: `f[i][j]` is the relationship of ith in first sequence and jth in second sequence

Initialization: consider about `f[i][0]` and `f[0][j]`

Example: Longest Common Substring, Longest Common Sequence, Edit Distance, Palindrome Boolean Matrix...

Backpack (Difficult Part)

Typical Question: Given N integers, one target, output if several number can be combined to get the sum as the target.

Status: `f[i][S]` the previous i numbers and whether it can sum to S

Function: `f[i][S]` should come from `f[i-1][S - arr[i]]` (taken) or `f[i-1][S]`(not taken)

Initialization: consider about f[i][0] and f[0][j]

House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Example

Given `[3, 8, 4]`, return 8.

Challenge

$O(n)$ time and $O(1)$ memory.

Think

- When Rob meet a house, he has two choices: taken or not taken, but if he had stolen the previous house, it cannnot taken.
- Set two variable: `include` and `exclude`
- `include` means it has taken the previous house, while the `exclude` means it didn't take in previous house.
- So each time we consider about the `exclude + cur_value` and `include`, the max of them should be new `include` while the new `exclude` come from the max value of original `include` (last taken, current not taken) and original `exclude` (not taken in both last and current house)

Solution

```
    /**
     * @param A: An array of non-negative integers.
     * return: The maximum amount of money you can rob tonight
     */
    public long houseRobber(int[] A) {
        if(A == null)
            return 0L;
        long exclude = 0L;
        long include = 0L;

        for(int i = 0; i < A.length; i++) {
            long tmp = include;
            include = Math.max(include, exclude + A[i]);
            exclude = Math.max(tmp, exclude);
        }

        return Math.max(include, exclude);
    }
```

Follow Up

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Think

The houses are in a cycle,so that when pick up the first element we cannot pick the last element because they are adjacent. So just seperate two parts for calculating: `0` to `last second element`, and `1` to `last element`, find the maximum from them.

Solution

```
    public int rob(int[] nums) {
        if(nums==null||nums.length==0)
            return 0;
        if(nums.length==1)
            return nums[0];
        return Math.max(helper(0, nums.length-2, nums), helper(1
,nums.length-1,nums));
    }

    private int helper(int l, int r, int[] nums){
        if(nums==null||nums.length==0)
            return 0;
        int incl = 0;
        int excl = 0;
        for(int i = l;i<=r;i++){
            int tmp = incl;
            incl = excl+nums[i];
            excl = Math.max(excl, tmp);
        }
        return Math.max(incl, excl);
    }
```

Backpack I

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

Example

If we have 4 items with size [2, 3, 5, 7], the backpack size is 11, we can select [2, 3, 5], so that the max size we can fill this backpack is 10. If the backpack size is 12. we can select [2, 3, 7] so that we can fulfill the backpack.

You function should return the max size we can fill in the given backpack.

Note

You can not divide any item into small pieces.

Challenge

□ time and $O(m)$ memory.

□ memory is also acceptable if you do not know how to optimize memory.

Think

- Setup 2-D memorized array with length is `memo[items.length][bag size]`
- `memo[i][S]` means what is the max size we can fill in when get first i items.
- Then we should check from zero to `M` size when got `i` th items and evaluate the max size when taken or not taken current `i` th item.

Solution

```
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    public int backPack(int M, int[] A) {
        int[][] bp = new int[N + 1][M + 1];

        for (int i = 0; i < A.length; i++) {
            for (int j = 0; j <= M; 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]); // add the A[i]- size itself
                }
            }
        }
        return bp[N][M];
    }
```

Backpack II

Given n items with size Ai and value Vi, and a backpack with size m. What's the maximum value can you put into the backpack?

Example

Given 4 items with size `[2, 3, 5, 7]` and value `[1, 5, 2, 4]`, and a backpack with size `10`. The maximum value is `9`.

Note

You cannot divide item into small pieces and the total size of items you choose should smaller or equal to m.

Challenge

memory is acceptable, can you do it in $O(m)$ memory?

Think

- The same idea as the Backpack I.
- But the value on memo array should be the value of items in bag

Solution

```
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */
    public int backPackII(int m, int[] A, int V[]) {
        // write your code here
        int[][] memo = new int[A.length+1][m+1];

         for(int i = 0; i < A.length; i++) {
            for(int j = 0;j <= m; j++) {
                if(j - A[i] >= 0)
                    memo[i+1][j] = Math.max(memo[i][j], memo[i][
j - A[i]] + V[i]); // add the value
                else
                    memo[i+1][j] = memo[i][j];
            }
        }

        return memo[A.length][m];
    }
```

Think (Space Optimized)

- 1-D array with length of `m`
- `memo[i]` should be the max value with `i` size items
- NOTE: the iterate on size should be reversed, from `m` to `0` since the value come from `j - A[i]` that can be updated if we look the previous index

Solution

```
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */
    public int backPackII(int m, int[] A, int V[]) {
        // write your code here
        int[] memo = new int[m+1];

         for(int i = 0; i < A.length; i++) {
            for(int j = m; j >= 0; j--) { // lookup by reversed
 order
                if(j - A[i] >= 0)
                    memo[j] = Math.max(memo[j], memo[j - A[i]] +
 V[i]);
            }
        }

        int maxVal = 0;
        for(int i = m; i >= 0; i--)
            maxVal = Math.max(maxVal, memo[i]);

        return maxVal;
    }
```

Edit Distance

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Example

Given `word1 = mart` and `word2 = karma` , return `3` .

Think

- Two sequence DP
- Setup a 2-D array
- `memo[i][j]` means the edit distance between `word1(1, i)` and `word2(1, j)` on the current position.
- When `word1[i] != word2[j]` , it need increase the distance, so the value of `memo[i][j]` comes from the minimum of `memo[i-1][j]` , `memo[i][j-1]` or `memo[i-1][j-1]` then plus one.
- Initialized value is `i` or `j` when `word1(0,0)` or `word2(0,0)` (`memo[i][0]` or `memo[0][j]`)
- The final output is `memo[word1.length][word2.length]`

Solution

```
    /**
     * @param word1 & word2: Two string.
     * @return: The minimum number of steps.
     */
    public int minDistance(String A, String B) {
        // write your code here
        if(A == null || B == null)
            return 0;

        int[][] memo = new int[A.length()+1][B.length()+1];
        for(int i = 0; i <= A.length(); i ++) {
            for(int j = 0; j <= B.length(); j ++) {
                if(i == 0 || j == 0)
                    memo[i][j] = (i == 0? (j == 0? 0 : j ): i);
                else
                    memo[i][j] =  (A.charAt(i-1) == B.charAt(j-1
) ? memo[i-1][j-1] : Math.min(memo[i-1][j-1], Math.min(memo[i][j-
1],memo[i-1][j])) + 1);
            }
        }
        return memo[A.length()][B.length()];
    }
```

Longest Increasing Sub-sequence

Given an unsorted array of integers, find the length of longest increasing sub-sequence.

Example

Given `[10, 9, 2, 5, 3, 7, 101, 18]`, The longest increasing sub-sequence is `[2, 3, 7, 101]`, therefore the length is `4`. Note that there may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in $O(n^2)$ complexity.

Follow up

Improve it to O(n log n) time complexity?

Think #1

- Setup one dimensional array to memorize the longest sub-sequence on each element
- One pass on each element,
- But it need to go through the previous indexes to check any elements less than current element and compare the maximum by `max(current, prev +1)`

Solution #1

```
    /**
     * @param nums: The integer array
     * @return: The length of LIS (longest increasing subsequenc
e)
     */
    public int longestIncreasingSubsequence(int[] nums) {
        if(nums == null || nums.length == 0)
            return 0;
        int[] memo = new int[nums.length];
        int max = 0;
        for(int i = 0; i < nums.length; i ++) {
            memo[i] = 1;
            for(int j = 0; j < i; j++) {
                if(nums[j] < nums[i]) {
                    memo[i] = Math.max(memo[i], memo[j] + 1);
                }
                max = Math.max(memo[i], max);
            }
        }
        return max;
    }
```

Think #2

- Setup an array `table` with the length of `nums`
- One pass on each element and initial max cursor is `0` in `table`
- Replace the element in `table` is just larger than current passing element
- If current element is largest, increase the cursor in `table`
- Finally, the the cursor + 1 is the max length.

Solution #2

```
    /**
     * @param nums: The integer array
     * @return: The length of LIS (longest increasing subsequenc
e)
```

```
     */
    public int longestIncreasingSubsequence(int[] nums) {
        if(nums == null || nums.length == 0)
            return 0;
        int[] memo = new int[nums.length];
        int max = 0;
        memo[0] = nums[0];
        for(int i = 1; i < nums.length; i++) {
            if(nums[i] < memo[0])
                memo[0] = nums[i];
            else if(memo[max] <= nums[i])
                // equal or not depend on question requirement
                memo[++max] = nums[i];
            else{
                int idx = findCeil(memo, max, nums[i]);
                memo[idx] = nums[i];
            }
        }
        return ++max;
    }

    /**
     * @param arr: the memo array
     * @param right index: current max in the memo array
     * @param val: target value
     * @return: where should val put in memo array
     */
    private int findCeil(int[] arr, int r, int val){
        int l = 0;
        while(l + 1 < r) {
            int m = l + ((r - l)>>1);
            if(arr[m] < val)
                l = m;
            else
                r = m;
        }
        return r;
    }
```

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.

If the array before adjustment is A, the array after adjustment is `B`, you should minimize the sum of `|A[i]-B[i]|`

Example

Given `[1,4,2,3]` and target = 1, one of the solutions is `[2,3,2,3]`, the adjustment cost is 2 and it's minimal.

Return `2`.

Note You can assume each number in the array is a positive integer and not greater than `100`.

Think

- There is invisible condition on Note part: each number in the array is a positive integer and not greater than `100`
- So the above condition giving the memorized data structure a length on enumeration.
- memo data should be `memo[arr.size() + 1][100]`, which means the minimum cost on modify the ith num in A to `j`
- Pass each element in input array
- Then enumeration `j` on 0 to 99, so that we can consider the previous number `p` should inside the range between `j + target` and `j-target`.
- So the previous cost should be `memo[i-1][p]` and for current, it should be `memo[i-1][p] + Math.abs(j-A.get(i-1))`
- After arrived the last element and got all possible cost, the minimum total cost should come from `memo[last element][i]`

Solution

```
    /**
     * @param A: An integer array.
     * @param target: An integer.
     */
    public int MinAdjustmentCost(ArrayList<Integer> A, int targe
t) {
        // the minimum cost on modify the ith num in A to j
        int[][] memo = new int[A.size() + 1][100];

        for(int i = 1; i <= A.size(); i++) {
            // enumeration on 0 to 99
            for(int j=0; j<=99; j++) {
                // initial the minimum cost on ith as the Intege
r.MAX
                memo[i][j] = Integer.MAX_VALUE;
                // get the range by target based on j
                int lowerRange = Math.max(0, j-target);
                int upperRange = Math.min(99, j+target);
                // p is the possible num based on previous index
                // check all cost between current index on A and
 j
                for (int p=lowerRange; p<=upperRange; p++) {

                    memo[i][j] = Math.min(memo[i][j], memo[i-1][
p]+Math.abs(j-A.get(i-1)));
                }
            }
        }
        int minCost = Integer.MAX_VALUE;
        // check all minimum cost on index equal to A's last ele
ment
        for(int i = 0; i <= 99; i++)
            minCost = Math.min(minCost, memo[A.size()][i]);
        return minCost;
    }
```

Coins in a Line

Problem I

There are n coins in a line. Two players take turns to take one or two coins from right side until there are no more coins left. The player who take the last coin wins.

Could you please decide the first play will win or lose?

Example

n = `1` , return `true` .

n = `2` , return `true` .

n = `3` , return `false` .

n = `4` , return `true` .

n = `5` , return `true` .

Challenge

$O(n)$ time and $O(1)$ memory

Think

- Only if the amount of coin is the multiply of `3` , first player cannot win.

Solution

```
    /**
     * @param n: an integer
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int n) {
        return n%3!=0;
    }
```

Problem II

There are n coins with different value in a line. Two players take turns to take one or two coins from left side until there are no more coins left. The player who take the coins with the most value wins.

Could you please decide the first player will win or lose?

Example

Given values array A = `[1,2,2]`, return true.

Given A = `[1,2,4]`, return false.

Think

State

`dp[i]` represent the max value it can get from `i` to the end

Function

Each time when iterate `i`, we have two choice:

- takes `values[i]`
- takes `values[i] + values[i+1]`

Here are what we need to think:

- 1.If we took `values[i]`, opposite has two choice: `values[i+1]` or `values[i+1] + values[i+2]` so the rest values for our own sides are `DP[i+2]` or `DP[i+3]`, however it should choose the minimum so that the opposite can get the maximum.

  $$value1 = values[i] + min(DP[i+2], DP[i+3])$$

- 2.If we took `values[i] + values[i+1]`

  $$value2 = values[i] + values[i+1] + min(DP[i+3], DP[i+4])$$

- 3.

  $$dp[I] = max(value1, value2)$$

Solution

```
    /**
     * @param values: an array of integers
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int[] values) {
        if (values == null || values.length <= 2) {
            return true;
        }
        int dp[] = new int[values.length];

        dp[values.length-1] = values[values.length-1];
        dp[values.length-2] = values[values.length-1] + values[v
alues.length-2];
        int total = dp[values.length-2];

        for(int i = values.length - 3; i >= 0; i--) {
            total += values[i];
            int value1 = values[i] + Math.min(dp[i+2], i+3<value
s.length?dp[i+3]:0);
            int value2 = values[i] + values[i+1] + Math.min(i+3<
values.length?dp[i+3]:0, i+4<values.length?dp[i+4]:0);
            dp[i] = Math.max(value1, value2);
        }

        return dp[0] > (total - dp[0]);
    }
```

Problem III

There are n coins in a line. Two players take turns to take a coin from one of the ends of the line until there are no more coins left. The player with the larger amount of money wins.

Could you please decide the first player will win or lose?

Example

Given array A = `[3,2,2]` , return true.

Given array A = `[1,2,4]` , return true.

Given array A = `[1,20,4]` , return false.

Challenge

Follow Up Question: If `n` is even. Is there any hacky algorithm that can decide whether first player will win or lose in $O(1)$ memory and $O(n)$ time?

Think

State

`dp[i][j]` represent the max value it can get from `i` to `j`

`sum[i][j]` represent the sum value from `i` to `j`

Function

Each time when taken coins, we have two choice:

- takes `values[i]`
- takes `values[j]`

$$dp[i][j] = max(values[i] + sum[i+1][j] - dp[i+1][j], values[j] + sum[i][j-1] - dp[i][j-1])$$

Since `values[i]+sum[i+1][j]` or `values[j]+sum[i][j-1]` equals to `sum[i][j]`

So the equation can be $dp[i][j] = sum[i][j] - min(dp[i+1][j], dp[i][j-1])$

Solution

```
    /**
     * @param values: an array of integers
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int[] values) {
        if (values == null || values.length <= 2) {
            return true;
        }

        int[][] dp = new int[values.length][values.length];
        int[][] sum = new int[values.length][values.length];
        // get the sum of i to j
        for(int i = 0; i < values.length; i++){
            for(int j  = i; j < values.length; j++) {
                if(i == j)
                    sum[i][j] = values[i];
                else
                    sum[i][j] = values[j] + sum[i][j-1];
            }
        }

        // to do the dp
        for(int i = values.length - 1; i >= 0; i--){
            for(int j  = i; j < values.length; j++) {
                if(i == j)
                    dp[i][j] = values[i];
                else
                    dp[i][j] = sum[i][j] - Math.min(dp[i+1][j],d
p[i][j-1]);
            }
        }

        return dp[0][values.length - 1] > sum[0][values.length -
1] - dp[0][values.length - 1];
    }
```

Distinct Subsequences

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. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Example

Given S = "rabbbit", T = "rabbit", return 3.

Challenge

Do it in $O(n^2)$ time and $O(n)$ memory.

$O(n^2)$ memory is also acceptable if you do not know how to optimize memory.

Think

- Two sequence DP, setup 2-D array for memorizing
- `dp[i][j]` means the distinct count when `S(1,i)` and `T(1,j)`
- If S has non character `dp[0][j]` should be 0, while T has non character `dp[i][0]` should be 1.
- Each time we add `S[i]` on `dp[i-1][j]`, iterate `j` from `0` to `len - 1`, if `S[i] == T[j]` we should look up the the result from `dp[i-1][j-1]` and add it on `dp[i-1][j]`, otherwise we add nothing.

Solution

```
    /**
     * @param S, T: Two string.
     * @return: Count the number of distinct subsequences
     */
    public int numDistinct(String S, String T) {
        int[][] memo = new int[S.length() + 1][T.length() + 1];
        for(int i = 0; i <= S.length(); i++) {
            for(int j = 0; j <= T.length(); j++) {
                if(i == 0)
                    memo[i][j] = 0;
                if(j == 0)
                    memo[i][j] = 1;
                else{
                    memo[i][j] = memo[i-1][j] + (S.charAt(i-1) =
= T.charAt(j-1)) ? memo[i-1][j-1] : 0;
                }
            }
        }
        return memo[S.length()][T.length()];
    }
```

Maximal Square

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

Example

For example, given the following matrix:

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

Return `4`.

Think

- Use itself as memorized array (modifying value directly on matrix)
- Ignore the top and left boundary
- If current point `[i][j]` is one, look up all three directions from `[i-1][j]`, `[i-1][j-1]` and `[i][j-1]` are not zero, get the minimum value from them so that the value plus one is the maximum length of square on current point.
- However, if one of three is zero, current point should keep zero or one
- Set a max value to track the max length.
- Don't forget make a square on final max result, since that result is just for length

Solution

```
    /**
     * @param matrix: a matrix of 0 and 1
     * @return: an integer
     */
    public int maxSquare(int[][] matrix) {
        if(matrix == null || matrix.length == 0)
            return 0;

        int max = 0;
        for(int i = 0; i < matrix.length; i++) {
            for(int j = 0; j < matrix[i].length; j++) {
                if(i != 0 && j != 0 && matrix[i][j]!=0 && matrix
[i-1][j] != 0 && matrix[i][j-1] != 0 && matrix[i-1][j-1] != 0)
                    matrix[i][j] = 1 + Math.min(matrix[i-1][j-1]
,Math.min(matrix[i-1][j],matrix[i][j-1]));
                max = Math.max(max, matrix[i][j]);
            }
        }
        return max*max;
    }
```

Interleaving String

Given three strings: s1, s2, s3, determine whether s3 is formed by the interleaving of s1 and s2.

Example

For s1 = `"aabcc"` , s2 = `"dbbca"`

When s3 = `"aadbbcbcac"` , return true. When s3 = `"aadbbbaccc"` , return false.

Challenge

$O(n^2)$ time or better

Think

- Two sequence DP, 2-D memorized boolean array
- `memo[i][j]` means from `word1(1,i)` and `word2(1,j)` can form the `word3(1, i+j)`
- Initial the case `memo[0][j]` and `memo[i][0]` is false and `memo[0][0]` should be true.
- Pass through `i` from `0` to `word1.length` and `j` from `0` to `word2.length` , check if `word3[i+j-1]== word2[j-1]` or `word3[i+j-1]== word1[i-1]`
- We also have to make sure when `word3[i+j-1]== word2[j-1]` , the `memo[i][j-1]` should be true or `word3[i+j-1]== word1[i-1]` , the `memo[i-1][j]` should be true
- Get the `memo[len1][len2]` as the final boolean result

Solution

```
    /**
     * Determine whether s3 is formed by interleaving of s1 and
s2.
     * @param s1, s2, s3: As description.
     * @return: true or false.
     */
    public boolean isInterleave(String s1, String s2, String s3)
{

        if(s1 == null || s2 == null)
            return false;
        if(s1.length() + s2.length() != s3.length())
            return false;

        boolean[][] memo = new boolean[s1.length() + 1][s2.lengt
h() + 1];
        for(int i = 0; i <= s1.length(); i++) {
            for(int j = 0; j <= s2.length(); j++) {
                if(i == 0 && j == 0)
                    memo[i][j] = true;
                else{
                    if(i > 0 && s1.charAt(i - 1) == s3.charAt(i
+ j - 1))
                        memo[i][j] = memo[i - 1][j];
                    if(j > 0 &&s2.charAt(j - 1) == s3.charAt(i +
 j - 1))
                        memo[i][j] |= memo[i][j - 1];
                }
            }
        }

        return memo[s1.length()][s2.length()];
    }
```

Word Break

Given a string s and a dictionary of words dict, determine if s can be break into a space-separated sequence of one or more dictionary words.

Example

Given s = `"lintcode"` , dict = `["lint", "code"]` .

Return true because "lintcode" can be break as "lint code".

Think

- One sequence DP, a 1-D boolean array.
- `memo[i]` means from `0` to `i` has valid word break or not.
- Tricky part is when we pass at `i` position, we don't need to use `j` from `0` to `i` for checking word existed in dictionary. We can use the length of word in dictionary as an length evaluation.
-

Solution

```
/**
 * @param s: A string s
 * @param dict: A dictionary of words dict
 */
public boolean wordBreak(String s, Set<String> dict) {

    if(s == null || s.length() == 0){
        if(dict == null || dict.size() == 0)
            return true;
        return false;
    }

    boolean[] memo = new boolean[s.length() + 1];
    memo[0] = true;
    for(int i = 1; i <= s.length(); i ++) {
        for(String ss : dict){
            int start = i - ss.length();
            if( start >= 0 && memo[start]){
                String str = s.substring(start, i);
                if(dict.contains(str))
                     memo[i] = true;
            }
        }
    }
    return memo[s.length()];
}
```

Scramble String

Given a string s1, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of s1 = `"great"`:

```
    great
   /    \
  gr    eat
 / \    /  \
g   r  e   at
           / \
          a   t
```

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string `"rgeat"`.

```
    rgeat
   /    \
  rg    eat
 / \    /  \
r   g  e   at
           / \
          a   t
```

We say that `"rgeat"` is a scrambled string of `"great"`.

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string `"rgtae"`.

```
      rgtae
     /    \
    rg     tae
   / \    /  \
  r   g  ta   e
        / \
       t   a
```

We say that `"rgtae"` is a scrambled string of `"great"`.

Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1.

Think - DP Version

- Ignore two situations: both length not equal and the characters not the same
- Two sequence but 3-D memorized array
- `memo[i][j][k]` means state: for s1.substring(i, i + k) and s2.substring(j, j + k), if they are scramble string
- Two conditions we can regard as scramble, for range of `word1(i -> i+k)` or `word2(j -> j+k)`:
  - `i -> i + split` = `j -> j + split` (len = split) and `split + i -> i + k` = `split + i -> j + k` (len = k - split)
  - `i -> i + split` = `j + (k - split) -> j+k` [len = split] and `i + split -> i+k` = `j -> j + (k - split)` (len = k - split)
- Consider about the initialization:
  - for `k == 1`, we only check if `word1[i] == word2[j]`

Solution

```
    /**
     * @param s1 A string
     * @param s2 Another string
     * @return whether s2 is a scrambled string of s1
     */
    public boolean isScramble(String s1, String s2) {
        // check length
        if(s1==null||s2==null||s1.length()!=s2.length())
            return false;
        // check anagram
        char[] c1 = s1.toCharArray();
        char[] c2 = s2.toCharArray();
        Arrays.sort(c1);
        Arrays.sort(c2);
        if (!Arrays.equals(c1, c2))
            return false;

        if(s1.length() != s2.length())
            return false;
        int len = s1.length();

        // state: for s1.substring(i, i + k) and s2.substring(j,
 j + k), if they are scramble string
        boolean[][][] memo = new boolean[len][len][len+1];

        // initial, only check if s1[i] == s2[j]
        for(int i=0;i<s1.length();i++)
            for(int j=0;j<s2.length();j++)
                memo[i][j][1] = (s1.charAt(i) == s2.charAt(j));

        for(int k = 2; k <= len; k++) {

            for(int i = 0; i <= len - k; i++) {
                for(int j = 0; j <= len - k; j++) {
                    // split point should start from 1 to k - 1
                    for(int split = 1; split < k; split++) {
                        memo[i][j][k] |= (memo[i][j][split]&&mem
o[i+split][j+split][k-split])||(memo[i][j+(k - split)][split]&&m
emo[i+split][j][k-split]);
                    }
                }
            }
        }
        return memo[0][0][len];
    }
```

Maximum Subarray III

Given an array of integers and a number k, find k non-overlapping subarrays which have the largest sum.

The number in each subarray should be contiguous.

Return the largest sum.

Example

Given `[-1,4,-2,3,-2,3]` , k= `2` , return `8`

Note

The subarray should contain at least one number

Think

- State: `memo[k][i]` means the max subarray sum value from index 0 to i-1 with taken k subarrays
- Since `k` is the amount of subarray, so at least we have to take `k` elements in array
- Initilize with two case:
  - Taken first `k` elements, each element is a subarray, so we only consider first kth element.
  - Taken only one subarray, with two cases: start from previous to current or start from itself.
- For the rest of elements, consider two cases:
  - Taken from the previous element with no subarray amount increase (`memo[k][i-1] + current element`)
  - Taken from current element but increase `k` so `memo[k-1][n] + current element` , however, here `n` can be any number from `k-1` to `current index - 1`

Solution

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @param k: An integer denote to find k non-overlapping sub
arrays
     * @return: An integer denote the sum of max k non-overlappi
ng subarrays
     */
    public int maxSubArray(ArrayList<Integer> nums, int k) {

        // like backpack, memo[k][i] means the max value from in
dex 0 to i-1 with taken k subarray
        int[][] memo = new int[k+1][nums.size()];

        // count previous kth num, which means the each item as
a subarray
        for(int i = 1; i<= k; i++) {
            int sum = 0;
            for(int j = 0; j < i; j++) {
                sum += nums.get(j);
            }
            memo[i][i-1] = sum;
        }

        // when k = 1, calculate the max value has two choices:
        // 1) max value from previous index (also come from two
choices) and current index
        // 2) just start from current index
        for(int i = 1; i < nums.size(); i++) {
            memo[1][i] = Math.max(memo[1][i-1] + nums.get(i), nu
ms.get(i));
        }

        // also has two cases:
        // 1) continous get from previous index to current
        // 2) not continuous but we have to scan each index betw
een current and (k-1) position
```

```
        for(int subarr = 2; subarr <= k; subarr++) {
            for(int j = subarr; j < nums.size(); j++) {
                // 1) continous get so directly add current numb
er
                memo[subarr][j] =  memo[subarr][j-1] + nums.get(
j);
                // 2) resume in previous index from subarr - 2 (
one more minus for index)
                for(int s = subarr - 2; s < j; s++) {
                    memo[subarr][j] = Math.max(memo[subarr][j],
memo[subarr-1][s] + nums.get(j));
                }
            }
        }

        // pass through all case from k-1 to array length
        int max = Integer.MIN_VALUE;
        for(int i = k-1; i < nums.size(); i++) {
            max = Math.max(max, memo[k][i]);
        }
        return max;
    }
}
```

Painter Problems

Paint Fence

There is a fence with `n` posts, each post can be painted with one of the `k` colors.

You have to paint all the posts such that **no more than two adjacent fence posts** have the same color.

Return the total number of ways you can paint the fence.

Note

`n` and `k` are non-negative integers.

Think

- Two cases:
  - If `[n-1] == [n]`, `[n]` has `1` choices
  - If `[n-1] != [n]`, `[n]` has `k-1` choices with consider the result from both `[n-2] == [n-1]` OR `[n-2] != [n-1]`.

Solution

```
    public int numWays(int n, int k) {
        if(n == 0 || k == 0)
            return 0;
        int same = k;
        if(n == 1)
            return same;
        int noSame = k*(k-1);
        for(int i = 2; i < n; i++) {
            int tmp = noSame;
            noSame = (same + noSame) * (k-1);
            same = tmp * 1;
        }
        return noSame + same;
    }
```

Paint House

There are a row of n houses, each house can be painted with one of the three colors: `red` , `blue` or `green` . The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a `n x 3` cost matrix. For example, `costs[0][0]` is the cost of painting house 0 with color red; `costs[1][2]` is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Follow Up: 2 Colors -> K Colors

There are a row of n houses, each house can be painted with one of the k colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a `n x k` cost matrix. For example, `costs[0][0]` is the cost of painting house 0 with color 0; `costs[1][2]` is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

Think

- Current paint only come from the previous different paint cost
- `minCost[i][j] = costs[i][j] + min(minCost[i-1][k])` (k != j)
- We can do it in-place, update the newest value on the original memorized array.

Solution (General for Two Cases: 2 or K colors)

```
public int minCost(int[][] costs) {
    if (costs == null || costs.length == 0)
            return 0;

    for(int i = 1; i < costs.length; i++) {
        for(int j = 0; j < costs[i].length; j++) {
            int cur = Integer.MIN_VALUE;
            for(int k = 0; j < costs[i].length; k++) {
                if(k == j)
                    continue;
                cur = Math.max(cur, costs[i-1][k] + costs[i][j])
;
            }
            costs[i][j] = cur;
        }
    }

    int max = 0;
    for(int i = 0; i < costs[costs.length - 1].length; i++)
        max = Math.max(max, costs[costs.length - 1][i]);

    return max;
}
```

Linked List

Linked List is common data structure:

```
public class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
        next = null;
    }
 }
```

But it covers a lot of algorithm strategies with manipulating this kind of data structure.

Routines

- Two Pointers
  - Runner Node and Walker Node
    - Find a certain point in list
      - e.g. Remove Nth Node from End of List
      - e.g. Palindrome List
    - Find Cycle in Linked List
  - Reverse Node
  - Remove Duplicates in List
- Dummy Node
  - e.g. Copy Random Pointer on Linked List
- Sort Algorithms
  - e.g. Insertion Sort on Linked List
  - e.g. Merge K Sorted Linked List (Assisted with Heap)
- Divide and Conquer
  - e.g. Merge Sort Linked List
  - e.g. Build a Tree from Linked List

Remove Element from Linked List

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

Solution

```
public class Solution {
    /**
     * @param head a ListNode
     * @param val an integer
     * @return a ListNode
     */
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy;
        while(head!=null) {
            if(head.val != val) {
                pre.next = head;
                pre = pre.next;
            }
            head = head.next;
        }
        // this is important
        pre.next = null;
        return dummy.next;
    }
}
```

Remove Nth Node From End of List

Given a linked list, remove the nth node from the end of list and return its head.

Example

Given linked list: `1->2->3->4->5->null`, and n = `2`.

After removing the second node from the end, the linked list becomes `1->2->3->5->null`.

Note

The minimum number of nodes in list is n.

Challenge

O(n) time

Think

- Typical idea on runner and walker linked list question
- Let runner node run for N step further than walker node.
- Get the N + 1 th position from end of list.
- Remove walker.next which is the target node.

Solution

```
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @param n: An integer.
     * @return: The head of linked list.
     */
    ListNode removeNthFromEnd(ListNode head, int n) {
        if(head==null)
            return head;
        ListNode runner = head;
        ListNode pre = new ListNode(0);
        pre.next = head;
        ListNode walker = pre;
        while(n>0&&runner!=null){
            runner = runner.next;
            n--;
        }
        while(runner!=null){
            runner = runner.next;
            walker = walker.next;
        }
        walker.next = walker.next.next;
        return pre.next;
    }
}
```

Swap pairs in Linked List

Given a linked list, swap every two adjacent nodes and return its head.

Example

Given 1->2->3->4, you should return the list as 2->1->4->3.

Challenge

Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

Think

- Record next two node: `dummy pre -> A -> B -> C`.
- Get C as next, change pre.next to B(head.next), B.next to A(head) and A(head.next) to C.
- Do while loop when head and head.next are both not null node

Solution

```
public class Solution {
    /**
     * @param head a ListNode
     * @return a ListNode
     */
    public ListNode swapPairs(ListNode head) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy;

        while(head != null && head.next != null) {

            ListNode next = head.next.next;
            pre.next = head.next;
            pre.next.next = head;
            pre = head;
            head.next = next;
            head = next;
        }
        return dummy.next;
    }
}
```

Intersection of Two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.

Example

The following two linked lists:

```
A:          a1 → a2
                   ↘
                     c1 → c2 → c3
                   ↗
B:     b1 → b2 → b3
```

begin to intersect at node c1.

Note

If the two linked lists have no intersection at all, return null. The linked lists must retain their original structure after the function returns. You may assume there are no cycles anywhere in the entire linked structure.

Think

- Count the length of each linked list
- Make two counts to be equal, then start moving and check if there are two nodes the same as each other.

Solution

```
public class Solution {
    /**
```

```
     * @param headA: the first list
     * @param headB: the second list
     * @return: a ListNode
     */
    public ListNode getIntersectionNode(ListNode headA, ListNode
 headB) {

        ListNode a = headA;
        ListNode b = headB;
        int alen = 0, blen = 0;

        while(a!=null) {
            a = a.next;
            alen++;
        }
        while(b!=null) {
            b = b.next;
            blen++;
        }

        while(alen > blen) {
            headA = headA.next;
            alen--;
        }

        while(alen < blen) {
            headB = headB.next;
            blen--;
        }

        while(headA != null && headB != null) {
            if(headA == headB)
                return headA;
            headA = headA.next;
            headB = headB.next;
        }
        return null;
    }
}
```

Flatten a Binary Tree to a Linked List

Flatten a binary tree to a fake "linked list" in pre-order traversal.

Here we use the right pointer in TreeNode as the next pointer in ListNode.

Example

```
              1
               \
     1          2
    / \          \
   2   5    =>    3
  / \   \          \
 3   4   6          4
                     \
                      5
                       \
                        6
```

Note

Don't forget to mark the left child of each node to null. Or you will get Time Limit Exceeded or Memory Limit Exceeded.

Challenge

Do it in-place without any extra memory.

Think

- One pass with iterate each right node.
- Get current root's left child(`left`) if it is not null.
- Put current root's right child (`right`) into the most right leaf position of left

child(`left`).
- Swap the `left` in to `right` position if `left` is not null.

Solution

```
public class Solution {
    /**
     * @param root: a TreeNode, the root of the binary tree
     * @return: nothing
     */
    public void flatten(TreeNode root) {
        TreeNode node = root;
        while(node!=null) {
            if(node.left != null) {
                TreeNode left = node.left;
                while(left.right != null) {
                    left = left.right;
                }
                left.right = node.right;
                node.right = node.left;
                node.left = null;
            }
            node = node.right;
        }
    }
}
```

Implement a function to check if a linked list is a palindrome.

Example

Given 1->2->1, return true

Challenge

Could you do it in O(n) time and O(1) space?

Solution

}

```
public class Solution {
    /**
     * @param head a ListNode
     * @return a boolean
     */
    public boolean isPalindrome(ListNode head) {

        // get half
        ListNode runner = head;
        ListNode walker = new ListNode(0);
        walker.next = head;
        while(runner != null && runner.next != null) {
            runner = runner.next.next;
            walker = walker.next;
        }

        // reverse half tail
        ListNode tail = null;
        ListNode move = walker.next;

        while(move!=null){
            ListNode next = move.next;
            move.next = tail;
            tail = move;
            move = next;
        }

        // check palindrome
        while(head != null&& tail != null) {
            if(head.val != tail.val)
                return false;
            head = head.next;
            tail = tail.next;
        }
        return true;
    }
}
```

Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.

You may not alter the values in the nodes, only nodes itself may be changed. Only constant memory is allowed.

Example

Given this linked list: `1->2->3->4->5`

For k = `2`, you should return: `2->1->4->3->5`

For k = `3`, you should return: `3->2->1->4->5`

Think

- Consider the list like following:

  ```
  dummy(pre) -> l1 -> l2 -> l3 -> l4 -> l5 -> null
  ```

- Get the reversed segment:

```
when cnt % k == 0:

      pre.next      node  node.next
         |            |     |
  pre -> l1 -> l2 -> l3 -> l4 -> l5 -> null

        last      ---->         end
          |                  |
  pre -> l1 -> l2 -> l3 -> l4 -> l5 -> null
          |     |
      pre.next pre.next(final status)
             \     \
                |  -> |     |
               cur   cur     cur.next (final status)
```

Solution

```
public class Solution {
    /**
     * @param head a ListNode
     * @param k an integer
     * @return a ListNode
     */
    public ListNode reverseKGroup(ListNode head, int k) {
        if(head == null || head.next == null)
            return head;
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy;
        int cnt = 1;
        ListNode front = head;
        while(head != null) {
            if(cnt % k == 0) {
                pre = reverse(pre, head.next);
                head = pre.next;
            }else
                head = head.next;
            cnt++;
        }
        return dummy.next;
    }

    private ListNode reverse(ListNode pre, ListNode end) {
        ListNode last = pre.next, cur = last.next;
        while(cur != end) {
            last.next = cur.next;
            cur.next = pre.next;
            pre.next = cur;
            cur = last.next;
        }
        return last;
    }
}
```

Copy List with Random Pointer

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

Challenge

Could you solve it with O(1) space?

Think

- Three pass:
  - Clone every node and attach it right next of original node,
  - Copy the random pointer for clone node (the next of origianl node's random pointer)
  - Cut down the original and clone one

Solution

```
public class Solution {
    /**
     * @param head: The head of linked list with a random pointe
r.
     * @return: A new head of a deep copy of the list.
     */
    public RandomListNode copyRandomList(RandomListNode head) {
        // write your code here
        if(head == null)
            return null;
        RandomListNode dummyHead = head;
        while(head != null) {
            RandomListNode clone = new RandomListNode(head.label
);
```

```
                RandomListNode next = head.next;
                head.next = clone;
                clone.next = next;
                head = next;
            }
            head = dummyHead;
            while(head != null) {
                RandomListNode clone = head.next;
                if(head.random != null)
                    clone.random = head.random.next;
                head = clone.next;
            }
            head = dummyHead;
            RandomListNode resHead = dummyHead.next;
            while(head != null) {
                RandomListNode clone = head.next;
                head.next = clone.next;
                head = head.next;
                if(head != null)
                    clone.next = head.next;
            }

            return resHead;
        }
    }
```

Merge k Sorted Lists

Merge k sorted linked lists and return it as one sorted list.

Analyze and describe its complexity.

Example

Given lists:

```
[
  2->4->null,
  null,
  -1->null
],
```

return `-1->2->4->null`.

Think

- Use a heap to receive element from linked list
- Tricky part:
  - Just entered k node in heap instead of pass all nodes in lists.
  - When poll out element, it also need to push back the next node of polled node.

Solution

```
public class Solution {
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    public ListNode mergeKLists(List<ListNode> lists) {
        if(lists == null)
            return null;

        PriorityQueue<ListNode> queue = new PriorityQueue<ListNo
de>(10, new Comparator<ListNode>(){
            @Override
            public int compare(ListNode o1, ListNode o2){
                return Integer.compare(o1.val, o2.val);
            }
        });
        // O(n) : n total nodes
        for(ListNode node : lists){
            if(node != null)
                queue.offer(node);
        }

        ListNode dummy = new ListNode(0);
        ListNode pre = dummy;
        while(!queue.isEmpty()){
            ListNode cur = queue.remove();
            if(cur.next != null)
                queue.offer(cur.next);
            pre.next = cur;
            pre = cur;
        }

        return dummy.next;
    }
}
```

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.

Example

```
               2
1->2->3  =>   / \
             1   3
```

Think

- Find the middle point in list
- Divide and Conquer to build left child and right child node

Solution

```
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @return: a tree node
     */
    public TreeNode sortedListToBST(ListNode head) {
        // write your code here
        if(head == null)
            return null;
        if(head.next == null)
            return new TreeNode(head.val);

        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode runner = head;
        ListNode walker = dummy;
        while(runner!=null && runner.next!=null) {
            runner = runner.next.next;
            walker = walker.next;
        }

        ListNode m = walker.next;
        TreeNode root = new TreeNode(m.val);
        ListNode left = dummy.next;
        ListNode right = walker.next.next;
        walker.next = null;
        root.left = sortedListToBST(left);
        root.right = sortedListToBST(right);
        return root;
    }
}
```

Binary Tree

Binary Tree typically can be regarded as two categories:

- Binary Tree

Tree node contains its value and two children: left and right

```
class TreeNode{
    int val;
    TreeNode left, right;
}
```

- Binary Search Tree

Binary Search Tree is similar with the binary tree in structure, but just one special character -- left child's value always less than root, while the right child has greater value than root.

Routines on Binary Tree:

- Traversal
    - Pre-order
    - In-order
    - Post-order
    - Level order
- Depth of Tree
    - Maximum Depth
    - Minimum Depth
    - Balanced Tree Check
- Sub-tree Check
- Find a Node and return the Path from Root to target Node

- Insert a Node in Tree
- Delete a Node in Tree (Difficult)
- Construct tree by in-order and pre-order
- Construct tree by in-order and post-order

Routines on Binary Search Tree

- Search range in Binary Search Tree
- Build Binary Search Tree
  - By Array
  - By Linked List

Trick

- Design recursion function with `boolean` type return
- Consider the in-order when meet the Binary Search Tree problem

Tree Treversal

Pre-order

Example

Given binary tree {1,#,2,3}:

```
   1
    \
     2
    /
   3
```

return [1,2,3].

Solution

```
public class Solution{
    // Way one: recursion and divide conquer
    public List<Integer> preorder(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null)
            return res;
        res.add(root.val);
        res.addAll(preorder(root.left));
        res.addAll(preorder(root.right));
        return res;
    }

    // Way two: stack
    public List<Integer> preorder(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null)
            return res;
        Stack<TreeNode> stack = new Stack<>();
        while(root != null) {
            res.add(root.val);
            stack.push(root);
            root = root.left;
        }
        while(!stack.isEmpty()) {
            TreeNode pop = stack.pop();
            TreeNode extend = pop.right;
            while(extend != null){
                res.add(extend.val);
                stack.push(extend);
                extend = extend.left;
            }
        }
        return res;
    }

    // Way Three: concise and stack
    public List<Integer> preorder(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null)
            return res;

        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while(!stack.isEmpty()) {
            TreeNode pop = stack.pop();
            res.add(pop.val);
            if(pop.right != null)
                stack.push(pop.right);
            if(pop.left != null)
                stack.push(pop.left);
        }
        return res;
    }
}
```

In-order

Example

Given binary tree {1,#,2,3}:

```
   1
    \
     2
    /
   3
```

return [1,3,2].

Solution

```
public class Solution{
    // Way one: recursion and divide conquer
    public List<Integer> inorder(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null)
            return res;
        res.addAll(inorder(root.left));
        res.add(root.val);
        res.addAll(inorder(root.right));
        return res;
    }

    // Way two: stack
    public List<Integer> inorder(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null)
            return res;

        Stack<TreeNode> stack = new Stack<>();
        do{
            while(root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            res.add(root.val);
            if(root.right != null)
                root = root.right;
            else
                root = null;
        }while(!stack.isEmpty() || root != null);
        // NOTE the loop should be continue to do when root is n
ot null

        return res;
    }
}
```

Post-order

Example

Given binary tree {1,#,2,3}:

```
    1
     \
      2
     /
    3
```

return [3,2,1].

Solution

```
public class Solution{
    // Way one: recursion and divide conquer
    public List<Integer> postorder(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null)
            return res;
        res.addAll(postorder(root.left));
        res.addAll(postorder(root.right));
        res.add(root.val);
        return res;
    }

    // Way two: stack
    public List<Integer> inorder(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null)
            return res;

        Stack<TreeNode> stack = new Stack<>();
        do{
            // tricky part 1: push the right child (if exist) fi
rst then push the root
            while(root != null) {
                if(root.right != null)
                    stack.push(root.right);
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            // check if the right child exist and didn't be trav
eled
            if(!stack.isEmpty() && root.right != null && root.ri
ght == stack.peek()) {
                // swap the root and right child
                stack.pop();
                stack.push(root);
                root = root.right;
            }else {
                res.add(root.val);
                root = null;
            }
        }while(!stack.isEmpty());

        return res;
    }
}
```

Level Order

Example

Given binary tree {3,9,20,#,#,15,7},

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

return its level order traversal as:

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

Solution

```
public class Solution {
    /**
     * One queue
     * @param root: The root of binary tree.
     * @return: Level order a list of lists of integer
     */
    public ArrayList<ArrayList<Integer>> levelOrder(TreeNode roo
t) {
        // write your code here
        ArrayList<ArrayList<Integer>> res = new ArrayList<>();
        if(root == null)
            return res;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        int cur = 1;
        int next = 0;
        ArrayList<Integer> row = new ArrayList<>();
        while(!queue.isEmpty()) {
            TreeNode node = queue.remove();
            cur--;
            row.add(node.val);
            if(node.left!=null) {
                queue.offer(node.left);
                next++;
            }
            if(node.right!=null) {
                queue.offer(node.right);
                next++;
            }

            if(cur == 0) {
                cur = next;
                next = 0;
                res.add(new ArrayList<>(row));
                row = new ArrayList<>();
            }
        }
        return res;
    }


    /**
     * Use DFS algorithm
     * @param root: The root of binary tree.
     * @return: Level order a list of lists of integer
     */
    public ArrayList<ArrayList<Integer>> levelOrder(TreeNode roo
t) {
        // write your code here
        ArrayList<ArrayList<Integer>> res = new ArrayList<>();
        if(root == null)
            return res;
        DFSUtil(res, 0, root);
        return res;
    }

    private void DFSUtil(ArrayList<ArrayList<Integer>> res, int
level, TreeNode cur) {
        if(cur == null)
            return;
        if(level >= res.size()) {
            ArrayList<Integer> line = new ArrayList<>();
            line.add(cur.val);
            res.add(line);
        }else{
            ArrayList<Integer> line = res.get(level);
            line.add(cur.val);
            res.set(level, line);
        }
        DFSUtil(res, level + 1, cur.left);
        DFSUtil(res, level + 1, cur.right);
    }
}
```

Invert Binary Tree

Invert a binary tree.

Have you met this question in a real interview? Yes

Example

```
    1         1
   / \       / \
  2   3  => 3   2
     /       \
    4         4
```

Challenge

Do it in recursion is acceptable, can you do it without recursion?

Solution

```
public class Solution {
    /**
     * Recursion
     * @param root: a TreeNode, the root of the binary tree
     * @return: nothing
     */
    public void invertBinaryTree(TreeNode root) {
        if(root == null)
            return;

        TreeNode tmp = root.left;
        root.left = root.right;
        root.right = tmp;
        invertBinaryTree(root.left);
        invertBinaryTree(root.right);
    }

    /**
     * Non-recursion, it likes tree preorder traversal
     * @param root: a TreeNode, the root of the binary tree
     * @return: nothing
     */
    public void invertBinaryTree(TreeNode root) {
        if(root == null)
            return;
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while(!stack.isEmpty()){
            TreeNode node = stack.pop();
            TreeNode tmp = node.left;
            node.left = node.right;
            node.right = tmp;
            if(node.right != null)
                stack.push(node.right);
            if(node.left != null)
                stack.push(node.left);
        }
    }
}
```

Depth of Tree

1. Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example

Given a binary tree as follow:

```
  1
 / \
2   3
   / \
  4   5
```

The maximum depth is 3.

Think

- Recursively check the depth
- Only if the node is a leaf node we stop recursion and return 0
- We compare the recursion results from left branch and right branch to get the maximum value.

Solution

```
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int maxDepth(TreeNode root) {
        if(root == null)
            return 0;
        int left = maxDepth(root.left);
        int right = maxDepth(root.right);
        return Math.max(left, right) + 1;
    }
}
```

2. Minimum Depth of Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Example

Given a binary tree as follow:

```
        1

     /     \

   2        3

          /    \

        4       5
```

The minimum depth is 2

Think

- Recursively check the depth
- Only if the node is a leaf node we stop recursion and return 0
- If a node just have one child, we should not compare the recursion result!
- If a node contains both left and right child, we compare the recursion results to get the minimum value.

Solution

```
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int minDepth(TreeNode root) {
        // write your code here
        if(root == null)
            return 0;

        if(root.left == null)
            return minDepth(root.right) + 1;
        else if(root.right == null)
            return minDepth(root.left) + 1;
        else
            return Math.min(minDepth(root.left), minDepth(root.r
ight)) + 1;
    }
}
```

Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

Example

Given binary tree A={3,9,20,#,#,15,7}, B={3,#,20,15,7}

```
A)  3            B)    3
   / \                  \
  9  20                 20
    /  \                / \
   15   7              15  7
```

The binary tree A is a height-balanced binary tree, but B is not.

Think

- Get two max depths from left branch and right branch
- Recursive from bottom to top and Compare two max depth on each node
- If the difference between two depth is larger than one, regard it as non-balanced tree.

Solution

```
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: True if this Binary tree is Balanced, or false.
     */
    public boolean isBalanced(TreeNode root) {
        if(root == null)
            return true;
        // write your code here
        int left =  height(root.left);
        int right = height(root.right);
        if(Math.abs(left - right) <= 1)
            return isBalanced(root.left) && isBalanced(root.righ
t);
        return false;
    }

    private int height(TreeNode node) {
        if(node == null)
            return 0;
        return Math.max(height(node.left), height(node.right)) +
1;
    }
}
```

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.

Example

For the following binary tree:

```
  4
 / \
3   7
   / \
  5   6
```

LCA(3, 5) = 4

LCA(5, 6) = 7

LCA(6, 7) = 7

Think

- Recursively traversal on every tree node
- Once it touched `null` or target A node or target B node, it return it self
- Divide and conquer to get return from left branch and right branch
- Check if both return result are not null, means current node is the common ancestor
- If it is not, return the not null one or return null if both are null

Solution

```
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param A and B: two nodes in a Binary.
     * @return: Return the least common ancestor(LCA) of the two
 nodes.
     */
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode
 A, TreeNode B) {
        if(root == null || root == A || root == B)
            return root;
        TreeNode left = lowestCommonAncestor(root.left, A, B);
        TreeNode right = lowestCommonAncestor(root.right, A, B);
        if(left != null && right != null) {
            return root;
        }else{
            return left == null? right : left;
        }
    }
}
```

Search Range in Binary Search Tree

Given two values k1 and k2 (where k1 < k2) and a root pointer to a Binary Search Tree. Find all the keys of tree in range k1 to k2. i.e. print all x such that k1<=x<=k2 and x is a key of given BST. Return all the keys in ascending order.

Example

If k1 = 10 and k2 = 22, then your function should return [12, 20, 22].

```
    20
   /  \
  8   22
 / \
4   12
```

Think

- Recursion on each valid node.
- For invalid node, if it is less than k1, check its right child, while if it is larger than k2, check its left child
- Add the result from left and itself and right

Solution

```
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param k1 and k2: range k1 to k2.
     * @return: Return all keys that k1<=key<=k2 in ascending or
der.
     */
    public ArrayList<Integer> searchRange(TreeNode root, int k1,
int k2) {
        ArrayList<Integer> res = new ArrayList<>();
        if(root == null)
            return res;
        ArrayList<Integer> left = searchRange(root.left, k1, k2)
;
        ArrayList<Integer> right = searchRange(root.right, k1, k
2);
        // current value is less than k1, check its right child
        if(root.val < k1)
            return right;
        // current value is larger than k2, check its left child
        if(root.val > k2)
            return left;
        // add left branch first then itself and then right bran
ch
        res.addAll(left);
        res.add(root.val);
        res.addAll(right);
        return res;
    }
}
```

Count Univalue Subtrees

Given a binary tree, count the number of uni-value subtrees.

A Uni-value subtree means all nodes of the subtree have the same value.

For example:

Given binary tree,

```
      5
     / \
    1   5
   / \   \
  5   5   5
```

return `4`.

Think

- Typical Tree problem with recursion idea
- Four cases in recursion:
    - Leaf node: if true case and increase the counter
    - Node with on one child:
        - Left Only: check return value from left child and value of left child should be the same as current node.
        - Right Only: check return value from right child and value of right child should be the same as current node.
    - Node with two children: check return value from both side and both child node value should be the same as current node.

Solution

```
public class CountUnivalueSubtrees {
    public int countUnivalSubtrees(TreeNode root) {
        if (root == null)
            return 0;
        int[] cnt = new int[1];
        helper(root, cnt);
        return cnt[0];
    }

    private boolean helper(TreeNode node, int[] cnt) {
        if (node.left == null && node.right == null) {
            cnt[0]++;
            return true;
        } else if (node.left == null) {
            if (helper(node.right, cnt) && node.right.val == nod
e.val) {
                cnt[0]++;
                return true;
            } else
                return false;
        } else if (node.right == null) {
            if (helper(node.left, cnt) && node.left.val == node.
val) {
                cnt[0]++;
                return true;
            } else
                return false;
        } else {
            if (helper(node.left, cnt) && helper(node.right, cnt
)
                    && node.left.val == node.val && node.right.v
al == node.val) {
                cnt[0]++;
                return true;
            } else
                return false;
        }
    }
}
```

Verify Preorder Sequence in Binary Search Tree

Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree.

You may assume each number in the sequence is unique.

Follow up

Could you do it using only constant space complexity?

Think #1

- The first element should be the root node.
- Find the bound that all previous element are small than root value by checking the first larger element.
- So the left of this bound should be the left tree of root, and the rest of it should be the right tree of root.
- Check left and right recursively.
- Time Complexity: $O(n^2)$, Space Complexity: $O(n)$

Solution #1

```
    public boolean verifyPreorder(int[] preorder) {
        if (preorder == null || preorder.length <= 1)
            return true;
        return helper(preorder, 0, preorder.length - 1);
    }

    private boolean helper(int[] preorder, int l, int r) {
        int root = preorder[l];
        int divide = l;
        for (int i = l + 1; i <= r; i++) {
            if (preorder[i] < root && divide != l)
                return false;
            else if (preorder[i] > root && divide != l)
                divide = i;
        }
        return helper(preorder, l + 1, divide - 1)
                && helper(preorder, divide, r);
    }
```

Think #2

- Preorder in BST has a regular pattern:
  - When going to left node, it must be a descending order
  - When going to right node, it should be a ascending order
- Setting a stack, to store the previous path. Iterate throught the array:
  - When it getting smaller element make it just push into stack
  - When it find the larger element (larger than the peek of stack), pop the stack and set the minimum limit as the value of popped element.
- Time Complexity: $O(n)$, Space Complexity: $O(n)$
- For Example, 10 5 2 7 6 8 12 11 -> BST

```
          10
        /    \
      5       12
     / \      /
    2   7   11
       / \
      6   8
```

The procedure in Stack:

```
10 -> 10 5 -> 10 5 2 -> 10 7 (min=5) -> 10 7 6 (min = 5) ->
10 8 (min = 7)
-> 12 (min = 10) -> 12 11 (min = 10)
```

Solution #2

```
    public boolean verifyPreorderII(int[] preorder) {

        Stack<Integer> stack = new Stack<Integer>();
        int min = Integer.MIN_VALUE;
        for (int num : preorder) {
            if (num < min)
                return false;
            while (!stack.isEmpty() && num > stack.peek())
                min = stack.pop();
            stack.push(num);
        }
        return true;
    }
```

Think #3

- Optimized on the #2 solution
- Use a pointer to replace the stack peek position.
- 指针模拟栈

Solution #3

```
    public boolean verifyPreorderII(int[] preorder) {

        int idx = -1;
        int min = Integer.MIN_VALUE;
        for (int num : preorder) {
            if (num < min)
                return false;
            while (idx >= 0 && num > preorder[idx]) {
                min = preorder[idx--];
            }
            preorder[++idx] = num;
        }
        return true;
    }
```

Closest Binary Search Tree Value

Problem I

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note

Given target value is a floating point. You are guaranteed to have only one unique value in the BST that is closest to the target.

Solution

```
public class ClosestBSTValue {
    double min = Double.MAX_VALUE;
    TreeNode closest = null;

    public int closestValue(TreeNode root, double target) {
        if (root == null) {
            return Integer.MAX_VALUE;
        }
        helper(root, target);
        return closest.val;
    }

    private void helper(TreeNode node, double target) {
        if (node == null)
            return;
        if (Math.abs((double) node.val - target) < min) {
            min = Math.abs((double) node.val - target);
            closest = node;
        }

        if((double) node.val > target) {
            helper(node.left, target);
        }else{
            helper(node.right, target);
        }
    }
}
```

Problem II

Given a non-empty binary search tree and a target value, find **k values** in the BST that are closest to the target.

Note

- Given target value is a floating point.

- You may assume `k` is always valid, that is: `k ≤ total` nodes.
- You are guaranteed to have only one unique set of `k` values in the BST that are closest to the target.

Follow up

Assume that the BST is balanced, could you solve it in less than $O(n)$ runtime (where n = total nodes)?

Hint

- Consider implement these two helper functions:
  - `getPredecessor(N)` , which returns the next smaller node to N.
  - `getSuccessor(N)` , which returns the next larger node to N.
- Try to assume that each node has a parent pointer, it makes the problem much easier.
- Without parent pointer we just need to keep track of the path from the root to the current node using a stack.
- You would need two stacks to track the path in finding predecessor and successor node separately.

Think #1

- The straight-forward solution would be to use a **heap**.
- We just treat the BST just as a usual array and do a in-order traverse.
- Then we compare the current element with the minimum element in the heap, the same as top k problem.

Solution #1

```
class Solution {
    private PriorityQueue<Integer> minPQ;
    private int count = 0;
    public List<Integer> closestKValues(TreeNode root, double ta
rget, int k) {
        minPQ = new PriorityQueue<;Integer>(k);
        List<Integer> result = new ArrayList<Integer>();
        inorderTraverse(root, target, k);

        // Dump the pq into result list
        for (Integer elem : minPQ) {
            result.add(elem);
        }

        return result;
    }

    private void inorderTraverse(TreeNode root, double target, i
nt k) {
        if (root == null) {
            return;
        }
        // recursive way
        inorderTraverse(root.left, target, k);

        if (count < k) {
            minPQ.offer(root.val);
        } else {
            if (Math.abs((double) root.val - target) < Math.abs((
double) minPQ.peek() - target)) {
                minPQ.poll();
                minPQ.offer(root.val);
            }
        }
        count++;

        inorderTraverse(root.right, target, k);
    }
}
```

Think #2

- Stack solution to replace the recursion way for inorder traversal
- Maintain a queue with K size, compare the head of queue and current value

  to make sure the minimum difference.
- When the head of queue has less difference to target than current node, stop iterate.

Solution #2

```
    public List<Integer> closestKValuesII(TreeNode root, double
target, int k) {
        Queue<Integer> values = new LinkedList<>();

        // build inorder traversal
        Stack<TreeNode> stack = new Stack<>();
        do {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();

            // compare and insert to the queue
            if(values.size() < k) {
                values.add(root.val);
            }else{
                int pop = values.peek();
                if(Math.abs((double)root.val - target) < Math.ab
s((double)pop - target)) {
                    values.poll();
                    values.add(root.val);
                }else
                    break;
            }

            if(root.right !=null) {
                root = root.right;
            }else
                root = null;
        } while (!stack.isEmpty());
        return new ArrayList<Integer>(values);
    }
```

Route Between Directed Graph

Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

Think

- Most typical Graph algorithm question!
- Try two ways: DFS, BFS.

Solution

```
public class Solution {
   /**
     * @param graph: A list of Directed graph node
     * @param s: the starting Directed graph node
     * @param t: the terminal Directed graph node
     * @return: a boolean value
     */

     // BFS
     public boolean hasRoute(ArrayList<DirectedGraphNode> graph,

                            DirectedGraphNode s, DirectedGraphNo
de t) {

        if(s == t)
            return true;

        Queue<DirectedGraphNode> queue = new LinkedList<>();
        queue.offer(s);
        graph.remove(s);
        while(!queue.isEmpty()) {
            DirectedGraphNode cur = queue.remove();
               graph.remove(cur);
            for(DirectedGraphNode next : cur.neighbors) {
                if(!graph.contains(next))
                    continue;
                if(next == t)
                    return true;
                queue.offer(next);
            }
        }
        return false;
    }


    // DFS
    public boolean hasRoute(ArrayList<DirectedGraphNode> graph,
                            DirectedGraphNode s, DirectedGraphNo
de t) {
        // write your code here
        if(s == t)
            return true;

        for(DirectedGraphNode next : s.neighbors) {
            if(hasRoute(graph, next, t))
                return true;
        }
        return false;
    }
}
```

Graph Valid Tree

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

Example

Given n = `5` and edges = `[[0, 1], [0, 2], [0, 3], [1, 4]]`, return true.
Given n = `5` and edges = `[[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]]`, return false.

Hint

The definition of tree on Wikipedia:

> a tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.

Note: You can assume that no duplicate edges will appear in edges. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]` and thus will not appear together inedges.

Think

Given n = `5` and edges = `[[0, 1], [1, 2], [3, 4]]`, what should your return? Is this case a valid tree?

No, isolate node shouldn't be allowed.

Design a Node class:

```
    private class Node {
        int val;
        List<Integer> neighbors;

        public Node(int val) {
            this.val = val;
            this.neighbors = new ArrayList<>();
        }
    }
```

Solution #BFS

```
public boolean validTree(int n, int[][] edges) {
    Node[] nodes = new Node[n];
    for (int i = 0; i < nodes.length; i++)
        nodes[i] = new Node(i);
    for (int[] edge : edges) {
        nodes[edge[0]].neighbors.add(edge[1]);
        nodes[edge[1]].neighbors.add(edge[0]);
    }

    boolean[] visited = new boolean[n];
    Queue<Integer> queue = new LinkedList<Integer>();
    queue.offer(0);

    while (!queue.isEmpty()) {
        int vertexId = queue.poll();
        // touch the cycle
        if (visited[vertexId])
            return false;

        visited[vertexId] = true;
        for (int neighbor : nodes[vertexId].neighbors) {
            if (!visited[neighbor])
                queue.offer(neighbor);
        }
    }

    // Check the isolate
    for (boolean v : visited) {
        if (!v)
            return false;
    }
    return true;
}
```

Solution #DFS

```
    public boolean validTreeDFS(int n, int[][] edges) {
        Node[] nodes = new Node[n];
        for (int i = 0; i < nodes.length; i++)
            nodes[i] = new Node(i);
        for (int[] edge : edges) {
            nodes[edge[0]].neighbors.add(edge[1]);
            nodes[edge[1]].neighbors.add(edge[0]);
        }

        // all node should connected from zero
        boolean[] visited = new boolean[n];
        if (!dfsHelper(nodes, visited, 0, -1))
            return false;

        // Check the isolate
        for (boolean v : visited) {
            if (!v)
                return false;
        }
        return true;
    }

    private boolean dfsHelper(Node[] nodes, boolean[] visited, i
nt idx,
            int parentIdx) {
        if (visited[idx])
            return false;
        visited[idx] = true;
        for (int i = 0; i < nodes[idx].neighbors.size(); i++) {
            if (nodes[idx].neighbors.get(i) == parentIdx)
                continue;
            if (!dfsHelper(nodes, visited, nodes[idx].neighbors.
get(i), idx))
                return false;
        }
        return true;
    }
```

Alien Dictionary

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of words from the dictionary, where words are sorted lexicographically by the rules of this new language. Derive the order of letters in this language.

Example

Given the following words in dictionary, `[ "wrt", "wrf", "er", "ett", "rftt" ]` The correct order is: `"wertf"`.

Note

- You may assume all letters are in lowercase.
- If the order is invalid, return an empty string.
- There may be multiple valid order of letters, return any one of them is fine.

Think

- Typical topological problem

Solution

```
    public static String alienOrder(String[] words) {
        // build up the node map, find node according to the char

        HashMap<Character, Node> map = new HashMap<>();
        // iterate through all provided words
        for (String str : words) {
            // read each word and learn their order
            map.put(str.charAt(0),
                    map.containsKey(str.charAt(0)) ? map.get(str
.charAt(0))
```

```
                             : new Node(str.charAt(0)));
              for (int i = 1; i < str.length(); i++) {
                  char cur = str.charAt(i);
                  // ignore the adjacent equal characters
                  if(cur == str.charAt(i-1))
                      continue;
                  Node node = map.containsKey(cur) ? map.get(cur)
: new Node(cur);
                  Node prev = map.get(str.charAt(i - 1));
                  // make current node indegree plus one only if t
he previous node doesn't have current node in its neighborhood l
ist
                  if (!prev.neighbors.contains(node) ) {
                      node.indegree++;
                      map.get(str.charAt(i - 1)).neighbors.add(nod
e);
                  }
                  map.put(cur, node);
              }
          }

          // find the node with zero indegree
          Queue<Node> queue = new LinkedList<>();
          for (Node node : map.values())
              if (node.indegree == 0)
                  queue.offer(node);
          // build the final string,
          StringBuilder sb = new StringBuilder();
          // each time pop the node with zero indegree
          // reduce their neighbor's indegree and push node when i
t has zero indegree
          while (!queue.isEmpty()) {
              Node node = queue.remove();
              sb.append(node.c);
              map.remove(node.c);
              for (Node nb : node.neighbors) {
                  nb.indegree--;
                  if (nb.indegree == 0 && map.containsKey(nb.c))
                      queue.offer(nb);
              }
```

```
        }
        // if map has any entry means the cycle existed
        if (map.size() > 0)
            return "";

        return sb.toString();
    }

    private static class Node {
        char c;
        int indegree;
        List<Node> neighbors;

        public Node(char c) {
            this.c = c;
            this.indegree = 0;
            this.neighbors = new ArrayList<>();
        }
    }
```

The structure of Segment Tree is a binary tree which each node has two attributes `start` and `end` denote an segment / interval.

start and end are both integers, they should be assigned in following rules:

- The root's start and end is given by `build` method.
- The left child of node A has `start=A.left, end=(A.left + A.right) / 2`.
- The right child of node A has `start=(A.left + A.right) / 2 + 1, end=A.right`.
- if start equals to end, there will be no children for this node.

Implement a `build` method with two parameters start and end, so that we can create a corresponding segment tree with every node has the correct start and end value, return the root of this segment tree.

Example

Given start=0, end=3. The segment tree will be:

```
               [0,  3]
             /        \
      [0,  1]           [2, 3]
      /     \           /     \
   [0, 0]  [1, 1]     [2, 2]  [3, 3]
```

Given start=1, end=6. The segment tree will be:

```
               [1,  6]
             /        \
      [1,  3]           [4,  6]
      /     \           /     \
   [1, 2]  [3,3]     [4, 5]   [6,6]
   /    \           /     \
[1,1]   [2,2]     [4,4]   [5,5]
```

Clarification

Segment Tree (a.k.a Interval Tree) is an advanced data structure which can support queries like:

- which of these intervals contain a given point
- which of these points are in a given interval

Solution

```
public class SegmentTreeNode {
    public int start, end, max;
    public SegmentTreeNode left, right;
    public SegmentTreeNode(int start, int end, int max) {
        this.start = start;
        this.end = end;
        this.max = max
        this.left = this.right = null;
    }
}

public class Solution {
    /**
     *@param start, end: Denote an segment / interval
     *@return: The root of Segment Tree
     */
    public SegmentTreeNode build(int start, int end) {
        if(end < start)
            return null;
        SegmentTreeNode node = new SegmentTreeNode(start, end);
        if(start == end)
            return node;
        int m = left + ((right - left)>>1);
        node.left = build(start, m);
        node.right = build(m+1,end);
        return node;
    }
}
```

The structure of Segment Tree is a binary tree which each node has two attributes `start` and `end` denote an segment / interval.

start and end are both integers, they should be assigned in following rules:

- The root's start and end is given by `build` method.
- The left child of node A has `start=A.left, end=(A.left + A.right) / 2`.
- The right child of node A has `start=(A.left + A.right) / 2 + 1, end=A.right`.
- if start equals to end, there will be no children for this node.

Implement a build method with a given array, so that we can create a corresponding segment tree with every node value represent the corresponding interval max value in the array, return the root of this segment tree.

Example

Given [3,2,1,4]. The segment tree will be:

```
                 [0,  3] (max = 4)
                  /            \
        [0,  1] (max = 3)     [2, 3]  (max = 4)
        /        \                /             \
[0, 0](max = 3)  [1, 1](max = 2)[2, 2](max = 1) [3, 3] (max = 4)
```

Clarification

Segment Tree (a.k.a Interval Tree) is an advanced data structure which can support queries like:

- which of these intervals contain a given point
- which of these points are in a given interval

Solution

```
public class Solution {
    /**
     *@param A: a list of integer
     *@return: The root of Segment Tree
     */
    public SegmentTreeNode build(int[] A) {
        return builder(A, 0, A.length - 1);
    }

    private SegmentTreeNode builder(int[] A, int left, int right
) {
        if(left > right)
            return null;
        if(left == right)
            return new SegmentTreeNode(left, right, A[left]);
        int m = left + ((right - left)>>1);
        SegmentTreeNode leftNode = builder(A, left, m);
        SegmentTreeNode rightNode = builder(A, m+1, right);
        SegmentTreeNode node = new SegmentTreeNode(left, right,
Math.max(leftNode.max, rightNode.max));
        node.left = leftNode;
        node.right = rightNode;
        return node;
    }
}
```

For an integer array (index from 0 to n-1, where n is the size of this array), in the corresponding SegmentTree, each node stores an extra attribute max to denote the maximum number in the interval of the array (index from start to end).

Design a query method with three parameters root, start and end, find the maximum number in the interval [start, end] by the given root of segment tree.

Example

For array [1, 4, 2, 3], the corresponding Segment Tree is:

```
                  [0, 3, max=4]
                 /             \
          [0,1,max=4]        [2,3,max=3]
          /         \        /         \
   [0,0,max=1] [1,1,max=4] [2,2,max=2], [3,3,max=3]
```

`query(root, 1, 1)`, return `4`

`query(root, 1, 2)`, return `4`

`query(root, 2, 3)`, return `3`

`query(root, 0, 2)`, return `4`

Note

It is much easier to understand this problem if you finished Segment Tree Build first.

Solution:

```
public class SegmentTreeNode {
    public int start, end, max;
    public SegmentTreeNode left, right;
    public SegmentTreeNode(int start, int end, int max) {
        this.start = start;
        this.end = end;
        this.max = max
        this.left = this.right = null;
    }
}

public class Solution {
    /**
     *@param root, start, end: The root of segment tree and
     *                         an segment / interval
     *@return: The maximum number in the interval [start, end]
     */
    public int query(SegmentTreeNode root, int start, int end) {
        if(root == null)
            return Integer.MIN_VALUE;

        if(root.start >= start && root.end <= end)
            return root.max;
        int m = root.start + ((root.end - root.start)>>1);
        if(start > m)
            return query(root.right, start, end);
        else if(end <= m)
            return query(root.left, start, end);
        else
            return Math.max(query(root.left, start, m), query(ro
ot.right, m+1, end));
    }
}
```

For an array, we can build a SegmentTree for it, each node stores an extra attribute count to denote the number of elements in the the array which value is between interval start and end. (The array may not fully filled by elements)

Design a query method with three parameters root, start and end, find the number of elements in the in array's interval [start, end] by the given root of value SegmentTree.

Example

For array [0, 2, 3], the corresponding value Segment Tree is:

```
                       [0, 3, count=3]
                       /             \
            [0,1,count=1]             [2,3,count=2]
            /         \               /            \
     [0,0,count=1] [1,1,count=0] [2,2,count=1], [3,3,count=1]
```

`query(1, 1)`, return `0`

`query(1, 2)`, return `1`

`query(2, 3)`, return `2`

`query(0, 2)`, return `2`

Note

It is much easier to understand this problem if you finished Segment Tree Buildand Segment Tree Query first.

Solution

```
public class SegmentTreeNode {
    public int start, end, max;
    public SegmentTreeNode left, right;
    public SegmentTreeNode(int start, int end, int max) {
        this.start = start;
        this.end = end;
        this.max = max
        this.left = this.right = null;
    }
}

public class Solution {
    /**
     *@param root, start, end: The root of segment tree and
     *                         an segment / interval
     *@return: The count number in the interval [start, end]
     */
    public int query(SegmentTreeNode root, int start, int end) {
        // write your code here
        if(root == null)
            return 0;

        if(root.start >= start && root.end <= end)
            return root.count;

        int m = root.start + ((root.end - root.start)>>1);

        if(start > m)
            return query(root.right, start, end);
        else if(end <= m)
            return query(root.left, start, end);
        else
            return query(root.left, start, m) + query(root.right
, m+1, end);
    }
}
```

For a Maximum Segment Tree, which each node has an extra value max to store the maximum value in this node's interval.

Implement a modify function with three parameter root, index and value to change the node's value with [start, end] = [index, index] to the new given value. Make sure after this change, every node in segment tree still has the max attribute with the correct value.

Example

For segment tree:

```
                      [1, 4, max=3]
                    /                \
         [1, 2, max=2]                [3, 4, max=3]
        /              \             /             \
 [1, 1, max=2], [2, 2, max=1], [3, 3, max=0], [4, 4, max=3]
```

if call modify(root, 2, 4), we can get:

```
                      [1, 4, max=4]
                    /                \
         [1, 2, max=4]                [3, 4, max=3]
        /              \             /             \
 [1, 1, max=2], [2, 2, max=4], [3, 3, max=0], [4, 4, max=3]
```

or call modify(root, 4, 0), we can get:

```
                      [1, 4, max=2]
                    /                \
         [1, 2, max=2]                [3, 4, max=0]
        /              \             /             \
 [1, 1, max=2], [2, 2, max=1], [3, 3, max=0], [4, 4, max=0]
```

Note

We suggest you finish problem Segment Tree Build and Segment Tree Query first.

Challenge

Do it in O(h) time, h is the height of the segment tree.

Solution:

```
public class Solution {
    /**
     *@param root, index, value: The root of segment tree and
     *@ change the node's value with [index, index] to the new g
iven value
     *@return: void
     */
    public void modify(SegmentTreeNode root, int index, int valu
e) {
        if(root.start == root.end) {
            root.max = value;
            return;
        }
        int m = root.start + ((root.end - root.start)>>1);
        if(m < index)
            modify(root.right, index, value);
        else
            modify(root.left, index, value);

        // update max each time, this is important
        root.max = Math.max(root.left.max, root.right.max);
    }
}
```

Given an integer array (index from 0 to n-1, where n is the size of this array), and an query list. Each query has two integers [start, end]. For each query, calculate the sum number between index start and end in the given array, return the result list.

Example

For array `[1,2,7,8,5]`, and queries `[(0,4),(1,2),(2,4)]`, return `[23,9,20]`

Challenge

O(logN) time for each query

Solution

```
// Segment tree for sum
class IntervalTree{
        IntervalNode root;

        public IntervalTree(int[] A) {
            root = build(A, 0, A.length - 1);
        }

        private IntervalNode build(int[] A, int start, int end)
{
            if(start > end)
                return null;
            IntervalNode node = new IntervalNode(start, end);

            if(start == end) {
                node.val = (long)A[start];
                return node;
            }

            int m = start + ((end - start)>>1);
            node.left = build(A, start, m);
            node.right = build(A, m+1, end);
            node.val = node.left.val + node.right.val;
            return node;
        }

        public long query(int start, int end) {
            return queryhelper(root, start, end);
        }

        private long queryhelper(IntervalNode node, int start, i
nt end) {
            if(start <= node.start && end >= node.end)
                return node.val;

            int m = node.start + ((node.end - node.start)>>1);
            if(m < start) {
                return queryhelper(node.right, start, end);
            }else if(m >= end){
                return queryhelper(node.left, start, end);
            }else
                return queryhelper(node.left, start, m) + queryh
elper(node.right, m+1, end);
        }

        private class IntervalNode {
            int start, end;
            long val;
            IntervalNode left, right;
            IntervalNode(int start, int end) {
                this.start = start;
                this.end = end;
            }
        }
}

public class Solution {
    /**
     *@param A, queries: Given an integer array and an query lis
t
     *@return: The result list
     */
    public ArrayList<Long> intervalSum(int[] A,
                                       ArrayList<Interval> queri
es) {
        IntervalTree tree = new IntervalTree(A);
        ArrayList<Long> res = new ArrayList<>();
        for(Interval interval : queries) {
            res.add(tree.query(interval.start, interval.end));
        }
        return res;
    }
}
```

Given an integer array in the construct method, implement two methods query(start, end) and modify(index, value):

- For query(start, end), return the sum from index start to index end in the given array.
- For modify(index, value), modify the number in the given index to value.

Example

Given array A = `[1,2,7,8,5]`.

- `query(0, 2)`, return `10`.
- `modify(0, 4)`, change `A[0]` from `1` to `4`.
- `query(0, 1)`, return `6`.
- `modify(2, 1)`, change `A[2]` from `7` to `1`.
- `query(2, 4)`, return `14`.

Challenge

O(logN) time for query and modify.

Solution:

```
public class Solution {
    /* you may need to use some attributes here */

    IntervalNode root;
    /**
     * @param A: An integer array
     */
    public Solution(int[] A) {
        root = build(A, 0, A.length - 1);
    }

    private IntervalNode build(int[] A, int start, int end) {
            if(start > end)
                return null;
            IntervalNode node = new IntervalNode(start, end);

            if(start == end) {
                node.val = (long)A[start];
                return node;
            }

            int m = start + ((end - start)>>1);
            node.left = build(A, start, m);
            node.right = build(A, m+1, end);
            node.val = node.left.val + node.right.val;
            return node;
    }

    /**
     * @param start, end: Indices
     * @return: The sum from start to end
     */
    public long query(int start, int end) {
        return query(root, start, end);
    }

    private long query(IntervalNode node, int start, int end) {
        if(start <= node.start && end >= node.end)
            return node.val;

        int m = node.start + ((node.end - node.start)>>1);
        if(m < start) {
            return query(node.right, start, end);
        }else if(m >= end){
            return query(node.left, start, end);
        }else
            return query(node.left, start, m) + query(node.right
, m+1, end);
    }

    /**
     * @param index, value: modify A[index] to value.
     */
    public void modify(int index, int value) {
        modify(root, index, value);
    }

    private void modify(IntervalNode node, int index, int value)
 {
        if(node.start == node.end) {
            node.val = value;
            return;
        }

        int m = node.start + ((node.end - node.start)>>1);
         if(m < index)
            modify(node.right, index, value);
        else
            modify(node.left, index, value);
        node.val = node.right.val + node.left.val;
    }



    private class IntervalNode {
        int start, end;
        long val;
        IntervalNode left, right;
        IntervalNode(int start, int end) {
            this.start = start;
            this.end = end;
        }
    }
}
```

Given an integer array (index from 0 to n-1, where n is the size of this array), and an query list. Each query has two integers [start, end]. For each query, calculate the minimum number between index start and end in the given array, return the result list.

Example

For array `[1,2,7,8,5]`, and queries `[(1,2),(0,4),(2,4)]`, return `[2,1,5]`

Challenge

O(logN) time for each query (Segment Tree)

Solution

```
public class Solution {
    /**
     *@param A, queries: Given an integer array and an query lis
t
     *@return: The result list
     */
    public ArrayList<Integer> intervalMinNumber(int[] A,
                                                ArrayList<Interv
al> queries) {
        IntervalTree tree = new IntervalTree(A);
        ArrayList<Integer> res = new ArrayList<>();
        for(Interval interval : queries) {
            res.add(tree.query(interval.start, interval.end));
        }
        return res;
    }
}

/**
* Build Interval Tree with Min of segment!
*
```

```
*/
class IntervalTree{
        IntervalNode root;

        public IntervalTree(int[] A) {
            root = build(A, 0, A.length - 1);
        }

        private IntervalNode build(int[] A, int start, int end)
{
            if(start > end)
                return null;
            IntervalNode node = new IntervalNode(start, end);

            if(start == end) {
                node.min = A[start];
                return node;
            }

            int m = start + ((end - start)>>1);
            node.left = build(A, start, m);
            node.right = build(A, m+1, end);
            node.min = Math.min(node.left.min, node.right.min);
            return node;
        }

        public int query(int start, int end) {
            return queryhelper(root, start, end);
        }

        private int queryhelper(IntervalNode node, int start, in
t end) {
            if(start <= node.start && end >= node.end)
                return node.min;

            int m = node.start + ((node.end - node.start)>>1);
            if(m < start) {
                return queryhelper(node.right, start, end);
            }else if(m >= end){
                return queryhelper(node.left, start, end);
            }else
                return Math.min(queryhelper(node.left, start, m)
, queryhelper(node.right, m+1, end));
        }

        private class IntervalNode {
            int start, end, min;
            IntervalNode left, right;
            IntervalNode(int start, int end) {
                this.start = start;
                this.end = end;
            }
        }
}
```

Count of Smaller Number before itself

Give you an integer array (index from 0 to n-1, where n is the size of this array, value from 0 to 10000) . For each element Ai in the array, count the number of element before this element Ai is smaller than it and return count number array.

Example

For array `[1,2,7,8,5]` , return `[0,1,2,3,2]`

Think

- Segment Tree
- Initial with the range (0 to 10000)
- Count array elements included in a certain tree node
- Dynamic count and make a query.
    - Query a value, evaluate the value and node's middle value,
    - if larger, that means the left node's count should be included and also enter into the right node;
    - if less, just enter the left node;
    - recursive until touch the null node;

Solution

```
class SegmentTree{
    Node root;

    public SegmentTree(){
        root = build(0, 20000);
    }

    public Node build(int left, int right) {
        if(right < left)
            return null;
        if(left == right)
            return new Node(left, right);

        int m = left + ((right - left)>>1);
        Node cur = new Node(left, right);
        cur.leftNode = build(left, m);
        cur.rightNode = build(m+1, right);
        return cur;
    }

    public void count(int val){
        count(root, val);
    }

    private void count(Node node, int val) {
        if(node == null)
            return;
        int m = node.left + ((node.right - node.left)>>1);
        node.cnt++;
        if(val > m)
            count(node.rightNode, val);
        else
            count(node.leftNode, val);
    }

    public int query(int val){
        return query(root, val);
    }

    private int query(Node node, int val) {
        int cnt = 0;
        if(node == null)
            return cnt;
        int m = node.left + ((node.right - node.left)>>1);
        cnt += (node.leftNode != null ? node.leftNode.cnt : 0);
        if(val > m) // if larger:
            return cnt + query(node.rightNode, val);
        else // if less or equal: includes val == m
            return query(node.leftNode, val);
    }
```

```
}

class Node{
    int left, right;
    long cnt;
    Node leftNode, rightNode;
    public Node(int left, int right){
        this.cnt = 0;
        this.left = left;
        this.right = right;
    }
}


public class Solution {
   /**
     * @param A: An integer array
     * @return: Count the number of element before this element
'ai' is
     *          smaller than it and return count number array
     */
    SegmentTree tree;
    public ArrayList<Integer> countOfSmallerNumberII(int[] A) {
        tree = new SegmentTree();
        ArrayList<Integer> res = new ArrayList<>();
        for(int i = 0; i < A.length; i++) {
            tree.count(A[i]);
            res.add(tree.query(A[i]));
        }
        return res;
    }

}
```

Integer to Roman

Given an integer, convert it to a roman numeral.

The number is guaranteed to be within the range from `1` to `3999`.

Example

`4` -> `IV`

`12` -> `XII`

`21` -> `XXI`

`99` -> `XCIX`

Think

- Store special integer value and its corresponding roman value;
- Get a loop to minus the integer value from large to small (if `n` greater than that integer value), then get index and the corresponding roman value;

Solution:

```
public class Solution {
    /**
     * @param n The integer
     * @return Roman representation
     */
    public String intToRoman(int n) {
        //
        int[] numTab = {1,4,5,9,10,40,50,90,100,400,500,900,1000
};
        String[] romanTab = {"I", "IV", "V", "IX", "X", "XL", "L"
, "LC", "C", "CD", "D", "DM","M"};

        StringBuilder sb = new StringBuilder();
        for(int i = numTab.length - 1; i >= 0; i--) {
            while(n >= numTab[i]) {
                sb.append(romanTab[i]);
                n -= numTab[i];
            }
        }
        return sb.toString();
    }
}
```

Roman to Integer

Given a roman numeral, convert it to an integer.

The answer is guaranteed to be within the range from 1 to 3999.

Example

`IV` -> `4`

`XII` -> `12`

`XXI` -> `21`

`XCIX` -> `99`

Clarification

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1,000

- 同一数码最多只能出现三次，如40不可表示为XXXX，而要表示为XL。
- 在较大的罗马数字的右边记上较小的罗马数字，表示大数字加小数字。
- 在较大的罗马数字的左边记上较小的罗马数字，表示大数字减小数字。
- 左减的数字有限制，仅限于I、X、C。比如45不可以写成VL，只能是XLV。
- 但是，左减时不可跨越一个位数。比如，99不可以用IC（100 - 1）表示，是用XCIX（[100 - 10] + [10 - 1]）表示。
- 左减数字必须为一位，比如8写成VIII，而非IIX。
- 右加数字不可连续超过三位，比如14写成XIV，而非XIIII。（见下方“数码限

制”一项。）

Solution

```
public class Solution {
    /**
     * @param s Roman representation
     * @return an integer
     */
    public int romanToInt(String s) {
        // Use a hashmap to store roman-integer pair
        HashMap<Character, Integer> map = new HashMap<>();
        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 res = 0;
        for (int i = 0; i < s.length() ; i++) {
            if(i < s.length() - 1 && map.get(s.charAt(i)) < map.
get(s.charAt(i+1)) )
                res -= map.get(s.charAt(i));
            else
                res += map.get(s.charAt(i));
        }

        return res;
    }
}
```

Ugly Number I

Write a program to check whether a given number is an ugly number.

Ugly numbers are positive numbers whose prime factors only include `2, 3, 5`. For example, `6, 8` are ugly while `14` is not ugly since it includes another prime factor `7`.

Note that 1 is typically treated as an ugly number.

Think

- Make sure the value can be divided exactly by the divisor in array `2, 3, 5`;
- So iterate the division while the value can be divided exactly, otherwise change another divisor from array.

Solution

```
public class Solution {
    public boolean isUgly(int num) {
        if(num<=0)
            return false;
        int[] factors = {2,3,5};
        for(int i = factors.length - 1; i >= 0; i--) {
            while(num % factors[i] == 0) {
                num /= factors[i];
            }
        }
        return num == 1;
    }
}
```

Ugly Number II

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

Ugly numbers are positive numbers whose prime factors only include `2, 3, 5`. For example, `1, 2, 3, 4, 5, 6, 8, 9, 10, 12` is the sequence of the first `10` ugly numbers.

Note that `1` is typically treated as an ugly number.

Think

- Declare an array for ugly numbers: ugly[150]
- Initialize first ugly no: ugly[1] = 1
- Initialize three array index variables t2, t3, t5 to point to 1st element of the ugly array:

```
  i2 = i3 = i5 = 1;
```

- Initialize 3 choices for the next ugly no:

```
   next_mulitple_of_2 = ugly[i2]*2;
   next_mulitple_of_3 = ugly[i3]*3
   next_mulitple_of_5 = ugly[i5]*5;
```

- Choose the minimum from the aboved 3 choices as the next ugly number.
- Check which choice and increase that index.

Solution

```
public class Solution {
    public long nthUglyNumber(int k) {
        long[] memo = new long[k + 1];
            memo[1] = 1;
            int t2 = 1, t3 = 1, t5 = 1;
            for(int i = 2; i <= k; i++) {
                memo[i] = Math.min(memo[t2]*2, Math.min(memo[t3]*
3, memo[t5]*5));
                if(memo[i] == memo[t2]*2)
                    t2++;
                if(memo[i] == memo[t3]*3)
                    t3++;
                if(memo[i] == memo[t5]*5)
                    t5++;
            }
            return memo[k];
    }
}
```

Fast Power

Calculate the where a, b and n are all 32bit integers.

Example:

For 231 % 3 = 2 For 1001000 % 1000 = 0

Challenge:

O(logn)

Think

- Divide and Conquer
- Think about the two basic condition: n is 1 and n is 0;
- Each time we divide the n into two part (n/2);
- Then we got the combine value (divide * divide) from both parts (they're eqaul, actually);
- While n is odd, we need to add one more "a" time (*a);

Solution

```
class Solution {
    /*
     * @param a, b, n: 32bit integers
     * @return: An integer
     */
    public int fastPower(int a, int b, int n) {

        if(n == 1)
            return a % b;

        if(n == 0)
            return 1 % b;

        long divide = fastPower(a, b, n/2);
        long combine = divide * divide;
        combine %= b;
        if(n % 2 == 1)
            combine *= (long)a;
        combine %= b;
        return (int)combine;
    }
}
```

Sqrt(x)

Implement `int sqrt(int x)` .

Compute and return the square root of `x` .

Note

Try to make the time complexity less.

Think

- The square root should be between 1 to half of input value;
- Use binary search idea to search the `sqrt` inside that range;

Solution

```
public class Solution {
    public int mySqrt(int x) {
        if(x == 0)
            return 0;
        // binary search from 1 -> x/2
        int l = 1, r = (x>>1);
        while(l < r) {
            int m = l + ((r - l) >> 1);
            if( m <= x / m && (m + 1) > x / (m + 1)) {
                return m;
            }else if(m + 1 <= x / m) {
                l = m + 1;
            }else{
                r = m;
            }
        }
        return l;
    }
}
```

Binary Representation

Given a (decimal - e.g. 3.72) number that is passed in as a string, return the binary representation that is passed in as a string. If the fractional part of the number can not be represented accurately in binary with at most 32 characters, return `ERROR`.

Example

For n = `"3.72"`, return `"ERROR"`.

For n = `"3.5"`, return `"11.1"`.

Think

- For Integer part, we use `% 2` to get each digit from lowest bit to highest bit, or use a loop to make `&` with `1` and left shift until it get zero.
- For decimal part, we can use ☐ approach. For example: `int n = 0.75;` `n*2 = 1.5;` Therefore, the first digit of binary number after `.` is 1 (i.e. 0.1). After constructed the first digit, n= n*2-1;

Solution

```
public class Solution {
    /**
    * Therefore, the first digit of binary number after '.' is 1
  (i.e. 0.1).  After constructed the first digit, n= n*2-1
     *@param n: Given a decimal number that is passed in as a st
ring
     *@return: A string
     */
    public String binaryRepresentation(String n) {
        int intPart = Integer.parseInt(n.substring(0, n.indexOf(
'.')));
        StringBuilder res = new StringBuilder();
        while(intPart != 0) {
            res.insert(0, "" + (intPart & 1));
            intPart >>= 1;
        }
        if(res.length() == 0)
            res.append(0);
        double decPart = Double.parseDouble(n.substring(n.indexO
f('.')));
        String decBit = "";
        // if it has decimal part, creat '.' in result string
        if(decPart > 0.0)
            res.append(".");
        // to count how many digit in decimal binary result
        int cnt = 0;
        while(decPart > 0.0) {
            double cur = (decPart * 2);
            cnt++;
            if(cnt > 32)
                return "ERROR";
            if(cur >= 1) {
                res.append(1);
                decPart = cur - 1.0;
            }else {
                res.append(0);
                decPart = cur;
            }
        }

        return res.toString();
    }
}
```

Divide Two Integers

Divide two integers without using multiplication, division and mod operator.

If it is overflow, return `2147483647`

Example

Given dividend = `100` and divisor = `9`, return `11`.

Think

- Bitwise Idea:
  - Get the result sign (negative or positive) by `((dividend ^ divisor) >>> 31) == 1`
  - This question contains many corner cases!
  - Firstly, check the corner cases in following steps:
    - Divisor is zero? return `Integer.MAX_VALUE`;
    - Dividend is `Integer.MIN_VALUE`?
      - if divisor is negative one? you cannot get the positive `MIN_VALUE` so return `Integer.MAX_VALUE`;
      - `dividend += Math.abs(divisor)` so that the dividend become away from overflow but that leads the res increase one;
    - Divisor is `Integer.MIN_VALUE`? return res; To avoid the inaccurate from above operation;
  - Make dividend and divisor both positive;
  - Then, the main operation to do the binary substraction;
    - Get the most higher position(`digit`) for bit one with increasing the divisor until it is just larger than (`dividend >> 1`): divisor cannot larger than dividend so that we use the `dividend>>1`
    - Get result by add the `1<<digit` (current bit position should be one) and `dividend -= divisor` but if divisor larger than dividend which means current bit position should be zero so just reduce digit and divisor should shift right one position each time;

Solution

```
public class Solution {
    /**
     * @param dividend the dividend
     * @param divisor the divisor
     * @return the result
     */
    public int divide(int dividend, int divisor) {
        int res = 0;
        if(divisor == 0)
            return Integer.MAX_VALUE;
        boolean neg = ((dividend ^ divisor) >>> 31) == 1;
        if(dividend == Integer.MIN_VALUE) {
            // since the dividend is negative number now so we p
lus the abs(divisor)
            dividend += Math.abs(divisor);
            if(divisor == -1)
                return Integer.MAX_VALUE;
            res++;
        }

        if(divisor == Integer.MIN_VALUE)
            return res;

        // the highest position for bit in result
        int digit = 0;
        dividend = Math.abs(dividend);
        divisor = Math.abs(divisor);
        while(divisor <= (dividend>>1)) {
            divisor <<= 1;
            digit ++;
        }

        while(digit>=0){
            if(dividend>=divisor){
                res += (1<<digit);
                dividend-=divisor;
            }
```

```
                divisor>>=1;
                digit--;
            }
            return neg?-res:res;
        }
    }
```

Digit Counts

Count the number of k's between 0 and n. k can be 0 - 9.

Example

if n=12, k=1 in `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]` , we have `FIVE` 1's `(1, 10, 11, 12)`

Think #1

- Brute Force: Check each digit in number form (0 -> n) then get the count;

Solution #1

```
    public int digitCounts(int k, int n) {
        int[] record = new int[10];
        Arrays.fill(record,0);
        for (int i=0;i<=n;i++){
            String temp = Integer.toString(i);
            for (int j=0;j < temp.length();j++){
                int ind = (int) (temp.charAt(j)-'0');
                record[ind]++;
            }
        }
        return record[k];
    }
```

Think #2

- Math:
  - When current digit less than `k` , the current count should be `higher digits x digit position (pos x 10)` ;

- When current digit equal to `k`, the current count should be `higher digits x digit position + lower digits + 1`;
- When current digit larger than `k`, the current count should be `higher digits + 1(itself) x digit position`;
- When `k` == 0 and the current digit larger than `k`, the higher digits x digit position and it need to add one in the last result;

Solution #2

```
class Solution {
    /*
     * param k : As description.
     * param n : As description.
     * return: An integer denote the count of digit k in 1..n
     */
    public int digitCounts(int k, int n) {
        int digit = 1;
        int cnt = 0;
        while(digit <= n) {
            int low = n % digit; // lower digits;
            int high = n / (digit*10); // higher digits;
            int cur = n / digit % 10;
            if(cur == k) {
                // higher digits * digit + lower digits + 1;
                cnt += ((high * digit) + low + 1);
            }else if(cur < k) {
                // higher digits * digit
                cnt += (high * digit);
            }else{
                // (higher digits + 1: itself) * digit
                cnt += ((high + (k == 0?0:1)) * digit);
            }
            digit *= 10;
        }
        return cnt + (k == 0 ? 1 : 0);
    }
};
```

Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are `+`, `-`, `*`, `/`. Each operand may be an integer or another expression.

Example

```
    ["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
    ["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
```

Think

- Reverse Polish Notation problems always accompany with the stack as its data structure;
- Setup stack only to store the integer value;
- One pass the tokens array, when get the integer, insert the stack, while get the operator, to do the evaluate with pop two value from stack and push back the result after calculated;
- Its good to use `switch` rather than `if`;

Solution

```
public class Solution {
    /**
     * @param tokens The Reverse Polish Notation
     * @return the value
     */
    public int evalRPN(String[] tokens) {
        // By use stack
        Stack<Integer> stack = new Stack<>();
        for(String str : tokens) {
            int curRes = 0;
            switch(str){
                case "+":
                    curRes = (stack.isEmpty()?0:stack.pop()) + (
stack.isEmpty()?0:stack.pop());
                    break;
                case "-":
                    curRes = stack.isEmpty()?0:stack.pop();
                    curRes = (stack.isEmpty()?0:stack.pop()) - c
urRes;
                    break;
                case "*":
                    curRes = (stack.isEmpty()?0:stack.pop()) *
(stack.isEmpty()?0:stack.pop());
                    break;
                case "/":
                    curRes = stack.isEmpty()?0:stack.pop();
                    if(curRes == 0)
                        break;
                    curRes = (stack.isEmpty()?0:stack.pop()) / c
urRes;
                    break;
                default:
                    curRes = Integer.parseInt(str);
            }
            stack.push(curRes);
        }

        return stack.isEmpty()?0:stack.pop();
    }
}
```

Convert Expression to Reverse Polish Notation

Given an expression string array, return the Reverse Polish notation of this expression. (remove the parentheses)

Example

For the expression `[3 - 4 + 5]` (which denote by `["3", "-", "4", "+", "5"]`), return `[3 4 - 5 +]` (which denote by `["3", "4", "-", "5", "+"]`)

Think

- Always consider stack firstly, when meet a reverse polish notation problem.
- The operator priority (`*` , `/`) > (`+` , `-`), when get the operator is less priority than stack top element, pop the stack util the element has the same priority as the current operator and output the pop element in result list.
- Push the "(" always but pop the stack when get the ")" until stack has pop the corresponding "(".
- When get the operand, just output in result list.

Solution

```
public class Solution {
    /**
     * @param expression: A string array
     * @return: The Reverse Polish notation of this expression
     */
    public ArrayList<String> convertToRPN(String[] expression) {
        Stack<String> ops = new Stack<>();
        ArrayList<String> res = new ArrayList<>();
        for(String str : expression) {
            if(str.equals("+") || str.equals("-") ||str.equals("
*") ||str.equals("/")) {
                while(!ops.isEmpty() && operatorLevel(ops.peek()
) >= operatorLevel(str))
                    res.add(ops.pop());
                ops.push(str);
            }else if(str.equals("(")){
                ops.push(str);
            }else if(str.equals(")")) {
                while(!ops.isEmpty() && !ops.peek().equals("("))
                    res.add(ops.pop());
                if(!ops.isEmpty())
                    ops.pop(); // pop the "("
            }else
                res.add(str);
        }
        while(!ops.isEmpty())
            res.add(ops.pop());
        return res;
    }

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

Convert Expression to Polish Notation

Given an expression string array, return the Polish notation of this expression. (remove the parentheses)

> Polish notation, also known as Polish prefix notation or simply prefix notation, is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands. If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity. The Polish logician Jan Łukasiewicz invented this notation in 1924 in order to simplify sentential logic.

Example

For the expression `[(5 − 6) * 7]` (which represented by `["(", "5", "−", "6", ")", "*", "7"]`), the corresponding polish notation is `[* - 5 6 7]` (which the return value should be `["*", "−", "5", "6", "7"]`).

Think

- The idea is not complext if you took the reverse polish notation question.
- Just two things changed:
    - Pass from rear to head;
    - Every insertion is to the first index in result list

Solution

```
public class Solution {
    /**
     * @param expression: A string array
     * @return: The Polish notation of this expression
     */
    public ArrayList<String> convertToPN(String[] expression) {
        Stack<String> ops = new Stack<>();
        ArrayList<String> res = new ArrayList<>();
        for(int i = expression.length - 1; i >= 0; i--) {
            String str = expression[i];
            if(str.equals("+") || str.equals("-") ||str.equals("
*") ||str.equals("/")) {
                while(!ops.isEmpty() && operatorLevel(ops.peek()
) > operatorLevel(str))
                    res.add(0, ops.pop());
                ops.push(str);
            }else if(str.equals(")")){
                ops.push(str);
            }else if(str.equals("(")) {
                while(!ops.isEmpty() && !ops.peek().equals(")"))
                    res.add(0, ops.pop());
                if(!ops.isEmpty())
                    ops.pop(); // pop the ")"
            }else
                res.add(0, str);
        }
        while(!ops.isEmpty())
            res.add(0, ops.pop());
        return res;
    }

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

Expression Evaluation (Calculator)

Given an expression string array, return the final result of this expression

Example

For the expression `2*6-(23+7)/(1+2)`, input is

```
[
  "2", "*", "6", "-", "(",
  "23", "+", "7", ")", "/",
  (", "1", "+", "2", ")"
],
```

return `2`

Note

The expression contains only `integer`, `+`, `-`, `*`, `/`, `(`, `)`.

Think

- This question is combination of read string array to RPN and then read RPN to get the integer result.
- So.... two steps:
  - first, build RPN
  - second, read the RPN

Solution

```
public class Solution {
    /**
     * @param expression: an array of strings;
     * @return: an integer
     */
    public int evaluateExpression(String[] expression) {
        // two steps:
        // first, build RPN
        ArrayList<String> list = convertToRPN(expression);
        // second, read the RPN
        return RPNreader(list);
    }

    public ArrayList<String> convertToRPN(String[] expression) {
        Stack<String> ops = new Stack<>();
        ArrayList<String> res = new ArrayList<>();
        for(String str : expression) {
            if(str.equals("+") || str.equals("-") ||str.equals("
*") ||str.equals("/")) {
                while(!ops.isEmpty() && operatorLevel(ops.peek()
) >= operatorLevel(str))
                    res.add(ops.pop());
                ops.push(str);
            }else if(str.equals("(")){
                ops.push(str);
            }else if(str.equals(")")) {
                while(!ops.isEmpty() && !ops.peek().equals("("))
                    res.add(ops.pop());
                if(!ops.isEmpty())
                    ops.pop(); // pop the "("
            }else
                res.add(str);
        }
        while(!ops.isEmpty())
            res.add(ops.pop());
        return res;
    }

    private int operatorLevel(String op) {
        if(op.equals("+") || op.equals("-"))
            return 1;
        else if(op.equals("*") ||op.equals("/"))
            return 2;
        else
            return 0;
    }

    private int RPNreader(ArrayList<String> list) {
        Stack<Integer> stack = new Stack<>();
        for(String str : list) {
            int curRes = 0;
            switch(str){
                case "+":
                    curRes = (stack.isEmpty()?0:stack.pop()) + (
stack.isEmpty()?0:stack.pop());
                    break;
                case "-":
                    curRes = stack.isEmpty()?0:stack.pop();
                    curRes = (stack.isEmpty()?0:stack.pop()) - c
urRes;
                    break;
                case "*":
                    curRes = (stack.isEmpty()?0:stack.pop()) *
(stack.isEmpty()?0:stack.pop());
                    break;
                case "/":
                    curRes = stack.isEmpty()?0:stack.pop();
                    if(curRes == 0)
                        break;
                    curRes = (stack.isEmpty()?0:stack.pop()) / c
urRes;
                    break;
                default:
                    curRes = Integer.parseInt(str);
            }
            stack.push(curRes);
        }

        return stack.isEmpty()?0:stack.pop();
    }

};
```

Expression Tree Build

The structure of Expression Tree is a binary tree to evaluate certain expressions. All leaves of the Expression Tree have an number string value. All non-leaves of the Expression Tree have an operator string value.

Now, given an expression array, build the expression tree of this expression, return the root of this expression tree.

Example

For the expression `(2*6-(23+7)/(1+2))` (which can be represented by `["2" "*" "6" "-" "(" "23" "+" "7" ")" "/" "(" "1" "+" "2" ")"]`). The expression tree will be like

```
                  [ - ]
              /           \
         [ * ]              [ / ]
       /     \           /         \
     [ 2 ]  [ 6 ]      [ + ]        [ + ]
                      /    \       /      \
                    [ 23 ][ 7 ] [ 1 ]   [ 2 ] .
```

After building the tree, you just need to return root node `[-]` .

Clarification

A binary expression tree is a specific application of a binary tree to evaluate certain expressions. Two common types of expressions that a binary expression tree can represent are algebraic[1] and boolean. These trees can represent expressions that contain both unary and binary operators.

```
class ExpressionTreeNode {
    public String symbol;
     public ExpressionTreeNode left, right;
     public ExpressionTreeNode(String symbol) {
        this.symbol = symbol;
        this.left = this.right = null;
    }
}
```

Think

- Two stacks, one is only for operator, another is for build the result tree.
- The idea is similar with RPN.
- One pass the expression array, when it get a operand, insert as new treenode in result tree, when it get a operator, insert as new treenode in operator tree and compare the priority, if the current is less than stack top element, build a new node with operator in stack top and two node from result tree as its right and left child, then insert back to result tree with this new node.

Solution

```
public class Solution {
    /**
     * @param expression: A string array
     * @return: The root of expression tree
     */
    public ExpressionTreeNode build(String[] expression) {
        Stack<ExpressionTreeNode> ops = new Stack<>();
        Stack<ExpressionTreeNode> data = new Stack<>();
        for( int i = 0; i < expression.length; i++) {
            String str = expression[i];
            if(str.equals("+") || str.equals("-") ||str.equals("
*") ||str.equals("/")) {
                while(!ops.isEmpty() && operatorLevel(ops.peek()
.symbol) >= operatorLevel(str)) {
                    newNodeGtr(ops, data);
                }
                ops.push(new ExpressionTreeNode(str));
            }else if(str.equals("(")){
                ops.push(new ExpressionTreeNode(str));
            }else if(str.equals(")")) {
                while(!ops.isEmpty() && !ops.peek().symbol.equal
s("("))
                    newNodeGtr(ops, data);
                if(!ops.isEmpty())
                    ops.pop(); // pop the "("
            }else
                data.push(new ExpressionTreeNode(str));
        }

        while(!ops.isEmpty())
            newNodeGtr(ops, data);

        return data.isEmpty()? null : data.pop();
    }

    private void newNodeGtr(Stack<ExpressionTreeNode> ops, Stack
<ExpressionTreeNode> data) {
        if(ops.isEmpty())
            return;
        ExpressionTreeNode node = ops.pop();
        node.right = data.isEmpty() ? null : data.pop();
        node.left = data.isEmpty() ? null : data.pop();
        data.push(node);
    }

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

Update Bits

Given two 32-bit numbers, N and M, and two bit positions, i and j. Write a method to set all bits between i and j in N equal to M (e g , M becomes a substring of N located at i and starting at j).

Example

Given N= `(10000000000)2` , M= `(10101)2` , i= `2` , j= `6` return N= `(10001010100)2`

Note

In the function, the numbers N and M will given in decimal, you should also return a decimal number.

Challenge

Minimum number of operations?

Clarification

You can assume that the bits j through i have enough space to fit all of M. That is, if M = `10011` , you can assume that there are at least 5 bits between j and i. You would not, for example, have j=3 and i=2, because M could not fully fit between bit 3 and bit 2.

Think

- Set a mask:

```
Position:   31 30 ..~.. j+1  j ..~.. i  i-1 ..~.. 0
Bit Val:     1  1  ...   1   0  ...  0   1   ...  1
```

- Use that mask to do `&` with N, so that in the new N, the position i~j will be zero.
- Left shift `i` for M to make the position aligned.
- Do `|` for M and N, then get the final result.

Solution

```
class Solution {
    /**
     *@param n, m: Two integer
     *@param i, j: Two bit positions
     *return: An integer
     */
    public int updateBits(int n, int m, int i, int j) {

        int mask = 0;
        for(int lfShift = 31; lfShift > j; lfShift--)
            mask += (1<<lfShift);

        for(int lfShift = i - 1; lfShift >= 0; lfShift--)
            mask += (1<<lfShift);
        n &= mask;
        m <<= i;
        return n|m;
    }
}
```

Data Structure

People build so many data structure only for one purpose - easy to manipulate. Those structure is supporting the computer science, just like how we place the goods in warehouse. Here is how we place the data.

From Line to Multidimension

Array

It is the most basic structure. But the most easy things can be the most sophisticate, like traditional Chinese philosophy, like DNA build all creature in the world.

Matrix (2-D Array)

Two dimension array.

String (Character Array)

String is a special array with characters.

More Structural

Binary Tree

Heap

Stack

Queue

Disjoint Set

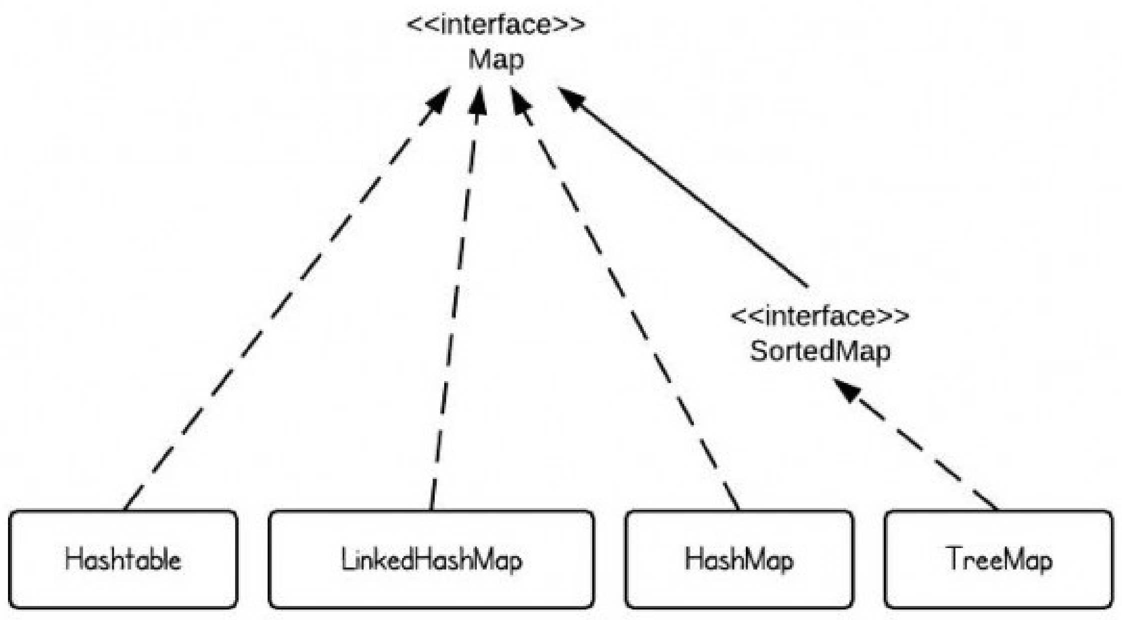
Graph

Map

Overview

There are 4 commonly used implementations of Map in Java SE - **HashMap**, **TreeMap**, **Hashtable** and **LinkedHashMap**. If we use only one sentence to describe each implementation, it would be the following:

- HashMap is implemented as a hash table, and there is no ordering on keys or values. It also allow null key or null value but hashtabl does not.
- TreeMap is implemented based on red-black tree structure, and it is ordered by the key.
- LinkedHashMap preserves the insertion order
- Hashtable is synchronized, in contrast to HashMap. It has an overhead for synchronization. This is the reason that Hashtable should be used if the program requires thread-safe.



Note that the HashSet doesn't implement Map interface but it is based on HashMap. It contains HashMap insided as the value storage.

Hash Code and Equals Functions

The Java super class java.lang.Object has two very important methods defined:

```
public boolean equals(Object obj)
public int hashCode()
```

They have been proved to be extremely important to understand, especially when user-defined objects are added to Maps.

Once you override the `equal()` function, you must be careful to treat the `hashCode()` function. The contract between `equals()` and `hashCode()` is that:

1. If two objects are equal, then they must have the same hash code.
2. If two objects have the same hashcode, they may or may not be equal.

The default implementation of `hashCode()` in Object class returns distinct integers for different objects. So you have to decide `hashCode()` function according to your requirement. Sometimes, the same properties can be regarded as the same object, then `hashCode()` function should just build by the character on these properties.

Hash Collision

Closed hashing:

Linear Probing, Quadratic Probing, Double hashing(Constant prime as the next hash size(x - val%x));

Open hashing:

Chaining Address. key-value pairs are stored in linked lists attached to cells of a hash table.

How Java Map classes handle collision?

Hash tables applied the chaining address

Step 1: By having each bucket contain a linked list of elements that are hashed to that bucket. This is why a bad hash function can make look-ups in hash tables very slow. Because the bad hash function make the items location less disperse and lead to some clusters.

Step 2: What if the nodes in HashMap are almost fulled? Two concept need to be introdued firstly:

- Threshold
- Load Factor

Threshold is a value when the number of entry nodes in HashMap touch this value, the `resize()` function should be triggered. Threshold value should be evaluated by the `loadFactor` which is set `0.75` as its default value.That means $threshold = size_{table} * 0.75$.

Step 3: Resize and rehash. The hash function returns an integer and the hash table has to take the result of the hash function and mod it against the size of the table that way it can be sure it will get to bucket. so by increasing the size it will rehash and run the modulo calculations which if you are lucky might send the objects to different buckets.

HashMap

How HashMap store data?

The least unit in hashmap data storage is the `Entry Node<K,V>` . The node implements `Map.Entry<K,V>`

```
 static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        // chaining address, point to next Entry Node
        Node<K,V> next;

        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }

        public final K getKey()        { return key; }
        public final V getValue()      { return value; }
        public final String toString() { return key + "=" + valu
e; }

        public final int hashCode() {
            return Objects.hashCode(key) ^ Objects.hashCode(valu
e);
        }

        public final V setValue(V newValue) {
            V oldValue = value;
            value = newValue;
            return oldValue;
        }
        public final boolean equals(Object o) {
            if (o == this)
                return true;
            if (o instanceof Map.Entry) {
                Map.Entry<?,?> e = (Map.Entry<?,?>)o;
                if (Objects.equals(key, e.getKey()) &&
                    Objects.equals(value, e.getValue()))
                    return true;
            }
            return false;
        }
    }
```

All of these `Node<K, V>` are stored in `table` as a Node Array

- Nodes Array `Node<K, V>[] table`

  ```
      /**
       * The table, initialized on first use, and resized as
       * necessary. When allocated, length is always a power of
  two.
       * (We also tolerate length zero in some operations to all
  ow
       * bootstrapping mechanics that are currently not needed.)
       */
      transient Node<K,V>[] table;
  ```

However, HashMap provides three views for retrieving the inside data: `keySet()` , `values()` and `entrySet()`

- KeySet: stores all keys in this hashmap

```
  /**
   * NOTE! This field is in AbstractMap class
   * Each of these fields are initialized to contain an inst
ance of the
   * appropriate view the first time this view is requested.
  The views are
   * stateless, so there's no reason to create more than one
 of each.
   */
  transient volatile Set<K>        keySet;
```

- Values: stores all values in this hash Map

```
  /**
   * NOTE! This field is in AbstractMap class
   */
  transient volatile Collection<V> values;
```

- EntrySet: stores Entry Node(K-V pair) in this HashMap

```
  /**
   * This field is in HashMap class
   * Holds cached entrySet(). Note that AbstractMap fields a
re used
   * for keySet() and values().
   */
  transient Set<Map.Entry<K,V>> entrySet;
```

How HashMap manipulate with a certain K-V pair?

Insertion

Here is the public entrance `put(key, value)` to insert K - V pair into HashMap. But we have to notice it has the return value, which is the previous value associated with this key.

```
    /**
     * @return the previous value associated with key, or
     *         null if there was no mapping for key.
     *         (A null return can also indicate that the map
     *         previously associated null with key.)
     */
    public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }
    /**
     * The original version about put k-v pair node implement.
     * Notice those two boolean value
     * @param hash hash for key
     * @param key the key
     * @param value the value to put
     * @param onlyIfAbsent if true, don't change existing value
     * @param evict if false, the table is in creation mode.
     * @return previous value, or null if none
     */
    final V putVal(int hash, K key, V value, boolean onlyIfAbsen
t,
                       boolean evict){...}
```

I made a Lite Edition about `putVal` function according to HashMap source code. The main procedure on insert a K-V pair to HashMap can be concluded as following:

- Check the Entry Node table if it's null, re-size it if necessary.
- Get the bucket index by `(table.length - 1) & hash`.
- Iterate the nodes in target bucket and check if there is any exist same node `exist.hash == hash && ((prevKey = exist.key) == key || (key != null && key.equals(prevKey))`. If exist the node with the same key, update its value.
- Record the previous node of insert position for returning the previous value.

- Insert the new node and update the previous node next pointer: `prev.next = new Node(hash, key, value, null)` .

```
/**
*  Lite Editon for HashMap input K - V pair function
**/
final V putVal(int hash, K key, V value) {
    // the reference for node table
    Node<K,V>[] tab;
    // the reference for node in current bucket
    Node<K,V> prev;
    int n; // the table length

    if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
    int idx = (n - 1) & hash; // the bucket index
    if((prev = table[idx]) == null) {
        table[idx] = new Node(hash, key, value, null);
    }else {
        Node<K,V> exist; K prevKey;
        // if key equal to the first node in current bucket
        if(prev.hash == hash && (prevKey = prev.key) == key || (
key != null && key.equals(prevKey)) {
            exist = prev;
        } else {
            // find the top node in current bucket and iterate n
odes in this bucket
            for (int binCount = 0; ; ++binCount) {
                // insert new node when find the next pointer of
 a node is null
                if((exist = prev.next)  == null) {
                    prev.next = new Node(hash, key, value, null)
;

                    break;
                }
                // if any node in this bucket is the same as the
 insert one
                if (exist.hash == hash && ((prevKey = exist.key)
 == key || (key != null && key.equals(prevKey))))
                    break;
                prev = exist;
            }
        }
        if (exist != null) { // existing mapping for key
            V oldValue = exist.value;
            if (oldValue == null)
                exist.value = value;
            return oldValue;
        }
    }
    // modification count for fail-fast
    ++modCount;
    // check current node amount, if larger than threshodl do re
size
    if (++size > threshold)
        resize();
    return null;
}
```

Retrieve

The public entrance `get(key)` returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. More formally, if this map contains a K - V pair `key==null` then this method returns `v` or `null` . Pleas note that return `null` doesn't necessarily indicates the map contains no mapping for the key! It's also possible that the map explicitly maps the key to `NULL` . Source code shown as below:

```
        public V get(Object key) {
            Node<K,V> e;
            return (e = getNode(hash(key), key)) == null ? null : e.
    value;
        }
```

The main procedure on get a K-V pair to HashMap can be concluded as following:

- Get the index by hash code `(table.length - 1) & hash` ;

- Check the first node in target bucket if it is the same as the input key and hash code
- Check the rest nodes iterative in target bucket if it matches the retrieval condition.

```
 final Node<K,V> getNode(int hash, Object key) {
        Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            if (first.hash == hash && // always check first node
 in the target bucket
                ((k = first.key) == key || (key != null && key.e
quals(k))))
                return first;
            if ((e = first.next) != null) {
                do { // check the rest nodes in this bucket
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && k
ey.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }
```

Deletion

Removes the mapping for the specified key from this map if present. Return the previous value associated with input key or `null` if there was no mapping for input key (A `null` return can also indicate that the map previously associated `null` key).

```
    public V remove(Object key) {
        Node<K,V> e;
        return (e = removeNode(hash(key), key, null, false, true
)) == null ?
            null : e.value;
    }

    /**
     * The original version about remove node implement.
     * Notice those two boolean value
     * @param hash hash for key
     * @param key the key
     * @param value the value to match if matchValue, else ignor
ed
     * @param matchValue if true only remove if value is equal
     * @param movable if false do not move other nodes while rem
oving
     * @return the node, or null if none
     */
    final Node<K,V> removeNode(int hash, Object key, Object valu
e,
                               boolean matchValue, boolean movab
le) {...}
```

Implements Map.remove and related methods with lite version.

```
    final Node<K,V> removeNode(int hash, Object key) {
        Node<K,V>[] tab; Node<K,V> p; int n, index;
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (p = tab[index = (n - 1) & hash]) != null) {
            Node<K,V> node = null, e; K k; V v;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equal
s(k))))
                node = p;
            else if ((e = p.next) != null) {
                do {
                    if (e.hash == hash && ((k = e.key) == key ||
                            (key != null && key.equals(k)))) {
                        node = e;
                        break;
                    }
                        p = e;
                } while ((e = e.next) != null);
            }

            if (node != null ) {
                if (node == p)
                    tab[index] = node.next;
                else
                    p.next = node.next;
                ++modCount;
                --size;
                return node;
            }
        }
        return null;
    }
```

How to hash by key?

Hash function is very important to HashMap. It requires quick, efficient and disperse distribution for nodes.

> Computes key.hashCode() and spreads (XORs) higher bits of hash to lower. Because the table uses power-of-two masking, sets of hashes that vary only in bits above the current mask will always collide. (Among known examples are sets of Float keys holding consecutive whole numbers in small tables.) So we apply a transform that spreads the impact of higher bits downward. There is a tradeoff between speed, utility, and quality of bit-spreading. Because many common sets of hashes are already reasonably distributed (so don't benefit from spreading), and because we use trees to handle large sets of collisions in bins, we just XOR some shifted bits in the cheapest possible way to reduce systematic lossage, as well as to incorporate impact of the highest bits that would otherwise never be used in index calculations because of table bounds.

```
    static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>>
16);
    }
```

Iterator

Iterator enables you to cycle through a collection, obtaining or removing elements. List Iterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an `iterator()` method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

Methods Declared by Iterator:

SN	Methods with Description
1	**boolean hasNext( )**
	Returns true if there are more elements. Otherwise, returns false.
2	**Object next( )**
	Returns the next element. Throws NoSuchElementException if there is not a next element.
3	**void remove( )**
	Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).

Methods Declared by ListIterator:

SN	Methods with Description
1	**void add(Object obj)**
	Inserts obj into the list in front of the element that will be returned by the next call to next( ).
2	**boolean hasNext( )**
	Returns true if there is a next element. Otherwise, returns false.
3	**boolean hasPrevious( )**
	Returns true if there is a previous element. Otherwise, returns false.
4	**Object next( )**
	Returns the next element. A NoSuchElementException is thrown if there is not a next element.
5	**int nextIndex( )**
	Returns the index of the next element. If there is not a next element, returns the size of the list.
6	**Object previous( )**
	Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
7	**int previousIndex( )**
	Returns the index of the previous element. If there is not a previous element, returns -1.
8	**void remove( )**
	Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked.
9	**void set(Object obj)**
	Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

Iterator for 2-D Array

Set up a iterator to iterate a 2-dimensional array. By given the following code, finish all class:

```
public class DeepIterator{

    public DeepIterator(int[][] listOfLists){
        ...
    }

    ...
    public static void main(String[] args) {
        int[][] listOfLists = {
          {},{},{1,2,3},{},{},{2,3,4}
        };
        DeepIterator it = new DeepIterator(listOfLists);
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

Think

- Iterate the 2-D array by `x` and `y`.
- Record current element during `hasNext()` function.
- Check the row if it is null.

Solution

```
public class DeepIterator{
    int cur; // this is important
    int row = 0, col = 0;

    int[][] listOfLists;

    public DeepIterator(int[][] listOfLists){
        if(listOfLists == null)
            throw new IllegalArgumentException("Null Input");
        this.listOfLists = listOfLists;
    }

    public Integer next(){
        return cur;
    }

    public boolean hasNext(){
        // make sure the row is not null
        while(row < listOfLists.length && col >= listOfLists[row
].length) {
                row ++; col = 0;
        }
        if(row < listOfLists.length) {
            cur = listOfLists[row][col++];
            return true;
        }else
            return false;
    }

    public static void main(String[] args) {
        int[][] listOfLists = {
          {},{},{1,2,3},{},{},{2,3,4}
        };
        DeepIterator it = new DeepIterator(listOfLists);
        while(it.hasNext()){
          System.out.println(it.next());
        }
    }
}
```

Iterator of Iterators

Given a Iterator which contain several iterator inside. The task is to iterate all elements inside these iterators. According to the following code, try to build up the entire class.

```
public class Iterators<T> implements Iterator<T>{
    public Iterators(Iterable<Iterator<T>> iterators){
        ...
    }
    ...
}
```

Think

- It's a class implements iterator interface but also contains several iterators.
- Image these iterator is a bucket, the unit is a iterator and the cursor of bucket index is a iterator element.
- Set a current index to point at a iterator.

Solution

```
public class Iterators<T> implements Iterator<T>{
    Iterators<T> current;
    Iterators<Iterators<T>> cursor;
    public Iterators(Iterable<Iterator<T>> iterators){
        if(iterators == null)
            throw new IllegalArgumentException("Illegal Argument
!");
        this.cursor = iterators;
    }

    @Override
    public T next(){
        return current.next();
    }

    @Override
    public boolean hasNext(){
        if(!current.hasNext())
            current = findNext();
        return current != null && current.hasNext();
    }

    private Iterator findNext(){
        while(cursor.hasNext()){
            current = cursor.next();
            if(current != null && current.hasNext())
                break;
        }
        return current;
    }

    @Override
    public void remove(){
        if(current!=null)
            current.remove();
    }
}
```

Peeking Iterator

Given an Iterator class interface with methods: `next()` and `hasNext()` , design and implement a PeekingIterator that support the peek() operation -- it essentially `peek()` at the element that will be returned by the next call to `next()` .

Here is an example. Assume that the iterator is initialized to the beginning of the list: `[1, 2, 3]` .

Call `next()` gets you 1, the first element in the list.

Now you call `peek()` and it returns 2, the next element. Calling `next()` after that still return 2.

You call `next()` the final time and it returns 3, the last element. Calling `hasNext()` after that should return false.

Solution

```
class PeekingIterator implements Iterator<Integer> {

    int cur;
    Iterator<Integer> it;
    public PeekingIterator(Iterator<Integer> iterator) {
        this.it = iterator;
        cur = it.hasNext() ? it.next() : null;
    }

    // Returns the next element in the iteration without advanci
ng the iterator.
    public Integer peek() {
        return cur;
    }

    // hasNext() and next() should behave the same as in the Ite
rator interface.
    // Override them if needed.
    @Override
    public Integer next() {
        int res = curl
        cur = it.next() ? it.next() : null;
        return res;
    }

    @Override
    public boolean hasNext() {
        return it.hasNext();
    }
}
```

Even Iterator

Implements an iterator only output the even number.

Example

```
public class EvenIterator implements Iterator<Integer> {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1); list.add(4); list.add(3); list.add(5);
        list.add(6); list.add(7); list.add(9); list.add(2);
        EvenIterator it = new EvenIterator(list.listIterator());
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

Output

```
4
6
2
```

Solution

```java
public class EvenIterator implements Iterator<Integer> {

    Iterator<Integer> iterator;

    public EvenIterator(Iterator<Integer> iterator) {
        this.iterator = iterator;
    }

    @Override
    public boolean hasNext() {
        return iterator.hasNext();
    }

    @Override
    public Integer next() {
        int res = 0;
        while (iterator.hasNext() && (res = iterator.next()) % 2
 != 0)
            ;
        return res;
    }


    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1); list.add(4);
        list.add(3); list.add(5);
        list.add(6); list.add(7);
        list.add(9); list.add(2);
        EvenIterator it = new EvenIterator(list.listIterator());
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

Jump Iterator

Implements an iterator in each output it print the number and skip the next one.

Partial Code

```
public class JumpIterator implements Iterator<Integer> {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1); list.add(4);
        list.add(3); list.add(5);
        list.add(6); list.add(7);
        list.add(2);
        JumpIterator it = new JumpIterator(list.listIterator());
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

Output

```
1
3
6
2
```

Solution

```
public class JumpIterator implements Iterator<Integer> {

    Iterator<Integer> iterator;

    public JumpIterator(Iterator<Integer> iterator) {
        this.iterator = iterator;
    }

    @Override
    public boolean hasNext() {
        return iterator.hasNext();
    }

    @Override
    public Integer next() {
        int res = iterator.next();
        if (iterator.hasNext())
            iterator.next();
        return res;
    }

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1); list.add(4);
        list.add(3); list.add(5);
        list.add(6); list.add(7);
        list.add(2);
        JumpIterator it = new JumpIterator(list.listIterator());
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

Object Oriented Programming

Principles

Open - Close

Open for extension but closed for modifications

> The design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

Example

```
class ShapeEditor{
    void drawShape(Shape s){
        if (s.m_type==1)
             drawRectangle(s);
         else if (s.m_type==2)
             drawCircle(s);
     };
    void drawRectangle(Shape s);
    void drawCircle(Shape s);
}
class Shape{}
class Rectangle extends Shape{}
class Circle extends Shape{}

//---> Every time if new shape added, we have to modify method i
n the editor class, which violates this rule!
// ---> change!!! write draw method in each shape concreted class

class Shape{ abstract void draw(); }
class Rectangle extends Shape{ public void draw() { /*draw the r
ectangle*/} }
class Circle extends Shape{ public void draw() { /*draw the circ
le*/} }
class ShapeEditor{
    void drawShape(Shape s){ s.draw(); }
    }
```

Dependency Inversion

The low level classes the classes which implement basic and primary operations(disk access, network protocols,...) and high level classes the classes which encapsulate complex logic(business flows, ...). The last ones rely on the low level classes.

> High-level modules should not depend on low-level modules. Both should depend on abstractions.
>
> Abstractions should not depend on details. Details should depend on abstractions.

When this principle is applied it means the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer.

Example

```
class WorkerWithTechOne{}
class Manager{
    WorkerWithTechOne worker;
}
// --> what if add other workers with other techniques?
// --> abstract worker!
interface Worker{}
class WorkerWithTechOne implements Worker{}
class WorkerWithTechTwo implements Worker{}
class Manager{
    Worker worker;
}
```

Interface Segregation

Clients should not be forced to depend upon interfaces that they don't use.

> Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.

Example

```
interface work{
    public void work();
    // too much methods!
    public void life();
    public void rest();
}
// ---> change!!!
interface work{public void work();}
interface life{public void life();}
interface rest{public void rest();}
```

Single Responsibility

> A class should have only one reason to change.

A simple and intuitive principle, but in practice it is sometimes hard to get it right.

This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

Example

```
interface Iemail{
    public void setSender(String sender);
    public void setReceiver(String receiver);
    public void setContent(String content);
    // --> change!!!
    public void setContent(Content content);
}
// --> Content can be change to HTML or JSON or other kinds of f
ormat
// so it should be splited
interface Content {
    public String getAsString(); // used for serialization
}
class email implement Iemail{
    public void setSender(String sender) { set sender; }
    public void setReceiver(String receiver) { set receiver; }
    public void setContent(String content) {set content; }
    // --> change!!!
    public void setContent(Content content) { set content; }
}
```

Liskov's Substitution

> Derived types must be completely substitutable for their base types.

This principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behavior.

This principle considers what kind of derived class should extends a base class.

Example

```
// Violation of Likov's Substitution Principle
class Rectangle {
    protected int m_width;
    protected int m_height;

    public void setWidth(int width){ m_width = width; }
    public void setHeight(int height){ m_height = height; }
}

class Square extends Rectangle {
    public void setWidth(int width){
        m_width = width;    m_height = width;
    }
    public void setHeight(int height){
        m_width = height;    m_height = height;
    }
}
```

Muddiest Points

Composite v.s Aggregation

Simple rules:

> A "owns" B = Composition : B has no meaning or purpose in the system without A
>
> A "uses" B = Aggregation : B exists independently (conceptually) from A
>
> **Example 1:**

A Company is an aggregation of People. A Company is a composition of Accounts. When a Company ceases to do business its Accounts cease to exist but its People continue to exist.

Example 2: (very simplified)

A Text Editor owns a Buffer (composition). A Text Editor uses a File (aggregation). When the Text Editor is closed, the Buffer is destroyed but the File itself is not destroyed.

Interface v.s Abstract Class

Interface

Characters

- Allow multiple inheritance.
- No concrete method.
- No constructor.
- No instance variable, it only allow static final constant with the assignment.
- Can extend interface.
- Cannot be initialized. But sometime the interface can initialized by providing anonymous inner class.

Guide

- When you think that the API will not change for a while.
- When you want to have something similar to multiple inheritance.
- Enforce developer to implement the methods defined in interface.
- Interface are used to represent adjective or behavior

Abstract Class

Characters

- Can extend abstract class.
- Allow concrete method, provide some default behavior.
- Allow abstract method.
- Cannot be initialized.

Guide

- On time critical application prefer abstract class is slightly faster than interface.

- If there is a genuine common behavior across the inheritance hierarchy which can be coded better at one place than abstract class is preferred choice.

All in all

Interface and abstract class can work together also where defining function in interface and default functionality on abstract class. It also called **Skeletal Implementations**.

Reference

1. OO Design Website: oodesign.com
2. OOD Question on Stackexchange: Composite v.s Aggregation

Design Patterns

Creational Patterns

- Object Pool Pattern
- Prototype Pattern
- Factory Method Pattern
- Builder Pattern
- Factory Pattern
- Abstract Factory Pattern
- Singleton Pattern

Behavioral Patterns

- Memento Pattern
- Mediator Pattern
- Observer Pattern
- Null Object Pattern
- Visitor Pattern
- Interpreter Pattern
- Iterator Pattern
- Strategy Pattern
- Command Pattern
- Template Method Pattern
- Chain of Responsibility Pattern

Structural Patterns

- Adapter Pattern
- Bridge Pattern
- Decorator Pattern
- Proxy Pattern
- Composite Pattern

- Flyweight Pattern

Deck of Card

Design the data structure for a generic deck of cards. How you would subclass it to implement particular card games?

Think

Card has suit and value; Suit has four kinds with Club, Spade, Heart and Diamond.

About Enum Type

1. enum 类型不支持 public 和 protected 修饰符的构造方法，因此构造函数一定要是 private 或 friendly 的。也正因为如此，所以枚举对象是无法在程序中通过直接调用其构造方法来初始化的。
2. 定义 enum 类型时候，如果是简单类型 (No more constructor)，那么最后一个枚举值后不用跟任何一个符号；但如果有定制方法，那么最后一个枚举值与后面代码要用分号';'隔开，不能用逗号或空格。
3. 由于 enum 类型的值实际上是通过运行期构造出对象来表示的，所以在 cluster 环境下，每个虚拟机都会构造出一个同义的枚举对象。因而在做比较操作时候就需要注意，如果直接通过使用等号 (‘ == ’) 操作符，这些看似一样的枚举值一定不相等，因为这不是同一个对象实例。

Solution

```
class Card {
    // Define the Suit by Enum type
    public enum Suit {
        CLUBS(1), SPADE(2), HEART(3), DIAMOND(4);
        int value;
        private Suit(int val) {
            this.value = val;
        }
    }

    // Card has suit and value, only two kind of data need to st
ore
    int val;
    Suit suit;


    public Card(int value, Suit suit) {
        this.val = value;
        this.suit = suit;
    }

    public int getVal(){
        return this.val;
    }

    public Suit getSuit(){
        return this.suit;
    }
}
```

BlackJack

- Face cards (kings, queens, and jacks) are counted as ten points.
- Ace can be counted as 1 point or 11 points
- Other cards with value less than ten should be counted as what it values.

```
class BlackJack extends Card{
    public BlackJack(int val, Suit suit) {
        super(val, suit);
    }

    @Override
    public int getVal(){
        int value = super.getVal();
        if(value < 10)
            return value;
        else if(value == 1)
            return 11;
        return 10;
    }

    public boolean isAce(){
        return super.getVal() == 1;
    }
}
```

Chess Game

Some Rules:

- Player chooses piece to move through the board.
- Piece makes legal move according to its own move rules.
- .
- If player captures a piece, remove the piece.
- If the piece is a pawn reaching the back rank, promote it.
- If the move is a castling, set the new position of the rook accordingly. But a king and rook can only castle if they haven't moved, so you need to keep track of that. And if the king moves through a check to castle, that's disallowed, too.
- If the move results in a stalemate or checkmate, the game is over.

Basic Object Design

- Game:
  - Contains the Board and 2 Players
  - Commands List (for history tracking)
- Board (Singleton):
  - Hold spots with 8*8
  - Initialize the piece when game start
  - Move Piece
  - Remove Piece
- Spot:
  - Hold Pieces
- Piece (Abstract):
  - Hold the color to represent the affiliation.
  - Extended by concreted classes with 8 Pawns, 2 Rooks, 2 Bishops, 2 Knights, 1 Queen, 1 King
  - Concreted classes define the detail step approach

- Player (Abstract):
    - Has a list of piece reference it owns.
    - Concreted classes for Human and Computer players
- Command
    - Piece
    - Destination x, y

Solution for Basic Part

Here we can achieve the step move and check the win for player

- Game:

```
public class Game{
    final static Board board;
    Player p1;
    Player p2;

    public Game() {
        board = new Board();
    }

    public boolean enterPlayer(Player p) {
        if(p1 == null)
            this.p1 = p;
        else if(p2 == null)
            this.p2 = p;
        else
            return false;

        board.initialize(p);
        return true;
    }

    public void processTurn(Player p) {
        // Player make a command and until it is valid
        // System input
        do{
            Command cmd = new Command(input);
            p.addCommand(cmd);
        }while(!board.executeMove(p));
    }

    public startGame(){
        // player enter the game:
        enterPlayer(new ComputerPlayer("Computer"));
        enterPlayer(new HumanPlayer("Bill"));

        while(true) {
            processTurn(p1);
            if(this.board.getWin()) {
                System.out.println("P1 win!");
                break;
            }
            processTurn(p2);
            if(this.board.getWin()) {
                System.out.println("P2 win!");
                break;
            }
        }
    }
}
```

- Board:

```
public class Board{

    private Spot[][] spots;
    private boolean win; // mark the win or not

    public Board(){
        win = false;
        spots = new Spot[8][8];
    }
    public void initialize(Player p){
        // put the pieces with initial status
        for(int i=0; i<p.getPieces().size(); i++){
            spots[p.getPieces().get(i).getX()][p.getPieces().get
(i).getY()].occupySpot(p.getPieces().get(i));
        }
    }

    public boolean executeMove(Player p) {
        Command cmd = p.getCurrentCmd();
        Piece piece = cmd.getPiece();

        // check the move step is valid for piece
        if(!piece.validMove(this, cmd.curX, cmd.curY, cmd.desX,
cmd.desY)) {
            // if not valid cmd remove the command and return fa
lse
            p.removeCurrentCmd();
            return false;
        }

        // check the two pieces side
        if(spot[cmd.desX][cmd.desY] != null && spot[cmd.desX][cm
d.desY].color == piece.color)
            return false;

        // check and change the state on spot
        Piece taken = spot[cmd.desX][cmd.desY].occupySpot(piece)
;
        if(taken != null &&taken.getClass().getName().equals("Ki
ng"))
            board.win = true;
        spot[cmd.curX][cmd.curY].releaseSpot;
        return true;
    }

    public boolean getWin() {
        return win;
    }
}
```

- Spot:

```
public class Spot {
    int x;
    int y;
    Piece piece;

    public Spot(int x, int y) {
        super();
        this.x = x;
        this.y = y;
        piece = null;
    }

    // return original piece
    public void occupySpot(Piece piece){
        Piece origin = this.piece;
        //if piece already here, delete it, i. e. set it dead
        if(this.piece != null) {
            this.piece.setAvailable(false);
        }
        //place piece here
        this.piece = piece;
        return origin;
    }

    public boolean isOccupied() {
        if(piece != null)
            return true;
        return false;
    }

    public Piece releaseSpot() {
        Piece releasedPiece = this.piece;
        this.piece = null;
        return releasedPiece;
    }

    public Piece getPiece() {
        return this.piece;
    }
}
```

- Pieces:

```
public class Piece {
    private int x;
    private int y;

    private boolean available; // mark the live or dead
    private int color; // mark the owner

    public Piece(boolean available, int x, int y, int color) {
        super();
        this.available = available;
        this.x = x;
        this.y = y;
        this.color = color;
    }


    public boolean isAvailable() {
        return available;
    }
    public void setAvailable(boolean available) {
        this.available = available;
    }
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }

    public boolean isValid(Board board, int fromX, int fromY, int
 toX, int toY){
        // different by character of piece
    }

}

public class King extends Piece{
    @Override
    public boolean isValid(Board board, int fromX, int fromY, int
 toX, int toY) {
    }
}
// ..... for Queen, Rook, Bishop, Pawn
```

- Player:

```
public class Player {

    public int color;

    private List<Piece> pieces = new ArrayList<>();

    private List<Command> cmds = new ArrayList<>();

    public Player(int color) {
        super();
        this.color = color;
        initializePieces();
    }

    public List<Piece> getPieces() {
        return pieces;
    }


    public void initializePieces(){
        if(this.color == 1){
            for(int i=0; i<8; i++){ // draw pawns
                pieces.add(new Pawn(true,i,2, 1));
            }
            pieces.add(new Rook(true, 0, 0, 1));
            pieces.add(new Rook(true, 7, 0, 1));
            pieces.add(new Bishop(true, 2, 0, 1));
            pieces.add(new Bishop(true, 5, 0, 1));
            pieces.add(new Knight(true, 1, 0, 1));
            pieces.add(new Knight(true, 6, 0, 1));
            pieces.add(new Queen(true, 3, 0, 1));
            pieces.add(new King(true, 4, 0, 1));
        }
        else{
            for(int i=0; i<8; i++){ // draw pawns
                pieces.add(new Pawn(true,i,6, 0));
            }
            pieces.add(new Rook(true, 0, 7, 0));
            pieces.add(new Rook(true, 7, 7, 0));
            pieces.add(new Bishop(true, 2, 7, 0));
            pieces.add(new Bishop(true, 5, 7, 0));
            pieces.add(new Knight(true, 1, 7, 0));
            pieces.add(new Knight(true, 6, 7, 0));
            pieces.add(new Queen(true, 3, 7, 0));
            pieces.add(new King(true, 4, 7, 0));
        }

    }
}
```

- Command

```
public class Command {
    Piece piece;
    int curX, curY, desX, desY;
    public Commanc(Piece piece, int curX, int curY, int desX, int
 desY) {
        this.piece = piece;
        this.curX = curX;
        this.curY = curY;
        this.desX = desX;
        this.desY = desY;
    }
}
```

Online Book Reader System

Introduction

This question comes from the book named "Cracking Code Interview", Chapter 7; It is very easy problem with thinking about the insert/remove/update/retrieve action.

Functionality

- User Membership Creation and Extension
- Search the book in memory
- Reading the book

Analysis

Objects

- Book:
    - ID
    - Title
    - Author
    - Content
    - Method
        - showContent
- Books: (In-memory storage for many book objects)
    - Set
    - Method
        - find
        - add
        - delete
        - update

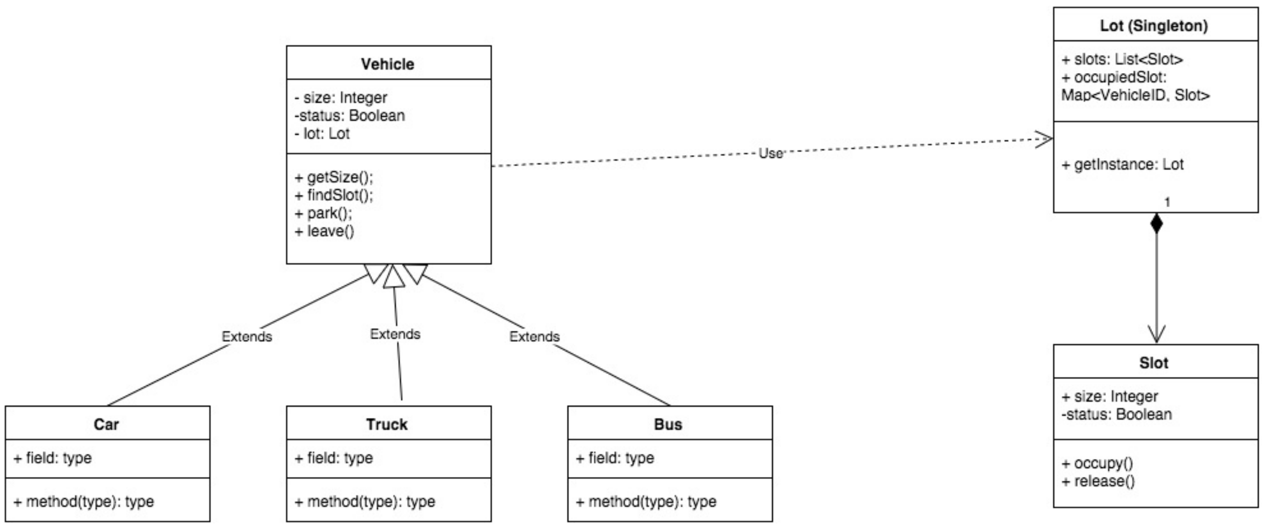
- User
    - ID
    - Name
    - accoutnType
    - Method
        - findBook
        - read (book.showContent)
- Users
    - Set
    - Method
        - find
        - add
        - delete
        - update

Parking Lot

Basic Object

- Vehicle
    - size of vehicle (small, medium, large)
    - status of vehicle (run or parked)
- Sedan, SUV, Bus, Truck... extends Vehicle
- Slot
    - size of slot
    - status (available or not)
- Lot
    - hold slots in lot

Diagram



Solution

- Vehicle

```
public class Vehicle {
    private final int size;
    private final int lisense;
    private boolean status;
    private Lot lot;

    public Vehicle(int size) {
        this.size = size;
        lisense = this.hashCode();
        lot = Lot.getInstance();
    }

    public void setStatus(boolean status) {
        this.status = status;
    }

    private Slot findSlot() {

        Slot slot;
        switch (this.size) {
        case 1:
            slot = lot.getSmallSlots().remove(0);
        case 2:
            slot = lot.getCompactSlots().remove(0);
        case 3:
            slot = lot.getLargeSlots().remove(0);
        default:
            slot = null;
        }
        return slot;
    }

    public void park() {
        Slot slot = findSlot();
        if (slot != null) {
            lot.occupiedSlots.put(this.lisense, slot);
            slot.occupy(this);
        }
    }

    public void leave() {
        Slot slot = lot.occupiedSlots.remove(this.lisense);
        slot.release();
        switch (this.size) {
        case 1:
            lot.getSmallSlots().add(slot);
        case 2:
            lot.getCompactSlots().add(slot);
        case 3:
            lot.getLargeSlots().add(slot);
        }
    }
}

public class Car extends Vehicle{
    public Car(){
        super(1);
    }
}
public class Truck extends Vehicle{
    public Truck(){
        super(2);
    }
}
// ... other type of vehicle
```

- Lot

```
public class Lot {
    private static Lot lot = null;

    private static final int NUMBER_OF_SMALL_SLOTS = 10;
    private static final int NUMBER_OF_COMPACT_SLOTS = 10;
    private static final int NUMBER_OF_LARGE_SLOTS = 10;

    public Map<Integer, Slot> occupiedSlots;
    private List<Slot> smallSlots;
    private List<Slot> compactSlots;
    private List<Slot> largeSlots;
    private Lot() {
        smallSlots = new LinkedList<>();
        compactSlots = new LinkedList<>();
        largeSlots = new LinkedList<>();
        occupiedSlots = new HashMap<>();
        for (int i = 1; i <= NUMBER_OF_SMALL_SLOTS; i++)
            smallSlots.add(new Slot(i, 1));

        for (int i = 1; i <= NUMBER_OF_COMPACT_SLOTS; i++)
            compactSlots.add(new Slot(i, 2));

        for (int i = 1; i <= NUMBER_OF_LARGE_SLOTS; i++)
            largeSlots.add(new Slot(i, 3));

    }

    public List<Slot> getSmallSlots() {
        return smallSlots;
    }

    public List<Slot> getCompactSlots() {
        return compactSlots;
    }

    public List<Slot> getLargeSlots() {
        return largeSlots;
    }

    public static Lot getInstance() {
        if (lot == null)
            lot = new Lot();
        return lot;
    }
}
```

- Slot

```
public class Slot {
    private final int id;
    private final int size;
    private boolean available;
    private Vehicle vehicle;

    public Slot(int id, int size) {
        this.id = id;
        this.size = size;
        this.available = true;
    }

    public void occupy(Vehicle v) {
        this.vehicle = v;
        this.available = false;
    }

    public void release() {
        this.vehicle = null;
        this.available = true;
    }
}
```

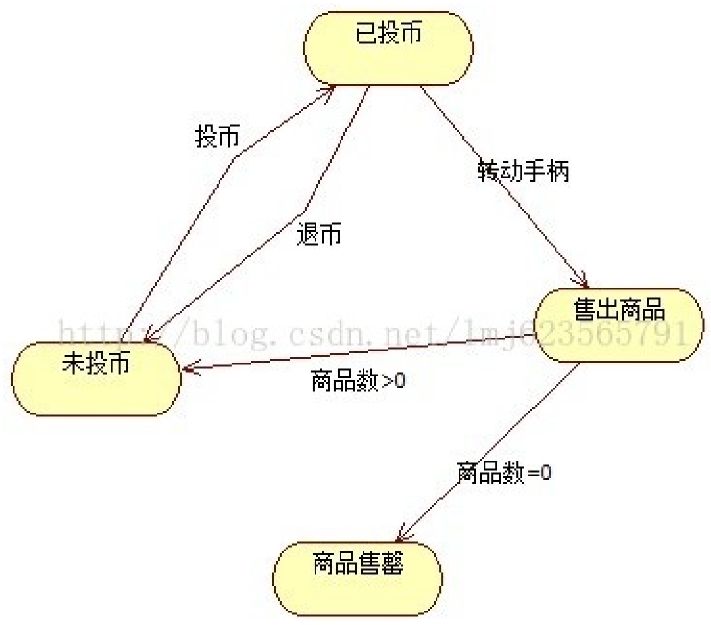
Vending Machine

Idea

The key point for this question is that you should recognize the states switching process. Essentially, it's a thinking about state pattern.

Four States for a vending machine:

1. Empty (Nothing in vending machine) 无商品
2. Coin Inserted 已投币
3. No Coin Inserted 未投币
4. Dispense a merchandise 出售商品



You can use integer to represent state like this:

```
        // 已投币
        private final static int HAS_MONEY = 0;
        // 未投币
        private final static int NO_MONEY = 1;
        // 售出商品
        private final static int SOLD = 2;
        // 商品售罄
        private final static int SOLD_OUT = 3;
```

BUT it'd be better to use classes with designed inheritance relationship, like following

```
      // 已投币
      State coninInsertedState = new CoinInsertedState(this);
      // 商品售罄
      State emptyState = new EmptyState(this);
      // 未投币
      State noCoinInsertedState = new NoCoinInsertedState(this);
      // 售出商品
      State dispensingState = new DispensingState(this);
```

Code

First We should think about state interface:

State.java

```
// State interface
import statepattern.exception.MachineWarning;
public interface State {

    public void insertCoin()throws MachineWarning;
    public void pressButton()throws MachineWarning;
    public void dispense()throws MachineWarning;
}
```

Concrete States implementations

ConinInsertedState.java

```
// Inserted Coin state
package statepattern;
import statepattern.exception.MachineWarning;
public class CoinInsertedState implements State{
    VendingMachine machine =null;
    public CoinInsertedState(VendingMachine machine) {
        this.machine =  machine;
    }
    public void insertCoin() throws MachineWarning{
        throw new MachineWarning("Coin is already inserted.");
    }
    public void dispense() throws MachineWarning{
        throw new MachineWarning("Dispense button is not pressed
.");

    }
    public void pressButton() throws MachineWarning{
        machine.setMachineState(machine.getDispensingState());
    }
}
```

DispensingState.java

```
// Dispensing Merchandise
package statepattern;
import statepattern.exception.MachineWarning;
public class DispensingState implements State{
    VendingMachine machine ;
    DispensingState(VendingMachine machine) {
        this.machine = machine;
    }
    public void insertCoin() throws MachineWarning {
        throw new MachineWarning("wait ... previous order is pro
cessing");
    }
    public void pressButton() throws MachineWarning {
        throw new MachineWarning("wait ... previous order is pro
cessing");
    }
    public void dispense() throws MachineWarning {
        machine.setMachineState(machine.getNoCoinInsertedState()
);
    }
}
```

EmptyState.java

```java
// Nothing in machine (Empty)
package statepattern;
import statepattern.exception.MachineWarning;
public class EmptyState implements State{
    VendingMachine machine;
    public EmptyState(VendingMachine machine) {
        this.machine =  machine;
    }
    public void insertCoin() throws MachineWarning{
        throw new MachineWarning("Can not process the request");
    }
    public void pressButton() throws MachineWarning{
        throw new MachineWarning("Invalid Action");
    }
    public void dispense() throws MachineWarning{
        throw new MachineWarning("Invalid Action");
    }
}
```

NoCoinInsertedState.java

```
// No coin inserted
package statepattern;
import statepattern.exception.MachineWarning;
public class NoCoinInsertedState implements State{
    VendingMachine machine;
    public NoCoinInsertedState(VendingMachine machine) {
        this.machine =  machine;
    }
    public void insertCoin() throws MachineWarning{
        if (!machine.isEmpty()) {
            machine.setMachineState(machine.getConinInsertedStat
e());
        }
        else {
            throw new MachineWarning("Can not process request ..
 Machine is out of stock");
        }
    }
    public void pressButton() throws MachineWarning{
        throw new MachineWarning("No coin inserted ..");
    }
    public void dispense() throws MachineWarning{
        throw new MachineWarning("Invalid Operation");
    }
}
```

Then, we have to think about how to structure of vending machine object.

VendingMachine.java

```
package statepattern;
import statepattern.exception.MachineWarning;
public class VendingMachine {
    // declare
    State coninInsertedState = new CoinInsertedState(this);
    State emptyState = new EmptyState(this);
    State noCoinInsertedState = new NoCoinInsertedState(this);
    State dispensingState = new DispensingState(this);
    State machineState = null;
    int capacity = 0;
    public VendingMachine() {
        machineState = noCoinInsertedState;
    }
    public void reFill(int count) {
        capacity += count;
        machineState = noCoinInsertedState;
    }
    /**
     * Two Actions performed by MAchine
     */
    public void insertCoin() throws MachineWarning {
        machineState.insertCoin();
    }

    public void pressButton() throws MachineWarning {
        machineState.pressButton();
        machineState.dispense();
        capacity--;
    }

    public boolean isEmpty(){
        if(capacity<=0)
            return true;
        else
            return false;
    }

    public void setMachineState(State machineState) {
        this.machineState = machineState;
    }
    public State getMachineState() {
        return machineState;
    }
    public void setConinInsertedState(State coninInsertedState)
{
        this.coninInsertedState = coninInsertedState;
    }
    public State getConinInsertedState() {
        return coninInsertedState;
    }
    public void setEmptyState(State emptyState) {
        this.emptyState = emptyState;
    }
    public State getEmptyState() {
        return emptyState;
    }
    public void setNoCoinInsertedState(State noCoinInsertedState)
 {
        this.noCoinInsertedState = noCoinInsertedState;
    }
    public State getNoCoinInsertedState() {
        return noCoinInsertedState;
    }
    public void setDispensingState(State dispensingState) {
        this.dispensingState = dispensingState;
    }
    public State getDispensingState() {
        return dispensingState;
    }
    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }
    public int getCapacity() {
        return capacity;
    }
}
```

System Design

System design is a wide topic. Even a experienced software engineer at top IT company may not be an expert on system design. Many books, articles, and solve real large scale system design problems may needed.

Steps (S - N - A - K - E)

- Scenario: (Case/Interface)
- Necessary: (Contain/Hypothesis)
- Application: (Services/Algorithms)
- Kilobyte: (Data)
- Evolution

Scenario

Enumerate All possible cases

- Register / Login
- Main Function One
- Main Function Two
- ...

Sort Requirement

Think about requirement and priority

Necessary

- Ask:
  - User Number
  - Active Users
- Predict
  - User

        - Concurrent User:
            - $Avg\_Concurrent\_Users = users_{daily\_active} / seconds_{daily} \times avg\_time_{user\_online}$
        - Peak User:
            - : c is a constant
        - Future Expand:
            - Users growing: max user amount in 3 months
    - Traffic
        - Traffic per user (bps)
        - MAX peak traffic
    - Memory (< 1 TB is acceptable)
        - Memory per user
        - MAX daily memory
    - Storage
        - Type of files (e.g movie, music need to store multiple qualities)
        - Amount of files

Application

- Replay the case, add a service for each request
- Merge the services

Kilobyte

- Append dataset for each request below a service
- Choose storage types

Evolution

- Analyze
    - with
        - Better: constrains
        - Broader: new cases
        - Deeper: details
    - from the views of
        - Performance

- Scalability
- Robustness

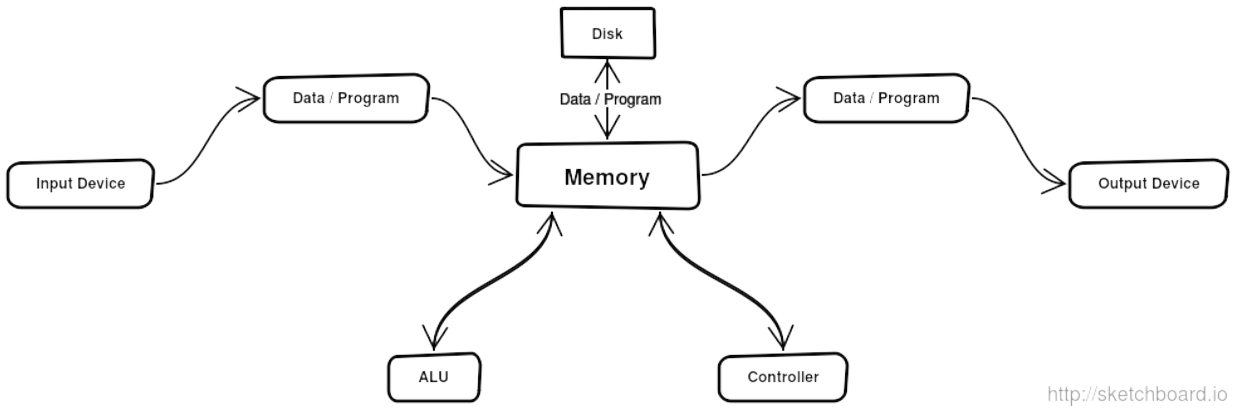
Distributed System

Principles

- Availability
- Performance
- Reliability
- Scalability
- Manageability
- Cost-Effective

How to distributed?

What is the basic computer structure?



It's clear to see three main parts:

- Controller
- ALU
- Storage

Extend the basic one computer model to multiple machine

The distributed system is also combined with those three parts.To design a system with distributed thinking can easier to extend based on the original Von Neumann structure.

Controller

Controlled the action and behavior between nodes in entire distributed system. Make nodes in system work collaboratively. There are several ways to achieve this point:

Hardware

- F5

Software

Transparent proxy: Both sides don't know each others. Only rely on the middle proxy. Cause two problems:1) Traffic overload; 2) The horrible situation when proxy machine is down;

- Linux Virtual System

Naming Service

Rule Server

Master-Workers

Load Balancer Algorithms

- Round Robin

  调度器通过"轮叫"调度算法将外部请求按顺序轮流分配到集群中的真实服务器上，它均等地对待每一台服务器，而不管服务器上实际的连接数和系统负载。

- Weighted Round Robin

  调度器通过"加权轮叫"调度算法根据真实服务器的不同处理能力来调度访问请求。这样可以保证处理能力强的服务器处理更多的访问流量。调度器可以自动问询真实服务器的负载情况，并动态地调整其权值。

- Least Connections

  调度器通过"最少连接"调度算法动态地将网络请求调度到已建立的链接数最少的服务器上。如果集群系统的真实服务器具有相近的系统性能，采用"最小连接"调度算法可以较好地均衡负载。

- Weighted Least Connections

  在集群系统中的服务器性能差异较大的情况下，调度器采用"加权最少链接"调度算法优化负载均衡性能，具有较高权值的服务器将承受较大比例的活动连接负载。调度器可以自动问询真实服务器的负载情况，并动态地调整其权值。

- Locality-Based Least Connections

  "基于局部性的最少链接" 调度算法是针对目标IP地址的负载均衡，目前主要用于Cache集群系统。该算法根据请求的目标IP地址找出该目标IP地址最近使用的服务器，若该服务器 是可用的且没有超载，将请求发送到该服务器；若服务器不存在，或者该服务器超载且有服务器处于一半的工作负载，则用"最少链接"的原则选出一个可用的服务 器，将请求发送到该服务器。

- Locality-Based Least Connections with Replication

  "带复制的基于局部性最少链接"调度算法也是针对目标IP地址的负载均衡，目前主要用于Cache集群系统。它与LBLC算法的不同之处是它要维护从一个 目标IP地址到一组服务器的映射，而LBLC算法维护从一个目标IP地址到一台服务器的映射。该算法根据请求的目标IP地址找出该目标IP地址对应的服务 器组，按"最小连接"原则从服务器组中选出一台服务器，若服务器没有超载，将请求发送到该服务器，若服务器超载；则按"最小连接"原则从这个集群中选出一 台服务器，将该服务器加入到服务器组中，将请求发送到该服务器。同时，当该服务器组有一段时间没有被修改，将最忙的服务器从服务器组中删除，以降低复制的 程度。

- Destination Hashing

  "目标地址散列"调度算法根据请求的目标IP地址，作为散列键（Hash Key）从静态分配的散列表找出对应的服务器，若该服务器是可用的且未超载，将请求发送到该服务器，否则返回空。

- Source Hashing

> "源地址散列"调度算法根据请求的源IP地址，作为散列键（Hash Key）从静态分配的散列表找出对应的服务器，若该服务器是可用的且未超载，将请求发送到该服务器，否则返回空。

ALU

Arithmetic Logical Unit can be regarded as the application logical services in distributed system.

- Vertical Partition: Break up a service into several services
- Horizontal Partition: Hold more machines to deal with the same service

Data Storage

Behavior Question

Introduction

Behavior questions are very common in every interview. It is an important part to evaluate you and company if both are fit in culture aspect.

Typical Questions

Why our company?

Keypoints:

- Look through the website of this company, 'about' page.
- Learn about the area, marketing trend around the product of this company.
- Promising area, especially combined with technology and big data.
- Product is amazing. handle those scalable problems.
- Engineer-driven environment, respect to technology and engineer.
- Start-up, enthusiastic young people.

Why this position? (Why Software Engineer?)

Keypoints:

- Because love engineering, love creative work, since childhood.
- Coding is an art or craft, not just engineering.
- Also, the academic background and experience make it so.
- Techniques matched or you're interested in techniques in this company.

What would you want to acquire from this position?

Keypoints:

- Knowledge, learn more about cutting-edge technology, talk about you like learning.
- Sense of achievement, conquer those tough problem.
- Improvement on personal ability on multiple aspects.

What should you do if you have different opinion with your colleague?

Keypoints:

- Talk about your opinion, do not hide your idea.
- Make sure your idea is reasonable, has enough resource to support it.
- But the way to talk should be careful.
- Show your opinion and discuss or compare the trade-off on different idea.
- Think about another side, consider if you were that person.

How would you convince other people to adopt your suggestion?

Keypoints:

- Think wider, diversity of solution.
- Evidence search.

Example:

I think I'm not a person like to convince somebody. I don't like to judge something. I prefer to provide different aspects or solutions to other people instead of giving them a judgment. There are lot of cases from me. But I think the most important thing is trying to think more wider and more potential solutions. Do more evidence research on that. Of course you can choose one aspect or solution you most prefer, but firstly you need have more evidence to support that. Think about how to convince yourself, then convince other people.

What would you do if your boss did something 100% wrong?

Keypoints:

- Depend on the character of your boss
- Different way for different boss
- But final purpose is point it out. Because you cannot veil the wrong thing.

What's your most challenge project or achievement?

This question is very common and almost every company will ask this question. It could be an attempt to see how will you identify weaknesses that you had and what did you do to overcome them. While figuring something out is how you see a challenge, this question is aimed a bit at seeing where did you grow the most from being pushed to your limit. Here would be the main points of the question:

Keypoints:

- Background of where you were at the start of this project.
- General nature of the project.
- Results from the project, so that this is concluding the basics of the story.
- Explain where I had problems and difficulties in getting this done, what did I do to conquer them and what other changes I'd make if I was in a similar situation now.

Example:

I'd like to talk about the experience when I did my research assistant. The following are typical difficulties:

1. Took a project totally from other people's idea and build the program upon the given prototype: It looks normal. Because when you enter a big company, this is the similar scenario. You'll start to build something sophisticate from others' idea and for others. But the idea on this project are based on many ideas from previous research. Not only about the programming, but also some user behavior and psychological research. Before everything start, you have to read many paper then you can start to do something but you cannot make excuse for slow progress. So quick learning across multiple fields to keep making progress.

2. Deadline and requirement from professor: The retrieval process speed is not ideal. When I almost finished the system, I need to reconstruct the system architecture. Then I figure it out by using a kind of user profile and action driven pattern. The data which concerning about user profile is loaded when user login and the task and topic data indexing start when user begin to choose their search task. So when user start to search, our indexed data is already loaded and also only partial data which only support the current task is loaded instead of load too much unnecessary data.

What would you do if you're facing something impossible? For example, a chanllenge thing but near the dealine.

- Communication, don't avoid difficulties, don't beat around the bush
-

Have you did more than was required during doing a project?

Yes, but not too much than required. You shouldn't forget this is a team project. Y

Have you did something creative?

Did you do something simplify?

Please describe what's your daily work look like?

What's your future plan?

- Learning on this field
- Learning outside of this field
- Interest

Questions on Past Projects

How to introduce your project?

General Introduction

- What (Overview) Non-technical brief intro.
- Why (The Purpose)
    - Problems you want to resolve
    - Value of this project
- How (Technical Backgroud)
    - Data
    - Services
    - Architecture

Individual Contribution

- What you did?
    - Modules
    - Function
- How you did?
    - Based on what kind of techniques
    - General Procedure

Chanllenge parts on Projects

Individual Difficulty

1. Knowledge Quick Learning
    - API document learning
    - Example Code on Github
2. Configuration Staff (e.g. Cloud Platform Devops)
    - Patient

Group Difficulty

1. Trade-off communication
2. Deadline Dilemma (Completeness and Flawless)