Algorithms Collection

kennyzhuang

Table of Contents

Algorithms Collections

A collection of algorithm problems and code bases for study and interview preparation.

Unsolved Problems

Collection of unsolved problems

Study carefully again !!!!

Burst Balloons

Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get nums[left] *nums[i]* nums[right] coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note: (1) You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not burst them. (2) 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

Example:

Given [3, 1, 5, 8]

Return 167

```
nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
coins =  3*1*5      +  3*5*8    +  1*3*8      + 1*8*1   = 167
```

https://leetcode.com/problems/burst-balloons/

Solution I: brute force

```
public class Solution {
    public int maxCoins(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        List<Integer> list = new ArrayList<>();
        for (int num : nums) {
            list.add(num);
        }

        return maxCoints(list);
    }

    public int maxCoints(List<Integer> list) {
        int max = 0;
        if (list.size() == 1) {
            return list.get(0);
        }
        for (int i = 0; i < list.size(); i++) {
            int left = i == 0 ? 1 : list.get(i-1);
            int right = i == list.size()-1 ? 1 : list.get(i+1);
            int n = left * list.get(i) * right;
            List<Integer> tmp = new ArrayList<>(list);
            tmp.remove(i);
            max = Math.max(max, n + maxCoints(tmp));
        }
        return max;
    }
}
```

Analysis

Be Naive First

When I first get this problem, it is far from dynamic programming to me. I started with the most naive idea the backtracking.

We have n balloons to burst, which mean we have n steps in the game. In the i th step we have n-i balloons to burst, i = 0~n-1. Therefore we are looking at an algorithm of O(n!). Well, it is slow, probably works for n < 12 only.

Of course this is not the point to implement it. We need to identify the redundant works we did in it and try to optimize.

Well, we can find that for any balloons left the maxCoins does not depends on the balloons already bursted. This indicate that we can use memorization (top down) or dynamic programming (bottom up) for all the cases from small numbers of balloon until n balloons. How many cases are there? For k balloons there are C(n, k) cases and for each case it need to scan the k balloons to compare. The sum is quite big still. It is better than O(n!) but worse than O(2^n).

Better idea

We then think can we apply the divide and conquer technique? After all there seems to be many self similar sub problems from the previous analysis.

Well, the nature way to divide the problem is burst one balloon and separate the balloons into 2 sub sections one on the left and one one the right. However, in this problem the left and right become adjacent and have effects on the maxCoins in the future.

Then another interesting idea come up. Which is quite often seen in dp problem analysis. That is reverse thinking. Like I said the coins you get for a balloon does not depend on the balloons already burst. Therefore instead of divide the problem by the first balloon to burst, we divide the problem by the last balloon to burst.

Why is that? Because only the first and last balloons we are sure of their adjacent balloons before hand!

For the first we have nums[i-1]*nums[i]*nums[i+1] for the last we have nums[-1]*nums[i]*nums[n].

OK. Think about n balloons if i is the last one to burst, what now?

We can see that the balloons is again separated into 2 sections. But this time since the balloon i is the last balloon of all to burst, the left and right section now has well defined boundary and do not affect each other! Therefore we can do either recursive method with memoization or dp.

Final

Here comes the final solutions. Note that we put 2 balloons with 1 as boundaries and also burst all the zero balloons in the first round since they won't give any coins. The algorithm runs in O(n^3) which can be easily seen from the 3 loops in dp solution.

https://leetcode.com/discuss/72216/share-some-analysis-and-explanations

Solution II: DP, memorize repeated search

```
public int maxCoins(int[] iNums) {
    int[] nums = new int[iNums.length + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int[][] memo = new int[n][n];
    return burst(memo, nums, 0, n - 1);
}

public int burst(int[][] memo, int[] nums, int left, int right) {
    if (left + 1 == right) return 0;
    if (memo[left][right] > 0) return memo[left][right];
    int ans = 0;
    for (int i = left + 1; i < right; ++i)
        ans = Math.max(ans, nums[left] * nums[i] * nums[right]
        + burst(memo, nums, left, i) + burst(memo, nums, i, right)
    memo[left][right] = ans;
    return ans;
}
```

Solution III: DP, memorize repeated search

```
public int maxCoins(int[] iNums) {
    int[] nums = new int[iNums.length + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int[][] dp = new int[n][n];
    for (int k = 2; k < n; ++k)
        for (int left = 0; left < n - k; ++left) {
            int right = left + k;
            for (int i = left + 1; i < right; ++i)
                dp[left][right] = Math.max(dp[left][right],
                nums[left] * nums[i] * nums[right] + dp[left][i] +
        }

    return dp[0][n - 1];
}
```

Longest increasing sequences

*Given an unsorted array of integers, find the length of longest increasing subsequence.

For example, Given [10, 9, 2, 5, 3, 7, 101, 18], The longest increasing subsequence 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(n2) complexity.

Follow up: Could you improve it to O(n log n) time complexity?

https://leetcode.com/problems/longest-increasing-subsequence/*

Solution:

1. Dynamic programming O(N^2) Complexity

```
state function: res[j] means the longest increasing sequence end wi

res[i] = max(res[j], 1) 0 <= j <= i


public int lengthOfLIS(int[] nums) {
    if (nums.length == 0) {
        return 0;
    }
    int[] res = new int[nums.length];
    res[0] = 1;
    int max = 1;
    for (int i = 1; i < nums.length; i++) {
        res[i] = 1;
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j] && res[i] < res[j] + 1) {
                res[i] = res[j]+1;
            }
            max = Math.max(res[i], max);
        }
    }
    return max;
}
```

2. Binary search O(NlgN) complexity

```
Detailed explanations:
http://www.geeksforgeeks.org/longest-monotonically-increasing-subse


Reference: without extra memory
https://leetcode.com/discuss/71129/space-log-time-short-solution-wi


public int lengthOfLIS(int[] nums) {
    int N = 0, idx, x;
    for(int i = 0; i < nums.length; i++) {
        x = nums[i];
        if (N < 1 || x > nums[N-1]) {
            nums[N++] = x;
        } else if ((idx = Arrays.binarySearch(nums, 0, N, x)) < 0)
            nums[-(idx + 1)] = x;
        }
    }
    return N;
}
```

Optimized solution:

```
public class Solution {
    public int lengthOfLIS(int[] nums) {
        int len = 0, n = nums.length;
        for (int i: nums) {
            int j = BinarySearch(nums, 0, len-1, i);
            nums[j] = i;
            if (j == len) len++;
        }
        return len;
    }

    public int BinarySearch(int[] a, int l, int r, int key) { // f
        while (l <= r) {
            int m = l+(r-l)/2;
            if (a[m] >= key)
                r = m-1;
            else l = m+1;
        }
        return l;
    }
}
```

Extended study: patience sorting

https://en.wikipedia.org/wiki/Patience_sorting
https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/LongestIncreasingSubsequence.pdf

Top K words in a document

Ref: http://stackoverflow.com/questions/185697/the-most-efficient-way-to-find-top-k-frequent-words-in-a-big-word-sequence

1. **Basic solution**

   a. use a Hash table to record all words' frequency while traverse the whole word sequence. In this phase, the key is "word" and the value is "word-frequency". This takes O(n) time.

   b. sort the (word, word-frequency) pair; and the key is "word-frequency". This takes O(n*lg(n)) time with normal sorting algorithm.

   c. After sorting, we just take the first K words. This takes O(K) time.

   the total time is O(n+nlg(n)+K)

2. **use heap instead of complete sort**

   b'. build a heap of (word, word-frequency) pair with "word-frequency" as key. It takes O(n) time to build a heap;

   c' extract top K words from the heap. Each extraction is O(lg(n)). So, total time is O(k*lg(n)).

   To summarize, this solution cost time **O(n+k*lg(n))**.

3. **use bucket sort to reduce the time complexity**
4. **large number of words**: Have n map workers count frequencies on 1/nth of the text each, and for each word, send it to one of m reducer workers calculated based on the hash of the word. The reducers then sum the counts. Merge sort over the reducers' outputs will give you the most popular words in order of popularity
5. **use trie and heap**

   ref: http://www.geeksforgeeks.org/find-the-k-most-frequent-words-from-a-file/

6. **Optimized: use quick select O(n) time and O(k) space**

ref: http://www.zrzahid.com/top-k-or-k-most-frequent-words-in-a-document/

```
    public String[] topKWordsSelect(final String stream, final int
    final Map<String, Integer> frequencyMap = new HashMap<String, 

    final String[] words = stream.toLowerCase().trim().split(" ");
    for (final String word : words) {
        int freq = 1;
        if (frequencyMap.containsKey(word)) {
            freq = frequencyMap.get(word) + 1;
        }

        // update the frequency map
        frequencyMap.put(word, freq);
    }

    // Find kth largest frequency which is same as (n-k)th smallest
    final int[] frequencies = new int[frequencyMap.size()];
    int i = 0;
    for (final int value : frequencyMap.values()) {
        frequencies[i++] = value;
    }
    final int kthLargestFreq = kthSmallest(frequencies, 0, i - 1, 

    // extract the top K
    final String[] topK = new String[k];
    i = 0;
    for (final java.util.Map.Entry<String, Integer> entry : frequen
        if (entry.getValue().intValue() >= kthLargestFreq) {
            topK[i++] = entry.getKey();
            if (i == k) {
                break;
            }
        }
    }

    return topK;
}
```

Sort Color

https://leetcode.com/problems/sort-colors/

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.

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

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

Solution:

```
public class Solution {
    public void sortColors(int[] nums) {
        if (nums.length <= 1) {
            return;
        }

        int i = 0, j = nums.length-1;
        int k = 0;
        while (k <= j) {
            if (nums[k] == 0) {
                swap(nums, i, k);
                i++;
                k++;
            }
            else if (nums[k] == 1) {
                k++;
            }
            else {
                swap(nums, j, k);
                j--;
            }
        }
    }

    public void swap(int[] nums, int i, int j) {
        int t = nums[i];
        nums[i] = nums[j];
        nums[j] = t;
    }
}
```

Permutations with duplicates

1. Non-recursive solution

```
public List<List<Integer>> permuteUnique(int[] nums) {
    List<List<Integer>> res=new ArrayList<List<Integer>>();
    if(nums.length==0) return res;
    res.add(new ArrayList<Integer>());
    for(int n:nums){
        List<List<Integer>> next=new ArrayList<List<Integer>>();
        for(List<Integer> list:res){
            for(int i=0;i<=list.size();i++){
                List<Integer> nextList=new ArrayList<Integer>(list
                if(i==list.size()) {nextList.add(n); next.add(next
                nextList.add(i,n);
                next.add(nextList);
                if(list.get(i)==n) break; // this line deal with t
            }
        }
        res=next;
    }
    return res;
}
```

1. Recursive solution

```
public class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        Arrays.sort(nums);
        LinkedList<Integer> list = new LinkedList<Integer>();
        for (int num : nums) list.add(num);
        permute(list, 0, res);
        return res;
    }
    private void permute(LinkedList<Integer> nums, int start, List<
        if (start == nums.size() - 1){
            res.add(new LinkedList<Integer>(nums));
            return;
        }
        for (int i = start; i < nums.size(); i++){
            if (i > start && nums.get(i) == nums.get(i - 1)) // thi
                continue;
            nums.add(start, nums.get(i)); // put ith element in the
            nums.remove(i + 1); // remove the moved element, now it
            permute(nums, start + 1, res); // permute the rest
            nums.add(i + 1, nums.get(start)); // put the moved elem
            nums.remove(start); // delete the residual
        }
    }
}
```

Find Kth node in BST

https://leetcode.com/problems/kth-smallest-element-in-a-bst/

Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.

Note: You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

Follow up: What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Solution 1: Inorder traversal

```
public class Solution {
    public int kthSmallest(TreeNode root, int k) {
        if (root == null || k <= 0) {
            return -1;
        }

        Stack<TreeNode> stack = new Stack();

        while (!stack.isEmpty() || root != null) {
            if (root != null) {
                stack.push(root);
                root = root.left;
            }
            else {
                root = stack.pop();
                k--;
                if (k == 0) {
                    return root.val;
                }
                root = root.right;
            }
        }

        return -1;
    }
}
```

Followup: data structure

ref: http://www.geeksforgeeks.org/find-k-th-smallest-element-in-bst-order-statistics-in-bst/

idea:

The idea is to maintain rank of each node. We can keep track of elements in a subtree of any node while building the tree. Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Assume that the root is having N nodes in its left subtree. If K = N + 1, root is K-th node. If K < N, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If K > N + 1, we continue our search in the right subtree for the (K – N – 1)-th smallest element. Note that we need the count of elements in left subtree only.

Time complexity: O(h) where h is height of tree.

Algorithm:

```
start:
if K = root.leftElement + 1
   root node is the K th node.
   goto stop
else if K > root.leftElements
   K = K - (root.leftElements + 1)
   root = root.right
   goto start
else
   root = root.left
   goto srart

stop:
```

Buy and sell stock with cooldown

Best Time to Buy and Sell Stock with Cooldown

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again). After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day) Example:

prices = [1, 2, 3, 0, 2]

maxProfit = 3

transactions = [buy, sell, cooldown, buy, sell]

Solution: DP

ref: https://leetcode.com/discuss/71391/easiest-java-solution-with-explanations

```
Here I share my no brainer weapon when it comes to this kind of pro

1. Define States

To represent the decision at index i:

buy[i]: Max profit till index i. The series of transaction is endin
sell[i]: Max profit till index i. The series of transaction is endi
To clarify:

Till index i, the buy / sell action must happen and must be the las
In the end n - 1, return sell[n - 1]. Apparently we cannot finally 
For special case no transaction at all, classify it as sell[i], so 
2. Define Recursion

buy[i]: To make a decision whether to buy at i, we either take a re
sell[i]: To make a decision whether to sell at i, we either take a 
So we get the following formula:

buy[i] = Math.max(buy[i - 1], sell[i - 2] - prices[i]);
sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]);
3. Optimize to O(1) Space

DP solution only depending on i - 1 and i - 2 can be optimized usin

Let b2, b1, b0 represent buy[i - 2], buy[i - 1], buy[i]
Let s2, s1, s0 represent sell[i - 2], sell[i - 1], sell[i]
Then arrays turn into Fibonacci like recursion:

b0 = Math.max(b1, s2 - prices[i]);
s0 = Math.max(s1, b1 + prices[i]);
4. Write Code in 5 Minutes

First we define the initial states at i = 0:

We can buy. The max profit at i = 0 ending with a buy is -prices[0]
We cannot sell. The max profit at i = 0 ending with a sell is 0.
```

Code:

```
public int maxProfit(int[] prices) {
    if(prices == null || prices.length <= 1) return 0;

    int b0 = -prices[0], b1 = b0;
    int s0 = 0, s1 = 0, s2 = 0;

    for(int i = 1; i < prices.length; i++) {
        b0 = Math.max(b1, s2 - prices[i]);
        s0 = Math.max(s1, b1 + prices[i]);
        b1 = b0; s2 = s1; s1 = s0;
    }
    return s0;
}
```

H index II

https://leetcode.com/problems/h-index-ii/

Solution I : Binary search

```
public class Solution {
    public int hIndex(int[] citations) {
        int start = 0;
        int end = citations.length-1;
        int len = citations.length;
        int result = 0;
        int mid;
        while(start <= end){
            mid = start + (end-start)/2;
            if(citations[mid] >= (len - mid)){
                result = (len-mid); // nice trick if cant find the
                end = mid-1;
            }
            else{
                start = mid + 1;
            }
        }
        return result;
    }
}
```

Solution II: Standard binary search

Mind the equal condition

```
public class Solution {
    public int hIndex(int[] citations) {
        if(citations == null || citations.length == 0) return 0;
        int l = 0, r = citations.length;
        int n = citations.length;
        while(l < r){
            int mid = l + (r - l) / 2;
            if(citations[mid] == n - mid) return n - mid;
            if(citations[mid] < citations.length - mid) l = mid + 
            else r = mid;
        }
        return n - l;
    }
}
```

```
public int hIndex(int[] citations) {
    int l = 0, r = citations.length - 1;
    for (int m = (l + r) / 2; l <= r; m = (l + r) / 2)
        if (citations[m] < citations.length - m) l = m + 1;
        else r = m - 1;
    return citations.length - l;
}
```

Check out more methods:

https://leetcode.com/discuss/56122/standard-binary-search

BackPack

I

http://www.lintcode.com/en/problem/backpack/#

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

Wrong solution:

State: res[i][j] means the maximum size using first i items to fill backpack of size j

why wrong:

the items are used repeatedly in this solution

```
public int backPack(int m, int[] A) {
        // write your code here
        if (A.length == 0) {
            return 0;
        }
        Arrays.sort(A);
        int[] a = new int[m+1];
        //int[] b = new int[A.length];
        a[0] = 0;
        for (int i = 1; i <= m; i++) {
            for (int j = A.length-1; j >= 0; j--) {
                if (i >= A[j]) {
                    a[i] = Math.max(a[i], a[i-A[j]] + A[j]);
                }
            }
        }
        return a[m];
    }
```

Solution: 2D DP I

Note: the state should be:

d[i][j] means whether size j can be filled using first i items

d[i][j] = d[i-1][j] || (j>=A[i-1] && d[i-1][j-A[i-1]]).

```
public int backPack(int m, int[] A) {
        // write your code here
        boolean[][] res = new boolean[A.length+1][m+1];
        res[0][0] = true;
        for (int i=1; i<=A.length; i++) {
            for (int j=0; j<=m; j++) {
                res[i][j] = res[i-1][j] || (j-A[i-1]>=0 && res[i
            }
        }
        for (int j=m; j>=0; j--) {
            if (res[A.length][j]) return j;
        }
        return 0;
    }
```

Solution II: 1D DP

d[i][j] = d[i-1][j] || (j>=A[i-1] && d[i-1][j-A[i-1]]) can be reduced to :

d[j] = d[j] || (j>=A[i-1] && d[j-A[i-1]])

since the state function only has d[i] and d[i-1], but note that j must decrease from m to 0 to avoid overwriting d[j-1]

```
public int backPack(int m, int[] A) {
    if (A.length==0) return 0;

    int len = A.length;
    boolean[] size = new boolean[m+1];
    Arrays.fill(size,false);
    size[0] = true;
    for (int i=1;i<=len;i++)
        for (int j=m;j>=0;j--){
            if (j-A[i-1]>=0 && size[j-A[i-1]])
                size[j] = size[j-A[i-1]];
        }

    for (int i=m; i>=0;i--)
        if (size[i]) return i;

    return 0;
}
```

II

http://www.lintcode.com/en/problem/backpack-ii/

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

1) Optimal Substructure: To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set. Therefore, the maximum value that can be obtained from n items is max of following two values. 1) Maximum value obtained by n-1 items and W weight (excluding nth item). 2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

```
public int backPackII(int m, int[] A, int V[]) {
        // write your code here
        if (A.length == 0 || A.length != V.length) {
            return 0;
        }

        int[] res = new int[m+1];
        for (int i = 0; i < A.length; i++) {
            for (int j = m; j >= 1; j--) {
                int k = j >= A[i] ? res[j-A[i]] + V[i] : 0;
                res[j] = Math.max(res[j], k);
            }
        }
        return res[m];
    }
```

DP - minimum adjustment cost

http://www.lintcode.com/en/problem/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.

Solution:

State: d[i][v] means for the first i numbers, the minimum cost to adjust number i to v while satisfying the difference smaller than target

transform function:

d[i][v] = min (d[i-1][v'] + abs(a[i]-v), d[i][v]) (|v-v'| <= target)

time complexity O(n A target)

```
public static int MinAdjustmentCost(ArrayList<Integer> A, int targe
        // write your code here
        if (A == null || A.size() == 0) {
            return 0;
        }

        // D[i][v]: 把index = i的值修改为v，所需要的最小花费
        int[][] D = new int[A.size()][101];

        int size = A.size();

        for (int i = 0; i < size; i++) {
            for (int j = 1; j <= 100; j++) {
                D[i][j] = Integer.MAX_VALUE;
                if (i == 0) {
                    // The first element.
                    D[i][j] = Math.abs(j - A.get(i));
                } else {
                    for (int k = 1; k <= 100; k++) {
                        // 不符合条件
                        if (Math.abs(j - k) > target) {
                            continue;
                        }

                        int dif = Math.abs(j - A.get(i)) + D[i
                        D[i][j] = Math.min(D[i][j], dif);
                    }
                }
            }
        }

        int ret = Integer.MAX_VALUE;
        for (int i = 1; i <= 100; i++) {
            ret = Math.min(ret, D[size - 1][i]);
        }

        return ret;
    }
```
复制代码

Search in Rotated sorted array with duplicates

https://leetcode.com/problems/search-in-rotated-sorted-array-ii/

Solution I: iterative

```
public boolean search(int[] nums, int target) {
    int left=0;
    int right=nums.length-1;

    while(left<=right){
        int mid = (left+right)/2;
        if(nums[mid]==target)
            return true;

        if(nums[left]<nums[mid]){
            if(nums[left]<=target&& target<nums[mid]){
                right=mid-1;
            }else{
                left=mid+1;
            }
        }else if(nums[left]>nums[mid]){
            if(nums[mid]<target&&target<=nums[right]){
                left=mid+1;
            }else{
                right=mid-1;
            }
        }else{
            left++;
        }
    }

    return false;
}
```

Solution II: recursive

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

    public boolean search(int[] nums, int left, int right, int targ
        if (left > right) {
            return false;
        }

        int mid = left + (right-left)/2;
        if (nums[mid] == target) {
            return true;
        }

        if (nums[left] < nums[mid]) {
            if (target >= nums[left] && target < nums[mid]) {
                return search(nums, left, mid-1, target);
            }
            return search(nums, mid+1, right, target);
        }
        else if (nums[left] > nums[mid]) {
            if (target > nums[mid] && target <= nums[right]) {
                return search(nums, mid+1, right, target);
            }
            return search(nums, left, mid-1, target);
        }
        else {
            return search(nums, left+1, right, target);
        }
    }
}
```

Flatten binary tree to linked list

https://leetcode.com/problems/flatten-binary-tree-to-linked-list/

Solution I: using stack, pre order traversal

```
public class Solution {
    public void flatten(TreeNode root) {
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode p = root;

        while(p != null || !stack.empty()){

            if(p.right != null){
                stack.push(p.right);
            }

            if(p.left != null){
                p.right = p.left;
                p.left = null;
            }else if(!stack.empty()){
                TreeNode temp = stack.pop();
                p.right=temp;
            }

            p = p.right;
        }
    }
}
```

Solution II: in place

note the use of variable post

思路是先遍历右子树，然后左子树，最后根节点。这样的好处是遍历完一个子树后，子树里的所有左右孩子指针都不需要了。 递归调用返回该子树中第一个节点，用于后续调用的后继。

```
public class Solution {
  public void flatten(TreeNode root){
        if (root == null){
            return;
        }

        flat(root, null);
    }

    private TreeNode flat(TreeNode root, TreeNode post){
        if (root == null){
            return post;
        }
        TreeNode right = flat(root.right, post);
        TreeNode left = flat(root.left, right);
        root.right = left;
        root.left = null;
        return root;
    }
}
```

Minimum height trees

Analysis: At most two minimum height trees

https://leetcode.com/discuss/71763/share-some-thoughts

First let's review some statement for tree in graph theory:

(1) A tree is an undirected graph in which any two vertices are connected by exactly one path. (2) Any connected graph who has n nodes with n-1 edges is a tree. (3) The degree of a vertex of a graph is the number of edges incident to the vertex. (4) A leaf is a vertex of degree 1. An internal vertex is a vertex of degree at least 2. (5) A path graph is a tree with two or more vertices that is not branched at all. (6) A tree is called a rooted tree if one vertex has been designated the root. (7) The height of a rooted tree is the number of edges on the longest downward path between root and a leaf. OK. Let's stop here and look at our problem.

Our problem want us to find the minimum height trees and return their root labels. First we can think about a simple case -- a path graph.

For a path graph of n nodes, find the minimum height trees is trivial. Just designate the middle point(s) as roots.

Despite its triviality, let design a algorithm to find them.

Suppose we don't know n, nor do we have random access of the nodes. We have to traversal. It is very easy to get the idea of two pointers. One from each end and move at the same speed. When they meet or they are one step away, (depends on the parity of n), we have the roots we want.

This gives us a lot of useful ideas to crack our real problem.

For a tree we can do some thing similar. We start from every end, by end we mean vertex of degree 1 (aka leaves). We let the pointers move the same speed. When two pointers meet, we keep only one of them, until the last two pointers meet or one step away we then find the roots.

It is easy to see that the last two pointers are from the two ends of the longest path in the graph.

The actual implementation is similar to the BFS topological sort. Remove the leaves, update the degrees of inner vertexes. Then remove the new leaves. Doing so level by level until there are 2 or 1 nodes left. What's left is our answer!

The time complexity and space complexity are both O(n).

Note that for a tree we always have V = n, E = n-1.

JAVA:

```java
public List<Integer> findMinHeightTrees(int n, int[][] edges) {
    if (n == 1) return Collections.singletonList(0);

    List<Set<Integer>> adj = new ArrayList<>(n);
    for (int i = 0; i < n; ++i) adj.add(new HashSet<>());
    for (int[] edge : edges) {
        adj.get(edge[0]).add(edge[1]);
        adj.get(edge[1]).add(edge[0]);
    }

    List<Integer> leaves = new ArrayList<>();
    for (int i = 0; i < n; ++i)
        if (adj.get(i).size() == 1) leaves.add(i);

    while (n > 2) {
        n -= leaves.size();
        List<Integer> newLeaves = new ArrayList<>();
        for (int i : leaves) {
            int j = adj.get(i).iterator().next();
            adj.get(j).remove(i);
            if (adj.get(j).size() == 1) newLeaves.add(j);
        }
        leaves = newLeaves;
    }
    return leaves;
}

// Runtime: 53 ms
```

Serialize and deserialize binary tree

Solution I: preorder, recursive

```
public String serialize(TreeNode root) {
    ArrayList<Integer> result = new ArrayList<Integer>();
    serializeHelper(root,result);
    return result.toString();
}

private void serializeHelper(TreeNode root, ArrayList<Integer> resu
    if (root == null) {
        result.add(null);
        return;
    }
    result.add(root.val);
    serializeHelper(root.left,result);
    serializeHelper(root.right,result);
}

// Decodes your encoded data to tree.
public TreeNode deserialize(String data) {
    String[] strArray = data.substring(1,data.length()-1).split(",
    Deque<String> strList = new LinkedList<String>(Arrays.asList(st
    return deserializeHelper(strList);
}

private TreeNode deserializeHelper(Deque<String> strList){
    if (strList.size() == 0) return null;
    String str = strList.pop();
    if (str.equals("null")) return null;
    TreeNode currentRoot = new TreeNode(Integer.parseInt(str));
    currentRoot.left = deserializeHelper(strList);
    currentRoot.right = deserializeHelper(strList);
    return currentRoot;
}
```

Find the Duplicate Number

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

Note: You must not modify the array (assume the array is read only). You must use only constant, O(1) extra space. Your runtime complexity should be less than O(n2). There is only one duplicate number in the array, but it could be repeated more than once.

https://leetcode.com/problems/find-the-duplicate-number/

Solution:

This approach essentially creates a graph out of the array (outgoing edge from each array value to the index denoted by that array value). Then, you run Floyd's cycle finding algorithm (tortoise and hare) to find the cycle (duplicate value in array). Look up Floyd's cycle finding algorithm for more info!

Simialr to find cycle in linkedList; i.e: 1 2 3 4 2

LinkedList: 0 -> 1 -> 2 -> 3 -> 4 -> 2

So cycle starts at 2, meaning 2 is the duplicate value

```
public class Solution {
    public int findDuplicate(int[] nums) {
        if (nums == null || nums.length <= 1) {
            return -1;
        }

    int slow = 0;
    int fast = 0;
    int finder = 0;

    while (true)
    {
        slow = nums[slow];
        fast = nums[nums[fast]];

        if (slow == fast)
            break;
    }
    while (true)
    {
        slow = nums[slow];
        finder = nums[finder];
        if (slow == finder)
            return slow;
    }
    }
}
```

Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example, Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.

The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. Thanks Marcos for contributing this image!

Solution I:

```
public class Solution {
    public int trap(int[] height) {
        if (height == null || height.length == 0) {
            return 0;
        }

        int[] left = new int[height.length];
        left[0] = height[0];

        for (int i = 1; i < height.length; i++) {
            left[i] = Math.max(left[i-1], height[i]);
        }

        int[] right = new int[height.length];
        right[height.length-1] = height[height.length-1];

        for (int i = height.length-2; i >= 0; i--) {
            right[i] = Math.max(right[i+1], height[i]);
        }

        int trappedWater = 0;

        for (int i = 0; i < height.length; i++) {
            int curr = Math.min(left[i], right[i]);
            if (curr > height[i]) {
                trappedWater +=  (curr - height[i]);
            }
        }

        return trappedWater;
    }
}
```

Solution II:

上面的方法非常容易理解，实现思路也很清晰，不过要进行两次扫描，复杂度前面的常数得是2，接下来我们要介绍另一种方法，相对不是那么好理解，但是只需要一次扫描就能完成。基本思路是这样的，用两个指针从两端往中间扫，在当前窗口下，如果哪一侧的高度是小的，那么从这里开始继续扫，如果比它还小的，肯定装

水的瓶颈就是它了，可以把装水量加入结果，如果遇到比它大的，立即停止，重新判断左右窗口的大小情况，重复上面的步骤。这里能作为停下来判断的窗口，说明肯定比前面的大了，所以目前肯定装不了水（不然前面会直接扫过去）。这样当左右窗口相遇时，就可以结束了，因为每个元素的装水量都已经记录过了。

http://blog.csdn.net/linhuanmars/article/details/20888505

```
public int trap(int[] A) {
    if(A==null || A.length ==0)
        return 0;
    int l = 0;
    int r = A.length-1;
    int res = 0;
    while(l<r)
    {
        int min = Math.min(A[l],A[r]);
        if(A[l] == min)
        {
            l++;
            while(l<r && A[l]<min)
            {
                res += min-A[l];
                l++;
            }
        }
        else
        {
            r--;
            while(l<r && A[r]<min)
            {
                res += min-A[r];
                r--;
            }
        }
    }
    return res;
}
```

First Missing Positive

Given an unsorted integer array, find the first missing positive integer.

For example, Given [1,2,0] return 3, and [3,4,-1,1] return 2.

Your algorithm should run in O(n) time and uses constant space.

Misunderstanding on the problem:

find the first missing positive, not the missing consecutive number !!!

i.e. 7 5 8 return 1 not 6

Solution:

zero based:

```
int firstMissingPositiveAnd0(int A[]) {
    int n = A.length;
    for (int i = 0; i < n; i++) {
        // when the ith element is not i
        while (A[i] != i) {
            // no need to swap when ith element is out of range [0,
            if (A[i] < 0 || A[i] >= n)
                break;

            //handle duplicate elements
            if(A[i]==A[A[i]])
                            break;
            // swap elements
            int temp = A[i];
            A[i] = A[temp];
            A[temp] = temp;
        }
    }

    for (int i = 0; i < n; i++) {
        if (A[i] != i)
            return i;
    }

    return n;
}
```

one based:

```
public int firstMissingPositive(int[] A) {
        int n = A.length;

        for (int i = 0; i < n; i++) {
            while (A[i] != i + 1) {
                if (A[i] <= 0 || A[i] >= n)
                    break;

                    if(A[i]==A[A[i]-1])
                            break;

                int temp = A[i];
                A[i] = A[temp - 1];
                A[temp - 1] = temp;
            }
        }

        for (int i = 0; i < n; i++){
            if (A[i] != i + 1){
                return i + 1;
            }
        }

        return n + 1;
}
```

Count of Smaller Numbers After Self

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

Example:

Given nums = [5, 2, 6, 1]

- To the right of 5 there are 2 smaller elements (2 and 1).
- To the right of 2 there is only 1 smaller element (1).
- To the right of 6 there is 1 smaller element (1).
- To the right of 1 there is 0 smaller element.

Return the array [2, 1, 1, 0].

Solution I: Bit manipulation

https://leetcode.com/discuss/74994/nlogn-divide-and-conquer-java-solution-based-bit-comparison

Solution II: Merge sort

https://leetcode.com/discuss/74110/11ms-java-solution-using-merge-sort-with-explanation

Solution III: Binary search tree

https://leetcode.com/discuss/73803/easiest-java-solution

https://leetcode.com/discuss/73762/9ms-short-java-bst-solution-get-answer-when-building-bst

10ms

```
public class Solution {
    public List<Integer> countSmaller(int[] nums) {
        List<Integer> res = new ArrayList<>();
        if(nums == null || nums.length == 0) return res;
        TreeNode root = new TreeNode(nums[nums.length - 1]);
        res.add(0);
        for(int i = nums.length - 2; i >= 0; i--) {
            int count = insertNode(root, nums[i]);
            res.add(count);
        }
        Collections.reverse(res);
        return res;
    }

    public int insertNode(TreeNode root, int val) {
        int thisCount = 0;
        while(true) {
            if(val <= root.val) {
                root.count++;
                if(root.left == null) {
                    root.left = new TreeNode(val); break;
                } else {
                    root = root.left;
                }
            } else {
                thisCount += root.count;
                if(root.right == null) {
                    root.right = new TreeNode(val); break;
                } else {
                    root = root.right;
                }
            }
        }
        return thisCount;
    }
}

class TreeNode {
    TreeNode left;
    TreeNode right;
    int val;
    int count = 1;
    public TreeNode(int val) {
        this.val = val;
    }
}
```

Solution IV: Binary search

53ms

```
public List<Integer> countSmaller(int[] nums) {
    Integer[] ans = new Integer[nums.length];
    List<Integer> sorted = new ArrayList<Integer>();
    for (int i = nums.length - 1; i >= 0; i--) {
        int index = findIndex(sorted, nums[i]);
        ans[i] = index;
        sorted.add(index, nums[i]);
    }
    return Arrays.asList(ans);
}
private int findIndex(List<Integer> sorted, int target) {
    if (sorted.size() == 0) return 0;
    int start = 0;
    int end = sorted.size() - 1;
    if (sorted.get(end) < target) return end + 1;
    if (sorted.get(start) >= target) return 0;
    while (start + 1 < end) {
        int mid = start + (end - start) / 2;
        if (sorted.get(mid) < target) {
            start = mid + 1;
        } else {
            end = mid;
        }
    }
    if (sorted.get(start) >= target) return start;
    return end;
}
```

Solution V: Segment tree

https://leetcode.com/discuss/73233/complicated-segmentree-solution-hope-to-find-a-better-one

```
public class Solution {
    static class segmentTreeNode {
        int start, end, count;
        segmentTreeNode left, right;
        segmentTreeNode(int start, int end, int count) {
            this.start = start;
            this.end = end;
            this.count = count;
            left = null;
            right = null;
        }
    }
    public static List<Integer> countSmaller(int[] nums) {
        // write your code here
        List<Integer> result = new ArrayList<Integer>();

        int min = Integer.MAX_VALUE, max = Integer.MIN_VALUE;
        for (int i : nums) {
            min = Math.min(min, i);

        }
        if (min < 0) {
            for (int i = 0; i < nums.length; i++) {
                nums[i] -= min;//deal with negative numbers, seems
            }
        }
        for (int i : nums) {
            max = Math.max(max, i);
        }
        segmentTreeNode root = build(0, max);
        for (int i = 0; i < nums.length; i++) {
            updateAdd(root, nums[i]);
        }
        for (int i = 0; i < nums.length; i++) {
            updateDel(root, nums[i]);
            result.add(query(root, 0, nums[i] - 1));
        }
        return result;
    }
    public static segmentTreeNode build(int start, int end) {
        if (start > end) return null;
        if (start == end) return new segmentTreeNode(start, end, 0)
        int mid = (start + end) / 2;
        segmentTreeNode root = new segmentTreeNode(start, end, 0);
        root.left = build(start, mid);
        root.right = build(mid + 1, end);
        root.count = root.left.count + root.right.count;
        return root;
    }

    public static int query(segmentTreeNode root, int start, int en
        if (root == null) return 0;
        if (root.start == start && root.end == end) return root.cou
        int mid = (root.start + root.end) / 2;
        if (end < mid) {
            return query(root.left, start, end);
        } else if (start > end) {
            return query(root.right, start, end);
        } else {
            return query(root.left, start, mid) + query(root.right,
        }
    }

    public static void updateAdd(segmentTreeNode root, int val) {
        if (root == null || root.start > val || root.end < val) ret
        if (root.start == val && root.end == val) {
            root.count ++;
            return;
        }
        int mid = (root.start + root.end) / 2;
        if (val <= mid) {
            updateAdd(root.left, val);
        } else {
            updateAdd(root.right, val);
        }
        root.count = root.left.count + root.right.count;
    }

    public static void updateDel(segmentTreeNode root, int val) {
        if (root == null || root.start > val || root.end < val) ret
        if (root.start == val && root.end == val) {
            root.count --;
            return;
        }
        int mid = (root.start + root.end) / 2;
        if (val <= mid) {
            updateDel(root.left, val);
        } else {
            updateDel(root.right, val);
        }
        root.count = root.left.count + root.right.count;
    }
}
```

Solution VI: Binary Indexed tree

https://leetcode.com/discuss/73233/complicated-segmentree-solution-hope-to-find-a-better-one

```
public class Solution {

    private void add(int[] bit, int i, int val) {
        for (; i < bit.length; i += i & -i) bit[i] += val;
    }

    private int query(int[] bit, int i) {
        int ans = 0;
        for (; i > 0; i -= i & -i) ans += bit[i];
        return ans;
    }

    public List<Integer> countSmaller(int[] nums) {
        int[] tmp = nums.clone();
        Arrays.sort(tmp);
        for (int i = 0; i < nums.length; i++) nums[i] = Arrays.bina
        int[] bit = new int[nums.length + 1];
        Integer[] ans = new Integer[nums.length];
        for (int i = nums.length - 1; i >= 0; i--) {
            ans[i] = query(bit, nums[i] - 1);
            add(bit, nums[i], 1);
        }
        return Arrays.asList(ans);
    }
}
```

Introduction to BIT:

https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

Largest Rectangle in Histogram

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].

The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example, Given height = [2,1,5,6,2,3], return 10.

https://leetcode.com/problems/largest-rectangle-in-histogram/

Solution I: Divide and conquer O(NlgN)

http://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/

A simple solution is to one by one consider all bars as starting points and calculate area of all rectangles starting with every bar. Finally return maximum of all possible areas. Time complexity of this solution would be O(n^2).

We can use Divide and Conquer to solve this in O(nLogn) time. The idea is to find the minimum value in the given array. Once we have index of the minimum value, the max area is maximum of following three values. a) Maximum area in left side of minimum value (Not including the min value) b) Maximum area in right side of minimum value (Not including the min value) c) Number of bars multiplied by minimum value. The areas in left and right of minimum value bar can be calculated recursively. If we use linear search to find the minimum value, then the worst case time complexity of this algorithm becomes O(n^2). In worst case, we always have (n-1) elements in one side and 0 elements in other side and if the finding minimum takes O(n) time, we get the recurrence similar to worst case of Quick Sort. How to find the minimum efficiently? Range Minimum Query using Segment Tree can be used for this. We build segment tree of the given histogram heights. Once the segment tree is built, all range minimum queries take O(Logn) time. So over all complexity of the algorithm becomes.

Overall Time = Time to build Segment Tree + Time to recursively find maximum area

Time to build segment tree is O(n). Let the time to recursively find max area be T(n). It can be written as following. T(n) = O(Logn) + T(n-1) The solution of above recurrence is O(nLogn). So overall time is O(n) + O(nLogn) which is O(nLogn).

Solution II: Use stack

```
public class Solution {
    public int largestRectangleArea(int[] height) {
        if (height == null || height.length == 0) {
            return 0;
        }

        int res = 0;

        Stack<Integer> stack = new Stack();
        int i = 0;
        while (i < height.length) {
            if (stack.isEmpty() || height[i] >= height[stack.peek(
                stack.push(i);
                i++;
            }
            else {
                int index = stack.pop();
                int len = stack.isEmpty() ? i : i - stack.peek() -
                res = Math.max(res, height[index]*len);
            }
        }
        // need to pop out all elements in the stack
        while (!stack.isEmpty()) {
            int index = stack.pop();
            int len = stack.isEmpty() ? i : i - stack.peek() - 1;
            res = Math.max(res, height[index]*len);
        }

        return res;
    }
}
```

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.

Solution I: Recursion

```
    public static boolean isScramble(String s1, String s2) {
        if(s1.length() != s2.length()){
            return false;
        }
        if(s1.length()==1 && s2.length()==1){
            return s1.charAt(0) == s2.charAt(0);
        }

        // 排序后可以通过
        char[] s1ch = s1.toCharArray();
        char[] s2ch = s2.toCharArray();
        Arrays.sort(s1ch);
        Arrays.sort(s2ch);
        if(!new String(s1ch).equals(new String(s2ch))){
            return false;
        }

        for(int i=1; i<s1.length(); i++){        // 至少分出一个字符出
            String s11 = s1.substring(0, i);
            String s12 = s1.substring(i);
            String s21 = s2.substring(0, i);
            String s22 = s2.substring(i);
//            System.out.println(s1 + "-" + s2 + ": "+ s11 + ", " -
            // 检测前半部是否匹配
            if(isScramble(s11, s21) && isScramble(s12, s22)){
                return true;
            }
            // 前半部不匹配，检测后半部是否匹配
            s21 = s2.substring(0, s2.length()-i);
            s22 = s2.substring(s2.length()-i);
            if(isScramble(s11, s22) && isScramble(s12, s21)){
                return true;
            }
        }
        return false;
    }
```

Solution II: Recursion with prune

这里的剪枝条件可以简单设为，所有字符的ASCII的值之和必须相等，这是成为scramble的一个充分条件

```
    public boolean isScramble2(String s1, String s2) {
        // Two quick recursion exits.
        if (s1.length() != s2.length())
            return false;
        if (s1.equals(s2))
            return true;

        // Prune candidates here to reduce candidate check times.
        int sum = 0;
        for (int i = 0; i < s1.length(); i++) {
            sum += s1.charAt(i) - 'a';
            sum -= s2.charAt(i) - 'a';
        }
        if (sum != 0)
            return false;

        // Partition and match recursively.
        for (int i = 1; i < s1.length(); ++i) {
            for (int j = 1; j < s2.length(); ++j) {
                // i and j are the partition indexes.
                String s1_left = s1.substring(0, i);
                String s1_right = s1.substring(i);
                String s2_left = s2.substring(0, j);
                String s2_right = s2.substring(j);

                if (isScramble2(s1_left, s2_right)
                        && isScramble2(s1_right, s2_left)
                        || isScramble2(s1_left, s2_left)
                        && isScramble2(s1_right, s2_right)) {
                    return true;
                }
            }
        }

        return false;
    }
```

Solution III: DP 3D

```
    public static boolean isScrambleDP(String s1, String s2) {
        int len = s1.length();
        if(len != s2.length()){
            return false;
        }
        if(len == 0){
            return true;
        }

        char[] c1 = s1.toCharArray();
        char[] c2 = s2.toCharArray();
        // canTransform 第一维为子串的长度delta，第二维为s1的起始索引，第
        // canTransform[k][i][j]表示s1[i...i+k]是否可以由s2[j...j+k]
        boolean[][][] canT = new boolean[len][len][len];
        for(int i=0; i<len; i++){
            for(int j=0; j<len; j++){    // 如果字符串总长度为1，则取决
                canT[0][i][j] = c1[i] == c2[j];
            }
        }

        for(int k=2; k<=len; k++){        // 子串的长度
            for(int i=len-k; i>=0; i--){            // s1[i...i+k]
                for(int j=len-k; j>=0; j--){    // s2[j...j+k]
                    boolean canTransform = false;
                    for(int m=1; m<k; m++){    // 尝试以m为长度分割子
                        // canT[k][i][j]
                        canTransform = (canT[m-1][i][j] && canT[k-
                                      (canT[m-1][i][j+k-m]
                        if(canTransform){
                            break;
                        }
                    }
                    canT[k-1][i][j] = canTransform;
                }
            }
        }

        return canT[len-1][0][0];
    }
```

Solution IV: use Map

不过以上解法非常的麻烦，而且其实这种3维的DP表里仍然会存储冗余的状态。自己想出了一种非常规的DP求解方法，发现可以用一个哈希表代替3维的DP表，减少对冗余状态的存储。另外这里的DP其实可以跟递归思路一样，自顶向下，而非所谓真正的自底向上的DP。

这里记录子问题求解结果的数据结构我用Map，前面其实偷了一下懒，理论上应该是类似于Pair一样的东西，就是任意两个字符串看成一个pair，然后记录是否为scramble的结果。但是如果真的手动写个类，还有覆盖equals方法，太麻烦了，于是这里偷懒一下，把Pair的形式替换成s1@s2，记成一种特殊字符串。 另外，这里提供了几个快速的递归出口，一个是如果两个字符串长度不相等，可以立刻返回false。另一个是如果两个字符串内容相等，则可以立刻返回true。

```
    // Memo for dp.
    private Map<String, Boolean> dp = new HashMap<String, Boolean>(
    public boolean isScramble(String s1, String s2) {
        String key = s1 + "@" + s2;
        if (dp.containsKey(key)) {
            return dp.get(key);
        }

        // Two quick recursion exits.
        if (s1.length() != s2.length()) {
            dp.put(key, false);
            return false;
        }
        if (s1.equals(s2)) {
            dp.put(key, true);
            return true;
        }

        // Partition and match recursively.
        for (int i = 1; i < s1.length(); ++i) {
            for (int j = 1; j < s2.length(); ++j) {
                // i and j are the partition indexes.
                String s1_left = s1.substring(0, i);
                String s1_right = s1.substring(i);
                String s2_left = s2.substring(0, j);
                String s2_right = s2.substring(j);

                if (isScramble2(s1_left, s2_right)
                        && isScramble2(s1_right, s2_left)
                        || isScramble2(s1_left, s2_left)
                        && isScramble2(s1_right, s2_right)) {
                    return true;
                }
            }
        }

        dp.put(key, false);
        return false;
    }
```

REF:

http://blog.csdn.net/fightforyourdream/article/details/17707187

http://blog.csdn.net/whuwangyi/article/details/14105063

N-Queens I, II

I:

https://leetcode.com/problems/n-queens/

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

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.

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

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

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

Solution:

N皇后问题是非常经典的问题了，记得当时搞竞赛第一道递归的题目就是N皇后。因为这个问题是典型的NP问题，所以在时间复杂度上就不用纠结了，肯定是指数量级的。下面我们来介绍这个题的基本思路。 主要思想就是一句话：用一个循环递归处理子问题。这个问题中，在每一层递归函数中，我们用一个循环把一个皇后填入对应行的某一列中，如果当前棋盘合法，我们就递归处理先一行，找到正确的棋盘我们就存储到结果集里面。 这种题目都是使用这个套路，就是用一个循环去枚举当前所有情况，然后把元素加入，递归，再把元素移除，这道题目中不用移除的原因是我们用一个一维数组去存皇后在对应行的哪一列，因为一行只能有一个皇后，如果二维数组，那么就需要把那一行那一列在递归结束后设回没有皇后，所以道理是一样的。 这道题最后一个细节就是怎么实现检查当前棋盘合法性的问题，因为除了刚加进来的那个皇后，前面都是合法的，我们只需要检查当前行和前面行是否冲突即可。检查是否同列很简单，检查对角线就是行的差和列的差的绝对值不要相等就可以。

这道题实现的方法是一个非常典型的套路，有许多题都会用到，基本上大部分NP问题的求解都是用这个方式，比如Sudoku Solver，Combination Sum，Combinations，Permutations，Word Break II，Palindrome Partitioning等，所以

大家只要把这个套路掌握熟练，那些题就都不在话下哈。

```
public class Solution {
    public List<List<String>> solveNQueens(int n) {
    List<List<String>> res = new ArrayList<List<String>>();
    helper(n,0,new int[n], res);
    return res;
}

private void helper(int n, int row, int[] columnForRow, List<List<S
{
    if(row == n)
    {
        List<String> item = new ArrayList<String>();
        for(int i=0;i<n;i++)
        {
            StringBuilder strRow = new StringBuilder();
            for(int j=0;j<n;j++)
            {
                if(columnForRow[i]==j)
                    strRow.append('Q');
                else
                    strRow.append('.');
            }
            item.add(strRow.toString());
        }
        res.add(item);
        return;
    }
    for(int i=0;i<n;i++)
    {
        columnForRow[row] = i;
        if(check(row,columnForRow))
        {
            helper(n,row+1,columnForRow,res);
        }
    }
}
private boolean check(int row, int[] columnForRow)
{
    for(int i=0;i<row;i++)
    {
        if(columnForRow[row]==columnForRow[i] || Math.abs(columnFo
            return false;
    }
    return true;
}
}
```

II: find count of solutions

use wrapper class

or use a unit length array int[1]

```
public class Solution {
    public class Count {
        int count;

        public Count() {
            count = 0;
        }
    }

    public int totalNQueens(int n) {
        if (n < 1) {
            return 0;
        }
        Count count = new Count();
        totalNQueens(n, 0, new int[n], count);

        return count.count;
    }

    public void totalNQueens(int n, int row, int[] colForRow, Count
        if (row == n) {
            count.count++;
            return;
        }
        for (int i = 0; i < n; i++) {
            colForRow[row] = i;
            if (isValid(row, colForRow)) {
                totalNQueens(n, row+1, colForRow, count);
            }
        }
    }

    public boolean isValid(int row, int[] colForRow) {
        for (int i = 0; i < row; i++) {
            if (colForRow[row] == colForRow[i] || Math.abs(colForRo
                return false;
            }
        }
        return true;
    }
}
```

Iterative solution:

```
public int totalNQueens(int n) {
    int ans = 0;
    int[] queens = new int[n];
    boolean[] c = new boolean[n + 1];
    boolean[] f = new boolean[2 * n];
    boolean[] b = new boolean[2 * n];
    c[n] = true; //dummy boundary
    int col = 0, row = 0;
    while (true) {
        if (c[col] || f[col + row] || b[col - row + n]) {
            if (row == n || col == n) {
                if (row == 0) return ans;
                if (row == n) ans++;
                col = queens[--row];
                c[col] = f[col + row] = b[col - row + n] = false;
            }
            col++;
        } else {
            c[col] = f[col + row] = b[col - row + n] = true;
            queens[row++] = col;
            col = 0;
        }
    }
}
```

Minimum Window Substring

https://leetcode.com/problems/minimum-window-substring/

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

For example, S = "ADOBECODEBANC" T = "ABC" Minimum window is "BANC".

Note: If there is no such window in S that covers all characters in T, return the empty string "".

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

注意：如果一个字母在T中出现了k次，那么在窗口中该字母至少出现k次

Solution:

这道题是字符串处理的题目，和Substring with Concatenation of All Words思路非常类似，同样是建立一个字典，然后维护一个窗口。区别是在这道题目中，因为可以跳过没在字典里面的字符（也就是这个串不需要包含且仅仅包含字典里面的字符，有一些不在字典的仍然可以满足要求），所以遇到没在字典里面的字符可以继续移动窗口右端，而移动窗口左端的条件是当找到满足条件的串之后，一直移动窗口左端直到有字典里的字符不再在窗口里。在实现中就是维护一个HashMap，一开始key包含字典中所有字符，value就是该字符的数量，然后遇到字典中字符时就将对应字符的数量减一。算法的时间复杂度是O(n),其中n是字符串的长度，因为每个字符再维护窗口的过程中不会被访问多于两次。空间复杂度则是O(字典的大小)，也就是代码中T的长度。

这个方法在Substring with Concatenation of All Words和Longest Substring Without Repeating Characters中都介绍过，属于一种类型的题目，只要掌握了思路便可以举一反三，都可以将这类问题降低到线性复杂度。

http://blog.csdn.net/linhuanmars/article/details/20343903

```
public String minWindow(String S, String T) {
    if(S==null || S.length()==0)
        return "";
    HashMap<Character, Integer> map = new HashMap<Character, Intege
    for(int i=0; i<T.length();i++)
    {
        if(map.containsKey(T.charAt(i)))
        {
            map.put(T.charAt(i),map.get(T.charAt(i))+1);
        }
        else
        {
            map.put(T.charAt(i),1);
        }
    }
    int left = 0;
    int count = 0;
    int minLen = S.length()+1;
    int minStart = 0;
    for(int right=0; right<S.length();right++)
    {
        if(map.containsKey(S.charAt(right)))
        {
            map.put(S.charAt(right),map.get(S.charAt(right))-1);
            if(map.get(S.charAt(right))>=0)
            {
                count++;
            }
            while(count == T.length())
            {
                if(right-left+1<minLen)
                {
                    minLen = right-left+1;
                    minStart = left;
                }
                if(map.containsKey(S.charAt(left)))
                {
                    map.put(S.charAt(left), map.get(S.charAt(left)
                    if(map.get(S.charAt(left))>0)
                    {
                        count--;
                    }
                }
                left++;
            }
        }
    }
    if(minLen>S.length())
    {
        return "";
    }
    return S.substring(minStart,minStart+minLen);
}
```

Solution II:

Notice how complicated the above solution is. It uses a hash table, a queue, and a sorted map. During an interview, the problems tend to be short and the solution usually can be coded in about 50 lines of code. So be sure that you say out loud what you are thinking and keep communication opened with the interviewer. Check if your approach is unnecessary complex, he/she might be able to give you guidance. The last thing you want to do is to get stuck in a corner and keep silent.

To help illustrate this approach, I use a different example: S = “acbbaca” and T = “aba“. The idea is mainly based on the help of two pointers (begin and end position of the window) and two tables (needToFind and hasFound) while traversing S. needToFind stores the total count of a character in T and hasFound stores the total count of a character met so far. We also use a count variable to store the total characters in T that’s met so far (not counting characters where hasFound[x] exceeds needToFind[x]). When count equals T‘s length, we know a valid window is found.

Each time we advance the end pointer (pointing to an element x), we increment hasFound[x] by one. We also increment count by one if hasFound[x] is less than or equal to needToFind[x]. Why? When the constraint is met (that is, count equals to T‘s size), we immediately advance begin pointer as far right as possible while maintaining the constraint.

How do we check if it is maintaining the constraint? Assume that begin points to an element x, we check if hasFound[x] is greater than needToFind[x]. If it is, we can decrement hasFound[x] by one and advancing begin pointer without breaking the constraint. On the other hand, if it is not, we stop immediately as advancing begin pointer breaks the window constraint.

Finally, we check if the minimum window length is less than the current minimum. Update the current minimum if a new minimum is found.

Essentially, the algorithm finds the first window that satisfies the constraint, then continue maintaining the constraint throughout.

```
public class Solution {
    public String minWindow(String S, String T) {
        HashMap<Character, Integer> dict = new HashMap<>();
        for (int i = 0; i < T.length(); i++) {
            char c = T.charAt(i);
            if (!dict.containsKey(c))
                dict.put(c, 1);
            else
                dict.put(c, dict.get(c) + 1);
        }
        HashMap<Character, Integer> found = new HashMap<>();
        int foundCounter = 0;
        int start = 0;
        int end = 0;
        int min = Integer.MAX_VALUE;
        String minWindow = "";
        while (end < S.length()) {
            char c = S.charAt(end);
            if (dict.containsKey(c)) {
                if (found.containsKey(c)) {
                    if (found.get(c) < dict.get(c))
                        foundCounter++;
                    found.put(c, found.get(c) + 1);
                } else {
                    found.put(c, 1);
                    foundCounter++;
                }
            }
            if (foundCounter == T.length()) {
                //When foundCounter equals to T.length(), in other
                char sc = S.charAt(start);
                while (!found.containsKey(sc) || found.get(sc) > di
                    if (found.containsKey(sc) && found.get(sc) > di
                        found.put(sc, found.get(sc) - 1);
                    start++;
                    sc = S.charAt(start);
                }
                if (end - start + 1 < min) {
                    minWindow = S.substring(start, end + 1);
                    min = end - start + 1;
                }
            }
            end++;
        }
        return minWindow;
    }
}
```

Solution III: O(1) space

```
public class Solution {
    public String minWindow(String S, String T) {
        if (S.length() == 0 || T.length() == 0 || S.length() < T.le
            return "";
        }

        int[] needToFind = new int[128];
        int[] hasFound = new int[128];

        for (char c : T.toCharArray()) {
            needToFind[c]++;
        }

        int count = 0, minWindowLen = S.length()+1;
        int minWindowBegin = 0, minWindowEnd = 0;

        int begin = 0, end = 0;

        for (end = 0; end < S.length(); end++) {
            // skip characters not in T
            if (needToFind[S.charAt(end)] == 0) continue;
            hasFound[S.charAt(end)]++;
            if (hasFound[S.charAt(end)] <= needToFind[S.charAt(end
                count++;

            // if window constraint is satisfied
            if (count == T.length()) {
            // advance begin index as far right as possible,
            // stop when advancing breaks window constraint.
                while (needToFind[S.charAt(begin)] == 0 ||
                    hasFound[S.charAt(begin)] > needToFind[S.charAt
                    if (hasFound[S.charAt(begin)] > needToFind[S.ch
                        hasFound[S.charAt(begin)]--;
                    begin++;
                }

                // update minWindow if a minimum length is met
                int windowLen = end - begin + 1;
                if (windowLen < minWindowLen) {
                    minWindowBegin = begin;
                    minWindowEnd = end;
                    minWindowLen = windowLen;
                } // end if
            } // end if
        } // end for

        if (minWindowLen == S.length()+1) {
            return "";
        }
        return S.substring(minWindowBegin, minWindowEnd+1);
    }
}
```

ref:

http://articles.leetcode.com/2010/11/finding-minimum-window-in-s-which.html

Substring with Concatenation of All Words

https://leetcode.com/problems/substring-with-concatenation-of-all-words/

You are given a string, s, and a list of words, words, that are all of the same length. Find all starting indices of substring(s) in s that is a concatenation of each word in words exactly once and without any intervening characters.

For example, given: s: "barfoothefoobarman" words: ["foo", "bar"]

You should return the indices: [0,9]. (order does not matter).

Similar problems:

- Longest Substring Which Contains 2 Unique Characters

  http://www.programcreek.com/2013/02/longest-substring-which-contains-2-unique-characters/

- Longest Substring Which Contains at most k Unique Characters
- Longest Substring Without Repeating Characters
- Minimum Window Substring

Solution:

```
public List<Integer> findSubstring(String s, String[] words) {
    ArrayList<Integer> result = new ArrayList<Integer>();
    if(s==null||s.length()==0||words==null||words.length==0){
        return result;
    }

    //frequency of words
    HashMap<String, Integer> map = new HashMap<String, Integer>();
    for(String w: words){
        if(map.containsKey(w)){
            map.put(w, map.get(w)+1);
        }else{
            map.put(w, 1);
        }
    }

    int len = words[0].length();

    for(int j=0; j<len; j++){
        HashMap<String, Integer> currentMap = new HashMap<String, I
        int start = j;//start index of start
        int count = 0;//count totoal qualified words so far

        for(int i=j; i<=s.length()-len; i=i+len){
            String sub = s.substring(i, i+len);
            if(map.containsKey(sub)){
                //set frequency in current map
                if(currentMap.containsKey(sub)){
                    currentMap.put(sub, currentMap.get(sub)+1);
                }else{
                    currentMap.put(sub, 1);
                }

                count++;

                while(currentMap.get(sub)>map.get(sub)){
                    String left = s.substring(start, start+len);
                    currentMap.put(left, currentMap.get(left)-1);

                    count--;
                    start = start + len;
                }


                if(count==words.length){
                    result.add(start); //add to result

                    //shift right and reset currentMap, count & sta
                    String left = s.substring(start, start+len);
                    currentMap.put(left, currentMap.get(left)-1);
                    count--;
                    start = start + len;
                }
            }else{
                currentMap.clear();
                start = i+len;
                count = 0;
            }
        }
    }

    return result;
}
```

Populating Next Right Pointers in Each Node II

Follow up for problem "Populating Next Right Pointers in Each Node".

What if the given tree could be any binary tree? Would your previous solution still work?

Note:

You may only use constant extra space. For example, Given the following binary tree,

```
         1
       /  \
      2    3
     / \    \
    4   5    7
```

After calling your function, the tree should look like:

```
         1 -> NULL
       /  \
      2 -> 3 -> NULL
     / \    \
    4-> 5 -> 7 -> NULL
```

https://leetcode.com/problems/populating-next-right-pointers-in-each-node-ii/

Solution:

I: with O(lgN) space:

level order traversal with a queue

II: with O(1) space:

首要是找到右孩子的第一个有效的next链接节点，然后再处理左孩子。然后依次递归处理右孩子，左孩子

```
/**
 * Definition for binary tree with next pointer.
 * public class TreeLinkNode {
 *     int val;
 *     TreeLinkNode left, right, next;
 *     TreeLinkNode(int x) { val = x; }
 * }
 */
public class Solution {
    public void connect(TreeLinkNode root) {
        // Note: The Solution object is instantiated only once and
        if (root == null) {
            return;
        }

        TreeLinkNode p = root.next;

        while (p != null) {
            if (p.left != null) {
                p = p.left;
                break;
            }
            if (p.right != null) {
                p = p.right;
                break;
            }
            p = p.next;
        }

        if (root.right != null) {
            root.right.next = p;
        }

        if (root.left != null) {
            root.left.next = root.right == null ? p : root.right;
        }
```

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

Remove Invalid Parentheses

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses (and).

Examples:

```
"()())()" -> ["()()()", "(())()"]
"(a)())()" -> ["(a)()()", "(a())()"]
")(" -> [""]
```

https://leetcode.com/problems/remove-invalid-parentheses/

Solution:

Analysis:

Here I share my DFS or backtracking solution. It's 10X faster than optimized BFS.

Limit max removal rmL and rmR for backtracking boundary. Otherwise it will exhaust all possible valid substrings, not shortest ones. Scan from left to right, avoiding invalid strs (on the fly) by checking num of open parens.

- If it's '(', either use it, or remove it.
- If it's '(', either use it, or remove it.
- Otherwise just append it.
- Lastly set StringBuilder to the last decision point.

In each step, make sure:

- i does not exceed s.length().
- Max removal rmL rmR and num of open parens are non negative.
- De-duplicate by adding to a HashSet.

Compared to 106 ms BFS (Queue & Set), it's faster and easier.

https://leetcode.com/discuss/72208/easiest-9ms-java-solution

```
public List<String> removeInvalidParentheses(String s) {
    Set<String> res = new HashSet<>();
    int rmL = 0, rmR = 0;
    for(int i = 0; i < s.length(); i++) {
        if(s.charAt(i) == '(') rmL++;
        if(s.charAt(i) == ')') {
            if(rmL != 0) rmL--;
            else rmR++;
        }
    }
    DFS(res, s, 0, rmL, rmR, 0, new StringBuilder());
    return new ArrayList<String>(res);
}

public void DFS(Set<String> res, String s, int i, int rmL, int rmR,
    if(i == s.length() && rmL == 0 && rmR == 0 && open == 0) {
        res.add(sb.toString());
        return;
    }
    if(i == s.length() || rmL < 0 || rmR < 0 || open < 0) return;

    char c = s.charAt(i);
    int len = sb.length();

    if(c == '(') {
        DFS(res, s, i + 1, rmL - 1, rmR, open, sb);
        DFS(res, s, i + 1, rmL, rmR, open + 1, sb.append(c));

    } else if(c == ')') {
        DFS(res, s, i + 1, rmL, rmR - 1, open, sb);
        DFS(res, s, i + 1, rmL, rmR, open - 1, sb.append(c));

    } else {
        DFS(res, s, i + 1, rmL, rmR, open, sb.append(c));
    }

    sb.setLength(len);
}
```

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).

Here is an example: S = "rabbbit", T = "rabbit"

Return 3.

https://leetcode.com/problems/distinct-subsequences/

Problem statement:

The problem itself is very difficult to understand. It can be stated like this: Give a sequence S and T, how many distinct sub sequences from S equals to T? How do you define "distinct" subsequence? Clearly, the 'distinct' here mean different operation combination, not the final string of subsequence. Otherwise, the result is always 0 or 1. -- from Jason's comment

When you see string problem that is about subsequence or matching, dynamic programming method should come to mind naturally. The key is to find the initial and changing condition.

Solution:

```
public int numDistincts(String S, String T) {
    int[][] table = new int[S.length() + 1][T.length() + 1];

    for (int i = 0; i < S.length(); i++)
        table[i][0] = 1;

    for (int i = 1; i <= S.length(); i++) {
        for (int j = 1; j <= T.length(); j++) {
            if (S.charAt(i - 1) == T.charAt(j - 1)) {
                table[i][j] += table[i - 1][j] + table[i - 1][j - 
            } else {
                table[i][j] += table[i - 1][j];
            }
        }
    }

    return table[S.length()][T.length()];
}
```

Solution II: O(N) space

how to reduce to O(N) space:

http://stackoverflow.com/questions/20459262/distinct-subsequences-dp-explanation

```
public class Solution {

public int numDistinct(String S, String T) {
    int[] table = new int[T.length() + 1];
    table[T.length()] = 1;

    for (int i = S.length()-1; i >= 0; i--) {
        for (int j = 0; j < T.length(); j++) {
            if (S.charAt(i) == T.charAt(j)) {
                table[j] += table[j+1];
            }
        }
    }

    return table[0]; // not return table[T.length()];
    }
}
```

Jump Game II

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

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

Your goal is to reach the last index in the minimum number of jumps.

For example: Given array A = [2,3,1,1,4]

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

https://leetcode.com/problems/jump-game-ii/

Solution: Greedy

```
public class Solution {
    public int jump(int[] nums) {
        if (nums.length <= 1) {
            return 0;
        }

        int i = 0, jump = 0;

        while (i < nums.length) {
            int max = -1;
            if (nums[i] == 0 && i < nums.length-1) {
                return -1;
            }
            if (i >= nums.length-1) {
                return jump;
            }

            jump++;

            if (i+nums[i] >= nums.length-1) {
                return jump;
            }

            int t = i; // this line is important
            for (int j = 1; j <= nums[t]; j++) {
                int k = t+j;
                if (k < nums.length && k + nums[k] > max) {
                    max = k+ nums[k];
                    i = k;
                }
            }
        }
        return -1;
    }
}
```

Note: why line "int t == i;" is necessary ??

Solution II:

```
public class Solution {

    public int jump(int[] A) {
        int jumps = 0, curEnd = 0, curFarthest = 0;
        for (int i = 0; i < A.length - 1; i++) {
            curFarthest = Math.max(curFarthest, i + A[i]);
            if (i == curEnd) {
                jumps++;
                curEnd = curFarthest;

                if (curEnd >= A.length - 1) {
                    break;
                }
            }
        }
        return jumps;
    }
}
```

https://leetcode.com/discuss/67047/concise-o-n-one-loop-java-solution-based-on-greedy

Candy

There are N children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

Each child must have at least one candy. Children with a higher rating get more candies than their neighbors. What is the minimum candies you must give?

https://leetcode.com/problems/candy/

Solution:

```
public class Solution {
    public int candy(int[] ratings) {
        if (ratings.length == 0) {
            return 0;
        }

        int[] res = new int[ratings.length];

        res[0] = 1;

        for (int i = 1; i < ratings.length; i++) {
            if (ratings[i] > ratings[i-1]) {
                res[i] = res[i-1] + 1;
            } else {
                res[i] = 1;
            }
        }

        int count = res[ratings.length-1];

        for (int i = ratings.length-2; i >= 0; i--) {
            if (ratings[i] > ratings[i+1]) {
                res[i] = Math.max(res[i+1] + 1, res[i]);
            }

            count += res[i];
        }

        return count;
    }
}
```

The Skyline Problem

similar problem: meeting room II

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are given the locations and height of all the buildings as shown on a cityscape photo (Figure A), write a program to output the skyline formed by these buildings collectively (Figure B).

Buildings Skyline Contour The geometric information of each building is represented by a triplet of integers [Li, Ri, Hi], where Li and Ri are the x coordinates of the left and right edge of the ith building, respectively, and Hi is its height. It is guaranteed that 0 ≤ Li, Ri ≤ INT_MAX, 0 < Hi ≤ INT_MAX, and Ri - Li > 0. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

For instance, the dimensions of all buildings in Figure A are recorded as: [[2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8]] .

The output is a list of "key points" (red dots in Figure B) in the format of [[x1,y1], [x2, y2], [x3, y3], ...] that uniquely defines a skyline. A key point is the left endpoint of a horizontal line segment. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the skyline in Figure B should be represented as:[[2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0]].

Notes:

The number of buildings in any input list is guaranteed to be in the range [0, 10000]. The input list is already sorted in ascending order by the left x position Li. The output list must be sorted by the x position. There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...[2 3], [4 5], [7 5], [11 5], [12 7]...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...[2 3], [4 5], [12 7], ...] Credits:

https://leetcode.com/problems/the-skyline-problem/

Solution:

Analysis:

check: https://leetcode.com/discuss/67091/once-for-all-explanation-with-clean-java-code-nlog-time-space

```
public List<int[]> getSkyline(int[][] buildings) {
    List<int[]> result = new ArrayList<>();
    List<int[]> height = new ArrayList<>();
    for(int[] b:buildings) {
        height.add(new int[]{b[0], -b[2]}); // start point has nega
        height.add(new int[]{b[1], b[2]}); // end point has normal
    }

    // sort $height, based on the first value, if necessary, use th
    Collections.sort(height, (a, b) -> {
            if(a[0] != b[0])
                return a[0] - b[0];
            return a[1] - b[1];
    });

    // Use a maxHeap to store possible heights
    Queue<Integer> pq = new PriorityQueue<>((a, b) -> (b - a));

    // Provide a initial value to make it more consistent
    pq.offer(0);

    // Before starting, the previous max height is 0;
    int prev = 0;

    // visit all points in order
    for(int[] h:height) {
        if(h[1] < 0) { // a start point, add height
            pq.offer(-h[1]);
        } else {  // a end point, remove height
            pq.remove(h[1]);
        }
        int cur = pq.peek(); // current max height;
```

```
            // compare current max height with previous max height, upd
            if(prev != cur) {
                result.add(new int[]{h[0], cur});
                prev = cur;
            }
        }
        return result;
    }
```

Longest Valid Parentheses

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

https://leetcode.com/problems/longest-valid-parentheses/

Solution:

Analysis:

https://leetcode.com/discuss/24045/simple-java-solution

- if open > 0 and current char is ')', then res[i] = res[i-1] + 2;
- divide the string into 2 parts: a. 0 to i - res[i], b. i-res[i] to i
- if the first half is valid ended at i, then res[i-res[i]] > 0; combine two half
- if not, res[i-res[i]] = 0

```
public class Solution {
    public int longestValidParentheses(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }

        // in this case, res[i] means the max length of parensis er
        int[] res = new int[s.length()];
        int open = 0, max = 0;

        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (c == '(') {
                open++;
                //res[i] = res[i-1];
            } else {
                if (open > 0) {
                    res[i] = res[i-1] + 2;
                    // important lines here
                    if (i >= res[i]) {
                        res[i] += res[i-res[i]];
                    }
                    open--;
                }
            }

            max = Math.max(max, res[i]);
        }

        return max;
    }
}
```

Interleaving String

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

For example, Given: s1 = "aabcc", s2 = "dbbca",

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

Solution:

When you see string problem that is about subsequence or matching, dynamic programming method should come to mind naturally. The key is to find the initial and changing condition.

```
public class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        if (s1.length() + s2.length() != s3.length()) {
            return false;
        }
        int m = s1.length();
        int n = s2.length();

        boolean[][] res = new boolean[m+1][n+1];
        res[0][0] = true;

        for (int i = 1; i <= m; i++) {
            res[i][0] = res[i-1][0] && (s1.charAt(i-1) == s3.charAt
        }

        for (int i = 1; i <= n; i++) {
            res[0][i] = res[0][i-1] && (s2.charAt(i-1) == s3.charAt
        }

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                res[i][j] = (res[i-1][j] && s1.charAt(i-1) == s3.ch
            }
        }

        return res[m][n];
    }
}
```

Expression Add Operators

Given a string that contains only digits 0-9 and a target value, return all possibilities to add binary operators (not unary) +, -, or * between the digits so they evaluate to the target value.

Examples:

- "123", 6 -> ["1+2+3", "123"]
- "232", 8 -> ["2*3+2″, ″2+3*2"]
- "105", 5 -> ["1*0+5","10-5"]
- "00", 0 -> ["0+0", "0-0", "0*0"]
- "3456237490", 9191 -> []

https://leetcode.com/problems/expression-add-operators/

Solution:

note: how to handle the multiply

```
public List<String> addOperators(String num, int target) {
    List<String> res = new ArrayList<>();
    StringBuilder sb = new StringBuilder();
    dfs(res, sb, num, 0, target, 0, 0);
    return res;

}
public void dfs(List<String> res, StringBuilder sb, String num, int
    if(pos == num.length()) {
        if(target == prev) res.add(sb.toString());
        return;
    }
    for(int i = pos; i < num.length(); i++) {
        if(num.charAt(pos) == '0' && i != pos) break;
        long curr = Long.parseLong(num.substring(pos, i + 1));
        int len = sb.length();
        if(pos == 0) {
            dfs(res, sb.append(curr), num, i + 1, target, curr, cu
            sb.setLength(len);
        } else {
            dfs(res, sb.append("+").append(curr), num, i + 1, targe
            sb.setLength(len);
            dfs(res, sb.append("-").append(curr), num, i + 1, targe
            sb.setLength(len);
            dfs(res, sb.append("*").append(curr), num, i + 1, targe
            sb.setLength(len);
        }
    }
}
```

Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

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

For example: Given the below binary tree,

```
       1
      / \
     2   3
```

Return 6.

Solution I:

Note: need to maintain a global max and local max because the max sum does not necessarily happen from root to leaf

Try the bottom up approach. At each node, the potential maximum path could be one of these cases:

- i. max(left subtree) + node
- ii. max(right subtree) + node
- iii. max(left subtree) + max(right subtree) + node
- iv. the node itself

Then, we need to return the maximum path sum that goes through this node and to either one of its left or right subtree to its parent. There’s a little trick here: If this maximum happens to be negative, we should return 0, which means: “Do not include this subtree as part of the maximum path of the parent node”, which greatly simplifies our code.

```
public class Solution {
    private int maxSum;
    public int maxPathSum(TreeNode root) {
        maxSum = Integer.MIN_VALUE;
        findMax(root);
        return maxSum;
    }

    private int findMax(TreeNode p) {
        if (p == null) return 0;
        int left = findMax(p.left);
        int right = findMax(p.right);
        maxSum = Math.max(p.val + left + right, maxSum);
        int ret = p.val + Math.max(left, right);
        return ret > 0 ? ret : 0;
    }
}
```

Solution II:

```
public class Solution {
    public int maxPathSum(TreeNode root) {
    int max[] = new int[1];
    max[0] = Integer.MIN_VALUE;
    calculateSum(root, max);
    return max[0];
}

public int calculateSum(TreeNode root, int[] max) {
    if (root == null)
        return 0;

    int left = calculateSum(root.left, max);
    int right = calculateSum(root.right, max);

    int current = Math.max(root.val, Math.max(root.val + left, root

    max[0] = Math.max(max[0], Math.max(current, left + root.val + r

    return current;
}
}
```

Regular Expression Matching

Implement regular expression matching with support for '.' and '*'.

'.' Matches any single character.

'*' Matches zero or more of the preceding element. "a/star" means zero to any number of a

The matching should cover the entire input string (not partial).

The function prototype should be: bool isMatch(const char *s, const char* p)

Some examples:

- isMatch("aa","a") → false
- isMatch("aa","aa") → true
- isMatch("aaa","aa") → false
- isMatch("aa", "a*") → true
- isMatch("aa", ".*") → true
- isMatch("ab", ".*") → true
- isMatch("aab", "cab") → true

https://leetcode.com/problems/regular-expression-matching/

Solution I:

```
public boolean isMatch(String s, String p) {
    // base case
    if (p.length() == 0) {
        return s.length() == 0;
    }

    // special case
    if (p.length() == 1) {

        // if the length of s is 0, return false
        if (s.length() < 1) {
            return false;
        }

        //if the first does not match, return false
        else if ((p.charAt(0) != s.charAt(0)) && (p.charAt(0) != '
            return false;
        }

        // otherwise, compare the rest of the string of s and p.
        else {
            return isMatch(s.substring(1), p.substring(1));
        }
    }

    // case 1: when the second char of p is not '*'
    if (p.charAt(1) != '*') {
        if (s.length() < 1) {
            return false;
        }
        if ((p.charAt(0) != s.charAt(0)) && (p.charAt(0) != '.')) 
            return false;
        } else {
            return isMatch(s.substring(1), p.substring(1));
        }
    }

    // case 2: when the second char of p is '*', complex case.
    else {
        //case 2.1: a char & '*' can stand for 0 element
        if (isMatch(s, p.substring(2))) {
            return true;
        }

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

Solution II: concise version

```
public class Solution {
    public boolean isMatch(String s, String p) {

        if(p.length() == 0)
            return s.length() == 0;

        //p's length 1 is special case
        if(p.length() == 1 || p.charAt(1) != '*'){
            if(s.length() < 1 || (p.charAt(0) != '.' && s.charAt(0)
                return false;
            return isMatch(s.substring(1), p.substring(1));

        }else{
            int len = s.length();

            int i = -1;
            while(i<len && (i < 0 || p.charAt(0) == '.' || p.charAt
                if(isMatch(s.substring(i+1), p.substring(2)))
                    return true;
                i++;
            }
            return false;
        }
    }
}
```

Palindrome Partitioning I, II

Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

For example, given s = "aab",

Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.

https://leetcode.com/problems/palindrome-partitioning-ii/

Analysis:

This problem is similar to Palindrome Partitioning. It can be efficiently solved by using dynamic programming. Unlike "Palindrome Partitioning", we need to maintain two cache arrays, one tracks the partition position and one tracks the number of minimum cut.

Solution:

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

    boolean dp[][] = new boolean[n][n];
    int cut[] = new int[n];

    for (int j = 0; j < n; j++) {
        cut[j] = j; //set maximum # of cut
        for (int i = 0; i <= j; i++) {
            if (s.charAt(i) == s.charAt(j) && (j - i <= 1 || dp[i+1
                dp[i][j] = true;

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

    return cut[n-1];
}
```

Solution II: TLE

```
public class Solution {
    public int minCut(String s) {
        if (s.length() == 0) {
            return 0;
        }

        int n = s.length();
        int[][] res = new int[n][n];

        for (int len = 2; len <= n; len++) {
            for (int i = 0; i <= n-len; i++) {
                int j = i + len - 1;

                if (len == 2) {
                    res[i][j] = s.charAt(i) == s.charAt(j) ? 0 : 1;
                }

                else {

                if (s.charAt(i) == s.charAt(j) && res[i+1][j-1] ==
                    res[i][j] = 0;

                } else {
                    res[i][j] = len;
                    for (int l = 1; l < len; l++) {
                        res[i][j] = Math.min(res[i][i+l-1] + res[i-
                        if (res[i][j] == 0) {
                            break;
                        }
                    }
                }
                }
            }
        }

        return res[0][n-1];

    }
}
```

Shortest Palindrome

Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.

For example:

Given "aacecaaa", return "aaacecaaa".

Given "abcd", return "dcbabcd".

https://leetcode.com/problems/shortest-palindrome/

Solution: O(N^2)

find the longest palindrome string start from index 0

```
public String shortestPalindrome(String s) {
    if (s == null || s.length() <= 1)
        return s;

    String result = null;

    int len = s.length();
    int mid = len / 2;

    for (int i = mid; i >= 1; i--) {
        if (s.charAt(i) == s.charAt(i - 1)) {
            if ((result = scanFromCenter(s, i - 1, i)) != null)
                return result;
        } else {
            if ((result = scanFromCenter(s, i - 1, i - 1)) != null
                return result;
        }
    }

    return result;
}
```

```
private String scanFromCenter(String s, int l, int r) {
    int i = 1;

    //scan from center to both sides
    for (; l - i >= 0; i++) {
        if (s.charAt(l - i) != s.charAt(r + i))
            break;
    }

    //if not end at the beginning of s, return null
    if (l - i >= 0)
        return null;

    StringBuilder sb = new StringBuilder(s.substring(r + i));
    sb.reverse();

    return sb.append(s).toString();
}
```

Solution: O(N) but not explanation

https://leetcode.com/discuss/51223/my-7-lines-recursive-java-solution

```
public class Solution {
    public String shortestPalindrome(String s) {
        int j = 0;

        for (int i = s.length() - 1; i >= 0; i--) {
            if (s.charAt(i) == s.charAt(j)) { j += 1; }
        }

        if (j == s.length()) { return s; }
        String suffix = s.substring(j);
        return new StringBuilder(suffix).reverse().toString() + sho
    }
}
```

Solution: KMP, O(N)

https://leetcode.com/discuss/52564/a-kmp-based-java-solution-with-explanation

```
public class Solution {
    public String shortestPalindrome(String s) {
        if(s.length()<=1) return s;
        String new_s = s+"#"+new StringBuilder(s).reverse().toStrin
        int[] position = new int[new_s.length()];

        for(int i=1;i<position.length;i++)
        {
            int pre_pos = position[i-1];
            while(pre_pos>0 && new_s.charAt(pre_pos)!=new_s.charAt(
                pre_pos = position[pre_pos-1];
            position[i] = pre_pos+((new_s.charAt(pre_pos)==new_s.ch
        }

        return new StringBuilder(s.substring(position[position.leng
    }
}
```

Wildcard Matching

Implement wildcard pattern matching with support for '?' and '*'.

'?' Matches any single character. '*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

The function prototype should be: bool isMatch(const char *s, const char* p)

Some examples:

- isMatch("aa","a") → false
- isMatch("aa","aa") → true
- isMatch("aaa","aa") → false
- isMatch("aa", "*") → true
- isMatch("aa", "a*") → true
- isMatch("ab", "?*") → true
- isMatch("aab", "cab") → false

https://leetcode.com/problems/wildcard-matching/

Solution I: adopt from Regular Expression Matching, TLE

```
public class Solution {
    public boolean isMatch(String s, String p) {
        if (s.length() == 0 && p.length() == 0) {
            return true;
        }

        if (p.length() == 0) {
            return false;
        }

        if (p.length() == 1) {
            if (s.length() == 0 || (s.charAt(0) != p.charAt(0) && p
                return false;
            }
            if (p.charAt(0) == '*') {
                return true;
            }
            return isMatch(s.substring(1), p.substring(1));
        }

        char c = p.charAt(1);

        if (c != '*') {
            if (s.length() == 0 || (s.charAt(0) != p.charAt(0) && p
                return false;
            }

            return isMatch(s.substring(1), p.substring(1));
        } else {
            if (isMatch(s, p.substring(2))) {
                return true;
            }

            int i = 0;

            while (i < s.length()) {
                if (isMatch(s.substring(i+1), p.substring(2))) {
                    return true;
                }
                i++;
            }

            return false;
        }
    }
}
```

Solution II: DP (2D)

https://leetcode.com/discuss/66038/java-solution-o-n-2-dp-solution-with-some-explanations

```
public class Solution {
    public boolean isMatch(String s, String p) {

    if(p.length()==0)
        return s.length()==0;

    boolean[][] res = new boolean[p.length()+1][s.length()+1];

    res[0][0] = true;

    for (int i = 1; i <= p.length(); i++) {
        boolean flag = false;
        for (int j = 0; j <= s.length(); j++) { // note that j star
            char c = p.charAt(i-1);
            flag = flag || res[i-1][j];

            if (c != '*') {
                res[i][j] = j>0 && res[i-1][j-1] && (c == '?' || s.
            } else {
                // For k>=0 and k<=j, if any dp[i-1][k] is true,
                // then '*' will match the rest sequence in s after
                res[i][j] = i==1 || flag; // note that if i == 1 &&
            }
        }
    }

    return res[p.length()][s.length()];

    }
}
```

Solution III: DP(1D)

http://blog.csdn.net/linhuanmars/article/details/21198049

```
public class Solution {
    public boolean isMatch(String s, String p) {
    if(p.length()==0)
        return s.length()==0;
    boolean[] res = new boolean[s.length()+1];
    res[0] = true;
    for(int j=0;j<p.length();j++)
    {
        if(p.charAt(j)!='*')
        {
            for(int i=s.length()-1;i>=0;i--)
            {
                res[i+1] = res[i]&&(p.charAt(j)=='?'||s.charAt(i)==
            }
        }
        else
        {
            int i = 0;
            while(i<=s.length() && !res[i])
                i++;
            for(;i<=s.length();i++)
            {
                res[i] = true;
            }
        }
        res[0] = res[0]&&p.charAt(j)=='*';
    }
    return res[s.length()];
}
}
```

Max Points on a Line

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

https://leetcode.com/problems/max-points-on-a-line/

Solution:

```
public class Solution {
    public int maxPoints(Point[] points) {
        if (points.length == 0) {
            return 0;
        }
        int n = points.length;
        int max = 0;
        Map<Double, Integer> map = new HashMap<>();

        for (int i = 0; i < n; i++) {
            int duplicate = 1, vertical = 0; // note that duplicate

            for (int j = i+1; j < n; j++) {
                Point a = points[i];
                Point b = points[j];

                if (a.x == b.x) {
                    if (a.y == b.y) {
                        duplicate++;
                    } else {
                        vertical++;
                    }
                } else {

                    double slope = getSlope(a, b);

                    if (!map.containsKey(slope)) {
                        map.put(slope, 1);
                    } else {
                        map.put(slope, map.get(slope) + 1);
                    }
                }
            }

            for (Integer count : map.values()) {
                max = Math.max(max, count+duplicate);
            }

            max = Math.max(max, vertical+duplicate);
            map.clear();
        }

        return max;
    }

    public double getSlope(Point a, Point b) {
        if (a.y == b.y) {
            return 0.0;
        } // this is necessary
        return 1.0 * (a.y - b.y) / (a.x - b.x);
    }
}
```

Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character '.'.

You may assume that there will be only one unique solution.

A sudoku puzzle...

...and its solution numbers marked in red.

https://leetcode.com/problems/sudoku-solver/

Solution :

https://leetcode.com/discuss/9929/a-java-solution-with-notes

```
public class Solution {
    public void solveSudoku(char[][] board) {
        if (board.length == 0) {
            return;
        }
        solve(board);
    }

    public boolean solve(char[][] board) {
        int m = board.length;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < m; j++) {
                if (board[i][j] == '.') {
                    for (char c = '1'; c <= '9'; c++) {
                        if (isValid(board, i, j, c)) {
                            board[i][j] = c;
                            if (solve(board)) {
                                return true;
                            } else {
                            board[i][j] = '.';
                            }
                        }
                    }
                    return false;
                }
            }
        }
        return true;
    }

    public boolean isValid(char[][] board, int i, int j, char c) {
        // check col
        for (int row = 0; row < board.length; row++) {
            if (board[row][j] == c) {
                return false;
            }
        }
        // check row
        for (int col = 0; col < board.length; col++) {
            if (board[i][col] == c) {
                return false;
            }
        }
        // check block

        for (int row = (i/3)*3; row < (i/3)*3+3; row++) {
            for (int col = (j/3)*3; col < (j/3)*3+3; col++) {
                if (board[row][col] == c) {
                    return false;
                }
            }
        }

        return true;
    }
}
```

Word Ladder II

Given two words (beginWord and endWord), and a dictionary's word list, find all shortest transformation sequence(s) from beginWord to endWord, such that:

Only one letter can be changed at a time Each intermediate word must exist in the word list For example,

Given:

```
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]
```

Return

```
  [
    ["hit","hot","dot","dog","cog"],
    ["hit","hot","lot","log","cog"]
  ]
```

Note:

All words have the same length.

All words contain only lowercase alphabetic characters.

https://leetcode.com/problems/word-ladder-ii/

Solution:

https://leetcode.com/discuss/64808/my-concise-java-solution-based-on-bfs-and-dfs

```
public List<List<String>> findLadders(String start, String end, Set
   List<List<String>> res = new ArrayList<List<String>>();
   HashMap<String, ArrayList<String>> nodeNeighbors = new HashMap<S
   HashMap<String, Integer> distance = new HashMap<String, Integer>
```

```
    ArrayList<String> solution = new ArrayList<String>();

    dict.add(end);
    bfs(start, end, dict, nodeNeighbors, distance);
    dfs(start, end, dict, nodeNeighbors, distance, solution, res);
    return res;
 }

 // BFS: Trace every node's distance from the start node (level by l
 private void bfs(String start, String end, Set<String> dict, HashMa
   for (String str : dict)
       nodeNeighbors.put(str, new ArrayList<String>());

   Queue<String> queue = new LinkedList<String>();
   queue.offer(start);
   distance.put(start, 0);

   while (!queue.isEmpty()) {
       int count = queue.size();
       boolean foundEnd = false;
       for (int i = 0; i < count; i++) {
           String cur = queue.poll();
           int curDistance = distance.get(cur);
           ArrayList<String> neighbors = getNeighbors(cur, dict);

           for (String neighbor : neighbors) {
               nodeNeighbors.get(cur).add(neighbor);
               if (!distance.containsKey(neighbor)) {// Check if vis
                   distance.put(neighbor, curDistance + 1);
                   if (end.equals(neighbor))// Found the shortest pa
                       foundEnd = true;
                   else
                       queue.offer(neighbor);
                   }
               }
           }

           if (foundEnd)
               break;
       }
```

```
  }

// Find all next level nodes.
private ArrayList<String> getNeighbors(String node, Set<String> dic
  ArrayList<String> res = new ArrayList<String>();
  char chs[] = node.toCharArray();

  for (char ch ='a'; ch <= 'z'; ch++) {
      for (int i = 0; i < chs.length; i++) {
          if (chs[i] == ch) continue;
          char old_ch = chs[i];
          chs[i] = ch;
          if (dict.contains(String.valueOf(chs))) {
              res.add(String.valueOf(chs));
          }
          chs[i] = old_ch;
      }

  }
  return res;
}

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

Create Maximum Number

Given two arrays of length m and n with digits 0-9 representing two numbers. Create the maximum number of length k <= m + n from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the k digits. You should try to optimize your time and space complexity.

Example 1: nums1 = [3, 4, 6, 5] nums2 = [9, 1, 2, 5, 8, 3] k = 5 return [9, 8, 6, 5, 3]

Example 2: nums1 = [6, 7] nums2 = [6, 0, 4] k = 5 return [6, 7, 6, 0, 4]

Example 3: nums1 = [3, 9] nums2 = [8, 9] k = 3 return [9, 8, 9]

https://leetcode.com/problems/create-maximum-number/

Solution:

Detailed explanation:

Sub porblems are also interesting:

- Given one array of length n, create the maximum number of length k.
- Given two array of length m and n, create maximum number of length k = m + n.

http://algobox.org/create-maximum-number/

https://leetcode.com/discuss/75756/share-my-greedy-solution

Maximal Rectangle

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

https://leetcode.com/problems/maximal-rectangle/

Solution I: brute force O(N^3)

取所有前面最小的延伸距离乘以当前离第一行的行数

http://www.cnblogs.com/lichen782/p/leetcode_maximal_rectangle.html

```
/**
     * 以给出的坐标作为左上角，计算其中的最大矩形面积
     * @param matrix
     * @param row 给出坐标的行
     * @param col 给出坐标的列
     * @return 返回最大矩形的面积
     */
    private int maxRectangle(char[][] matrix, int row, int col) {
        int minWidth = Integer.MAX_VALUE;
        int maxArea = 0;
        for (int i = row; i < matrix.length && matrix[i][col] == '
            int width = 0;
            while (col + width < matrix[row].length
                    && matrix[i][col + width] == '1') {
                width++;
            }
            if (width < minWidth) {// 如果当前宽度小于了以前的最小宽度，
                minWidth = width;
            }
            int area = minWidth * (i - row + 1);
            if (area > maxArea)
                maxArea = area;
        }
        return maxArea;
    }
```

Solution II: Optimized O(N^2)

find max area in histogram as a routine to calculate each row

why n+1 in height:

height[i][n] == 0, manually push 0 to stack to calculate all the elements in stack

check the code for max rectangle for histogram

```
public class Solution {
    public int maximalRectangle(char[][] matrix) {
        int m = matrix.length;

        if (m == 0) {
            return m;
        }

        int n = matrix[0].length;

        int[][] height = new int[m][n+1]; // why n+1

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == '0') {
                    height[i][j] = 0;
                } else {
                    height[i][j] = i == 0 ? 1 : height[i-1][j] + 1;
                }
            }
        }

        int max = 0;

        for (int i = 0; i < m; i++) {
            int area = maxInLine(height[i]);
            max = Math.max(area, max);
        }

        return max;
    }
    public int maxInLine(int[] height) {
        int max = 0;

        Stack<Integer> stack = new Stack<>();
        int i = 0;
        while (i < height.length) {
            if (stack.isEmpty() || height[i] >= height[stack.peek()
                stack.push(i);
                i++;
            } else {
                int n = stack.pop();
                int curr = height[n] * (stack.isEmpty() ? i : i - s
                max = Math.max(max, curr);
            }
        }

        return max;
    }
}
```

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.

For 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

https://leetcode.com/problems/reverse-nodes-in-k-group/

Solution:

tricky part: break the link into piece of size k and reverse then assembly

dont forget the last part

```
public class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        if (head == null || k <= 1) {
            return head;
        }

        int count = 0;
        ListNode dummyHead = new ListNode(0);
        ListNode n = dummyHead;
        ListNode currHead = head;
        while (head != null) {
            count++;

            if (count == k) {
                ListNode t = head.next;
                head.next = null; // break the link first
                n.next = reverse(currHead);
                n = currHead;
                currHead = t;
                head = t;
                count = 0;
            } else {
                head = head.next;
            }

        }
        // the last part with size < k
        if (count < k) {
            n.next = currHead;
        }

        return dummyHead.next;
    }

    public ListNode reverse(ListNode head) {
        ListNode dummyHead = new ListNode(0);

        while (head != null) {
            ListNode n = head.next;
            head.next = dummyHead.next;
            dummyHead.next = head;
            head = n;
        }

        return dummyHead.next;
    }
}
```

Best Time to Buy and Sell Stock IV

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most k transactions.

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

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/

Solution I: 2D DP, TLE

The local array tracks maximum profit of j transactions & the last transaction is on ith day. The global array tracks the maximum profit of j transactions until ith day.

```
public class Solution {
    public int maxProfit(int k, int[] prices) {
        if (prices.length == 0) {
            return 0;
        }

    // solve the TLE problem
    if(k>=prices.length/2){
        int maxProfit = 0;
        for(int i=1; i<prices.length; i++){
            if(prices[i]>prices[i-1]) maxProfit += prices[i]-prices
        }
        return maxProfit;
    }


        int[][] res = new int[prices.length][k+1];
        int[][] local = new int[prices.length][k+1];

        for (int j = 1; j <= k; j++) {
            for (int i = 1; i < prices.length; i++) {
                int diff = prices[i] - prices[i-1];
                local[i][j] = Math.max(res[i-1][j-1] + Math.max(dif
                res[i][j] = Math.max(res[i-1][j], local[i][j]);
            }
        }

        return res[prices.length-1][k];
    }
}
```

Solution II: General version of Best Time to Buy and Sell Stock III

https://leetcode.com/discuss/69340/clean-java-dp-o-nk-solution-with-o-k-space

```
public int maxProfit(int k, int[] prices) {
    if(k>=prices.length/2){
        int maxProfit = 0;
        for(int i=1; i<prices.length; i++){
            if(prices[i]>prices[i-1]) maxProfit += prices[i]-price
        }
        return maxProfit;
    }

    int[] maxProfit = new int[k+1];
    int[] lowPrice = new int[k+1];
    for(int i=0; i<lowPrice.length; i++) lowPrice[i]=Integer.MAX_V
    for(int p : prices){
        for(int i=k; i>=1; i--){
            maxProfit[i] = Math.max(maxProfit[i],p-lowPrice[i]);
            lowPrice[i] = Math.min(lowPrice[i],p-maxProfit[i-1]);
        }
    }
    return maxProfit[k];
}
```

Code Bases

1. Remove duplicates from an sorted array (allow at most k duplicates)

```
    public int removeDuplicates(int[] nums) {
        return removeDuplicates(nums, 2);
    }

    private int removeDuplicates(int[] nums, int k) {
        if (nums.length <= k)
            return nums.length;
        int index = k;
        for (int i = k; i < nums.length; i++) {
            if (nums[i] != nums[index - k]) {
                //index++;
                nums[index++] = nums[i];
            }
        }
        return index;
    }
```

Binary search variations/applications

1. find target position, return index, or -1 if not exist

```
public int BinarySearch(int[] a, int l, int r, int key) { // f
    while (l <= r) {
        int m = l+(r-l)/2;
        if (a[m] == key)
            return m;
        if (a[m] > key)
            r = m-1;
        else l = m+1;
    }
    return -1;
    }
}
```

2. find target insert position, return insert index; if target is in the array, the return value is index of the target, if not, return value is the insert position

   this method can be used for both search and insert

```
public int BinarySearch(int[] a, int l, int r, int key) { // fi
    while (l <= r) {
        int m = l+(r-l)/2;
        if (a[m] >= key)
            r = m-1;
        else l = m+1;
    }
    return l;
}
```

1. find leftbound of target, return target index; return -1 if not exist

```
public int BinarySearch(int[] a, int l, int r, int key) {
     while (l <= r) {
         int m = l+(r-l)/2;
         if (a[m] >= key)
             r = m-1;
         else l = m+1;
     }
     if (l >= 0 && l < a.length && a[l] == target) {
         return l;
     return -1;
     }
 }
```

2. find rightbound of target, return target index; return -1 if not exist

```
public int BinarySearch(int[] a, int l, int r, int key) {
        while (l <= r) {
            int m = l+(r-l)/2;
            if (a[m] <= key)
                l = m+1;
            else r = m-1;
        }
        if (r >= 0 && r < a.length && a[r] == target) {
            return r;
        return -1;
        }
    }
```

1. find minimum in rotated sorted array

   Without Duplicates

```
public class Solution {
 public int findMin(int[] nums) {
     if (nums.length == 0) {
         return -1;
     }

     int left = 0, right = nums.length-1;

     while (left <= right) {
         int mid = left + (right-left)/2;

         if (nums[left] <= nums[mid] && nums[mid] <= nums[right
             return nums[left];
         }

         if (nums[left] <= nums[mid]) {
             left = mid + 1;
         }
         else {
             right = mid;
         }
     }
     return -1;
 }
}
```

2. if the target is dynamic, cant find exact equal condition to return

   idea: use an extra variable to store the answer each time

   Example: H index II

```
public class Solution {
    public int hIndex(int[] citations) {
        int start = 0;
        int end = citations.length-1;
        int len = citations.length;
        int result = 0;
        int mid;
        while(start <= end){
            mid = start + (end-start)/2;
            if(citations[mid] >= (len - mid)){
                result = (len-mid);
                end = mid-1;
            }
            else{
                start = mid + 1;
            }
        }
        return result;
    }
}
```

Binary Tree Traversal

1. BST iterator (Inorder traversal)

   ref: https://leetcode.com/problems/binary-search-tree-iterator/

```
public class BSTIterator {
private Stack<TreeNode> stack;
private TreeNode prev;

public BSTIterator(TreeNode root) {
    prev = null;
    stack = new Stack<TreeNode>();
    while (root != null) {
        stack.push(root);
        root = root.left;
    }
}

/** @return whether we have a next smallest number */
public boolean hasNext() {
    return !stack.isEmpty() || (prev != null && prev.right !=
}

/** @return the next smallest number */
public int next() {

    if (prev != null && prev.right != null) {
        prev = prev.right;
        while (prev != null) {
            stack.push(prev);
            prev = prev.left;
        }

    }
    prev = stack.pop();
    return prev.val;
}
}
```

Remove Duplicates

Dynamic programming collection

G&G search result

http://www.geeksforgeeks.org/dynamic-programming-set-9-binomial-coefficient/

Dynamic Programming | Set 6 (Min Cost Path)

http://www.geeksforgeeks.org/dynamic-programming-set-6-min-cost-path/

Dynamic Programming | Set 7 (Coin Change)

http://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/

Dynamic Programming | Set 8 (Matrix Chain Multiplication)

http://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/

Dynamic Programming | Set 9 (Binomial Coefficient)

http://www.geeksforgeeks.org/dynamic-programming-set-9-binomial-coefficient/

Dynamic Programming | Set 13 (Cutting a Rod)

http://www.geeksforgeeks.org/dynamic-programming-set-13-cutting-a-rod/

Two versions of state functions:

I:

cutRod(n) = max(price[i] + cutRod(n-i-1)) for all i in {0, 1 .. n-1}

II:

d[i][j] = max(d[i-1][j], d[i-1][j-i-1] + V[i])

reduced to 1D:

d[i] = max(d[i], d[i-j-i]+V[j])

note: i should increase from 1 to n

Dynamic Programming | Set 11 (Egg Dropping Puzzle) http://www.geeksforgeeks.org/dynamic-programming-set-11-egg-dropping-puzzle/

Dynamic Programming | Set 19 (Word Wrap Problem)

http://www.geeksforgeeks.org/dynamic-programming-set-18-word-wrap/

Dynamic Programming | Set 31 (Optimal Strategy for a Game)

http://www.geeksforgeeks.org/dynamic-programming-set-31-optimal-strategy-for-a-game/

Bit manipulations

1. Multiply two Numbers Without Using * Operator

```
m=0;
          while (a)
          {
                if (a&1)
                   m+=b;
                a>>=1;
                b<<=1;
          }
          return m;
```

1. check number is even or odd

   http://www.programmingsimplified.com/c/source-code/c-program-check-odd-even

2. check if a num is power of 2

```
 public class Solution {
 public boolean isPowerOfTwo(int n) {
     if(n<=0)
     return false;

     while(n>2){
         int t = n>>1;
         int c = t<<1;

         if(n-c != 0)
             return false;

         n = n>>1;
     }

     return true;
 }
}
```

solution II:

```
 public class Solution {
 public boolean isPowerOfTwo(int n) {
     if(n<=0)
         return false;
     return ((n & n-1) == 0);
 }
}
```

3. check a number is positive or negative

   int k = (a >> 31) & 1; k == 1 negative, k == 0 positive

4. find max of two numbers without comparison

   check Cracking coding interview at page 476

```
int findMax( int x, int y)
{
   int z = x - y;
   int i  = (z  >>  31)  &  0x1;
   int  max  =  x - i  *  z;
   return max;
}
```

Good tricks

1. swap two number without temporary variable

   I:

   a = a - b; b = a + b; a = b - a;

   II:

   a = a ^ b; b = a ^ b; a = a ^ b;

Segment Tree

Applications:

http://poj.org/summerschool/gw_interval_tree.pdf

1. find range minimum/maximum

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.

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

1. Build segment tree

```
/**
 * Definition of SegmentTreeNode:
 * public class SegmentTreeNode {
 *     public int start, end;
 *     public SegmentTreeNode left, right;
 *     public SegmentTreeNode(int start, int end) {
 *         this.start = start, this.end = end;
 *         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) {
        // write your code here
        if (start > end) {
            return null;
        }
        if (start == end) {
            return new SegmentTreeNode(start, end);
        }

        int mid = start + (end-start)/2;
        SegmentTreeNode root = new SegmentTreeNode(start, end);
        root.left = build(start, mid);
        root.right = build(mid+1, end);

        return root;
    }
}
```

2. Build segment tree with max value of the interval

```
/**
 * Definition of SegmentTreeNode:
 * 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 A: a list of integer
     *@return: The root of Segment Tree
     */
    public SegmentTreeNode build(int[] A) {
        // write your code here
        if (A == null || A.length == 0) {
            return null;
        }
        return build(A, 0, A.length-1);
    }

    public SegmentTreeNode build(int[]A, int start, int end) {
        // write your code here
        if (start > end) {
            return null;
        }
        if (start == end) {
            return new SegmentTreeNode(start, end, A[start]);
        }

        int mid = start + (end-start)/2;
        SegmentTreeNode root = new SegmentTreeNode(start, end, 0);
        root.left = build(A, start, mid);
        root.right = build(A, mid+1, end);
        root.max = Math.max(root.left.max, root.right.max);
        return root;
    }
```

```
}
```

3. Query the max value in a given interval

```
Runtime 10s

/**
 * Definition of SegmentTreeNode:
 * 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) {
        // write your code here
        if (root == null || start > root.end || end < root.start)
            return 0;
        }
        if (root.start == root.end) {
            return root.max;
        }
        int mid = ((root.start + root.end) / 2);
        return Math.max(query(root.left, start, end), query(root.ri
    }
}
```

```
Runtime 5s

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) {
        // write your code here
        if (root == null || start > root.end || end < root.start)
            return 0;
        }
        if (root.start >= start && root.end <= end) {
            return root.max;
        }
        int mid = ((root.start + root.end) / 2);
        return Math.max(query(root.left, start, Math.min(mid, end))
    }
}
```

4. Query the count of numbers in a given interval

```
/**
 * Definition of SegmentTreeNode:
 * public class SegmentTreeNode {
 *     public int start, end, count;
 *     public SegmentTreeNode left, right;
 *     public SegmentTreeNode(int start, int end, int count) {
 *         this.start = start;
 *         this.end = end;
 *         this.count = count;
 *         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 || start > end) {
            return 0;
        }

        if (start <= root.start && end >= root.end) {
            return root.count;
        }

        int mid = root.start + (root.end-root.start)/2;
        return query(root.left, start, Math.min(mid, end)) + query(
    }
}
```

Good blogs

Blogs to study

AlgoBox by dietpepsi: leetcode solutions

http://algobox.org/

Leetcode Frequency

http://www.acmerblog.com/leetcode

Knowledge Bases

OS: Process, thread

Process, Threads and Synchronization

http://www.zrzahid.com/process-threads-and-synchronization/

Java Concurrency Tutorial

http://howtodoinjava.com/java-concurrency-tutorial/

Network

1. What really happens when you navigate to a URL

   http://igoro.com/archive/what-really-happens-when-you-navigate-to-a-url/

   https://web.stanford.edu/class/msande91si/www-spr04/readings/week1/InternetWhitepaper.htm

System Design

1. singleton pattern

```
Singleton design pattern in java

http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-ja

**a thorough explanation of singleton pattern**
```

2. factory pattern

```
Factory design pattern in java

http://howtodoinjava.com/2012/10/23/implementing-factory-design-pat
```

3. abstract factory pattern

```
Abstract factory pattern in java

http://howtodoinjava.com/2012/10/29/abstract-factory-pattern-in-ja
```

4. prototype pattern

```
Prototype design pattern in java

http://howtodoinjava.com/2013/01/04/prototype-design-pattern-in-ja
```

Java data structures

How hashmap works in java

http://howtodoinjava.com/2012/10/09/how-hashmap-works-in-java/

useful-java-collection-interview-questions

http://howtodoinjava.com/2013/07/09/useful-java-collection-interview-questions/

Java Syntax

1. define min-heap or max-heap in Java

min-heap:

```
Queue<Integer> min-heap = new PriorityQueue<>();
```

max-heap:

```
Queue<Integer> max-heap = new PriorityQueue<>(Collections.reverseOr
```

Summary

Binary Search Tree

Part I: Basic operations

1. Insert Node

```
public static TreeNode insert(TreeNode head, int val) {
     if (head == null) {
         head = new TreeNode(val);
         return head;
     }
     TreeNode n = head;
     while (n != null) {
         if (val < n.val) {
             if (n.left == null) {
                 n.left = new TreeNode(val);
                 break;
             }
             n = n.left;
         } else {
             if (n.right == null) {
                 n.right = new TreeNode(val);
                 break;
             }
             n = n.right;
         }
     }

     return head;
}
```

2. Delete value

```
public static TreeNode delete(TreeNode head, int val) {
     if (head == null) {
         return null;
     }
     TreeNode n = head;
     TreeNode prev = null;

     while (val != n.val) { // find the node to be deleted, pre
         prev = n;
         if (val < n.val) {
             //prev = n;
             n = n.left;
         } else {
             //prev = n;
             n = n.right;
         }
     }
     // move the left subtree to be left of the left most node
     TreeNode t = n.right;
     while (t != null && t.left != null) {
         t = t.left;
     }

     t.left = n.left;

     if (prev == null) { // to delete head
         head = n.right;
     } else {
         if (prev.left == n) {
             prev.left = n.right;
         } else {
             prev.right = n.right;
         }
     }

     return head;
}
```

3. Search value
4. In-order successor/preceder

```
public static TreeNode successor(TreeNode root, TreeNode node)
     if (root == null) {
         return null;
     }

     if (node.right != null) {
         node = node.right;
         while (node.left != null) {
             node = node.left;
         }
         return node;
     }

     TreeNode n = null;
     TreeNode successor = null;
     n = root;
     while (n != node) {
         if (n.val > node.val) {
             successor = n;
             n = n.left;
         } else {
             n = n.right;
         }
     }

     return successor;
 }
```

5. In-order precessor

```
public static TreeNode precessor(TreeNode root, TreeNode node)
     if (root == null) {
         return null;
     }

     if (node.left != null) {
         node = node.left;
         while (node.right != null) {
             node = node.right;
         }
         return node;
     }

     TreeNode n = root;
     TreeNode precessor = null;

     while (n != node) {
         if (n.val > node.val) {
             n = n.left;
         } else {
             precessor = n;
             n = n.right;
         }
     }

     return precessor;
 }
```

6. Verify Preorder Sequence in Binary Search Tree

```
public boolean verifyPreorder(int[] preorder) {
 int low = Integer.MIN_VALUE, i = -1;
 for (int p : preorder) {
     if (p < low)
         return false;
     while (i >= 0 && p > preorder[i])
         low = preorder[i--];
     preorder[++i] = p;
 }
 return true;
}
```

Postorder version

```
public boolean verifyPostorder(int[] postorder) {
    int high = Integer.Max_VALUE, i = postorder;
    for (int p : postorder) {
        if (p > high)
            return false;
        while (i < postorder.length && p < preorder[i])
            high = preorder[i++];
        preorder[--i] = p;
    }
    return true;
}
```

Part II: Traversal

1. Pre-order

```
public List<Integer> preorderTraversal(TreeNode root) {
     List<Integer> res = new ArrayList<>();
     if (root == null) {
         return res;
     }

     Stack<TreeNode> q = new Stack<>();
     q.push(root);

     while (!q.isEmpty()) {
         TreeNode n = q.pop();
         res.add(n.val);
         if (n.right != null) {
             q.push(n.right);
         }
         if (n.left != null) {
             q.push(n.left);
         }
     }
     return res;
}
```

2. In-order

```
public List<Integer> inorderTraversal(TreeNode root) {
     List<Integer> res = new ArrayList<>();
     if (root == null) {
         return res;
     }

     Stack<TreeNode> stack = new Stack<>();

     while (!stack.isEmpty() || root != null) {
         if (root != null) {
             stack.push(root);
             root = root.left;
         }

         else {
             root = stack.pop();
             res.add(root.val);
             root = root.right;
         }
     }
     return res;
 }
```

3. Post-order

```
public List<Integer> postorderTraversal(TreeNode root) {
     List<Integer> res = new ArrayList<>();
     if (root == null) {
         return res;
     }

     Stack<TreeNode> stack = new Stack<>();
     TreeNode prev = null;

     while (!stack.isEmpty() || root != null) {
         if (root != null) {
             stack.push(root);
             root = root.left;
         } else {
             TreeNode peek = stack.peek();
             if (peek.right != null && prev != peek.right) {
                 root = peek.right;
             } else {
                 stack.pop();
                 res.add(peek.val);
                 prev = peek;
             }
         }
     }

     return res;
 }
```

4. Level/vertical/zigzag order traversal:

```
public List<List<Integer>> levelOrder(TreeNode root) {
     List<List<Integer>> res = new ArrayList<List<Integer>>();
     if (root == null) {
         return res;
     }

     Queue<TreeNode> q = new LinkedList<>();
     q.offer(root);

     while (!q.isEmpty()) {
         int size = q.size();
         List<Integer> tmp = new ArrayList<>();
         for (int i = 0; i < size; i++) {
             TreeNode n = q.poll();
             tmp.add(n.val);  // change this for different orde
             if (n.left != null) {
                 q.offer(n.left);
             }
             if (n.right != null) {
                 q.offer(n.right);
             }
         }
         res.add(tmp); // change this for different orders
     }

     return res;
 }
```

Part III: array/list <-> binary tree

1. sorted array to BST

```
public TreeNode sortedArrayToBST(int[] nums) {
     if (nums.length == 0) {
         return null;
     }

     return sortedArrayToBST(nums, 0, nums.length-1);
 }
     public TreeNode sortedArrayToBST(int[] nums, int left, int
         if (left > right) {
             return null;
         }

         int mid = left + (right-left)/2;
         TreeNode node = new TreeNode(nums[mid]);
         node.left = sortedArrayToBST(nums, left, mid - 1);
         node.right = sortedArrayToBST(nums, mid + 1, right);
         return node;
     }
```

2. sorted list to BST

Iterative version: merge sort like

```
public TreeNode sortedListToBST(ListNode head) {
        if (head == null) {
            return null;
        }

        if (head.next == null) {
            return new TreeNode(head.val);
        }

        ListNode slow = head;
        ListNode fast = head.next;

        while (fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        TreeNode root = new TreeNode(slow.next.val);

        ListNode second = slow.next.next;
        slow.next = null;

        root.left = sortedListToBST(head);
        root.right = sortedListToBST(second);

        return root;
    }
```

1. Serialize/deserialize Binary tree

easiest way to do: BFS, parent-children with a separating symbol

```
public String serialize(TreeNode root) {
        if (root == null) {
            return "";
        }
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        StringBuilder sb = new StringBuilder();
        while (!q.isEmpty()) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                TreeNode n = q.poll();
                if (n == null) {
                    sb.append("#,");
                } else {
                    sb.append(n.val + ",");
                    q.offer(n.left);
                    q.offer(n.right);
                }

            }
        }

        return sb.toString().substring(0, sb.length()-1);
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if (data == null || data.length() == 0) {
            return null;
        }
        String[] str = data.split(",");
        TreeNode root = new TreeNode(Integer.parseInt(str[0]));
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);

        int index = 1;
        while (index < str.length) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                TreeNode n = q.poll();
                if (!str[index].equals("#")) {
                    n.left = new TreeNode(Integer.parseInt(str[inde
                    q.offer(n.left);
                }
                index++;
                if (!str[index].equals("#")) {
                    n.right = new TreeNode(Integer.parseInt(str[ind
                    q.offer(n.right);
                }
                index++;
            }
        }

        return root;
    }
```

1. Serialize/deserialize N-ary tree

```
// parent-all children BFS
 public static String serializeTreeIte2(TreeNode root) {
     if (root == null) {
         return new String();
     }
     StringBuilder sb = new StringBuilder();
     Queue<TreeNode> queue = new LinkedList<>();
     queue.offer(root);

     while (!queue.isEmpty()) {
         int size = queue.size();

         for (int i = 0; i < size; i++) {
             TreeNode n = queue.poll();
             sb.append(n.c);
             for (TreeNode t : n.children) {
                 queue.offer(t);
             }
         }
         sb.append(')');
     }

     return sb.toString();
 }

 public static TreeNode deserializeTreeIte2(String str) {
     if (str == null || str.length() == 0) {
         return null;
     }

     Queue<TreeNode> queue = new LinkedList<>();
     TreeNode root = new TreeNode(str.charAt(0));
     queue.offer(root);
     int index = 2;
     while (index < str.length()) {
         int size = queue.size();

         for (int i = 0; i < size; i++) {
             TreeNode n = queue.poll();
             while (str.charAt(index) != ')') {
                 TreeNode t = new TreeNode(str.charAt(index++))
                 n.children.add(t);
                 queue.offer(t);
             }
         }
         index++;
     }

     return root;
 }
```

2. construct binary tree from postoder+inorder traversal

```
public TreeNode buildTree(int[] inorder, int[] postorder) {
     if (inorder.length != postorder.length || inorder.length =
         return null;
     }
     return buildTree(inorder, 0, inorder.length-1, postorder,
 }

 public TreeNode buildTree(int[] inorder, int inStart, int inEr
     if (inStart > inEnd || postStart > postEnd) {
         return null;
     }

     TreeNode root = new TreeNode(postorder[postEnd]);
     int index = 0;
     for (int i = inStart; i <= inEnd; i++) {
         if (inorder[i] == postorder[postEnd]) {
             index = i;
             break;
         }
     }

     root.left = buildTree(inorder, inStart, index - 1, postorc
     root.right = buildTree(inorder, index + 1, inEnd, postorde

     return root;
 }
```

3. construct binary tree from preoder+inorder traversal

```
public TreeNode buildTree(int[] preorder, int[] inorder) {
     if (preorder.length == 0 || inorder.length == 0) {
         return null;
     }

     return buildTree(preorder, 0, preorder.length-1, inorder,
 }

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

     TreeNode node = new TreeNode(preorder[preStart]);
     int index = 0;
     for (int i = inStart; i <= inEnd; i++) {
         if (inorder[i] == preorder[preStart]) {
             index = i;
             break;
         }
     }
     int k = index-inStart;
     node.left = buildTree(preorder, preStart+1, preStart+k, ir
     node.right = buildTree(preorder, preStart+k+1, preEnd, inc
     return node;
 }
```

Binary Search

Expression evaluation

use stack

1. Basic calculator

```
public int calculate(String s) {
     if (s == null || s.length() == 0) {
         return -1;
     }

     Stack<Integer> stack = new Stack<>();
     int op = 1;
     int num = 0;
     int res = 0;
     int i = 0;

     while (i < s.length()) {
         char c = s.charAt(i);
         if (Character.isDigit(c)) {
             while (i < s.length() && Character.isDigit(s.charA
                 num = num * 10 + s.charAt(i) - '0';
                 i++;
             }
             res += num * op;
         }
         else if (c == '+') {
             op = 1;
             num = 0;
             i++;
         }
         else if (c == '-') {
             op = -1;
             num = 0;
             i++;
         }
         else if (c == '(') {
             stack.push(res);
```

```
                stack.push(op);
                res = 0;
                op = 1;
                i++;
            }
            else if (c == ')') {
                op = stack.pop();
                num = stack.pop();
                res = num + res * op;
                i++;
            }
            else {
                i++;
            }
        }
        return res;
    }
```

2. Basic calculator II

```
public int calculate(String s) {
     if (s == null || s.length() == 0) {
         return 0;
     }
     Stack<Integer> stack = new Stack<>();
     int op = 0, num = 0;
     int res = 0;
     int i = 0;
     while (i < s.length()) {
         char c = s.charAt(i);
         if (Character.isDigit(c)) {
                 num = num * 10 + s.charAt(i) - '0';
                 i++;
             }

             if (op == 3 || op == 4) {
                 int t = stack.pop();
                 if (op == 3)
                     res = t * num;
```

```
                else
                    res = t / num;
                stack.push(res);
            }
            else
                stack.push(num);
        }
        else if (c == '+') {
            op = 1;
            stack.push(op);
            num = 0;
            i++;
        }
        else if (c == '-') {
            op = 2;
            stack.push(op);
            num = 0;
            i++;
        }
        else if (c == '*') {
            op = 3;
            num = 0;
            i++;
        }
        else if (c == '/') {
            op = 4;
            num = 0;
            i++;
        }
        else {
            i++;
        }
    }
    res = 0;
    while (stack.size() > 1) {
        int a = stack.pop();
        op = stack.pop();
        res = op == 1 ? res + a : res - a;
    }
    res += stack.pop();
```

```
        return res;
    }
```

3. Evaluate Reverse Polish Notation

```
public int evalRPN(String[] tokens) {
        if (tokens == null || tokens.length == 0) {
            return 0;
        }

        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i < tokens.length; i++) {
            String s = tokens[i];
            if (s.equals("+") || s.equals("-") || s.equals("*") ||
                int b = stack.pop();
                int a = stack.pop();
                stack.push(evaluate(a, b, s));
            } else {
                stack.push(Integer.parseInt(s));
            }
        }
        return stack.pop();
    }

    public int evaluate(int a, int b, String token) {
        switch (token) {
            case "+" : return a + b;
            case "-" : return a - b;
            case "*" : return a * b;
            case "/" : return a / b;
            default  : return 0;
        }
    }
```

1. Expression Add Operators: add operators to a string of numbers

```
public List<String> addOperators(String num, int target) {
    List<String> res = new ArrayList<>();
    StringBuilder sb = new StringBuilder();
    dfs(res, sb, num, 0, target, 0, 0);
    return res;

}
public void dfs(List<String> res, StringBuilder sb, String num, int
    if(pos == num.length()) {
        if(target == prev) res.add(sb.toString());
        return;
    }
    for(int i = pos; i < num.length(); i++) {
        if(num.charAt(pos) == '0' && i != pos) break;
        long curr = Long.parseLong(num.substring(pos, i + 1));
        int len = sb.length();
        if(pos == 0) {
            dfs(res, sb.append(curr), num, i + 1, target, curr, cu
            sb.setLength(len);
        } else {
            dfs(res, sb.append("+").append(curr), num, i + 1, targe
            sb.setLength(len);
            dfs(res, sb.append("-").append(curr), num, i + 1, targe
            sb.setLength(len);
            dfs(res, sb.append("*").append(curr), num, i + 1, targe
            sb.setLength(len);
        }
    }
}
```

1. Different Ways to Add Parentheses:

divide into left and right half and recursively solve

```
public List<Integer> diffWaysToCompute(String input) {
        List<Integer> res = new ArrayList<>();
        if (input.length() == 0) {
            return res;
        }

        for (int i = 0; i < input.length(); i++) {
            char c = input.charAt(i);
            if (c == '+' || c == '-' || c == '*') {
                String left = input.substring(0, i);
                String right = input.substring(i+1);
                List<Integer> l = diffWaysToCompute(left);
                List<Integer> r = diffWaysToCompute(right);
                for (int a : l) {
                    for (int b : r) {
                        res.add(eval(a, b, c));
                    }
                }
            }
        }
        if (res.size() == 0) {
            res.add(Integer.parseInt(input));
        }

        return res;
    }

    public int eval(int a, int b, char c) {
        switch(c) {
            case '+' : return a + b;
            case '-' : return a - b;
            case '*' : return a * b;
            default  : return -1;
        }
    }
```

Combination/Permutation/Subset

Part I: Combination

1. k numbers from 1 to n

```
public List<List<Integer>> combine(int n, int k) {
     List<List<Integer>> res = new ArrayList<List<Integer>>();
     if (n < 1 || k > n) {
         return res;
     }
     combine(n, 1, new ArrayList<Integer>(), k, res);
     return res;
 }

 public void combine(int n, int index, List<Integer> list, int
     if (index > n && k != list.size()) {
         return;
     }
     if (k == list.size()) {
         res.add(new ArrayList<>(list));
         return;
     }

     for (int i = index; i <= n; i++) {
         list.add(i);
         combine(n, i+1, list, k, res);
         list.remove(list.size()-1);
     }
 }
```

2. Combination sum: element can be repeately used

```
public List<List<Integer>> combinationSum(int[] candidates, int
     List<List<Integer>> res = new ArrayList<List<Integer>>();
     if (candidates.length == 0) {
         return res;
     }
     Arrays.sort(candidates);
     combineSum(candidates, 0, target, new ArrayList<Integer>()
     return res;
 }

 public void combineSum(int[] candidates, int index, int target
     if (target < 0) {
         return;
     }
     if (target == 0) {
         res.add(new ArrayList<Integer>(curr));
         return;
     }

     for (int i = index; i < candidates.length; i++) {
         if (i > index && candidates[i] == candidates[i-1])  //
             continue;
         curr.add(candidates[i]);
         combineSum(candidates, i, target-candidates[i], curr,
         curr.remove(curr.size()-1);
     }
 }
```

3. Combination sum: element can be used once

```
public List<List<Integer>> combinationSum2(int[] candidates, ir
     List<List<Integer>> res = new ArrayList<List<Integer>>();
     if (candidates.length == 0) {
         return res;
     }
     Arrays.sort(candidates);
     combineSum(candidates, 0, target, new ArrayList<Integer>()
     return res;
 }

 public void combineSum(int[] candidates, int index, int target
     if (index > candidates.length || target < 0) {
         return;
     }
     if (target == 0) {
         res.add(new ArrayList<Integer>(curr));
         return;
     }

     for (int i = index; i < candidates.length; i++) {
         if (i > index && candidates[i] == candidates[i-1])
             continue;
         curr.add(candidates[i]);
         combineSum(candidates, i+1, target-candidates[i], curr
         curr.remove(curr.size()-1);
     }
 }
```

4. Factor combinations

```
public List<List<Integer>> getFactors(int n) {
     List<List<Integer>> ret = new ArrayList<List<Integer>> ();
     helper(ret, new ArrayList<Integer> (), n, 2);
     return ret;
 }

 private void helper(List<List<Integer>> ret, List<Integer> ite
     if (n == 1) {
         if (item.size() > 1) {
             ret.add(new ArrayList<Integer> (item));
         }
         return;
     }
     for (int i = start; i <= n; i++) {
         if (n % i == 0) {
             item.add(i);
             helper(ret, item, n/i, i);
             item.remove(item.size()-1);
         }
     }
 }
```

Part II: Subsets

1. distinct elements

a. insert element to the previous result

```
public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        res.add(new ArrayList<Integer>());
        if (nums.length == 0) {
            return res;
        }
        Arrays.sort(nums);
        for (int i = 0; i < nums.length; i++) {
            int num = nums[i];
            int size = res.size();

            for (int j = 0; j < size; j++) {
                List<Integer> tlist = new ArrayList<>(res.get(j));
                tlist.add(num);
                res.add(tlist);
            }
        }
        return res;
    }
```

b. DFS

```
public List<List<Integer>> subsets(int[] S) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();

    if(S.length == 0){
        return result;
    }

    Arrays.sort(S);
    dfs(S, 0, new ArrayList<Integer>(), result);
    return result;
}

public void dfs(int[] s, int index, List<Integer> path, List<List<I
    result.add(new ArrayList<Integer>(path));

    for(int i = index; i < s.length; i++){
        path.add(s[i]);
        dfs(s, i+1, path, result);
        path.remove(path.size()-1);
    }
}
```

c.bit manipulation

```
public List<List<Integer>> subsets(int[] nums) {
    int n = nums.length;
    List<List<Integer>> subsets = new ArrayList<>();
    for (int i = 0; i < Math.pow(2, n); i++)
    {
        List<Integer> subset = new ArrayList<>();
        for (int j = 0; j < n; j++)
        {
            if (((1 << j) & i) != 0)
                subset.add(nums[j]);
        }
        Collections.sort(subset);
        subsets.add(subset);
    }
    return subsets;
}
```

1. with duplicate elements

a. when encounter duplicate, add new element only to the right half: see the variable start

```
public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        res.add(new ArrayList<Integer>());
        if (nums.length == 0) {
            return res;
        }
        Arrays.sort(nums);
        int start = 0;
        for (int i = 0; i < nums.length; i++) {
            int num = nums[i];
            int size = res.size();

            for (int j = start; j < size; j++) {
                List<Integer> tlist = new ArrayList<>(res.get(j));
                tlist.add(num);
                res.add(tlist);
            }

            start = 0;
            if (i < nums.length-1 && nums[i] == nums[i+1]) {
                start = size;
            }

        }
        return res;
    }
```

b. check if the list is already in the result each time, less efficient

```
public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        res.add(new ArrayList<Integer>());
        if (nums.length == 0) {
            return res;
        }
        Arrays.sort(nums);
        for (int i = 0; i < nums.length; i++) {

            int num = nums[i];
            List<List<Integer>> tmp = new ArrayList<List<Integer>>(
            for (List<Integer> list : res) {
                List<Integer> tlist = new ArrayList<>(list);
                tlist.add(num);
                if (!res.contains(tlist))
                    tmp.add(tlist);
            }
            res.addAll(tmp);
        }
        return res;
    }
```

Part III: Permutations

1. Distinct elements

a. add new element to every possible position of the previous set

```
public List<List<Integer>> permute(int[] nums) {
         List<List<Integer>> res = new ArrayList<List<Integer>>();
         if (nums.length == 0) {
             return res;
         }

        res.add(new ArrayList<Integer>());

         for (int i = 0; i < nums.length; i++) {
             int num = nums[i];
             int size = res.size();

             List<List<Integer>> bak = new ArrayList<List<Integer>>

             for (int j = 0; j < size; j++) {
                 List<Integer> list = res.get(j);
                 for (int k = 0; k <= list.size(); k++) {
                     List<Integer> tmp = new ArrayList<>(list);
                     tmp.add(k, num);
                     bak.add(tmp);
                 }
                 //res.remove(j);
             }
             res = bak;
         }
         return res;
    }
```

b. swap every element once to the front and do the same thing for the rest

```
public List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        Arrays.sort(nums);
        LinkedList<Integer> list = new LinkedList<Integer>();
        for (int num : nums) list.add(num);
        permute(list, 0, res);
        return res;
    }
    private void permute(LinkedList<Integer> nums, int start, List<
        if (start == nums.size() - 1){
            res.add(new LinkedList<Integer>(nums));
            return;
        }
        for (int i = start; i < nums.size(); i++){
            if (i > start && nums.get(i) == nums.get(i - 1)) contir
            nums.add(start, nums.get(i));
            nums.remove(i + 1);
            permute(nums, start + 1, res);
            nums.add(i + 1, nums.get(start));
            nums.remove(start);
        }
    }
```

1. with duplicates: check previous code
2. find the kth permutation sequence of number 1 to n

```
public String getPermutation(int n, int k) {
     if (n <= 0 || k <= 0) {
         return new String();
     }

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

3. next permutation

```
public void nextPermutation(int[] nums) {
    if(nums == null || nums.length<2)
        return;

    int p=0;
    for(int i=nums.length-2; i>=0; i--){
        if(nums[i]<nums[i+1]){
            p=i;
            break;
        }
    }

    int q = 0;
    for(int i=nums.length-1; i>p; i--){
        if(nums[i]> nums[p]){
            q=i;
            break;
        }
    }

    if(p==0 && q==0){
        reverse(nums, 0, nums.length-1);
        return;
    }

    int temp=nums[p];
    nums[p]=nums[q];
    nums[q]=temp;

    if(p<nums.length-1){
        reverse(nums, p+1, nums.length-1);
    }
}

public void reverse(int[] nums, int left, int right){
    while(left<right){
        int temp = nums[left];
        nums[left]=nums[right];
        nums[right]=temp;
        left++;
        right--;
    }
}
```

String subsequence/substring problems

Commonly solved by dynamic programming

1. Longest common subsequence

http://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/

1. Longest common substring

http://www.geeksforgeeks.org/longest-common-substring/

1. Longest repeating substring

http://www.geeksforgeeks.org/suffix-tree-application-3-longest-repeated-substring/

1. Longest repeating subsequence

http://www.geeksforgeeks.org/longest-repeating-subsequence/

1. Distinct Subsequences

https://leetcode.com/problems/distinct-subsequences/

1. Longest Palindromic Subsequence

http://www.geeksforgeeks.org/dynamic-programming-set-12-longest-palindromic-subsequence/

1. Longest Palindromic Substring

http://www.geeksforgeeks.org/longest-palindrome-substring-set-1/