LeetBook

zhaoyuned

Published with GitBook

Table of Contents

My leetcode book

数组

- FirstMissingPositive
- LongestConsecutiveSequence
- MajorityElements
- MajorityElement II
- Pascal'sTriangle
- Pascal's Triangle II
- SpiralMatrix
- SpiralMatrix2
- RotateArray
- SummaryRanges
- Next Permutation
- Set Matrix Zeroes
- ProductOfArrayExcept
- RoateImage

FirstMissingPositive

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.

思路：

每次把数组的元素移动到其对应的位置之后从头遍历找不符合条件的 index(此题目在面试中遇到过)

移动的元素大小需要在[0,nums.size()]之间，大小不在这范围内的直接跳过即可。

```
int firstMissingPositive(vector<int>& nums)
{
        int n = nums.size();
        for(int i=0;i<n;i++)
        {
            while(nums[i]>0 && nums[i]<=n && nums[i] != nums[nums[i]-1])
                swap(nums[i],nums[nums[i]-1]);
        }
        for(int i=0;i<n;i++)
            if(nums[i]!=i+1)
                return i+1;
        return n+1;
}
```

longestConsecutiveSequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example, Given [100, 4, 200, 1, 3, 2], The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Your algorithm should run in O(n) complexity.

思路：

用一个map来存每个元素所在的最长连续序列的长度。 对于数组中的每个数s,找 s-1 与 s+1 的序列长度,分别为left，right，最终s所在最长连续序列的长度即为left+right+1

```
 int longestConsecutive(vector<int>& nums) {
        unordered_map<int, int> m;
        int r = 0;

        for(int s:nums)
        {
            if(m[s]) continue;

            int left = (m.find(s-1) == m.end()) ? 0: m[s-1];
            int right = (m.find(s+1) == m.end()) ? 0: m[s+1];

            int leng = right +left +1;
            m[s] =leng;

            r = max(leng,r);

            m[s-left] = leng;
            m[s+right] = leng;
        }
        return r;
}
```

精简版代码

```
int longestConsecutive(vector<int> &num)
{
    unordered_map<int, int> m;
    int r = 0;
    for (int i : num) {
        if (m[i]) continue;
        r = max(r, m[i] = m[i + m[i + 1]] = m[i - m[i - 1]] = m[i + 1] + m[i - 1] + 1);
    }
    return r;
}
```

majorityElement

Given an array of size n, find the majority element. The majority element is the element that appears more than n/2 times.

You may assume that the array is non-empty and the majority element always exist in the array.

```
int majorityElement(vector<int>& nums)
{

        int majority = nums[0];
        int count = 1;
        for(int i=1 ;i<nums.size();i++)
        {
            if(count==0)
            {
                    majority = nums[i];
                    count++;
            }
            else
            {
                if(nums[i] == majority)
                    count++;
                else
                    count--;
            }
        }
        return majority;
    }
```

majorityElement II

Given an integer array of size n, find all elements that appear more than n/3 times. The algorithm should run in linear time and in O(1) space.

Hint: How many majority elements could it possibly have? Do you have a better hint? Suggest it!

思路： 最多两个元素 出现超过n/3

```
vector<int> majorityElement(vector<int> &a) {
    int y = 0, z = 1, cy = 0, cz = 0;
    for (auto x: a) {
      if (x == y) cy++;
      else if (x == z) cz++;
      else if (! cy) y = x, cy = 1;
      else if (! cz) z = x, cz = 1;
      else cy--, cz--; //不同要相减
    }
    cy = cz = 0;
    for (auto x: a)
      if (x == y) cy++;
      else if (x == z) cz++;

    vector<int> r;
    if (cy > a.size()/3) r.push_back(y);
    if (cz > a.size()/3) r.push_back(z);
    return r;
  }
```

Psscal's Triangle

Given numRows, generate the first numRows of Pascal's triangle.

For example, given numRows = 5, Return

[[1], [1,1], [1,2,1], [1,3,3,1], [1,4,6,4,1]]

```
vector<vector<int>> generate(int numRows)
{
        vector<vector<int>> result;
        if(numRows<=0)
            return result;
        vector<int> first(1,1);

        result.push_back(first);
        for(int i=2;i<=numRows;i++)
        {
            vector<int> cur;
            cur.push_back(1);

            vector<int> pre = result[i-2];

            for(int j=0;j<pre.size()-1;j++)
            {
                cur.push_back(pre[j]+pre[j+1]);
            }

            cur.push_back(1);
            result.push_back(cur);
        }

        return result;
}
```

精简代码版

```
vector<vector<int> > generate(int numRows) {
          vector<vector<int> > r;
            for(int i = 0; i < numRows; i++){
                vector<int> tmp;
                for(int j = 0; j <= i; j++){
                    if(j == 0 || j == i){
                        tmp.push_back(1);
                    }
                    else{
                        tmp.push_back(r[i - 1][j - 1] + r[i - 1][j]);
                    }
                }
                r.push_back(tmp);
            }
            return r;
```

```
    }
```

Pascal's Triangle II

Given an index k, return the kth row of the Pascal's triangle.

For example, given k = 3, Return [1,3,3,1].

Note: Could you optimize your algorithm to use only O(k) extra space?

思路： 在原数组上进行操作

```
vector<int> getRow(int rowIndex)
{
        vector<int> result;
        if(rowIndex<0)
            return result;
        result.push_back(1);
        for(int i=1;i<=rowIndex;i++)
        {
            int size = result.size();
            for(int j=size-1; j>0;j--)
            {
                result[j] += result[j-1];
            }
            result.push_back(1);
        }

        return result;
}
```

SpiralMatrix

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.

For example, Given the following matrix:

[[1, 2, 3], [4, 5, 6], [7, 8, 9]] You should return [1,2,3,6,9,8,7,4,5].

思路： 用x1 x2 y1 y2 记录 索引 然后按照要求来遍历 二维数组

```
vector<int> spiralOrder(vector<vector<int> > &matrix) {
        vector<int> result;

        if(matrix.size() == 0)
            return result;


        int x1 = 0;
        int y1 = 0;
        int x2 = matrix.size()-1;
        int y2 = matrix[0].size()-1;

        while(x1 <= x2 && y1 <= y2)
        {
            //up row
            for(int i = y1; i <= y2; ++i) result.push_back(matrix[x1][i]);
            //right column
            for(int i = x1+1; i <= x2; ++i) result.push_back(matrix[i][y2]);
            //bottom row
            if(x2 != x1)
            for(int i = y2-1; i >= y1; --i) result.push_back(matrix[x2][i]);
            //left column
            if(y1 != y2)
            for(int i = x2-1; i > x1; --i) result.push_back(matrix[i][y1]);

            x1++, y1++, x2--, y2--;
        }


        return result;
}
```

SpiralMatrix2

Given an integer n, generate a square matrix filled with elements from 1 to n2 in spiral order.

For example, Given n = 3,

You should return the following matrix: [[1, 2, 3], [8, 9, 4], [7, 6, 5]]

```
vector<vector<int>> generateMatrix(int n)
{
        if(n<=0)
            return vector<vector<int> >();
        if(n==1)
            return vector<vector<int> >(1,vector<int>(1,1));

        vector<vector<int>> result(n,vector<int>(n,0));

        int x1=0,y1=0;
        int x2=n-1,y2=n-1;

        int num =1;

        while(x1<=x2 && y1<=y2)
        {
            for(int i=y1; i<=y2 ;i++)
                result[x1][i] = num++;
            for(int i=x1+1;i<=x2;i++)
                result[i][y2] = num++;
            for(int i= y2-1;i>=y1; i--)
                result[x2][i] = num++;
            for(int i= x2-1;i>x1;i--)
                result[i][y1]=num++;
            x1++;y1++;
            x2--;y2--;
        }

        return result;
}
```

RotateArray

Rotate an array of n elements to the right by k steps.

For example, with n = 7 and k = 3, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4].

Note: Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

方法1：

exactly O(n)time O(1) space

```
void rotate(vector<int>& nums, int k) {
        int n = nums.size();
        if(n == 0)return;
        k = k % n;

        //initialize
        int i = 0;
        int nowIndex = 0;
        int tmp = nums[0],stmp;

        //exactly n steps, n times loop
        for(int j = 0; j < n; j++)
        {
            //next index to exchange
            nowIndex = (nowIndex + k) % n;

            //exchange
            stmp = nums[nowIndex];
            nums[nowIndex] = tmp;
            tmp = stmp;

            //finish a circle,move to another circle
            if(nowIndex == i)
            {
                nowIndex = ++i;
                tmp = nums[nowIndex];
            }
        }
}
```

方法2

```
void rotate(vector<int>& nums, int k)
{
        if(nums.size()<=1 || k<0)
            return;
        k = k%nums.size();
        reverse(nums.begin(),nums.begin()+nums.size()-k);

        reverse(nums.begin()+nums.size()-k,nums.end());

        reverse(nums.begin(),nums.end());
}
```

SummaryRanges

Given a sorted integer array without duplicates, return the summary of its ranges.

For example, given [0,1,2,4,5,7], return ["0->2","4->5","7"].

```
vector<string> summaryRanges(vector<int>& nums) {
        int i = 0, size = nums.size();
        vector<string> result;
        while(i < size){
            int j = 1;
            while(i + j < size && nums[i + j] - nums[i] == j) ++j;
            result.push_back(j <= 1 ? to_string(nums[i]) : to_string(nums[i])
            + "->" + to_string(nums[i + j - 1]));
            i += j;
        }
        return result;
    }
```

NextPermutation

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

1,2,3 → 1,3,2

3,2,1 → 1,2,3

1,1,5 → 1,5,1

思路：

对于一个序列，从右向左找第一个升序的数字，其下标为pivot

- 若pivot>0 下一步我们就需要找到(从右向左找)第一个比nums[pivot-1]大的元素，假设其下表为large,交换nums[pivot-1]与nums[large],最后我们将序列下标从pivot到末尾的元素reverse
- 若pivot<=0 整个序列则是降序排列，该序列的下一个序列需要将整个现有序列reverse

举个例子可能会更容易理解

序列： 4 6 5 3 2 1 找到pivot元素为6

从末尾找第一个比4大的元素为5，交换这两个元素，为

5 6 4 3 2 1 reverse 从 6到末尾1的元素，最终为

5 1 2 3 4 6

```
void nextPermutation(vector<int>& nums)
{
        int end = nums.size()-1;
        int pivot = end;
        while(pivot>0)
        {
            if(nums[pivot] > nums[pivot-1] )  //从后找第一个升序的数字
                break;
            pivot--;
        }

        if(pivot>0)
        {
            pivot--;
            int large = end;
            //找到第一个比nums[pivot]大的数
            while(nums[large] <= nums[pivot]) large--;
            swap(nums[large],nums[pivot]);
            reverse(nums.begin()+pivot+1,nums.end());
        }else
        {
            reverse(nums.begin(),nums.end());
        }
}
```

SetMatrixZeros

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

click to show follow up.

Follow up: Did you use extra space? A straight forward solution using O(mn) space is probably a bad idea. A simple improvement uses O(m + n) space, but still not the best solution. Could you devise a constant space solution?

```
void setZeroes(vector<vector<int>>& matrix) {
        int col0 = 1, rows = matrix.size(), cols = matrix[0].size();

    for (int i = 0; i < rows; i++) {
        if (matrix[i][0] == 0) col0 = 0;
        for (int j = 1; j < cols; j++)
            if (matrix[i][j] == 0)
                matrix[i][0] = matrix[0][j] = 0; //把是0的元素的 第0行j列 和 第j行0列 设为0
    }

    for (int i = rows - 1; i >= 0; i--) {  //根绝已设为0的标志 来设置元素
        for (int j = cols - 1; j >= 1; j--)
            if (matrix[i][0] == 0 || matrix[0][j] == 0)
                matrix[i][j] = 0;
        if (col0 == 0) matrix[i][0] = 0;
    }
}
```

ProductofArrayExceptSelf

Given an array of n integers where n > 1, nums, return an array output such that output[i] is equal to the product of all the elements of nums except nums[i].

Solve it without division and in O(n).

For example, given [1,2,3,4], return [24,12,8,6].

Follow up: Could you solve it with constant space complexity? (Note: The output array does not count as extra space for the purpose of space complexity analysis.)

```
vector<int> productExceptSelf(vector<int>& nums) {
        int size = nums.size();
        vector<int> result(size,1);

        for(int i=1;i<size;i++)
            result[i] = result[i-1]*nums[i-1];
        result[0]= nums[size-1];
        for(int i=size-2; i>0;i--)
        {
            result[i] *= result[0];
            result[0] *= nums[i];
        }

        return result;
    }
```

RotateImage

You are given an n x n 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up: Could you do this in-place?

思路：

方法1 ：异或的操作

```
 void rotate(vector<vector<int> >& matrix) {
        //reverse
        for(int i = 0; i < matrix.size() / 2; i++)
        {
            for(int j = 0; j < matrix.size(); j++)
            {
                matrix[i][j] ^= matrix[matrix.size() - 1 - i][j];
                matrix[matrix.size() - 1 - i][j] ^= matrix[i][j];
                matrix[i][j] ^= matrix[matrix.size() - 1 - i][j];
            }
        }

        //swap
        for(int i = 0; i < matrix.size() - 1; i++)
        {
            for(int j = i + 1; j < matrix.size(); j++)
            {
                matrix[j][i] ^= matrix[i][j];
                matrix[i][j] ^= matrix[j][i];
                matrix[j][i] ^= matrix[i][j];
            }
        }
    }
```

方法2 ：同方法1其实是一个思路

```
void rotate(vector<vector<int> > &matrix)
{

     reverse(matrix.begin(), matrix.end());
     for (int i = 0; i < matrix.size(); ++i) {
            for (int j = i + 1; j < matrix[i].size(); ++j)
                swap(matrix[i][j], matrix[j][i]);
     }
}
```

方法3 :直接进行交换

```
void rotate(vector<vector<int>>& matrix)
{

        int n =matrix.size();
        for (int i=0; i<n/2; ++i)
        {
            for (int j=i; j<n-1-i; ++j)
            {
                int z = matrix[i][j];
                matrix[i][j] = matrix[n-j-1][i];
                matrix[n-j-1][i] = matrix[n-i-1][n-j-1];
                matrix[n-i-1][n-j-1] = matrix[j][n-i-1];
                matrix[j][n-i-1] = z;
            }
        }
}
```

双指针

- TrappingRainWater
- MergeSortedArray
- MinimumSizeSubarraySum
- ImplementstrStr
- RemoveDuplicateFromSortArray
- RemoveDulicateFromSortArray2
- LongestSubstringWithoutRepeatingChar
- PartitonList
- RemoveElement
- 3Sum
- 3SumCloest
- 4Sum
- ValidPalindrome
- MinimumWindowSubstring
- ContainerWithMostWater

TrappingRainWater

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.

方法1：two Pointers

```
int trap(vector<int>& height)
{
        int left =0;
        int right = height.size()-1;
        int maxleft=0,maxright=0;
        int result=0;
        while(left<right)
        {
            maxleft = max(maxleft,height[left]);
            maxright = max(maxright,height[right]);
            if(height[left]<height[right])
            {
                result += maxleft-height[left];
                left++;
            }else
            {
                result += maxright-height[right];
                right--;
            }
        }
        return result;
}
```

方法2：先遍历一遍数组，记录每个bar左右两边最高的bar高度

每个bar能装的容量即是min(leftMostHeight[i],rightMostHeight[i])-A[i]

该方法思路跟方法1的思路是等价的。

```
int trap(int A[], int n)
{
        if(A==NULL || n<1)return 0;

        int maxheight = 0;
        vector<int> leftMostHeight(n);
        for(int i =0; i<n;i++)
        {
            leftMostHeight[i]=maxheight;
            maxheight = maxheight > A[i] ? maxheight : A[i];
        }

        maxheight = 0;
        vector<int> rightMostHeight(n);
        for(int i =n-1;i>=0;i--)
        {
            rightMostHeight[i] = maxheight;
            maxheight = maxheight > A[i] ? maxheight : A[i];
        }

        int water = 0;
        for(int i =0; i < n; i++)
        {
            int high = min(leftMostHeight[i],rightMostHeight[i])-A[i];
            if(high>0)
                water += high;
        }
        return water;
}
```

方法3：利用stack

```
struct Node
 {
        int h, len;
        Node(int _h = 0, int _len = 0):h(_h), len(_len){};
 };

 int trap(vector<int>& A)
{

        int n= A.size();
        int areaSum = 0;
        stack<Node> S;
        S.push(Node(0, 0));
        for(int i = 0; i < n; ++i)
        {
            if(A[i] < S.top().h)
            {
                 S.push(Node(A[i], 1));
                 continue;
            }
            int curLen = 0;
            int heightSum = 0;
            while(S.size() > 1 && A[i] >= S.top().h)
            {
                curLen += S.top().len;
                heightSum += S.top().h*S.top().len;
                S.pop();
            }

            int minHeight = min(S.top().h, A[i]);
            int curArea = minHeight*curLen-heightSum;
            areaSum += curArea;
            if(A[i] > S.top().h)
            {
                S.pop();
                S.push(Node(A[i], 0));
            }
            else S.push(Node(A[i], curLen+1));
        }
        return areaSum ;
}
```

MergeSortedArray

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

Note: You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2. The number of elements initialized in nums1 and nums2 are m and n respectively.

思路:two pointer 算法 ,需要从 nums1的末尾开始 添加元素

```
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
       int index = m + n - 1, i = m - 1, j = n - 1;
        while(j >=0)
            if(i < 0 || nums1[i] < nums2[j])
                nums1[index--] = nums2[j--];
            else nums1[index--] = nums1[i--];
}
```

MinimumSizeSubArrayLen

Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the sum >= s. If there isn't one, return 0 instead.

For example, given the array [2,3,1,2,4,3] and s = 7, the subarray [4,3] has the minimal length under the problem constraint.

If you have figured out the O(n) solution, try coding another solution of which the time complexity is O(n log n).

思路： two pointer 时间复杂度为O(N)

```
int minSubArrayLen(int s, vector<int>& nums)
{
        int start =0;
        int end =0;

        int minS=0;
        int sum =0;
        for(int i=0;i<nums.size();i++)
        {
            end =i;
            sum+= nums[i];
            if(sum >= s)
            {
                while(start <= end && sum>=s)
                {
                    if(minS == 0 ||  (end -start+1)<minS )
                        minS = end -start+1;
                    start++;
                    sum -= nums[start-1];
                }
            }
        }

        return minS;
}
```

方法2： 分治的方法，时间复杂度O(nlgn)

```
int minSubArrayLen(int s, vector<int>& nums)
{
        return findMinArray(s,0,nums.size()-1,nums);
}

int findMinArray(int s,int start,int end,vector<int>& nums)
{
        if(start >end) return 0;
        int mid = start + (end -start)/2;
        int minLeft = findMinArray(s,start,mid-1,nums);
```

```
          int minRight = findMinArray(s,mid+1,end,nums);
          int minS=0;

          minS=min(minLeft,minRight);
          if(minLeft ==0)
              minS = minRight;
          if(minRight ==0)
              minS = minLeft;




          int indexleft = mid -minS+1;
          int indexright =mid+minS-1;
           //panduan zhongjian quyu
           if(minS == 0)
          {
              indexleft = start;
              indexright = end;
          }



          int si= indexleft; int r = indexleft;
          int sum =0;
          for(int i=indexleft;i<=indexright;i++)
          {
              r = i;
              sum+= nums[i];
              if(sum>= s)
              {
                   while( si<=r && sum >=s )
                  {
                      if((minS ==0 || (r-si+1) < minS))
                               minS = r-si+1;
                      si++;
                      sum -= nums[si-1];
                  }
              }
          }

          return minS;
  }
```

Implement strStr

implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

方法1： 该题就是字符串匹配问题，该方法是暴力法，时间复杂度为O(N^2)

```
int strStr(string haystack, string needle)
{

        int len_a = haystack.size();
        int len_b = needle.size();

        if (len_a < len_b) return -1;

        for (int i = 0; i <= len_a-len_b; ++i) {
            int j = 0;
            while(j < len_b && haystack[j+i] == needle[j]) {
                j++;
            }
            if (j == len_b) {
                return i;
            }
        }
        return -1;
}
```

方法2： KMP算法 时间复杂度为O(N)

```
int strStr(string haystack, string needle)
{
        if(needle =="") return 0;

        int need_len = needle.size();
        vector<int> next(need_len,-1);

        int k = -1;
        for(int i=1; i<need_len ;i++)
        {
            while(k>-1 && needle[k+1] != needle[i])
                k = next[k];
            if(needle[k+1] == needle[i])
                k++;
            next[i] =k;
        }

        int q=-1;
        int h_len = haystack.size();
        for(int i=0 ; i<h_len ; i++)
        {
            while(q>-1 && needle[q+1] != haystack[i])
```

```
                q= next[q];
            if(needle[q+1] == haystack[i])
                q++;
            if(q == need_len-1)
            {
                return i-q;
            }
        }

        return -1;
}
```

Remove Duplicates from Sorted Array

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

For example, Given input array nums = [1,1,2],

Your function should return length = 2, with the first two elements of nums being 1 and 2 respectively. It doesn't matter what you leave beyond the new length.

```
int removeDuplicates(int A[], int n) {
    if(n<2) return n;
    int id=1;
    for(int i=1;i<n;i++)
    {
        if(A[i]!=A[i-1]) A[id++]=A[i];
    }
    return id;
}
```

Remove Duplicates from Sorted Array2

Follow up for "Remove Duplicates": What if duplicates are allowed at most twice?

For example, Given sorted array nums = [1,1,1,2,2,3],

Your function should return length = 5, with the first five elements of nums being 1, 1, 2, 2 and 3. It doesn't matter what you leave beyond the new length.

思路：这题目可以拓展到at most K 的情况

```
int removeDuplicates(vector<int>& nums) {
        int length = nums.size();
        if(length<=2)
            return length;
        int id=1;
        int count=1;
        for(int i=1;i<length;i++)
        {
            if(nums[i]!=nums[i-1])
            {
                nums[id++] = nums[i];
                count=1;
            }else if(count<2)
            {
                count++;
                nums[id++] = nums[i];
            }
        }

        return id;
}
```

LongestSubstringWithoutRepeatingChar

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

For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.

思路： 所有可能的字符 存在 256大小的vector中

对于串中的每一个字符找到它上次出现的位置，然后重新计算maxlength

```
int lengthOfLongestSubstring(string s)
{
        vector<int> vect(256,-1);
        int maxlength =0,start=0;
        for(int i=0;i<s.size();i++)
        {
            start = max(start,vect[s[i]]+1);
            vect[s[i]]=i;
            maxlength = max(maxlength,i-start+1);
        }
        return maxlength;
}
```

PartionList

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example, Given 1->4->3->2->5->2 and x = 3, return 1->2->2->4->3->5.

```
ListNode *partition(ListNode *head, int x)
{
        ListNode node1(0), node2(0);
    ListNode *p1 = &node1, *p2 = &node2;
    while (head) {
        if (head->val < x)
            p1 = p1->next = head;
        else
            p2 = p2->next = head;
        head = head->next;
    }
    p2->next = NULL;
    p1->next = node2.next;
    return node1.next;
}
```

RemoveElement

Given an array and a value, remove all instances of that value in place and return the new length.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

思路：

典型的two pointer 题目 和 removeDuplicateElements一样的思路

```
int removeElement(vector<int>& nums, int val)
{
    int index=0;//用来维护实际的元素数目
    for(int i=0;i<nums.size();i++)
       if(nums[i] != val)
          nums[index++] = nums[i];
    return index;
}
```

3Sum

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

Note: Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c) The solution set must not contain duplicate triplets. For example, given array S = {-1 0 1 2 -1 -4},

A solution set is: (-1, 0, 1) (-1, -1, 2)

思路：

类似 two sum

要注意 去重复元素的边界问题

```
vector<vector<int>> threeSum(vector<int>& nums)
{

        vector<vector<int>> result;

        sort(nums.begin(),nums.end());

        for (int i = 0; i < nums.size(); i++)
        {
            int target = -nums[i];

            int front = i+1;
            int end = nums.size()-1;
            while(front<end)
            {
                int sum = nums[front]+nums[end];
                if(sum ==target)
                {
                   vector<int> temp;
                   temp.push_back(nums[i]);
                   temp.push_back(nums[front]);
                   temp.push_back(nums[end]);

                   result.push_back(temp);
                   front++;


                }else if(sum >target)
                {
                    end--;
                }else{
                    front++;
                }

                while(front<end && front>i+1 && nums[front-1] == nums[front]) front++;
                while(front<end && end<nums.size()-1 && nums[end+1] == nums[end]) end--;
            }
            while (i + 1 < nums.size() && nums[i + 1] == nums[i])
                    i++;
        }

        return result;
}
```

3SumClosest

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

For example, given array S = {-1 2 1 -4}, and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

思路： 同3Sum思路相近，这里是求sum和target最相近的值

```
int threeSumClosest(vector<int> &num, int target)
{
    vector<int> v(num.begin(), num.end());
    int n = 0;
    int ans = 0;
    int sum;

    sort(v.begin(), v.end());

    // If less then 3 elements then return their sum
    while (v.size() <= 3) {
        return accumulate(v.begin(), v.end(), 0);
    }

    n = v.size();

    ans = v[0] + v[1] + v[2];
    for (int i = 0; i < n-2; i++) {
        int j = i + 1;
        int k = n - 1;
        while (j < k) {
            sum = v[i] + v[j] + v[k];
            if (abs(target - ans) > abs(target - sum)) {
                ans = sum;
                if (ans == target) return ans;
            }
            (sum > target) ? k-- : j++;
        }
    }
    return ans;
}
```

4Sum

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target.

Note: Elements in a quadruplet (a,b,c,d) must be in non-descending order. (ie, a ≤ b ≤ c ≤ d) The solution set must not contain duplicate quadruplets. For example, given array S = {1 0 -1 0 -2 2}, and target = 0.

A solution set is: (-1, 0, 0, 1) (-2, -1, 1, 2) (-2, 0, 0, 2)

思路： two pointer的思路,同3Sum一致

```
vector<vector<int> > fourSum(vector<int> &num, int target)
{
    vector<vector<int> > ret;

    if (num.size() == 0) return ret;

    sort(num.begin(), num.end());

    for (size_t a = 0; a < num.size(); ++a)
    {
        if (a != 0 && num[a] == num[a - 1])
        {
            continue;
        }

        for (size_t b = a + 1; b < num.size(); ++b)
        {
            if (b != a + 1 && num[b] == num[b - 1])
            {
                continue;
            }

            size_t c = b + 1;
            size_t d = num.size() - 1;

            while (c < d)
            {
                const int sum = num[a] + num[b] + num[c] + num[d];

                if (sum > target)
                {
                    --d;
                }
                else if (sum < target)
                {
                    ++c;
                }
                else if (c != b + 1 && num[c] == num[c - 1])
                {
                    ++c;
                }
                else if (d != num.size() - 1 && num[d] == num[d + 1])
                {
                    --d;
                }
                else
                {
                    vector<int> result;

                    result.push_back(num[a]);
                    result.push_back(num[b]);
                    result.push_back(num[c]);
                    result.push_back(num[d]);

                    ret.push_back(result);

                    ++c;
                    --d;
                }
            }
        }
    }

    return ret;
}
```

方法二 还有hashmap的思路，这里没贴代码

ValidPalindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example, "A man, a plan, a canal: Panama" is a palindrome. "race a car" is not a palindrome.

Note: Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

```
bool isPalindrome(string s)
{

        int st=0,e=s.size()-1;

        while(st<e)
        {

            if( !isalpha(s[st]) ) {
                ++st;
                continue;
              }
              if( !isalpha(s[e]) ) {
                --e;
                continue;
              }

            if(s[st] != s[e])
                return false;
            st++;
            e--;
        }

        return true;
}


bool isalpha(char &c)
{
    if((c>='A'&&c<='Z')){
      c = c-'A'+'a';
      return true;
    }
    return (c>='a'&&c<='z') || (c>='0'&&c<='9');
}
```

MinimumWindowSubstring

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 emtpy string "".

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

```
string minWindow(string s, string T)
{

        int sLen=s.size();
        int tLen=T.size();
        if(tLen==0 || sLen<tLen) return "";

        int needFind[256]={0};//¿ÉÓÃunordered_map ´úÌæ
        int hasFind[256]={0};

        for(auto &c : T)
        {
            needFind[c]++;
        }

        int minWindowLength=INT_MAX;
        int minBegin=0;
        int minEnd=sLen-1;

        int begin =0;
        int end =0;
        int count =0;
        for(;end<sLen;end++)
        {
            if(needFind[s[end]] == 0) continue;

            hasFind[s[end]]++;
            if(hasFind[s[end]] <=needFind[s[end]] )
                count++;

            if(count == tLen)
            {
                while(begin <end)
                {
                    if(needFind[s[begin]]==0)
                    {
                        begin++;
                        continue;
                    }
                    if(hasFind[s[begin]] >needFind[s[begin]])
                    {
                        hasFind[s[begin]]--;
                        begin++;
                        continue;
                    }else
                        break;
                }

                int tmpWindowLength=end-begin+1;

                if(tmpWindowLength < minWindowLength)
                {
                    minBegin=begin;
                    minEnd=end;
                    minWindowLength=tmpWindowLength;
                }
            }

        }

        if(minWindowLength==INT_MAX)
            return "";
        return s.substr(minBegin,minWindowLength);
}
```

ContainerWithMostWater

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

Note: You may not slant the container.

典型的TwoPointer题目

```
int maxArea(vector<int> &height)
{
    if (height.empty()) return 0;
    int result =0;

    int l=0;
    int r = height.size()-1;

    int area =0;
    while(l<r)
    {
        area = (r-l)*(min(height[l], height[r]));
        result = max(result,area);

        if(height[l] < height[r])
        {
            do
            {
               l++;
            } while (l < r && height[l-1] >= height[l]);
        }else
        {
             do
             {
                 r--;
             } while (r > l && height[r+1] >= height[r]);
        }
    }

    return result;
}
```

数据结构设计

ImplementTrie

Implement a trie with insert, search, and startsWith methods.

考察字典树的设计实现（答案不唯一，但要实现字典树的功能）

```
class TrieNode {
public:
    bool isWord;
    TrieNode *child[26];
    // Initialize your data structure here.
    TrieNode():isWord(false) {
        for (auto &a : child) a = NULL;
    }
};

class Trie {
public:
    Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    void insert(string s) {
        TrieNode* p =root;

        for(auto &sc : s)
        {
            if(p->child[sc-'a'] == NULL) p->child[sc-'a'] =new TrieNode();
            p = p->child[sc-'a'];
        }
        p->isWord = true;
    }

    // Returns if the word is in the trie.
    bool search(string key) {
        TrieNode* p= find(key);
        if(p&&p->isWord)
            return true;
        return false;
    }

    // Returns if there is any word in the trie
    // that starts with the given prefix.
    bool startsWith(string prefix) {
        if(find(prefix))
            return true;
        return false;
    }
private:
    TrieNode* root;

    TrieNode* find(string key)
    {
        TrieNode* p =root;
        for(auto &sc : key)
        {
            if(p->child[sc-'a'] != NULL)
                p = p->child[sc-'a'];
            else
                return NULL;
        }
        return p;
    }
};
```

Add and Search Word

Design a data structure that supports the following two operations:

void addWord(word) bool search(word) search(word) can search a literal word or a regular expression string containing only letters a-z or .. A . means it can represent any one letter.

For example:

addWord("bad")

addWord("dad")

addWord("mad")

search("pad") -> false

search("bad") -> true

search(".ad") -> true

search("b..") -> true

思路： 利用字典树进行查找插入，注意遇到“.”的时候需要递归处理所有字母情况

```
class TrieNode {
public:
    bool isWord;
    TrieNode *child[26];
    // Initialize your data structure here.
    TrieNode():isWord(false) {
        for (auto &a : child) a = NULL;
    }
};

class WordDictionary {
public:

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

    // Adds a word into the data structure.
    void addWord(string word) {
        TrieNode* p =root;

        for(auto &sc : word)
        {
            if(p->child[sc-'a'] == NULL) p->child[sc-'a'] =new TrieNode();
            p = p->child[sc-'a'];
        }
        p->isWord = true;
    }

    // Returns if the word is in the data structure. A word could
    // contain the dot character '.' to represent any one letter.
    bool search(string word) {

        return find(word,root);
    }

private:
    TrieNode* root;

    bool find(string key,TrieNode* root)
    {
        TrieNode* p =root;
        for(int i=0; i<key.size();i++)
        {
            char sc= key[i];
            if(sc == '.')
            {

                for(auto node : p->child)
                {

                   if(node != NULL)
                   {
                       bool is = find(key.substr(i+1),node);
                       if(is)
                           return true;
                   }
                }

                return false;

            }else if(p->child[sc-'a'] != NULL)
                p = p->child[sc-'a'];
            else
                return false;
        }
        return p->isWord;
    }
};
```

Word Search II

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

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

For example, Given words = ["oath","pea","eat","rain"]

and board =

[

['o','a','a','n'],

['e','t','a','e'],

['i','h','k','r'],

['i','f','l','v']

]

Return ["eat","oath"]. Note: You may assume that all inputs are consist of lowercase letters a-z.

思路： 通过trie树来辅助搜索

若在trie树中找不到对应的节点 直接回溯

```
class TrieNode
{
public:
    TrieNode *next[26];
    bool is_word;

    // Initialize your data structure here.
    TrieNode(bool b = false)
    {
        memset(next, 0, sizeof(next));
        is_word = b;
    }
};

class Trie
{

public:
    TrieNode *root;
    Trie()
    {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    void insert(string s)
    {
        TrieNode *p = root;
        for(int i = 0; i < s.size(); ++ i)
        {
            if(p -> next[s[i] - 'a'] == NULL)
                p -> next[s[i] - 'a'] = new TrieNode();
            p = p -> next[s[i] - 'a'];
        }
        p -> is_word = true;
    }

};


class Solution {
public:

    bool isInboard(int i, int j,vector<vector<char>>& board)
    {
            if(i < 0|| i >= board.size() || j < 0 || j >= board[i].size())
                return false;
            return true;
    }

    void DFS(TrieNode *root,int si, int sj,vector<vector<char>>& board,vector<vector<bool
    {
        if(isInboard(si,sj,board))
        {
            if(!b_board[si][sj] && (nullptr != root->next[board[si][sj]-'a']) )
                {

                    b_board[si][sj] = true;
                    s.push_back(board[si][sj]);

                    if(root->next[board[si][sj]-'a']->is_word)
                    {
                        res.push_back(s);
                        root->next[board[si][sj]-'a']->is_word = false; // set to false t
                    }

                    DFS(root->next[board[si][sj]-'a'],si+1, sj, board,b_board,res,s);
                    DFS(root->next[board[si][sj]-'a'],si-1, sj, board,b_board,res,s);
                    DFS(root->next[board[si][sj]-'a'],si, sj+1, board,b_board,res,s);
                    DFS(root->next[board[si][sj]-'a'],si, sj-1, board,b_board,res,s);

                    s.pop_back();
                    b_board[si][sj] = false; //important

                }
        }
    }



    vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
```

```
             vector<string> res;
             if(board.size()==0 || board[0].size()==0 || words.size() ==0)
                  return res;
             int row = board.size();
             int col = board[0].size();
             int wordCount = words.size();

             Trie *trie = new Trie();
             for(int i=0 ;i<wordCount; i++)
                  trie->insert(words[i]);
             vector<vector<bool>> b_board(row,vector<bool>(col,false));
            //xunhuan
            string s="";
            for(int i=0; i<row ;i++)
            for(int j=0 ;j<col && wordCount>res.size() ;j++)
              {
                  DFS(trie->root,i,j,board,b_board,res,s);
              }
           return res;
      }
  };
```

LRUCache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

方法1：

用一个链表list存储缓存信息的key值

用一个map来存储每个信息key值和value值和其对应在链表中的位置

需要注意的地方：

每来一个key值，若在链表中存在，就需要将其置在链首部的位置。

set一个key和value值得时候，若capacity达到最大值，需要将list末尾的元素删掉，然后插入到链首部

```
class LRUCache {
public:
    LRUCache(int capacity) : _capacity(capacity) {}

    int get(int key) {
        auto it = cache.find(key);
        if (it == cache.end()) return -1;
        touch(it);
        return it->second.first;
    }

    void set(int key, int value) {
        auto it = cache.find(key);
        if (it != cache.end()) touch(it);
        else {
            if (cache.size() == _capacity) {
                cache.erase(used.back());
                used.pop_back();
            }
            used.push_front(key);
        }
        cache[key] = { value, used.begin() };
    }

private:
    typedef list<int> LI;
    typedef pair<int, LI::iterator> PII;
    typedef unordered_map<int, PII> HIPII;
    void touch(HIPII::iterator it) {
        int key = it->first;
        used.erase(it->second.second);
        used.push_front(key);
        it->second.second = used.begin();
    }

    HIPII cache;
    LI used;
    int _capacity;
};
```

方法2：

个人感觉这个方法更容顺手些，比较符合人的直观地想法，同上述方法思路是一致的。

```
class LRUCache{
private:
    struct item_t{
        int key, val;
        item_t(int k, int v) :key(k), val(v){}
    };
    typedef list<item_t> list_t;
    typedef unordered_map<int, list_t::iterator> map_t;

    map_t   m_map;
    list_t  m_list;
    int     m_capacity;
public:
    LRUCache(int capacity) : m_capacity(capacity) {
    }
    int get(int key) {
        map_t::iterator i = m_map.find(key);
        if (i == m_map.end()) return -1;
        m_map[key] = promote(i->second);
        return m_map[key]->val;
    }
    void set(int key, int value) {
        map_t::iterator i = m_map.find(key);
        if (i != m_map.end()){
            m_map[key] = promote(i->second);
            m_map[key]->val = value;
        }
        else {
            if (m_map.size() < m_capacity){
                m_map[key] = m_list.insert(m_list.end(), item_t(key, value));
            }
            else {
                m_map.erase(m_list.front().key);
                m_list.pop_front();
                m_map[key] = m_list.insert(m_list.end(), item_t(key, value));
            }
        }
    }
    list_t::iterator promote(list_t::iterator i){
        list_t::iterator inew = m_list.insert(m_list.end(), *i);
        m_list.erase(i);
        return inew;
    }
};
```

Implement Queue using Stacks

Implement the following operations of a queue using stacks.

push(x) -- Push element x to the back of queue.

pop() -- Removes the element from in front of queue.

peek() -- Get the front element.

empty() -- Return whether the queue is empty.

Notes: You must use only standard operations of a stack -- which means only push to top, peek/pop from top, size, and is empty operations are valid. Depending on your language, stack may not be supported natively. You may simulate a stack by using a list or deque (double-ended queue), as long as you use only standard operations of a stack.

You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue).

思路：

利用两个stack 来实现

插入元素时先放入input 返回队首元素时先看看output栈里是否有元素，若有的话，直接返回output栈顶元素，若没有的话先将input栈里元素一个个push到output栈中，再返回栈顶元素。

```
class Queue {
    stack<int> input,output;
public:
    // Push element x to the back of queue.
    void push(int x) {
        input.push(x);
    }

    // Removes the element from in front of queue.
    void pop(void) {
        peek();
        output.pop();
    }

    // Get the front element.
    int peek(void) {
        if(output.empty())
        {
            while(!input.empty())
             output.push(input.top()),input.pop();
        }
        return output.top();
    }
    // Return whether the queue is empty.
    bool empty(void) {
        return input.empty() && output.empty();
    }
};
```

Implement Stack using Queue

Implement the following operations of a stack using queues.

push(x) -- Push element x onto stack.

pop() -- Removes the element on top of the stack.

top() -- Get the top element.

empty() -- Return whether the stack is empty.

Notes: You must use only standard operations of a queue -- which means

only push to back, peek/pop from front, size, and is empty operations

are valid. Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue. You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

```
class Stack {
public:
    // Push element x onto stack.
    void push(int x) {
        int len = nums.size();
        nums.push(x);
        for (int i = 0; i != len; ++i) {
            nums.push(nums.front());
            nums.pop();
        }
    }
    // Removes the element on top of the stack.
    void pop() {
        nums.pop();
    }
    // Get the top element.
    int top() {
        return nums.front();
    }
    // Return whether the stack is empty.
    bool empty() {
        return nums.empty();
    }
private:
    std::queue<int> nums;

};
```

MinStack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

push(x) -- Push element x onto stack.

pop() -- Removes the element on top of the stack.

top() -- Get the top element.

getMin() -- Retrieve the minimum element in the stack.

此题也是编程之美上的一道题目,这里列举了多种方法（参考网上）

方法1：

```
class MinStack {
stack<int> data;
stack<int> min;

public:

    void push(int x) {

        // If empty
        if (min.empty()) {
            data.push(x);
            min.push(x);
        }

        // Not empty
        else {
            data.push(x);
            if (x <= min.top())
                min.push(x);
        }

    }

    void pop() {

        if (!min.empty()) {
            if (data.top() == min.top())
                min.pop();
            data.pop();
        }
    }

    int top() {
        return data.top();
    }

    int getMin() {
        return min.top();
    }
}
```

方法2：

```
class MinStack {
    int min=Integer.MAX_VALUE;
    Stack<Integer> stack = new Stack<Integer>();
    public void push(int x) {
       // only push the old minimum value when the current
       // minimum value changes after pushing the new value x
        if(x <= min){
            stack.push(min);
            min=x;
        }
        stack.push(x);
    }

    public void pop() {
       // if pop operation could result in the changing of the current minimum value,
       // pop twice and change the current minimum value to the last minimum value.
        if(stack.peek()==min) {
            stack.pop();
            min=stack.peek();
            stack.pop();
        }else{
            stack.pop();
        }
        if(stack.empty()){
            min=Integer.MAX_VALUE;
        }
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return min;
    }
}
```

方法3：

```
public class MinStack {
    long min;
    Stack<Long> stack;

    public MinStack(){
        stack=new Stack<>();
    }

    public void push(int x) {
        if (stack.isEmpty()){
            stack.push(0L);
            min=x;
        }else{
            stack.push(x-min);//Could be negative if min value needs to change
            if (x<min) min=x;
        }
    }

    public void pop() {
        if (stack.isEmpty()) return;

        long pop=stack.pop();

        if (pop<0)  min=min-pop;//If negative, increase the min value

    }

    public int top() {
        long top=stack.peek();
        if (top>0){
            return (int)(top+min);
        }else{
           return (int)(min);
        }
    }

    public int getMin() {
        return (int)min;
    }
}
```

链表

AddTwoNumbers

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4) Output: 7 -> 0 -> 8

```
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
     if(l1 == NULL) return l2;
     if(l2 == NULL) return l1;
     ListNode *preHead = new ListNode(0);
     ListNode *p = preHead;
     int extra=0;
     while(l1 || l2 || extra )
    {
        int sum = (l1?l1->val:0)+(l2?l2->val:0)+extra;
        extra = sum /10;
        p->next = new ListNode(sum%10);
        p = p->next;
        l1 = l1?l1->next:l1;
        l2 = l2?l2->next:l2;
    }

    return preHead->next;
}
```

CopyListwithRandomPointer

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

Return a deep copy of the list.

```
RandomListNode *copyRandomList(RandomListNode *head)
{
        if(head == NULL) return NULL;
    RandomListNode *p = head;
    do {
        RandomListNode *q = p->next;
        p->next = new RandomListNode(p->label);
        p->next->next = q;
        p = q;
    } while(p != NULL);
    p = head;
    do {
        p->next->random = (p->random == NULL) ? NULL : p->random->next;
        p = p->next->next;
    } while(p != NULL);
    p = head;
    RandomListNode *r = head->next;
    for(RandomListNode *q = r;;) {
        p->next = q->next;
        p = p->next;
        if(p == NULL) break;
        q->next = p->next;
        q = q->next;
    }
    return r;
}
```

```
RandomListNode *copyRandomList(RandomListNode *head) {
        map<RandomListNode*, RandomListNode*> relation;

        RandomListNode* copiedHead= NULL;
        RandomListNode* copiedPtr=NULL;
        RandomListNode* ptr = head;

        while(ptr!=NULL)
        {
            RandomListNode* new_node = new RandomListNode(ptr->label);
            relation.insert(pair<RandomListNode*, RandomListNode*>(ptr, new_node));
            if(copiedHead==NULL)
            {
                copiedHead = new_node;
                copiedPtr = new_node;
            }
            else
            {
                copiedPtr->next = new_node;
                copiedPtr=copiedPtr->next;
            }
            copiedPtr->random = ptr->random;
            ptr=ptr->next;
        }

        ptr=head;
        copiedPtr = copiedHead;
        while(ptr!=NULL)
        {
            if(ptr->random!=NULL)
            {
                copiedPtr->random = relation[ptr->random];
            }
            copiedPtr=copiedPtr->next;
            ptr=ptr->next;
        }
        return copiedHead;

}
```

Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example, Given 1->1->2, return 1->2. Given 1->1->2->3->3, return 1->2->3.

方法1：有pre指针的

```
ListNode *deleteDuplicates(ListNode *head)
{
    if(head==NULL || head->next==NULL) return head;
    ListNode* c = head->next;
    ListNode* p = head;

    while(c!=NULL) // We assume that there is no loop in the list.
    {
        ListNode* n = c->next;

        if(p->val == c->val)
        {
            p->next = n;
            delete c;
            c = n;
        }
        else
        {
            p = c;
            c = n;
        }
    }
    return head;
}
```

方法2：双重指针

```
ListNode* deleteDuplicates(ListNode* head)
{
        if(head ==NULL || head->next ==NULL)
            return head;
        ListNode** pre =&head;

        while(*pre && (*pre)->next)
        {
            int pvalue =(*pre)->val;

            if((*pre)->next->val == pvalue)
            {
                ListNode *temp = *pre;
                *pre = (*pre)->next;
                delete temp;
            }else{
                pre =&(*pre)->next;
            }
        }

        return head;
}
```

方法3: 其实也可以不用双重指针 并没有考虑到delete的

```
ListNode *deleteDuplicates(ListNode *head)
{
        ListNode* cur = head;
        while (cur) {
            while (cur->next && cur->val == cur->next->val)
                cur->next = cur->next->next;
            cur = cur->next;
        }
        return head;
 }
```

RemoveDuplicatesFromSortedList2

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example, Given 1->2->3->3->4->4->5, return 1->2->5. Given 1->1->1->2->3, return 2->3.

方法1：递归方法

```
ListNode* deleteDuplicates(ListNode* head) {
        if (!head) return 0;
        if (!head->next) return head;

        int val = head->val;
        ListNode* p = head->next;

        if (p->val != val) {
            head->next = deleteDuplicates(p);
            return head;
        } else {
            while (p && p->val == val) p = p->next;
            return deleteDuplicates(p);
        }
    }
```

方法2：双重指针

```
ListNode *deleteDuplicates(ListNode *head) {
        ListNode **runner = &head;

        if(!head || !head->next)return head;

        while(*runner)
        {
            if((*runner)->next && (*runner)->next->val == (*runner)->val)
            {
                ListNode *temp = *runner;
                while(temp && (*runner)->val == temp->val)
                    temp = temp->next;

                *runner = temp;
            }
            else
                runner = &((*runner)->next);
        }

        return head;
}
```

方法3：类似的方法

```
ListNode* deleteDuplicates(ListNode* head) {
        ListNode** curNext = &head;
        ListNode* cur =head;

        if(head == NULL || head->next ==NULL)
            return head;
        while(cur!=NULL)
        {
            ListNode* temp = cur;
            while(NULL != cur->next && cur->next->val == cur->val)
                cur = cur->next;
            if( cur == temp)
            {
                *curNext = temp;
                curNext = &(*curNext)->next;
            }
            cur = cur->next;
        }
        *curNext =NULL;
        return head;
}
```

RemoveNthNodeFromEndofList

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

For example,

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

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

Note: Given n will always be valid. Try to do this in one pass.

方法1：普通方法， 保存pre指针

```
ListNode *removeNthFromEnd(ListNode *head, int n)
{
         if (head == NULL)
            return NULL;

        ListNode *pPre = NULL;
        ListNode *p = head;
        ListNode *q = head;
        for(int i = 0; i < n - 1; i++)  //先让q指针先走n步
            q = q->next;

        while(q->next)
        {
            pPre = p;
            p = p->next;
            q = q->next;
        }

        if (pPre == NULL)
        {
            head = p->next;
            delete p;
        }
        else
        {
            pPre->next = p->next;
            delete p;
        }

        return head;
}
```

方法2：Linus Torvalds 的双重指针方法

```
ListNode *removeNthFromEnd(ListNode *head, int n)
{
       ListNode* fast = head;
```

```
        for(int i=0;i<n;i++)
            fast = fast->next;

        ListNode** pre =&head;

        while(fast)
        {
            fast = fast->next;
            pre = &(*pre)->next; //pre 总是保存当前节点的next指针的地址
        }

        ListNode* current = *pre;
        *pre= current->next;
        delete current;

        return head;
}
```

Delete Node in a Linked List

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

Supposed the linked list is 1 -> 2 -> 3 -> 4 and you are given the third node with value 3, the linked list should become 1 -> 2 -> 4 after calling your function.

思路：修改value值，貌似是编程之美上的题目

```
void deleteNode(ListNode* node)
{
        if(node->next)
        {
            node->val = node->next->val;
            node->next = node->next->next;
        }else
          node =NULL;

}
```

RemoveLinkedListElements

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

Example Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, val = 6

Return: 1 --> 2 --> 3 --> 4 --> 5

思路：利用双重指针

```
ListNode* removeElements(ListNode* head, int val)
{
        ListNode** cur = &head;
        while(*cur)
        {
            ListNode* temp = *cur;
            if(temp->val == val)
            {
                *cur =(*cur)->next;
                delete temp;
            }else
                cur = &((*cur)->next);
        }
        return head;
}
```

思路：利用辅助头结点

```
ListNode* removeElements(ListNode* head, int val)
{
        ListNode *pseudo_head = new ListNode(0);
        pseudo_head->next = head;
        ListNode *cur = pseudo_head;
        while(cur)
        {
            if(cur->next && cur->next->val == val)
            {
                ListNode *temp = cur->next;
                cur->next = cur->next->next;
                delete temp;
            }else
                cur = cur->next;
        }

        return pseudo_head->next;
}
```

Palindrome Linked List

Given a singly linked list, determine if it is a palindrome.

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

思路：

是否存在O(1)空间的解法还存在争议

```
bool isPalindrome(ListNode* head) {
        if(head == NULL || head->next == NULL)
            return true;
        ListNode* slow=head;
        ListNode* fast=head;
         while(fast->next!=NULL&&fast->next->next!=NULL){
            slow=slow->next;
            fast=fast->next->next;
        }
        slow->next = reverseList(slow->next);
         slow=slow->next;
         fast = head;
        while(slow!=NULL){
            if(fast->val != slow->val)
                return false;
            slow = slow->next;
            fast = fast->next;
        }

        return true;
}

ListNode* reverseList(ListNode* head) {
        ListNode* pre =NULL;
        ListNode* next =NULL;
        while(head)
        {
            next = head ->next;
            head->next = pre;
            pre = head;
            head = next;
        }

        return pre;
}
```

Swap Nodes in Pairs

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

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

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

方法1 ：C++版本..

```
ListNode *swapPairs(ListNode *head)
{
        if(head==NULL) return head;
        //check single-node list;
        if(head->next==NULL) return head;

        ListNode *prev;
        ListNode *p1;
        ListNode *p2;
        p1=head;
        prev=head;
        do{
            p2=p1->next;
            p1->next=p2->next;
            p2->next=p1;
            if(prev!=head)
                prev->next=p2;
            else
                head=p2;
            prev=p1;
            p1=p1->next;
        }while(p1!=NULL && p1->next!=NULL);

        return head;
}
```

方法2：Java版本的方法

```
public ListNode swapPairs(ListNode head)
{
    ListNode start = new ListNode(0); //make head no longer a special case
    start.next = head;

    for(ListNode cur = start; cur.next != null && cur.next.next != null; cur = cur.next.n
      cur.next = swap(cur.next, cur.next.next);
    }
    return start.next;
}

 private Listnode swap(ListNode next1, ListNode next2)
```

```
   {
      next1.next = next2.next;
      next2.next = next1;
      return next2;
   }
```

ReverseLinkedList

Reverse a singly linked list.

A linked list can be reversed either iteratively or recursively. Could you implement both?

方法1：递归

```
ListNode* reverseList(ListNode* head)
{
        if(head == NULL || head->next ==NULL)
            return head;
        ListNode* newhead = reverseList(head->next);
        head->next->next =head;
        head->next =NULL;
       return newhead;
}
```

方法2,3：迭代的方法

```
ListNode *reverseList(ListNode *head)
{
    if (head == NULL || head->next == NULL)
        return head;

    ListNode *pCurr = head;
    ListNode *pPrev = NULL;
    ListNode *pNext = NULL;

    while (pCurr != NULL)
    {
        pNext = pCurr->next;  //save next node
        pCurr->next = pPrev;
        if (pNext == NULL)
            head = pCurr;
        pPrev = pCurr;
        pCurr = pNext;
    }

    return head;
}
```

```
ListNode* reverseList(ListNode* head)
{
     ListNode* pre =NULL;
      while(head)
      {
            ListNode* next = head->next;
            head->next = pre;
            pre = head;
            head = next;
        }
        return pre;
    }
```

ReverseLinkList2

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example: Given 1->2->3->4->5->NULL, m = 2 and n = 4,

return 1->4->3->2->5->NULL.

Note: Given m, n satisfy the following condition: 1 ≤ m ≤ n ≤ length of list.

思路： 保存第n,m个节点,第m节点的前一个节点，第n个节点的后一个节点

```
ListNode* reverseBetween(ListNode* head, int m, int n) {
        if(head == NULL || head->next ==NULL || m>=n)
            return head;
        ListNode* mPre=NULL;
        ListNode* nNext=NULL;

         ListNode* curr=head;
         ListNode* next=NULL;
         ListNode* pre=NULL;
         ListNode *mTh= NULL;
         ListNode *nTh= NULL;
        for(int i=1;i<=n;i++)
        {
            next =curr->next;

            if(i==m)
            {
                mTh = curr;
                mPre = pre;
            }

            if(i == n)
            {
                   nTh = curr;
                   nNext = next;
                   curr->next = pre;
             }
            if(i>m && i<n)
            {
                curr->next = pre;//
            }

            pre = curr;
            curr =next;
        }

        if(m==1)
        {
            mTh->next = nNext;
            head = nTh;
        }else{
            mPre->next = nTh;
            mTh->next = nNext;
        }

        return head;
}
```

ReverseNodesInK_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

```
ListNode *reverseKGroup(ListNode *head, int k)
{
        if(head==NULL||k==1) return head;
        int num=0;
        ListNode *preheader = new ListNode(-1);
        preheader->next = head;
        ListNode *cur = preheader, *nex, *pre = preheader;
        while(cur = cur->next)
            num++;
        while(num>=k) {
            cur = pre->next;
            nex = cur->next;
            for(int i=1;i<k;++i) {
                cur->next=nex->next;
                nex->next=pre->next;
                pre->next=nex;
                nex=cur->next;
            }
            pre = cur;
            num-=k;
        }
        return preheader->next;
}
```

RotateList

Given a list, rotate the list to the right by k places, where k is non-negative.

For example: Given 1->2->3->4->5->NULL and k = 2,

return 4->5->1->2->3->NULL.

方法1： make a circle

```
ListNode* rotateRight(ListNode* head, int k) {
        if(head == NULL || head->next == NULL||k==0) return head;

        ListNode* node = head;
        int size =1;
        while(node->next != NULL)
        {
            size++;
            node = node->next;
        }

        node->next = head; //make circle

        k = k%size;

        int num = size -k;

        while(num--)
        {
            node = node ->next;
        }

        ListNode* first = node->next;
        node->next=NULL;

        return first;
}
```

同样的思路

```


    ListNode *rotateRight(ListNode *head, int k) {
        if(k==0 || head ==NULL || head->next ==NULL)
            return head;

        ListNode* ptr =head;
        ListNode* tail =head;
        int length =0;

        while(ptr!=NULL)
        {
            tail = ptr;
            ptr =ptr->next;
            length++;
        }

        k= k%length;

        ptr=head;
        for (int i = 0; i < length - k - 1; i++) {
            ptr = ptr-> next;
        }

        tail->next = head;
        head = ptr->next;
        ptr->next = NULL;

        return head;
    }
```

ReorderList

Given a singly linked list L: L0→L1→…→Ln-1→Ln, reorder it to:L0→Ln→L1→Ln-1→L2→Ln-2→…

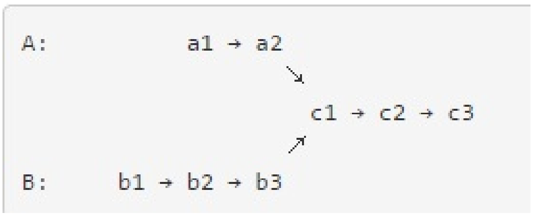
You must do this in-place without altering the nodes' values.

For example, Given {1,2,3,4}, reorder it to {1,4,2,3}.

```
void reorderList(ListNode* head)
{

        if (!head || !head->next) return;

        ListNode *p1 = head, *p2 = head->next;
        while (p2 && p2->next) {
            p1 = p1->next;
            p2 = p2->next->next;
        }

        // cut from the middle and reverse the second half: O(n)
        ListNode *head2 = p1->next;
        p1->next = NULL;

        p2 = head2->next;
        head2->next = NULL;
        while (p2) {
            p1 = p2->next;
            p2->next = head2;
            head2 = p2;
            p2 = p1;
        }

        // merge two lists: O(n)
        for (p1 = head, p2 = head2; p1; ) {
            auto t = p1->next;
            p1 = p1->next = p2;
            p2 = t;
        }

 }
```

IntersectionofTwoLinkedLists

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

For example, the following two linked lists: 

begin to intersect at node c1.

Notes:

If the two linked lists have no intersection at all, return null. The linked lists must retain their original structure after the function returns. You may assume there are no cycles anywhere in the entire linked structure. Your code should preferably run in O(n) time and use only O(1) memory.

方法1 先计算length

```
int getLen(ListNode *head) {
        int len = 0;
        while(head) { len++; head = head->next; }
        return len;
    }

ListNode *getIntersectionNode(ListNode *headA, ListNode *headB)
{
        if (!headA || !headB) return NULL;
        ListNode *p1 = headA, *p2 = headB;
        int lenA = getLen(headA), lenB = getLen(headB);

        if (lenA <= lenB) {
            for (int i = 0; i < lenB - lenA; ++i)
                p2 = p2->next;
        }
        else {
            for (int i=0; i < lenA - lenB; ++i)
                p1 = p1->next;
        }

        while (p1 && p2 && p1 != p2) p1 = p1->next, p2 = p2->next;
        if (!p1 || !p2) return NULL;
        return p1;
}
```

方法2: 其实 是利用了同样的原理 但是代码更加简练

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB)
```

```
{
         ListNode *ptrA = headA, *ptrB = headB;
        while (ptrA != ptrB) {
            ptrA = ptrA ? ptrA->next : headB;
            ptrB = ptrB ? ptrB->next : headA;
        }
        return ptrA;
}
```

LinkedListCircle

Given a linked list, determine if it has a cycle in it.

Follow up: Can you solve it without using extra space?

思路： 一个指针走两步 一个指针走一步

```
bool hasCycle(ListNode *head)
{
        ListNode* pfast = head;
        ListNode* pslow = head;
        do{
            if(pfast!=NULL)
                pfast=pfast->next;
            if(pfast!=NULL)
                pfast=pfast->next;
            if(pfast==NULL)
                return false;
            pslow = pslow->next;
        }while(pfast != pslow);
        return true;
}
```

题目拓展：

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

思路：画图即懂

```
ListNode *detectCycle(ListNode *head)
{
        ListNode *fast = head;
        ListNode *slow = head;
        while(fast != NULL && fast->next != NULL){
            fast = fast->next->next;
            slow = slow->next;
            if(fast == slow){
                fast = head;
                while(fast != slow){
                    fast = fast->next;
                    slow = slow->next;
                }
                return slow;
            }
        }
        return NULL;
}
```

MergeTwoSortedList

Merge two sorted linked lists and return it as a new list.

The new list should be made by splicing together the nodes of the first two lists.

```
ListNode *mergeTwoLists(ListNode *l1, ListNode *l2)
{
        ListNode *helper=new ListNode(0);
        ListNode *head=helper;
        while(l1 && l2)
        {
             if(l1->val<l2->val)
                helper->next=l1,l1=l1->next;
             else
                helper->next=l2,l2=l2->next;
             helper=helper->next;
        }
        if(l1) helper->next=l1;
        if(l2) helper->next=l2;
        return head->next;
}
```

动态规划

EditDistance

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

You have the following 3 operations permitted on a word:

a) Insert a character

b) Delete a character

c) Replace a character

思路：老题目了

方法1 O(mn)的空间

distance[i][j]表示word1[0..i]与word2[0..j]大的最小距离

我们可以得到动规方程：

当word1[i-1] == word2[j-1] distance[i][j] = distance[i-1][j-1];

当word1[i-1] ！= word2[j-1] distance[i][j] = 1+ min(distance[i-1][j-1], distance[i-1][j], distance[i][j-1])

```
int minDistance(string word1, string word2)
{
       vector<vector<int>> distance(word1.length()+1, vector<int>(word2.length()+1, 0));

       for(int i=0;i<=word1.length();i++)
            distance[i][0]=i;
       for(int i=0;i<=word2.length();i++)
            distance[0][i]=i;

     // when insert a char to word1 it means we are trying to match word1 at i-1 to word2
     // when delete a char from word1 it equals to insert a char to word2, which
     // means we are trying to match word1 at i to word2 at j-1
     // when replace a char to word1, then we add one step to previous i and j
       for(int i=1; i< distance.size(); i++)
       for(int j=1; j<distance[0].size(); j++){
            if(word1[i-1] == word2[j-1])
                distance[i][j] = distance[i-1][j-1];
            else
                distance[i][j] = 1+ min(distance[i-1][j-1], min(distance[i-1][j], distanc
        }

        return distance[word1.length()][word2.length()];
}
```

方法2： O(n)的空间复杂度

f[i][j] 只依赖 f[i-1][j-1], f[i-1][j] 和 f[i][j-1]，我们可以把算法的空间复杂度降到O(n)

```
int minDistance(string word1, string word2)
{
        int l1 = word1.size();
        int l2 = word2.size();

        vector<int> f(l2+1, 0);
        for (int j = 1; j <= l2; ++j)
            f[j] = j;

        for (int i = 1; i <= l1; ++i)
        {
            int prev = i;
            for (int j = 1; j <= l2; ++j)
            {
                int cur;
                if (word1[i-1] == word2[j-1]) {
                    cur = f[j-1];
                } else {
                    cur = min(min(f[j-1], prev), f[j]) + 1;
                }

                f[j-1] = prev;
                prev = cur;
            }
            f[l2] = prev;
        }
        return f[l2];
}
```

ScrambleString

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":

```
    /    \
   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.

方法1：递归法

```
bool isScramble(string s1, string s2) {
        if(s1==s2)
            return true;

        int len = s1.length();
        int count[26] = {0};
        for(int i=0; i<len; i++)
        {
            count[s1[i]-'a']++;
            count[s2[i]-'a']--;
        }

        for(int i=0; i<26; i++)
        {
            if(count[i]!=0)
                return false;
        }

        for(int i=1; i<=len-1; i++)
        {
            if( isScramble(s1.substr(0,i), s2.substr(0,i)) && isScramble(s1.substr(i), s2
                return true;
            if( isScramble(s1.substr(0,i), s2.substr(len-i)) && isScramble(s1.substr(i),
                return true;
        }
        return false;
}
```

方法2：动态规划法

isS[i][j][l] 表示 s2.substr(j,l) 是否是 s1.substr(i,l) 的一个Scramble string.

```
bool isScramble(string s1, string s2) {
        int sSize = s1.size(), len, i, j, k;
        if(0==sSize) return true;
        if(1==sSize) return s1==s2;
        bool isS[sSize+1][sSize][sSize];

        for(i=0; i<sSize; ++i)
            for(j=0; j<sSize; ++j)
                isS[1][i][j] = s1[i] == s2[j];

        for(len=2; len <=sSize; ++len)
            for(i=0; i<=sSize-len; ++i)
                for(j=0; j<=sSize-len; ++j)
                {
                    isS[len][i][j] = false;
                        for(k=1; k<len && !isS[len][i][j]; ++k)
                        {
                            isS[len][i][j] = isS[len][i][j] || (isS[k][i][j] && isS[len-k
                            isS[len][i][j] = isS[len][i][j] || (isS[k][i+len-k][j] && isS
                        }
                }
        return isS[sSize][0][0];

    }
```

UniquePath

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

方法1： 动态规划题目，很简单，空间复杂度O(m*n)

```
int uniquePaths(int m, int n) {
        int f[m][n];
        memset(f, 0, sizeof(int) * m * n);
        for (int i = 0; i < m; i++) {
            f[i][0] = 1;
        }
        for (int j = 0; j < n; j++) {
            f[0][j] = 1;
        }
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                f[i][j] = f[i - 1][j] + f[i][j - 1];
            }
        }
        return f[m - 1][n - 1];
}
```

方法2：将空间复杂度降到O(n)

```
int uniquePaths(int m, int n)
{
        if(m ==1 || n==1) return 1;

        vector<int> sum(n,1);

        for(int i=1; i<m ;i++)
        {
            for(int j=1 ; j<n; j++)
                sum[j] = sum[j-1]+sum[j];
        }

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

方法3：排列组合的方法,该题目其实就是计算$$C_{m+n-2}^{m-1}$$

公式：$$C_{m}^{n} = m!/(m-n)!n!$$

```
int uniquePaths(int m, int n)
{
        int A = m + n - 2, B = std::min(m - 1, n - 1);
        long long result = 1;
        for (int i = 1; i <= B; ++i)
            result = result * A-- / i;
        return static_cast<int>(result);
}
```

UniquePath2

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example, There is one obstacle in the middle of a 3x3 grid as illustrated below.

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

The total number of unique paths is 2.

Note: m and n will be at most 100.

思路：

同样的动态规划思路 要多考虑阻碍点

```
int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
    int m = obstacleGrid.size();
    int n = obstacleGrid[0].size();
    vector<int> cur(m, 0);
    for (int i = 0; i < m; i++) {
        if (obstacleGrid[i][0] != 1)
            cur[i] = 1;
        else break;
    }
    for (int j = 1; j < n; j++) {
        bool flag = false;
        if (obstacleGrid[0][j] == 1)
            cur[0] = 0;
        else flag = true;
        for (int i = 1; i < m; i++) {
            if (obstacleGrid[i][j] != 1) {
                cur[i] += cur[i - 1];
                if (cur[i]) flag = true;
            }
            else cur[i] = 0;
        }
        if (!flag) return 0;
    }
    return cur[m - 1];
}
```

RegularExpressionMatching

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

'.' Matches any single character. '*' Matches zero or more of the preceding element.

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

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", "c*a*b") → true
```

方法1：递归方法

```
bool isMatch(string s, string p)
{
        if ( p.empty() ) return s.empty();
        // p's next is not *
        if ( p[1]!='*' )
        {
            return ( s[0]==p[0] || (p[0]=='.' && !s.empty()) ) && Solution::isMatch(s.sub
        }
        // p's next is * and curr s match curr p
        int i = 0;
        for ( ; s[i]==p[0] || (p[0]=='.' && i<s.size()); ++i)
        {
            if ( Solution::isMatch(s.substr(i), p.substr(2)) ) return true;
        }
        // p's next is * but curr s not match curr p
        return Solution::isMatch(s.substr(i), p.substr(2));
}
```

方法2：动态规划

```
bool isMatch(string s, string p) {

        int m = s.size(), n = p.size();
        vector<vector<bool>> f(m + 1, vector<bool>(n + 1, false));

        f[0][0] = true;
        for (int i = 1; i <= m; i++)
            f[i][0] = false;
        // p[0.., j - 3, j - 2, j - 1] matches empty iff p[j - 1] is '*' and p[0..j - 3]
        for (int j = 1; j <= n; j++)
            f[0][j] = j > 1 && '*' == p[j - 1] && f[0][j - 2];

        for (int i = 1; i <= m; i++)
            for (int j = 1; j <= n; j++)
                if (p[j - 1] != '*')
                    f[i][j] = f[i - 1][j - 1] && (s[i - 1] == p[j - 1] || '.' == p[j - 1]
                else
                    // p[0] cannot be '*' so no need to check "j > 1" here
                    f[i][j] = f[i][j - 2] || (s[i - 1] == p[j - 2] || '.' == p[j - 2]) &&

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

此题详细分析可参考： Leetcode:RegularExpressionMatching

WildcardMatching

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") return false
isMatch("aa","aa") return true
isMatch("aaa","aa") return false
isMatch("aa", "*") return true
isMatch("aa", "a*") return true
isMatch("ab", "?*") return true
isMatch("aab", "c*a*b") return false
```

方法1：迭代法

```
bool isMatch(string s, string p)
{
        int s_size = s.size();
        int p_size =p.size();
        int s_i=0,p_i=0;

        int s_start=0;
        int p_start =-1;
        while(s_i<s_size)
        {
            if(s[s_i] == p[p_i] || ( p[p_i]=='?' ))
            {
                s_i++;
                p_i++;
                continue;
            }

            if(p[p_i] =='*')
            {
                p_start = p_i++;
                s_start = s_i;
                continue;
            }

            //current characters didn't match, last pattern pointer was *, current patter
            //only advancing pattern pointer
            if(p_start !=-1)
            {
                p_i =p_start+1;
                s_i = ++s_start;
                continue;
            }

            return false;
        }

        while(p[p_i] == '*') p_i++;

        return p_i>=p_size;
}
```

方法2： 递归法.

```
bool isMatch(const char *s, const char *p)
{
    if (s == NULL || p == NULL) return false;
    if (*p == '\0') return *s == '\0';

    if (*p == '*')
    {
        while (*p == '*') ++p;

        while (*s != '\0')
        {
            if (isMatch(s, p)) return true;
            ++s;
        }

        return isMatch(s, p);
    }
    else if ((*s != '\0' && *p == '?') || *p == *s)
    {
        return isMatch(s + 1, p + 1);
    }

    return false;
}
```

MaximumSubarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [−2,1,−3,4,−1,2,1,−5,4], the contiguous subarray [4,−1,2,1] has the largest sum = 6.

click to show more practice.

More practice: If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

编程之美的上的题目

方法1:

```
int maxSubArray(int A[], int n) {

        if(n<=0)
            return 0;

        int maxsum =A[0];
        int sum=0;
        for(int i=0; i<n ; i++)
        {
            sum+=A[i];
            maxsum = sum>maxsum ? sum :maxsum;
            sum = sum>0 ? sum: 0;
        }
        return maxsum;

}
```

更容易解释理解的 DP方法

```
public int maxSubArray(int[] A) {
        int n = A.length;
        int[] dp = new int[n];//dp[i] means the maximum subarray ending with A[i];
        dp[0] = A[0];
        int max = dp[0];

        for(int i = 1; i < n; i++){
            dp[i] = A[i] + (dp[i - 1] > 0 ? dp[i - 1] : 0);
            max = Math.max(max, dp[i]);
        }

        return max;
}
```

方法2：分治方法

```
int maxSubArray(vector<int>& nums) {
        int result =0;
        result = getMaxSub(nums,0,nums.size()-1);
        return result;
    }

int getMaxSub(vector<int>& nums,int left,int right)
{
        if(left >= right ) return nums[left];

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

        int leftans = getMaxSub(nums,left,mid-1);
        int rightans = getMaxSub(nums,mid+1,right);

        int leftmax = nums[mid];
        int rightmax =nums[mid];

        int temp =0;
        for(int i =mid ;i>=left;i--)
        {
            temp = temp+nums[i];
            leftmax = max(leftmax,temp);
        }
        temp =0;
        for(int i=mid;i<=right;i++)
        {
             temp = temp+nums[i];
            rightmax = max(rightmax,temp);
        }
        int midmax = max(max(leftmax,rightmax),leftmax+rightmax-nums[mid]);

        return max(max(leftans,rightans),midmax);
}
```

MinimumPathSum

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

典型的动态规划题目,在微软面试中我被问到过这个题目

方法1：O(n2)的空间复杂度

```
int minPathSum(vector<vector<int>>& grid)
    {
        if(grid.size()==0 || grid[0].size()==0)
            return 0;
        int m =grid.size();
        int n = grid[0].size();

        int minPath[m][n];
        minPath[0][0] = grid[0][0];

        for(int i=1;i<m;i++)
            minPath[i][0] = minPath[i-1][0]+grid[i][0];
        for(int i=1;i<n;i++)
            minPath[0][i] = minPath[0][i-1]+grid[0][i];

        for(int i=1 ;i<m;i++)
        for(int j=1; j<n;j++)
        {
            minPath[i][j] = min(minPath[i-1][j],minPath[i][j-1])+grid[i][j];
        }

        return minPath[m-1][n-1];
}
```

方法2：

动态规划开始计算新的一行的minPath的时候 只需要上一行的数据 所以只需要O(n)的空间复杂度

```
int minPathSum(vector<vector<int>>& grid)
{
    if(grid.size()==0 || grid[0].size()==0)
            return 0;
        int m =grid.size();
        int n = grid[0].size();

        int minPath[n];
        minPath[0] =grid[0][0];

        for(int i=1;i<n;i++)
            minPath[i] = minPath[i-1]+grid[0][i];
```

```
         for(int i=1;i<m;i++)
         {
             minPath[0] += grid[i][0];
             for(int j=1;j<n;j++)
             {
                 //minPath[j]表示对应的minPath[i-1][j]  minPath[j-1]表示对应的 minPath[i][j-
                 minPath[j] = min(minPath[j],minPath[j-1])+grid[i][j];
             }
         }
        return minPath[n-1];
}
```

ClimbingStairs

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

思路： 说是动规，其实就是求一个个裴波那契数f(n)

可用递归 或者迭代的方法

```
int climbStairs(int n)
{
        if(n==1) return 1;
        if(n==2) return 2;
        int pre1 =1;
        int pre2 =2;
        int result =0;

        for(int i=3; i<=n;i++)
        {
            result = pre1+pre2;
            pre1= pre2;
            pre2 =result;
        }

        return result;
}
```

WordBreak

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

For example, given s = "leetcode", dict = ["leet", "code"].

Return true because "leetcode" can be segmented as "leet code".

```
方法1：
动态规划方法
dp[i]表示s(0..i)是否能被分成多个dict中的串
```

```
bool wordBreak(string s, unordered_set<string> &dict)
{
        if(dict.size()==0) return false;

        vector<bool> dp(s.size()+1,false);
        dp[0]=true;

        for(int i=1;i<=s.size();i++)
        {
            for(int j=i-1;j>=0;j--)
            {
                if(dp[j])
                {
                    string word = s.substr(j,i-j);
                    if(dict.find(word)!= dict.end())
                    {
                        dp[i]=true;
                        break; //next i
                    }
                }
            }
        }

        return dp[s.size()];
}
```

```
动态规划方法2
优化版 只针对 字典里面的字符串的大小 进行遍历
```

```
bool wordBreak(string s, unordered_set<string> &dict) {
        int n = s.size();
        if(!n)
            return true;
        vector<bool> ret(n+1, false);
        ret[n] = true;
        unordered_set<int> dsize;
        for(auto d : dict)
            dsize.insert(d.size());
        ret[n] = true;
        for(int i=n-1; i>=0; i--){
            for(auto k : dsize){
                if(i+k<=n && dict.find(s.substr(i,k))!=dict.end() && ret[i+k]){
                    ret[i] = true;
                    break;
                }
            }
        }
        return ret[0];
}
```

```
BFS(广度优先搜索)的方法
```

```
bool wordBreak(string s, unordered_set<string> &dict) {
    // BFS
    queue<int> BFS;
    unordered_set<int> visited;

    BFS.push(0);
    while(BFS.size() > 0)
    {
        int start = BFS.front();
        BFS.pop();
        if(visited.find(start) == visited.end())
        {
            visited.insert(start);
            for(int j=start; j<s.size(); j++)
            {
                string word(s, start, j-start+1);
                if(dict.find(word) != dict.end())
                {
                    BFS.push(j+1);
                    if(j+1 == s.size())
                        return true;
                }
            }
        }
    }

    return false;
}
```

DecodeWays

A message containing letters from A-Z is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given an encoded message containing digits, determine the total number of ways to decode it.

For example, Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).

The number of ways decoding "12" is 2.

动态规划,空间复杂度O(N)

```
int numDecodings(string s)
{
        if(s.length() ==0 || s=="")
            return 0;
        vector<int> f;
        f.resize(s.length());

        for(int i=0;i<s.length();i++)
        {
            f[i]=0;
            if(i>=1)
            {
                 if ('1' <= s[i] && s[i] <= '9')
                    f[i] += f[i-1];

                 if(('1' == s[i-1] && s[i] <= '9') || ('2' == s[i-1] && s[i] <= '6'))
                    if(i==1) f[i] += 1;
                    else     f[i] +=f[i-2];

            }else //i==0
            {
                if ('1' <= s[i] && s[i] <= '9')
                    f[i] = 1;
            }

        }

        return f[s.length()-1];
}
```

空间复杂度O(1)

```
int numDecodings(string s)
```

```
{
    if (!s.size() || s.front() == '0') return 0;
    // r2: decode ways of s[i-2] , r1: decode ways of s[i-1]
    int r1 = 1, r2 = 1;

    for (int i = 1; i < s.size(); i++) {
        // zero voids ways of the last because zero cannot be used separately
        if (s[i] == '0') r1 = 0;

        // possible two-digit letter, so new r1 is sum of both while new r2 is the old r1
        if (s[i - 1] == '1' || s[i - 1] == '2' && s[i] <= '6') {
            r1 = r2 + r1;
            r2 = r1 - r2;
        }

        // one-digit letter, no new way added
        else {
            r2 = r1;
        }
    }

    return r1;
}
```

MaximalSquare

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

For example, given the following matrix:

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

Return 4.

```
int maximalSquare(vector<vector<char>>& matrix) {
        int nr = matrix.size(); if (nr == 0) return 0;
        int nc = matrix[0].size(); if (nc == 0) return 0;

        vector<int> dp(nc+1, 0);

        int maxsize = 0;

        int last_topleft = 0;  // This is 'pre[i-1]' for the current element

        for (int ir = 1; ir <= nr; ++ir) {
            for (int ic = 1; ic <= nc; ++ic) {
                int temp = dp[ic];      // This is 'pre[i-1]' for the next element
                 if (matrix[ir-1][ic-1] == '0') dp[ic] = 0;
                else {
                    dp[ic] = min(min(dp[ic], dp[ic-1]), last_topleft) + 1;
                    maxsize = max(maxsize, dp[ic]);
                }
                last_topleft = temp;  // update 'pre[i-1]'
            }
        }
        return maxsize * maxsize;
}
```

PalindromePartitioningII

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.

```
思路：
dp[i][j]表示s[i..j]是否是一个回文串
count[i]表示s[0..i]需要的最小cut数目

不需要单独 先求一遍 dp数组
在 同一个双重for循环中 就可以 解决 最小值 和 是否是 回文子串
```

```
int minCut(string s)
{
        if(s.size()<=0)
            return 0;

        vector<vector<int>> dp(s.size(),vector<int>(s.size(),false));
        vector<int> count(s.size(),INT_MAX);
        int min= INT_MAX;

        for(int i=0;i<s.size();i++)
            for(int j=0;j<=i;j++)
            {
                if(i ==j || ( (s[i] == s[j]) && (i==j+1 || dp[i-1][j+1]) ))
                {
                    dp[i][j]= true;
                    int tempcount = j==0? 1: count[j-1]+1;
                    if(tempcount< count[i])
                        count[i] = tempcount;
                }
            }
        return count[s.size()-1]-1;
}
```

BestTimetobuyandSellStock

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

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

方法1

```
int maxProfit(vector<int> &prices)
{
    if(prices.size() <=0) return 0;
        int min=prices[0];
        int maxPro=0;
        for(int i=1 ;i<prices.size();i++)
        {
            if(prices[i]<min)
            {
                min = prices[i];
                continue;
            }
            maxPro = (prices[i]-min) >maxPro ? (prices[i]-min) :maxPro;
        }
    return maxPro;
}
```

方法2 :转换成求maxSumArray 数组间的差值数组 就转换成了和求maxSumArray一样的问题

```
int maxProfit(vector<int> &prices)
{
        if(prices.size() <=0) return 0;

        int sum =0;
        int result=0;
        for(int i=1;i<prices.size();i++)
        {
            sum+=prices[i]-prices[i-1];
            if(sum<0) sum=0;
            result = max(sum,result);
        }

        return result;
}
```

BestTimetobuyandSellStock3

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 two transactions.

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

方法1 动规(还可以把空间复杂度降到O(N))

```
int maxProfit(vector<int> &prices) {
        // f[k, ii] represents the max profit up until prices[ii] (Note: NOT ending with
        // f[k, ii] = max(f[k, ii-1], prices[ii] - prices[jj] + f[k-1, jj]) { jj in range
        //          = max(f[k, ii-1], prices[ii] + max(f[k-1, jj] - prices[jj]))
        // f[0, ii] = 0; 0 times transation makes 0 profit
        // f[k, 0] = 0; if there is only one price data point you can't make any money no
        if (prices.size() <= 1) return 0;
        else {
            int K = 2; // number of max transation allowed
            int maxProf = 0;
            vector<vector<int>> f(K+1, vector<int>(prices.size(), 0));
            for (int kk = 1; kk <= K; kk++) {
                int tmpMax = f[kk-1][0] - prices[0];
                for (int ii = 1; ii < prices.size(); ii++) {
                    f[kk][ii] = max(f[kk][ii-1], prices[ii] + tmpMax);
                    tmpMax = max(tmpMax, f[kk-1][ii] - prices[ii]);
                    maxProf = max(f[kk][ii], maxProf);
                }
            }
            return maxProf;
        }
}
```

方法2：类似 single number 2 题目

```
public int maxProfit(int[] prices)
{
        int hold1 = Integer.MIN_VALUE, hold2 = Integer.MIN_VALUE;
        int release1 = 0, release2 = 0;
        for(int i:prices){                              // Assume we only have 0 money at
            release2 = Math.max(release2, hold2+i);     // The maximum if we've just sold
            hold2    = Math.max(hold2,    release1-i);  // The maximum if we've just buy
            release1 = Math.max(release1, hold1+i);     // The maximum if we've just sold
            hold1    = Math.max(hold1,    -i);          // The maximum if we've just buy
        }
        return release2; ///Since release1 is initiated as 0, so release2 will always hig
}
```

方法3： 从两边 分别遍历一次

```
int maxProfit(vector<int> &prices)
{
         int maxPro = 0;
        int num = prices.size();

        if(1 >= num)
          return 0;
        if(2 == num)
        {
            if(prices[1]>prices[0])
                return prices[1]-prices[0];
            else
                return 0;
        }

        vector<int> pros(num,0);
        vector<int> futurepros(num,0);

        int lowprice =prices[0];

        for(int i=1 ;i<=num-1 ;i++)
        {

           if(prices[i]<lowprice)
           {
                lowprice = prices[i];
                pros[i] = pros[i-1];
           }else
           {
                if(prices[i] - lowprice >pros[i-1])
                {
                    pros[i] = prices[i] - lowprice;
                }else
                    pros[i] = pros[i-1];

           }
        }

        int highprice =prices[num-1];



            for(int j=num-2; j>=0; j--)
            {
                if(prices[j]>highprice)
                {
                    highprice = prices[j];
                    futurepros[j] = futurepros[j+1];
                }else
                {
                    if(highprice-prices[j] >futurepros[j+1])
                    {
                         futurepros[j] = highprice - prices[j] ;
                    }else
                         futurepros[j] = futurepros[j+1];
                }
```

```
                  if(futurepros[j] +pros[j] >maxPro)
                       maxPro = futurepros[j] +pros[j];

              }

          return maxPro;
  }
```

BestTimetobuyandSellStock4

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).

```
int solveMaxProfit(vector<int> &prices)
{
        int res = 0;
        for (int i = 1; i < prices.size(); ++i) {
            if (prices[i] - prices[i - 1] > 0) {
                res += prices[i] - prices[i - 1];
            }
        }
        return res;
}

int maxProfit(int k, vector<int> &prices)
{
        if (prices.empty()) return 0;
        if (k >= prices.size()) return solveMaxProfit(prices);
        int g[k + 1] = {0};
        int l[k + 1] = {0};
        for (int i = 0; i < prices.size() - 1; ++i) {
            int diff = prices[i + 1] - prices[i];
            for (int j = k; j >= 1; --j) {
                l[j] = max(g[j - 1] + max(diff, 0), l[j] + diff);
                g[j] = max(g[j], l[j]);
            }
        }
        return g[k];
}
```

此题目附上同学的解析链接： BestTimetobuyandSellStock题解

DungeonGame

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

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

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

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

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

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

```
-2 (K)    -3    3
-5    -10    1
10    30    -5 (P)
```

方法1：

思路：该题目要求从左上角走到右下角所需要的最少的,思路类似 unique path,但是该题目需要从右下角开始动规

动贵过程：

```
  need = min(hp[i + 1][j], hp[i][j + 1]) - dungeon[i][j];
  hp[i][j] = need <= 0 ? 1 : need;

  need小于0表示从点(i,j)到右下角不需要额外的血补充, 设置为1即可
```

O(MN)的空间复杂度

```
int calculateMinimumHP(vector<vector<int> > &dungeon) {
        int M = dungeon.size();
        int N = dungeon[0].size();
        // hp[i][j] represents the min hp needed at position (i, j)
        // Add dummy row and column at bottom and right side
        vector<vector<int> > hp(M + 1, vector<int>(N + 1, INT_MAX));
        hp[M][N - 1] = 1;
        hp[M - 1][N] = 1;
        for (int i = M - 1; i >= 0; i--) {
            for (int j = N - 1; j >= 0; j--) {
                int need = min(hp[i + 1][j], hp[i][j + 1]) - dungeon[i][j];
                hp[i][j] = need <= 0 ? 1 : need;
            }
        }
        return hp[0][0];
}
```

O(N)空间复杂度

```
int calculateMinimumHP(vector<vector<int> > &dungeon)
{
        const int m = dungeon.size();
        const int n = dungeon[0].size();
        vector<int> dp(n + 1, INT_MAX);
        dp[n - 1] = 1;
        for(int i = m - 1; i >= 0; --i)
            for(int j = n - 1; j >= 0; --j)
                dp[j] = getMin(min(dp[j], dp[j + 1]) - dungeon[i][j]);
        return dp[0];
}

int getMin(int n)
{
       return n <= 0 ? 1 : n;
}
```

方法2：在原有的数组上进行操作(不太合适)

```
int calculateMinimumHP(vector<vector<int> > &dungeon)
{
     int m = dungeon.size();
          int n = dungeon[0].size();
          dungeon[m-1][n-1] = max(0-dungeon[m-1][n-1], 0);
         for (int i = m - 2; i >= 0; --i) {
             dungeon[i][n-1] = max(dungeon[i+1][n-1]-dungeon[i][n-1], 0);
          }
         for (int j = n - 2; j >= 0; --j) {
             dungeon[m-1][j] = max(dungeon[m-1][j+1]-dungeon[m-1][j], 0);
         }
         for (int i = m - 2; i >= 0; --i) {
             for (int j = n - 2; j >= 0; --j) {
                 dungeon[i][j] = max(min(dungeon[i][j+1], dungeon[i+1][j])-dungeon[i][j],
             }
         }
         return dungeon[0][0] + 1;
}
```

Distinct Subsequences

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

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

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

Return 3.

题解： 不错的动规题目，借用了leetcode上大神的代码以及解释

```
/**
 * Solution (DP):
 * We keep a m*n matrix and scanning through string S, while
 * m = T.length() + 1 and n = S.length() + 1
 * and each cell in matrix Path[i][j] means the number of distinct subsequences of
 * T.substr(1...i) in S(1...j)
 *
 * Path[i][j] = Path[i][j-1]            (discard S[j])
 *              +     Path[i-1][j-1]    (S[j] == T[i] and we are going to use S[j])
 *                 or 0                 (S[j] != T[i] so we could not use S[j])
 * while Path[0][j] = 1 and Path[i][0] = 0.
 */
int numDistinct(string S, string T) {
    int m = T.length();
    int n = S.length();
    if (m > n) return 0;    // impossible for subsequence
    vector<vector<int>> path(m+1, vector<int>(n+1, 0));
    for (int k = 0; k <= n; k++) path[0][k] = 1;    // initialization

    for (int j = 1; j <= n; j++) {
        for (int i = 1; i <= m; i++) {
            path[i][j] = path[i][j-1] + (T[i-1] == S[j-1] ? path[i-1][j-1] : 0);
        }
    }

    return path[m][n];
}


/**  O(m) 空间复杂度
 * Further optimization could be made that we can use only 1D array instead of a
 * matrix, since we only need data from last time step.
 */

int numDistinct(string S, string T) {
    int m = T.length();
    int n = S.length();
    if (m > n) return 0;    // impossible for subsequence
    vector<int> path(m+1, 0);
    path[0] = 1;            // initial condition

    for (int j = 1; j <= n; j++) {
        // traversing backwards so we are using path[i-1] from last time step
        for (int i = m; i >= 1; i--) {
            path[i] = path[i] + (T[i-1] == S[j-1] ? path[i-1] : 0);
        }
    }

    return path[m];
}
```

HouseRobber

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

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

```
思路：
result[i]表示偷到第i家为止能获得的最大价值
得到动规方程：
result[i]=max(result[i-1],num[i]+result[i-2])
很简单的一道题目，看看我们能不能用O(1)的空间完成呢？

int rob(vector<int> &num)
{

        if(num.size()<=0)
        {
            return 0;
        }
        if(num.size()==1)
            return num[0];
        vector<int> result(num.size(),0);
        result[0]=num[0];
        result[1]=max(num[0],num[1]);

        for(int i=2;i<num.size();i++)
        {
            result[i]=max(result[i-1],num[i]+result[i-2]);
        }

        return result[num.size()-1];
}
```

HouseRobber2

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

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

```
思路：
O (1) 空间 O(n)时间
构成环的环的话考虑两种情况，一种选最后一个；一种不选最后一个

int rob(vector<int>& nums)
{
       if(nums.size() == 0)
        return 0;
        if(nums.size() == 1)
            return nums[0];

        int pre2=0,pre1 = 0, cur1 = 0;
        for(int i = 0; i < nums.size() - 1; ++ i)
        {
            pre2 = pre1;
            pre1 = cur1;
            cur1 = max(pre2 + nums[i], pre1);
        }

        int pre3=0,pre4 = 0, cur2 = 0;
        for(int i = 1; i < nums.size(); ++ i)
        {
            pre4 = pre3;
            pre3 = cur2;
            cur2 = max(pre4 + nums[i], pre3);
        }

        return max(cur1, cur2);
 }
```

```
方法2:
感觉写的比较简练 思路是一样的

private int rob(int[] num, int lo, int hi) {
    int include = 0, exclude = 0;
    for (int j = lo; j <= hi; j++) {
        int i = include, e = exclude;
        include = e + num[j];
        exclude = Math.max(e, i);
    }
    return Math.max(include, exclude);
}

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

InterleavingString

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.

```
思路：
典型的DP题目
V[i][j]  表示 s1(i,n1)与s2(j,n2) 能否拼装成 s3(i+j,n3)

动规方程：
V[i][j] = ((s1[i] == s3[i+j]) && V[i+1][j]) | ((s2[j] == s3[i+j]) && V[i][j+1]);

bool isInterleave(string s1, string s2, string s3)
{
        int n1 = s1.length(), n2 = s2.length(), n3 = s3.length();
        if (n1+n2 != n3) return false;

        vector<vector<bool>> V(n1+1, vector<bool>(n2+1, false));

        V[n1][n2] =true;

        for(int i=n1-1 ; i>=0 ;i--)
            V[i][n2] = V[i+1][n2] && (s3[n2+i] == s1[i]) ;

        for(int i= n2-1 ;i>=0;i--)
            V[n1][i] = V[n1][i+1] && (s3[n1+i] == s2[i]);

        for(int i=n1-1;i>=0;i--)
            for(int j=n2-1;j>=0;j--)
            {
                V[i][j] = ((s1[i] == s3[i+j]) && V[i+1][j]) | ((s2[j] == s3[i+j]) && V[i]
            }

        return V[0][0];
}
```

MaximumProductSubarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

```
方法1
假设数组A中没有0， 最大乘积的值对应subarray在数组中的分布 一定是：
A[0] A[1] .... A[i] 或者 A[j] *A[j+1] A[n - 1].

假设有0的话(比如说 A[k] == 0). 那么 A[0],A[1]...A[k - 1 ]
以及 A[k + 1] A[k + 2]...A[n-1] 可以作为两个新数组来考虑
整理一下代码如下：

int maxProduct(int A[], int n)
{
        int frontProduct = 1;
        int backProduct = 1;
        int ans = INT_MIN;
        for (int i = 0; i < n; ++i)
        {
            frontProduct *= A[i];
            backProduct *= A[n - i - 1];
            ans = max(ans,max(frontProduct,backProduct));
            frontProduct = frontProduct == 0 ? 1 : frontProduct;
            backProduct = backProduct == 0 ? 1 : backProduct;
        }
        return ans;
}
```

```
方法2
int maxProduct(vector<int>& A)
{
         // store the result that is the max we have found so far
        int r = A[0];

        // imax/imin stores the max/min product of
        // subarray that ends with the current number A[i]
        for (int i = 1, imax = r, imin = r; i < A.size(); i++) {
            // multiplied by a negative makes big number smaller, small number bigger
            // so we redefine the extremums by swapping them
            if (A[i] < 0)
                swap(imax, imin);

            // max/min product for the current number is either the current number itself
            // or the max/min by the previous number times the current one
            imax = max(A[i], imax * A[i]);
            imin = min(A[i], imin * A[i]);

            // the newly computed max value is a candidate for our global result
            r = max(r, imax);
        }
        return r;
}
```

```
方法2的复杂版本
public int maxProduct(int[] A)
{
        if(A.length<=0) return 0;
        if(A.length==1) return A[0];
        int curMax = A[0];
        int curMin = A[0];
        int ans = A[0];
        for(int i=1;i<A.length;i++){
            int temp = curMin *A[i];
            curMin = Math.min(A[i], Math.min(temp, curMax*A[i]));
            curMax = Math.max(A[i], Math.max(temp, curMax*A[i]));
            ans = Math.max(ans, curMax);
        }
        return ans;
}
```

longestValidParentheses

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.

```
longest[i]表示s(0..i)最长的validParentheses的长度

int longestValidParentheses(string s)
{
        if(s.length() <= 1) return 0;
        int curMax = 0;
        vector<int> longest(s.size(),0);
        for(int i=1; i < s.length(); i++){
            if(s[i] == ')' && i-longest[i-1]-1 >= 0 && s[i-longest[i-1]-1] == '('){
                    longest[i] = longest[i-1] + 2 + ((i-longest[i-1]-2 >= 0)?longest[i-lo
                    curMax = max(longest[i],curMax);
            }
        }
        return curMax;
}
```

Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

Note: Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

```
典型的DP题目 同时只需要 O(n)的空间复杂度
自底向上解决代码简练

int minimumTotal(vector<vector<int> > &triangle) {
    int n = triangle.size();
    vector<int> minlen(triangle.back());
    for (int layer = n-2; layer >= 0; layer--)
    {
        for (int i = 0; i <= layer; i++) //每一行有layer个元素
        {
            minlen[i] = min(minlen[i], minlen[i+1]) + triangle[layer][i];
        }
    }
    return minlen[0];
}
```

数学

ValidNumber

Validate if a given string is numeric.

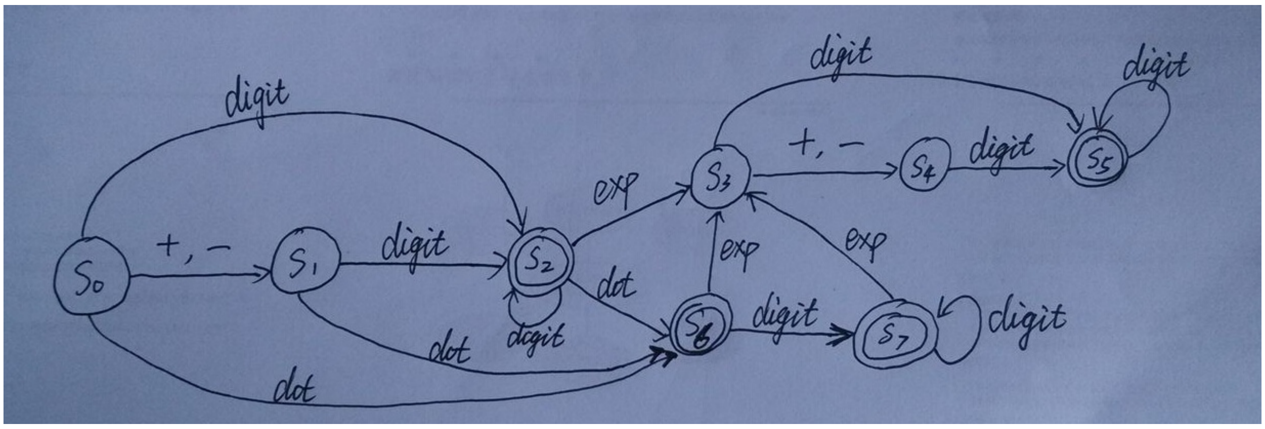
Some examples:

```
"0" => true
" 0.1 " => true
"abc" => false
"1 a" => false
"2e10" => true
```

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

方法1：

利用自动机的方法进行验证 DFA如下图(代码和图是摘自leetcode)：



```
bool isNumber(string str) {
        int state=0, flag=0; // flag to judge the special case "."
        while(str[0]==' ')  str.erase(0,1);//delete the  prefix whitespace
        while(str[str.length()-1]==' ') str.erase(str.length()-1, 1);//delete the suffix
        for(int i=0; i<str.length(); i++){
            if('0'<=str[i] && str[i]<='9'){
                flag=1;
                if(state<=2) state=2;
                else state=(state<=5)?5:7;
            }
            else if('+'==str[i] || '-'==str[i]){
                if(state==0 || state==3) state++;
                else return false;
            }
            else if('.'==str[i]){
                if(state<=2) state=6;
                else return false;
            }
            else if('e'==str[i]){
                if(flag&&(state==2 || state==6 || state==7)) state=3;
                else return false;
            }
            else return false;
        }
        return (state==2 || state==5 || (flag&&state==6) || state==7);
}
```

方法2：

直接对字符串进行判断(自己比较懒....代码也是摘自leetcode)

```
 public boolean isNumber(String s) {
        s = s.trim();
        if (s.length() == 0) return false;
        boolean hasE = false;
        boolean hasDot = false;
        boolean hasNumber = false;

        for (int i = 0; i < s.length(); i++) {
            // e cannot be the first character
            if (i == 0 && s.charAt(i) == 'e') return false;
            if (s.charAt(i) == 'e') {
                // e cannot be replicated nor placed before number
                if (hasE == true || hasNumber == false) {
                    return false;
                } else {
                    hasE = true;
                }
            }

            if (s.charAt(i) == '.') {
                // '.' cannot be replicated nor placed after 'e'
                if (hasDot == true || hasE == true) {
                    return false;
                } else {
                    hasDot = true;
                }
            }
            // the sign can be placed at the beginning or after 'e'
            if (i != 0 && s.charAt(i - 1) != 'e' && (s.charAt(i) == '+' || s.charAt(i) ==

            // no other chacraters except '+', '-', '.', and 'e'
            if ((s.charAt(i) > '9' || s.charAt(i) < '0') && s.charAt(i) != '+' && s.charA
            return false;

            // check whether numbers are included.
            if (s.charAt(i) <= '9' && s.charAt(i) >= '0') {
                hasNumber = true;
            }
        }
        // '+', '-', and 'e' cannot be the last character
        if (s.charAt(s.length() - 1) == '-' || s.charAt(s.length() - 1) == '+' || s.charA

        return hasNumber;
}
```

CountPrimes

Description:

Count the number of prime numbers less than a non-negative number, n.

思路： 埃拉托色尼选筛法(Sieve of Eratosthenes)

```
用一个大小为n的表来记录每个数的素数性

对于每个质数n，其2*n,3*n,4*n 就都不是质数
遍历每个数，判断一个数若是质数，那就把其所有的的倍数的值(小于n)设置为false.

最后统计素数的数目即可

int countPrimes(int n)
{
      vector<bool> isPrime(n,true);

      for(int i=2;i*i<n;i++)
        {
            if(!isPrime[i]) continue;
            for(int j=i*i ;j<n ;j +=i)//j 从i*i开始即可
                isPrime[j] = false;
      }

     int count =0;//统计数目
     for(int i=2;i<n;i++)
        if(isPrime[i])
          count++;

     return count;
}
```

AddBinary

Given two binary strings, return their sum (also a binary string).

For example,

```
a = "11"
b = "1"
Return "100".
```

```
思路:不用考虑a与b的长度谁大,直接进行二进制加法操作
string addBinary(string a, string b)
{
       int c=0;
       int i = a.size()-1;
       int j = b.size()-1;

       string s="";
       int vala =0;
       int valb =0;
       while(c>0 || i >=0 || j>=0)
       {
           vala = i>=0 ? a[i--]-'0': 0;
           valb = j>=0 ? b[j--]-'0': 0;

           c += vala+valb;
           s.push_back(c%2+'0');
           c = c/2;
       }

       reverse(s.begin(),s.end());
       return s;
}
```

Number of Digit One

Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n.

For example: Given n = 13, Return 6, because digit 1 occurred in the following numbers: 1, 10, 11, 12, 13.

```
思路：编程之美上的题目


对于每一位上能出现的1次数进行统计
比如说：10312
十位1出现1的次数即为103*10加上3(10310, 10311,10312)

百位3出现1的次数为：(10+1)*100

千位0 出现1的次数为：（1*1000）

每一位根据0,1和大于1 分成不同的情况 个有着不同的计算出现1次数方式

int countDigitOne(int n)
{
        long long res =0;
        if(n<=0) return res;

        long long fact =1;
        long long low=0;
        long long curr=0;
        long long high=0;

        while(n/fact !=0)
        {
            low = n- (n/fact)*fact;
            curr = (n/fact)%10;
            high = n/(fact*10);

            switch(curr)
            {
                case 0 : res += high*fact;  break;
                case 1: res+= high*fact +low+1; break;
                default :
                    res += (high+1)*fact;
            }
            fact = fact *10;
        }
        return res;
}
```

Sqrt

Implement int sqrt(int x).

Compute and return the square root of x.

```
方法1：
二分查找
int sqrt(int x)
{
        unsigned long long begin = 0;
        unsigned long long end = (x+1)/2;  //sqrt(x)一定小于x/2
        unsigned long long mid;
        unsigned long long tmp;
        while(begin < end)
        {
            mid = begin + (end-begin)/2;
            tmp = mid*mid;
            if(tmp==x)return mid;
            else if(tmp<x) begin = mid+1;
            else end = mid-1;
        }
        tmp = end*end;
        if(tmp > x)
            return end-1;
        else
            return end;
}
```

```
方法2：
位运算
int mySqrt(int x)
{
     unsigned int res = 0;
     for (int i = 15; i >= 0; i--)
     {
        if ((res + (1 << i)) * (res + (1 << i)) <= x)
            res += (1 << i);
     }
     return res;
}
```

```
方法3：牛顿迭代算法
int sqrt(int x)
{
    double ans    = x;
    double delta  = 0.0001;
    while (fabs(pow(ans, 2) - x) > delta) {
        ans = (ans + x / ans) / 2;
    }
    return ans;
}
```

```
x(n+1) = xn - f(xn) / f'(xn)

x = sqrt(S) where S is the parameter x x^2 = S

f(xn) = xn^2 - S = 0 f'(xn) = 2 * xn

From the first sentence, x(n+1) = xn - (xn^2 - S) / 2 * xn x(n+1) = (xn + S / x_n) / 2

From the code, ans = (ans + x / ans) / 2;

I.E. get x where f(x) = x^7 + x^4 + x^2 + 1 = 0

f(x) = x^7 + x^4 + x^2 + 1 f'(x) = 7x^6 + 4x^3 + 2*x

while( > delta) { x(n+1) = xn - (xn^7 + xn^4 + xn^2 + 1) / (7*xn^6 + 4x_n^3 + 2x_n) }

return x(n_1) // depending initial value and the status of the graph, the result may vary
```

MultiplyStrings

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note: The numbers can be arbitrarily large and are non-negative.

```
思路：
按照传统的乘法方法 从string末尾 开始算

string multiply(string num1, string num2)
{
        string sum(num1.size() + num2.size(), '0');

        for (int i = num1.size() - 1; 0 <= i; --i) {
            int carry =0;
            for(int j=num2.size()-1 ; 0<=j ; j--)
            {
                int temp = sum[i+j+1]-'0' + (num1[i]-'0')*(num2[j]-'0')+carry;
                sum[i+j+1] = temp%10+'0';
                carry = temp/10;
            }
            sum[i] += carry; //
        }

        size_t startpos = sum.find_first_not_of("0");
        if (string::npos != startpos) {
            return sum.substr(startpos);
        }
        return "0";
}
```

ReverseInteger

Reverse digits of an integer.

Example1: x = 123, return 321 Example2: x = -123, return -321

Have you thought about this? Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!

If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.

Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?

For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.

Update (2014-11-10): Test cases had been added to test the overflow behavior.

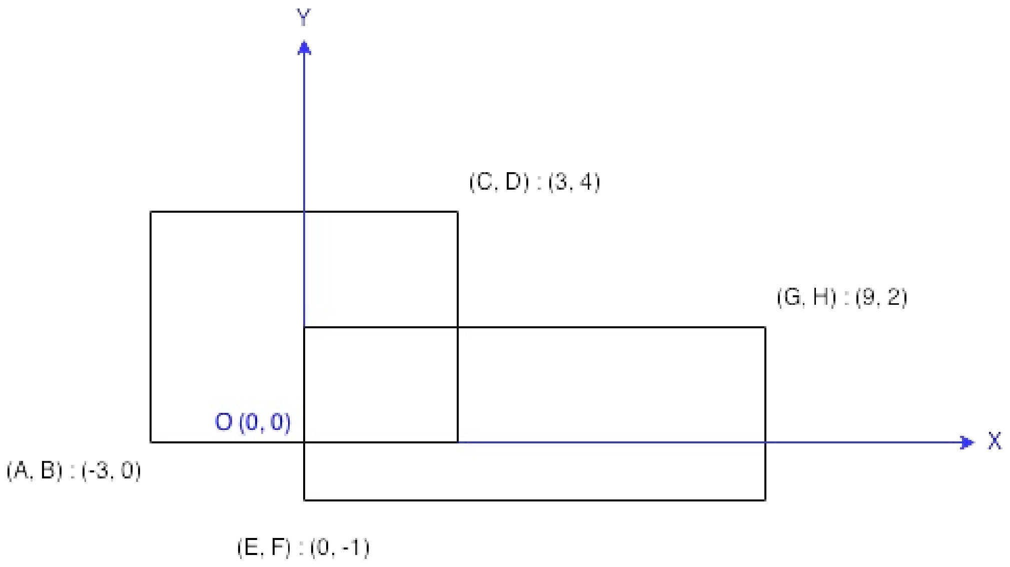
```
方法1：代码来自leetcode

int reverse(int x)
{
    long long val = 0;
    do
    {
        val = val * 10 + x % 10;
        x /= 10;
    } while (x);

    return (val > INT_MAX || val < INT_MIN) ? 0 : val;
}
```

```
方法2：

int reverse(int x)
{
     int sign = 1;
     if (x<0)
     {
            sign = -1;
            x = -x; // make it positive
            if (x<0) // x was numeric_limits<int>::min() -2^31
                return 0;
     }

     int res = 0;
     int compare = numeric_limits<int>::max() / 10;
     while (x)
     {
            if (res > compare)
                return 0;
            res = res*10 + x% 10;
            x /= 10;
        }

        return res*sign;
    }
```

RectangleArea

Find the total area covered by two rectilinear rectangles in a 2D plane.

Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.



Rectangle Area

Assume that the total area is never beyond the maximum possible value of int.

```
思路:
两个面积相加减掉重复的部分

int computeArea(int A, int B, int C, int D, int E, int F, int G, int H)
{

        int S1 = (C-A)*(D-B);
        int S2 = (G-E)*(H-F);
        if(C < E || G < A || H < B || D < F)
        {
            return S1 + S2; // no intersection
        }
        int xi_left = max(A, E);
        int yi_left = max(B, F);
        int xi_right = min(C, G);
        int yi_right = min(D, H);

        return S1 + S2 - (xi_right - xi_left)*(yi_right - yi_left);
}
```

ExcelSheetColumnTitle

Given a positive integer, return its corresponding column title as appear in an Excel sheet.

For example:

```
1 -> A
2 -> B
3 -> C
...
26 -> Z
27 -> AA
28 -> AB
```

```
string convertToTitle(int n)
{
    string res = "";
    while(n)
    {
        res = (char)('A' + (n-1)%26) + res; //注意不要加反了
        n = (n-1) / 26;
    }
    return res;
}
```

ExcelSheetColumnTitle

Related to question Excel Sheet Column Title

Given a column title as appear in an Excel sheet, return its corresponding column number.

For example:

```
A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
```

```
思路:
相当于一个26进制数
int titleToNumber(string s)
{
        int res=0;
        for(auto ss :s)
        {
            res = res*26 + ss-'A'+1;
        }
        return res;
}
```

FactorialTrailingZeroes

Given an integer n, return the number of trailing zeroes in n!.

```
思路：
又是一道编程之美上的题目
n的阶乘 末尾能造成出现0 只有可能2*5形成.
也就是变成了1...n之间的数 有多少个2的因子和5因子

而其中出现5因子的次数是远远少于2的，因此最终问题就变成了求
1...n中含有多少个5因子（像25这样的数是提供2个5因子）

int trailingZeroes(int n)
{
        if(n<=0) return 0;
        int ret=0;
        while(n)
        {
            ret += n/5;
            n= n/5;
        }
        return ret;
}
```

DivideTwoNumbers

Divide two integers without using multiplication, division and mod operator.

If it is overflow, return MAX_INT.

```
思路：
必须用减法来实现

A=B*(2^k1+2^k2+2^k3+2^k4...)
转换为求result = (2^k1+2^k2+2^k3+2^k4...)


int divide(int dividend, int divisor)
{
        auto sign = [=](int x){
            return x>=0 ? 1 : -1;
        };

        int d1 = sign(dividend);
        int d2 = sign(divisor);

        if(divisor == 0 || (dividend == INT_MIN && divisor ==-1))
            return INT_MAX;

        long long n1 = abs(static_cast<long long>(dividend));
        long long n2 = abs(static_cast<long long>(divisor));
        long long ans = 0;

        while(n1>=n2)
        {
            long long base = n2;
            for(int i=0;base<=n1;i++)
            {
                n1 -= base;
                base <<=1;
                ans += 1LL<<i;
            }
        }

        int res = static_cast<int>(ans);

        if(d1 != d2) return -res;
        else return res;
}
```

Basic Calculator

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open (and closing parentheses), the plus + or minus sign -, non-negative integers and empty spaces .

You may assume that the given expression is always valid.

Some examples:

```
"1 + 1" = 2
" 2-1 + 2 " = 3
"(1+(4+5+2)-3)+(6+8)" = 23
Note: Do not use the eval built-in library function.
```

```
int calculate(string s)
{
        stack<int> signs; //用来存放 正负号
        int sign = 1;
        int num = 0;
        int ans = 0;

        signs.push(1);

        for(auto c:s)
        {
            if(c>='0' && c<='9')
            {
                num = num*10+c-'0';
            }else if( c == '+' || c=='-')
            {
                ans = ans + signs.top() * sign * num;
                sign = c=='+' ? 1 :-1;
                num =0;
            }else if(c ==('(')
            {
                signs.push(sign* signs.top());
                sign =1;

            }else if( c == ')')
            {
                ans = ans + signs.top() * sign * num;
                signs.pop();
                num=0;
                sign =1;
            }
        }

        if (num) {
            ans = ans + signs.top() * sign * num;
        }
        return ans;
}
```

Power(x,n)

Implement pow(x, n).

```
思路:
利用的是 蒙哥马利快速模乘算法
将幂转换为二进制
注意INT_MIN的绝对值比INT_Max大1，因此nINT_MIN转换为正数求幂时，要多乘以一个x

double myPow(double x, int n) {

    if(n<0)
     {
        if(n == INT_MIN)
            return 1.0/(myPow(x,INT_MAX))*x;
        else
            return 1.0/myPow(x,-n);
     }

     if(n==0)
         return 1.0;

     double result =1.0;
     double ass =x;

     for( ;n>0;n >>= 1)
     {
        if(n & 0x1)
            result *=ass;
        ass *=ass;
     }
     return result;
 }
```

StringToInteger

Implement atoi to convert a string to an integer.

Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

```
int myAtoi(string str)
{
        int index=0;
        int n = str.size();

        //去掉空格
        while(index<n && str[index]==' ') index++;

        if(index>=n) return 0;

        //判断正负
        int sign =1;
        if(str[index]=='+')
        {
            sign=1;
            index++;
        }else if(str[index]=='-')
        {
            sign =-1;
            index++;
        }
        long result =0;

        //计算整数值 只关注是数字的字符
        while(index<n && result< INT_MAX && isdigit(str[index]) )
        {
            result = result*10+ str[index]-'0';
            index++;
        }

        if(result >INT_MAX)
        {
            return sign ==1 ? INT_MAX:INT_MIN;
        }

        return result*sign;

}
```

FractionToRecurringDecimal

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

Given numerator = 1, denominator = 2, return "0.5". Given numerator = 2, denominator = 1, return "2". Given numerator = 2, denominator = 3, return "0.(6)".

```
string fractionToDecimal(int64_t n, int64_t d)
{
    // zero numerator
    if (n == 0) return "0";

    string res;
    // determine the sign
    if (n < 0 ^ d < 0) res += '-';

    // remove sign of operands
    n = abs(n), d = abs(d);

    // append integral part
    res += to_string(n / d);

    // in case no fractional part
    if (n % d == 0) return res;

    res += '.';

    unordered_map<int, int> map;

    // simulate the division process
    for (int64_t r = n % d; r; r %= d) {

        // meet a known remainder
        // so we reach the end of the repeating part
        if (map.count(r) > 0) {
            res.insert(map[r], 1, '(');
            res += ')';
            break;
        }

        // the remainder is first seen
        // remember the current position for it
        map[r] = res.size();

        r *= 10;

        // append the quotient digit
        res += to_string(r / d);
    }

    return res;
}
```

MaxPointsonaLine

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

```
思路：
每个点计算和其在一条直线上点的数目，最后统计最大值，在同一条直线上的判断标准是斜率相同

但是 切记 不要将斜率直接作为 hash 的key值
因为double 作为 key值 是有 风险的，因此将斜率的分数值得分子，分母作为key值存到map中


int maxPoints(vector<Point> &points) {

        if(points.size()<2) return points.size();

        int result=0;

        for(int i=0; i<points.size(); i++) {

            map<pair<int, int>, int> lines;
            int localmax=0, overlap=0, vertical=0;

            for(int j=i+1; j<points.size(); j++) {

                if(points[j].x==points[i].x && points[j].y==points[i].y) {

                    overlap++;
                    continue;
                }
                else if(points[j].x==points[i].x) vertical++;
                else {

                    int a=points[j].x-points[i].x, b=points[j].y-points[i].y;
                    int gcd=GCD(a, b);

                    a/=gcd;
                    b/=gcd;

                    lines[make_pair(a, b)]++;
                    localmax=max(lines[make_pair(a, b)], localmax);
                }

                localmax=max(vertical, localmax);
            }

            result=max(result, localmax+overlap+1);
        }

        return result;
    }

private:
    int GCD(int a, int b) {
        if(b==0) return a;
        else return GCD(b, a%b);
    }
```

PlusOne

Given a non-negative number represented as an array of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

```
思路：
从末尾判断每个数字加1后是否有进位
不是则跳出返回

vector<int> plusOne(vector<int>& digits) {
        bool carry =true;
        for(int i=digits.size()-1; i>=0 && carry; i--)
        {
            carry = (++digits[i]%10 == 0) ;
            digits[i] %= 10;
        }

        if(carry) {
            digits[0]=1;
            digits.push_back(0);
        }

        return digits;
}
```

HappyNumber

Write an algorithm to determine if a number is "happy".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.

```
Example: 19 is a happy number

12 + 92 = 82
82 + 22 = 68
62 + 82 = 100
12 + 02 + 02 = 1
```

```
bool isHappy(int n)
{
        if(n ==1) return true;

        unordered_set<int> set;

        int newNum =0;

        while((newNum = getNum(n)) !=1 )
        {
            if(set.find(newNum) == set.end())
                set.insert(newNum);
            else
                return false; //若是出现过，则发生循环返回false
            n = newNum ;
        }

        return true;
}

int getNum(int n)
{
     int result =0;
     while(n>0)
     {
         int temp = n%10;
         result += temp*temp;
         n = n/10;
     }
     return result;
}
```

PermutationSequence

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

By listing and labeling all of the permutations in order, We get the following sequence (ie, for n = 3):

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

Given n and k, return the kth permutation sequence.

Note: Given n will be between 1 and 9 inclusive.

```
string getPermutation(int n, int k)
{
        vector<int> nums(n);
        int pCount = 1;
        for(int i = 0 ; i < n; ++i) {
            nums[i] = i + 1;
            pCount *= (i + 1);
        }

        k--;
        string res = "";
        for(int i = 0 ; i < n; i++) {
            pCount = pCount/(n-i);
            int selected = k / pCount;
            res += ('0' + nums[selected]);

            for(int j = selected; j < n-i-1; j++)
                nums[j] = nums[j+1];
            k = k % pCount;
        }
        return res;
}
```

排序

栈

Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another expression.

```
Some examples:
  ["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
  ["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
```

//stack µÄ¾µäÀûÓÃ

```
int perop(int a,int b,string op)
{
        if(op=="+")
            return a+b;
        else if(op=="-")
            return a-b;
        else if(op=="*")
            return a*b;
        else if(op=="/")
            return a/b;
        return 0;
}

int evalRPN(vector<string>& tokens)
{
        stack<int> ss;
        for(auto str : tokens)
        {
            if(str == "+" || str =="-" || str =="*" || str =="/")
            {
                int a = ss.top();ss.pop();
                int b = ss.top();ss.pop();
                int c = perop(b,a,str);
                ss.push(c);
            }else{
                ss.push(atoi(str.c_str()));
            }
        }

        return ss.top();
}
```

LargestRectangleinHistogram

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.

```
思路:典型的stack使用

int largestRectangleArea(vector<int> &height)
{

     if(height.size() == 0) return 0;

        int res = 0;
        vector<int> tmp = height;
        tmp.push_back(0);  // Important  让 最终所有的元素出栈

        stack<int> s;
        for(int i = 0; i < tmp.size(); i++)
        {
            if(s.empty() || (!s.empty() && tmp[i] >= tmp[s.top()])) s.push(i);
            else{
                while(!s.empty() && tmp[s.top()] > tmp[i])
                {
                    int idx = s.top(); s.pop();
                    int width = s.empty() ? i : (i-s.top()-1);
                    res = max(res, tmp[idx] * width);
                }
                s.push(i);  // Important
            }
        }
     return res;
}
```

MaximalRectangle

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

```
方法1： DP 抄自leetcode上的一大神

height counts the number of successive '1's above (plus the current one).
The value of left & right means the boundaries of the rectangle which contains the curren
height of value height.

int maximalRectangle(vector<vector<char> > &matrix) {
    if(matrix.empty()) return 0;
    const int m = matrix.size();
    const int n = matrix[0].size();
    int left[n], right[n], height[n];
    fill_n(left,n,0); fill_n(right,n,n); fill_n(height,n,0);
    int maxA = 0;
    for(int i=0; i<m; i++) {
        int cur_left=0, cur_right=n;
        for(int j=0; j<n; j++) { // compute height (can do this from either side)
            if(matrix[i][j]=='1') height[j]++;
            else height[j]=0;
        }
        for(int j=0; j<n; j++) { // compute left (from left to right)
            if(matrix[i][j]=='1') left[j]=max(left[j],cur_left);
            else {left[j]=0; cur_left=j+1;}
        }
        // compute right (from right to left)
        for(int j=n-1; j>=0; j--) {
            if(matrix[i][j]=='1') right[j]=min(right[j],cur_right);
            else {right[j]=n; cur_right=j;}
        }
        // compute the area of rectangle (can do this from either side)
        for(int j=0; j<n; j++)
            maxA = max(maxA,(right[j]-left[j])*height[j]);
    }
    return maxA;
}
```

```
方法2思路：stack 方法 和 求histogram的方法一样
O(n)的空间复杂度
int maximalRectangle(vector<vector<char>>& matrix)
{
        if (matrix.size() <= 0 || matrix[0].size() <= 0)
        return 0;

        int m = matrix.size();
        int n = matrix[0].size()+1; //最后一个元素是0 方便让所有的元素出栈
        int h = 0, w = 0, ret = 0;
        vector<int> height(n, 0);
        for (int i = 0; i < m; ++i) {
            stack<int> s;
            for(int j=0;j<n;j++)
            {
                if(j<n-1)
                {
                    if(matrix[i][j] == '1')
                        height[j] ++;
                    else
                        height[j]=0;
                }


                while(!s.empty() && height[s.top()]>= height[j]) //计算所有高于最低点的可能的
                {
                    int idx = s.top();
                    s.pop();
                    int width = s.empty()? j: j-1-s.top();
                    ret = max(ret,width*height[idx]);
                }

                s.push(j);
            }
        }

        return ret;
}
```

SimplifyPath

Given an absolute path for a file (Unix-style), simplify it.

For example, path = "/home/", => "/home" path = "/a/./b/../../c/", => "/c" click to show corner cases.

Corner Cases: Did you consider the case where path = "/../"? In this case, you should return "/". Another corner case is the path might contain multiple slashes '/' together, such as "/home//foo/". In this case, you should ignore redundant slashes and return "/home/foo".

```
方法1
string simplifyPath(string path)
{
       vector<string>   nameVect;
        string name;

        path.push_back('/');
        for(int i=0;i<path.size();i++){
            if(path[i]=='/'){
                if(name.size()==0)continue;
                if(name==".."){     //special case 1：double dot, pop dir
                     if(nameVect.size()>0)nameVect.pop_back();
                }else if(name=="."){//special case 2:singel dot, don`t push
                }else{
                    nameVect.push_back(name);
                }
                name.clear();
            }else{
                name.push_back(path[i]);//record the name
            }
        }

        string result;
        if(nameVect.empty())return "/";
        for(int i=0;i<nameVect.size();i++){
            result.append("/"+nameVect[i]);
        }
        return result;
}
```

```
方法2 ：
思路 是利用 stringstream 的 getline方法
类似于 java 的split  操作起来相比方法1要简单很多
string simplifyPath(string path)
{
    string res, tmp;
    vector<string> stk;
    stringstream ss(path);
    while(getline(ss,tmp,'/')) {
        if (tmp == "" or tmp == ".") continue;
        if (tmp == ".." and !stk.empty()) stk.pop_back();
```

```
            else if (tmp != "..") stk.push_back(tmp);
        }
        for(auto str : stk) res += "/"+str;
        return res.empty() ? "/" : res;
    }
```

ValidParentheses

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

```
思路： 这个题目很简单，用一个栈来保存 "左边"类型的括号,遇到对应的"右边"括号则弹出
bool isValid(string s)
{
        stack <char> stk;
        for(char& c: s)
        {
            switch(c)
            {
                case '(':
                case '[':
                case '{': stk.push(c);break;
                case ')': if(stk.empty()|| stk.top() != '(') return false; else stk.pop()
                case ']': if(stk.empty() || stk.top() != '[' )return false; else stk.pop(
                case '}': if(stk.empty() || stk.top() != '{' )return false; else stk.pop(
                default : ;
            }
        }

        return stk.empty();
}
```

回溯

- permutations
- permutations2
- GenerateParentheses
- Subsets
- Subsets2
- WordBreak2
- Combinations
- Combination Sum
- CombinationSum2
- CombinationSum3
- LetterCombinationsofaPhoneNumber
- WordSearch
- PalindromePartitioning
- GrayCode
- RestoreIPAddresses
- SudokuSolver
- N-Queen
- N-Queen2

广度优先搜索

NumberofIslands

Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

```
Example 1:

11110
11010
11000
00000
Answer: 1

Example 2:

11000
11000
00100
00011
Answer: 3
```

```
以下代码都摘自leetcode
方法1 ：DFS
从某个“1”点出发，一直深度遍历周围所有为1的grid

void contaminate(vector<vector<char> > &grid, int i, int j)
{
        if(i>0&&grid[i-1][j]=='1'){
            grid[i-1][j]='0';
            contaminate(grid, i-1, j);
        }
        if(j>0&&grid[i][j-1]=='1'){
            grid[i][j-1]='0';
            contaminate(grid, i, j-1);
        }
        if(i<grid.size()-1&&grid[i+1][j]=='1'){
            grid[i+1][j]='0';
            contaminate(grid, i+1, j);
        }
        if(j<grid[0].size()-1&&grid[i][j+1]=='1'){
            grid[i][j+1]='0';
            contaminate(grid, i, j+1);
        }
}

int numIslands(vector<vector<char>> &grid)
{
        int n=grid.size();
        if(n==0) return 0;
        int m=grid[0].size();
        int cnt=0;
        for(int i=0; i<n; i++){
            for(int j=0; j<m; j++){
                if(grid[i][j]=='1'){
                    cnt++;
                    contaminate(grid, i, j);
                }
            }
        }
        return cnt;
}


方法2：广度优先搜索 用一个队列来保存

int numIslands(vector<vector<char>> &grid)
{
        if(grid.size() == 0 || grid[0].size() == 0)
            return 0;

        int res = 0;
        for(int i = 0; i < grid.size(); ++ i)
            for(int j = 0; j < grid[0].size(); ++ j)
                if(grid[i][j] == '1')
                {
                    ++ res;
                    BFS(grid, i, j);
                }
        return res;
}

private:
void BFS(vector<vector<char>> &grid, int x, int y)
{
        queue<vector<int>> q;
        q.push({x, y});
        grid[x][y] = '0';

        while(!q.empty())
        {
            x = q.front()[0], y = q.front()[1];
            q.pop();

            if(x > 0 && grid[x - 1][y] == '1')
            {
                q.push({x - 1, y});
                grid[x - 1][y] = '0';
            }
            if(x < grid.size() - 1 && grid[x + 1][y] == '1')
            {
                q.push({x + 1, y});
                grid[x + 1][y] = '0';
            }
            if(y > 0 && grid[x][y - 1] == '1')
            {
                q.push({x, y - 1});
                grid[x][y - 1] = '0';
            }
            if(y < grid[0].size() - 1 && grid[x][y + 1] == '1')
            {
                q.push({x, y + 1});
                grid[x][y + 1] = '0';
            }
        }
}
```

SurroundedRegions

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

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

```
For example,
X X X X
X O O X
X X O X
X O X X
After running your function, the board should be:

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

```
方法1：利用并查集
代码来自 leetcode

class UF
{
private:
    int* id;     // id[i] = parent of i
    int* rank;  // rank[i] = rank of subtree rooted at i (cannot be more than 31)
    int count;    // number of components
public:
    UF(int N)
    {
        count = N;
        id = new int[N];
        rank = new int[N];
        for (int i = 0; i < N; i++) {
            id[i] = i;
            rank[i] = 0;
        }
    }
    ~UF()
    {
        delete [] id;
        delete [] rank;
    }
    int find(int p) {
        while (p != id[p]) {
            id[p] = id[id[p]];    // 路径压缩
            p = id[p];
        }
        return p;
    }
    int getCount() {
        return count;
    }
    bool connected(int p, int q) {
        return find(p) == find(q);
    }
    void connect(int p, int q) {
        int i = find(p);
        int j = find(q);
        if (i == j) return;
        if (rank[i] < rank[j]) id[i] = j;
        else if (rank[i] > rank[j]) id[j] = i;
        else {
            id[j] = i;
            rank[i]++;
        }
        count--;
    }
};

class Solution {
public:
    void solve(vector<vector<char>> &board) {
        int n = board.size();
        if(n==0)    return;
        int m = board[0].size();
        UF uf = UF(n*m+1);

        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if((i==0||i==n-1||j==0||j==m-1)&&board[i][j]=='O') // if a 'O' node is on
                    uf.connect(i*m+j,n*m);
                else if(board[i][j]=='O') // connect a 'O' node to its neighbour 'O' node
                {
                    if(board[i-1][j]=='O')
                        uf.connect(i*m+j,(i-1)*m+j);
                    if(board[i+1][j]=='O')
                        uf.connect(i*m+j,(i+1)*m+j);
                    if(board[i][j-1]=='O')
                        uf.connect(i*m+j,i*m+j-1);
                    if(board[i][j+1]=='O')
                        uf.connect(i*m+j,i*m+j+1);
                }
            }
        }

        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(!uf.connected(i*m+j,n*m)){ // if a 'O' node is not connected to the du
                    board[i][j]='X';
                }
            }
        }
    }
};



方法2：广度优先搜索

void bfsBoundary(vector<vector<char> >& board, int w, int l)
```

```
{
    int width = board.size();
    int length = board[0].size();
    deque<pair<int, int> > q;
    q.push_back(make_pair(w, l));
    board[w][l] = 'B';
    while (!q.empty()) {
        pair<int, int> cur = q.front();
        q.pop_front();
        pair<int, int> adjs[4] = {{cur.first-1, cur.second},
            {cur.first+1, cur.second},
            {cur.first, cur.second-1},
            {cur.first, cur.second+1}};
        for (int i = 0; i < 4; ++i)
        {
            int adjW = adjs[i].first;
            int adjL = adjs[i].second;
            if ((adjW >= 0) && (adjW < width) && (adjL >= 0)
                    && (adjL < length)
                    && (board[adjW][adjL] == 'O')) {
                q.push_back(make_pair(adjW, adjL));
                board[adjW][adjL] = 'B';
            }
        }
    }
}

void solve(vector<vector<char> > &board) {
    int width = board.size();
    if (width == 0) //Add this to prevent run-time error!
        return;
    int length = board[0].size();
    if  (length == 0) // Add this to prevent run-time error!
        return;

    for (int i = 0; i < length; ++i)
    {
        if (board[0][i] == 'O')
            bfsBoundary(board, 0, i);

        if (board[width-1][i] == 'O')
            bfsBoundary(board, width-1, i);
    }

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

    for (int i = 0; i < width; ++i)
    {
        for (int j = 0; j < length; ++j)
        {
            if (board[i][j] == 'O')
                board[i][j] = 'X';
            else if (board[i][j] == 'B')
```

```
                    board[i][j] = 'O';
            }
        }
    }
```

WordLadder

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

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the word list

For example,

```
Given:
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",
return its length 5.
```

Note: Return 0 if there is no such transformation sequence. All words have the same length.

```
思路：

利用广度优先搜索，每次找和队列中字符串相差1的字符串 并添加到队列中，直到找到end字符串

 public:
    int ladderLength(string start, string end, unordered_set<string> &dict)
{

    if (dict.empty() || dict.find(start) == dict.end() || dict.find(end) == dict.end()) r

    //每次找和队列中字符串相差1的字符串 并添加到队列中
    queue<string> q;
        q.push(start);

    unordered_map<string,int> map_visit;//用来保存每个字符串是否被访问过以及需要改变的次数 用un_m
        map_visit[start] =1;
    unordered_set<string> left_dict = dict; //保存没有被访问过的字符串
    left_dict.erase(start);

    while(q.size()!=0)
    {
        string word = q.front();
        q.pop();
        auto it = left_dict.begin();
        while(it != left_dict.end())
        {
            if(oneCharDiff(*it,word))
            {
                map_visit[*it] = map_visit[word] + 1; //距离加1
                if(*it == end) return map_visit[*it];
                q.push(*it); //加到队列中
```

```
                  it = left_dict.erase(it);
              }else
              {
                it++;
              }
          }
      }

      return 0;//没有找到
  }

  inline bool oneCharDiff(const string& str1, const string& str2)
  {
      int diff =0;
      for(int i=0;i<str1.size();i++)
      {
          if(str1[i] !=str2[i])
              diff++;
          if(diff>1) break;
      }
      return diff == 1;
  }
```

WordLadder2

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:

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

```
Given:
start = "hit"
end = "cog"
dict = ["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.

```
方法1 是通过不断的找可能的字符与现有比较字符的距离是否为1


vector<vector<string>> findLadders(string start, string end, unordered_set<string> &dict)
          vector<vector<string> >ans;
          if(start == end) return ans;
          unordered_set<string>current , next;
          unordered_set<string> flag;
          unordered_map<string,vector<string> > father;

          current.insert(start);

          bool found = false;

          while(!current.empty() && !found) {
              //expand
              for(const auto &x : current) {
                  flag.insert(x);
              }

              for(auto x : current) {
                  for(int i = 0 ; i < x.size() ; ++i) {
                      for(int j = 'a' ; j <= 'z' ; ++j) {
                          if(x[i] == j) continue;
                          string tmp = x;
                          tmp[i] = j;
                          if(tmp == end) found = true;
                          if(dict.find(tmp) != dict.end() && flag.find(tmp) == flag.end()
                              next.insert(tmp);
                              father[tmp].push_back(x);
                          }
```

```
                    }
                }
            }
            //end expand

            current.clear();
            swap(current, next);
        }
        //start foudn father

        if(found) {
            vector<string> c;
            dfs(ans , father , c , start , end);
        }
        return ans;
    }
private:
    void dfs(vector<vector<string> >&ans,
             unordered_map<string,vector<string> >& father ,
             vector<string>& c ,
             string& start ,
             string& now) {

        c.push_back(now);
        if(now == start) {
            ans.push_back(c);
            reverse(ans.back().begin() , ans.back().end());
            c.pop_back();
            return;
        }
        auto que = father.find(now) -> second;
        for(auto& x : que) {
            dfs(ans , father , c , start , x);
        }
        c.pop_back();
    }
```

方法2 是通过不断的剩余字典中的字符串与现有比较字符的距离是否为1，若这个字典很大话时间会很长

```
vector<vector<string>> findLadders(string start, string end, unordered_set<string> &dict)
{
    vector<vector<string> >ans;
          if(start == end) return ans;
          unordered_set<string>current , next;
          unordered_set<string> flag;
          unordered_map<string,vector<string> > father;
          unordered_set<string> left_dict = dict;
          current.insert(start);

          bool found = false;

          while(!current.empty() && !found) {
              //expand
              for(const auto &x : current) {
                  left_dict.erase(x);
              }

              for(auto x : current) {
                  auto it = left_dict.begin();
```

```
                     while(it != left_dict.end())
                    {
                         if(oneCharDiff(*it,x))
                         {
                             father[*it].push_back(x);
                             if(*it == end) found =true;
                             next.insert(*it); //加到队列中
                             it++;
                         }else
                         {
                           it++;
                         }
                    }
               }
               //end expand

               current.clear();
               swap(current, next);
           }
           //start foudn father

           if(found) {
               vector<string> c;
               dfs( start , end,c,ans , father );
           }
           return ans;
  }

  inline bool oneCharDiff(const string& str1, const string& str2)
  {
      int diff =0;
      for(int i=0;i<str1.size();i++)
      {
          if(str1[i] !=str2[i])
              diff++;
          if(diff>1) break;
      }
      return diff == 1;
  }

  private:
  void dfs(string start, string now,
          vector<string>& c ,
          vector<vector<string> >&ans,
               unordered_map<string,vector<string> >& father)
  {
      c.push_back(now);
      if(start == now)
      {

          ans.push_back(c);
          reverse(ans.back().begin() , ans.back().end());
          c.pop_back();
          return;
      }

      auto vect = father.find(now)->second;
      for(auto& x : vect)
      {
          dfs(start , x , c , ans,father);
      }
      c.pop_back();
  }
```

二分搜索

FindPeakElement

A peak element is an element that is greater than its neighbors.

Given an input array where num[i] ≠ num[i+1], find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that num[-1] = num[n] = -∞.

For example, in array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

click to show spoilers.

Note: Your solution should be in logarithmic complexity.

题目要求时间复杂度在O(lgn)

```
方法1：标准的二分查找方法
int findPeakElement(const vector<int> &num)
{
        int low = 0, high = num.size() - 1;
        while (low < high ) {
            int mid = low + (high-low) / 2;
            if (num[mid] > num[mid - 1] && num[mid] > num[mid + 1])
                return mid;
            else if (num[mid] > num[mid + 1])
                    high = mid - 1;
                 else
                    low = mid + 1;
        }
        return  low;
 }


方法2

和方法一1思路一致
若 num[i-1] < num[i] > num[i+1], 那么 num[i] 是一个峰点
若 num[i-1] < num[i] < num[i+1], 那么 num[i+1...n-1] 一定包含一个峰点
若 num[i-1] > num[i] > num[i+1], 那么 num[0...i-1] 一定包含一个峰点
若 num[i-1] > num[i] < num[i+1], 那么两边都有可能包含峰值
但代码实际写起来和方法1是一致的
public int findPeakElement(int[] num) {
    return helper(num,0,num.length-1);
}

public int helper(int[] num,int start,int end){
    if(start == end){
        return start;
    }else if(start+1 == end){
        if(num[start] > num[end]) return start;
        return end;
    }else{

        int m = (start+end)/2;

        if(num[m] > num[m-1] && num[m] > num[m+1]){

            return m;

        }else if(num[m-1] > num[m] && num[m] > num[m+1]){

            return helper(num,start,m-1);

        }else{

            return helper(num,m+1,end);

        }

    }
}


方法3
always 寻找 局部最大的那个元素
int findPeakElement(const vector<int> &num)
{
        int low = 0;
        int high = num.size()-1;

        while(low < high)
        {
            int mid1 = low + (high-low) / 2;
            int mid2 = mid1+1;
            if(num[mid1] < num[mid2])
                low = mid2;
            else
                high = mid1;
        }
        return low;
}
```

SearchinRotatedSortedArray

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

```
方法1:
int search(vector<int>& nums, int target)
{
        int lo = 0;
        int hi = nums.size() - 1;
        while (lo < hi) { //基本也是 二分查找 但是要加特殊的判断 画个图 就比较容易理解
            int mid =lo+(hi-lo)/2;
            if (nums[mid] == target) return mid;

            if (nums[lo] <= nums[mid]) { //因为rotatded了 所以不一定哪个大
                if (target >= nums[lo] && target < nums[mid]) {
                    hi = mid - 1;
                } else {
                    lo = mid + 1;
                }
            } else {
                if (target > nums[mid] && target <= nums[hi]) {
                    lo = mid + 1;
                } else {
                    hi = mid - 1;
                }
            }
        }
        return nums[lo] == target ? lo : -1;
}

方法2
先找到旋转后数组中元素最小的值的下标low, 这样数组旋转(移动)的距离就是low

之后用传统的二分查找来查询元素target, 每次通过
            mid=lo+(hi-lo)/2;
            realmid=(mid+rot)%nums.size();
找到实际的realmid来进行二分

int search(vector<int>& nums, int target)
{
        int lo=0,hi=nums.size()-1;

        while(lo<hi){
            int mid=lo+(hi-lo)/2;
            if(nums[mid]>nums[hi]) lo=mid+1;
            else hi=mid;
        }//先找到拐点, 然后再用传统的二分查找方法
        int rot=lo;
        lo=0;hi=nums.size()-1;

        while(lo<=hi){
            int mid=lo+(hi-lo)/2;
            int realmid=(mid+rot)%nums.size();
            if(nums[realmid]==target)return realmid;
            if(nums[realmid]<target)lo=mid+1;
            else hi=mid-1;
        }
        return -1;
}
```

SearchinRotatedSortedArray2

Follow up for "Search in Rotated Sorted Array": What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

```
思路：
基本思路是和SearchinRotatedSortedArray一致的。但是允许重复元素出现后该算法的最差时间复杂度变为了O(n)

考虑 11111113 rotate之后 变为  11311111
此时 A[m] == A[l] == A[h] 无法判断要查找的target落在哪个区域内因此只能让l++,继续重新计算mid

因此最差的时间复杂度为O(N)



bool search(vector<int>& A, int key) {

        int l = 0, r = A.size()-1;

        while (l <= r) {
            int m = l + (r - l)/2;
            if (A[m] == key) return true; //return m
            if (A[l] < A[m]) { //左边是排好序的
                if (A[l] <= key && key < A[m])
                    r = m - 1;
                else
                    l = m + 1;
            } else if (A[l] > A[m]) { //右边是排好序的
                if (A[m] < key && key <= A[r])
                    l = m + 1;
                else
                    r = m - 1;
            } else if(A[m] != A[r])
            {
                l=m+1;
            }else
                l++;
        }
        return false;
}
```

SearchInsertPosition

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

```
Here are few examples.
[1,3,5,6], 5 → 2
[1,3,5,6], 2 → 1
[1,3,5,6], 7 → 4
[1,3,5,6], 0 → 0
```

```
方法：
标准的二分查找...
int searchInsert(vector<int>& nums, int target)
{
        int left =0,right = nums.size()-1;
        while(left<=right)
        {
            int mid = left+(right-left)/2;

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

SearchforaRange

Given a sorted array of integers, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of O(log n).

If the target is not found in the array, return [-1, -1].

For example, Given [5, 7, 7, 8, 8, 10] and target value 8, return [3, 4].

```
思路：
先找区间的左边界，再找区间的右边界

vector<int> searchRange(int A[], int n, int target) {
        vector<int> res;
        if(n <= 0)
            return res;

        int leftIndex = lSearch(A, n, target);
        int rightIndex = rSearch(A, n, target);

        res.push_back(leftIndex);
        res.push_back(rightIndex);
        return res;
    }

    int lSearch(int A[], int n, int target) {
        int low = 0, high = n-1;
        while(low <= high) { // q1
            int mid = low+(high-low)/2;
            if(A[mid] < target) { // q2
                low = mid+1;
            }else{
                high = mid-1;
            }
        }
        if(A[low] != target)
            return -1;
        return low; // q3   若只有一个元素 最终A[low]= target and low>high
    }

    int rSearch(int A[], int n, int target) {
        int low = 0, high = n-1;
        while(low <= high) { // q1
            int mid = low+(high-low)/2;
            if(A[mid] > target) { // q2
                high = mid-1;
            }else{
                low = mid +1;
            }
        }
        if(A[high] != target)
            return -1;
        return high; // q3 若只有一个元素 最终A[high]= target and low>high 返回high
    }
```

FindMinimuminRotatedSortedArray

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

You may assume no duplicate exists in the array.

```
int findMin(vector<int>& nums)
{
        int left = 0;
        int right = nums.size()-1;

        while(left < right)
        {
            if(nums[left] <= nums[right])
                return nums[left];

            int mid = left +(right-left)/2;
            if(nums[mid] < nums[left])
                right = mid;
            else
                left = mid+1;
        }

        return nums[left];
}
```

FindMinimuminRotatedSortedArrayII

Follow up for "Find Minimum in Rotated Sorted Array": What if duplicates are allowed?

Would this affect the run-time complexity? How and why? Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

The array may contain duplicates.

```
思路：
同理Search in Rotate array 2
只有 mid left right 都相等的时候 才无法判断 此时只能left ++

int findMin(vector<int>& nums) {
        int left =0;
        int right = nums.size()-1;
        while(left < right)
        {
            if(nums[left] < nums[right])
                return nums[left];
            int mid = left + (right -left)/2;

            if(nums[mid] >nums[left])
                left = mid+1;
            else if( nums[mid] <nums[left])
                right = mid;
            else if(nums[right] != nums[mid])//[mid] == [left]
                left = mid+1;
            else //都相等
                left++;
        }

        return nums[left];
}
```

Searcha2DMatrix

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

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

```
Consider the following matrix:

[
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
Given target = 3, return true.
```

```
The first integer of each row is greater than the last integer of the previous row
表明了这个数组其实就是一个排好序的数列，只不过用二维数组的形式表示了出来，所以我们方法一 可以用两次二分查

方法1 两次二分查找

bool searchMatrix(vector<vector<int> > &matrix, int target) {
        if(matrix.size()<=0 || matrix[0].size()<=0)
            return false;
        int left = 0;
        int right = matrix.size()-1;

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

            if(matrix[mid][0]>target)
            {
                right = mid-1;
            }else if(matrix[mid][0] < target)
            {
                left = mid+1;
            }else{
                return true;
            }
        }

        if(right == -1)
            return false;
        else{

            int row = right;
            int left = 0;
            int right = matrix[row].size()-1;

            while(left<=right)
            {
                int mid = left+(right-left)/2;
                if(matrix[row][mid]>target)
                {
                    right = mid-1;
                }else if(matrix[row][mid] < target)
                {
                    left = mid+1;
                }else{
                    return true;
                }
            }

            return false;
        }
    }


方法2 直接看成一个连贯的array

bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int n = matrix.size();
        int m = matrix[0].size();
        int l = 0, r = m * n - 1;
        while (l <= r){
            int mid = l+((r-l)>>1);
            int row = mid /m;
            int col = mid %m;
            if(matrix[row][col] == target)
                return true;
            else if (matrix[row][col] < target)
                l = mid + 1;
            else
                r = mid-1;
        }
        return false;
    }
```

Searcha2DMatrix2

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

Integers in each row are sorted in ascending from left to right. Integers in each column are sorted in ascending from top to bottom. For example,

```
Consider the following matrix:

[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
Given target = 5, return true.

Given target = 20, return false.
```

```
 bool searchMatrix(vector<vector<int>>& matrix, int target)
 {
    return searchMatrix(matrix, target, 0, matrix.size() - 1);
 }

    bool searchMatrix(vector<vector<int>>& matrix, int target, int top, int bottom)
    {
        if (top > bottom)
            return false;

        int mid = top + (bottom - top) / 2;
        if (matrix[mid].front() <= target && target <= matrix[mid].back())
            if (searchVector(matrix[mid], target)) return true;

        if (searchMatrix(matrix, target, top, mid - 1)) return true;
        if (searchMatrix(matrix, target, mid + 1, bottom)) return true;

        return false;
    }

    bool searchVector(vector<int>& v, int target)
    {
        int left = 0, right = v.size() - 1;

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

```
                left = mid + 1;
            else
                right = mid - 1;
        }

        return false;
    }
```

位运算

- BitwiseANDofNumbersRange
- Numberof1Bits
- Power of Two
- ReverseBit
- SingleNumber
- SingleNumber2
- SingleNumber3
- MissingNumber

BitwiseANDofNumbersRange

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

For example, given the range [5, 7], you should return 4.

```
方法1
找最左边相同的部门 其余剩下的总是AND为0
int rangeBitwiseAnd(int m, int n)
{
        int i = 0;
        while(m!=n){
            m=m>>1;
            n=n>>1;
            ++i;
        }
        return m<<i;
}

方法2
不断把n的最后一位为1的位变成0 看是否和m相等，应该更快一些
int rangeBitwiseAnd(int m, int n)
{
    while(n > m) {
       n = n & (n-1);
    }
    return n;
}
```

Numberof1Bits

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer '11' has binary representation 00000000000000000000000000001011, so the function should return 3.

思路：

这是老题目了,n&(n-1) 是将最低位的为1的位变成0

```
int hammingWeight(uint32_t n)
{
        int res=0;
        while(n)
        {
            n = n&(n-1);
            res++;
        }
        return res;
}
```

Power of Two

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

```
思路：
判断一个数的是否是2的次幂 等价于求二进制中1的位数的数目是否为1
bool isPowerOfTwo(int n)
{
        if(n == INT_MIN)
            return false;
        int res=0;
        while(n)
        {
            n = n&(n-1);
            res++;
        }

        return res == 1;
}
```

ReverseBit

Reverse bits of a given 32 bits unsigned integer.

For example, given input 43261596 (represented in binary as 00000010100101000001111010011100), return 964176192 (represented in binary as 00111001011110000010100101000000).

Follow up: If this function is called many times, how would you optimize it?

我从leetcode上搜集了多个方法

```
//1
uint32_t reverseBits(uint32_t n)
{
        uint32_t m=0;
        for(int i=0;i<32;i++){
            m<<=1;
            m = m|(n & 1);//每次将二进制串的最后一位放到m的最后一位
            n>>=1;
        }
        return m;
}
//2
uint32_t reverseBits(uint32_t n)
{
    uint32_t bin=0;
    for (i = 0; i < 32; i++)
         bin+=(n >> i & 1)<<(31-i);
    return bin;
}
//3
uint32_t reverseBits(uint32_t n)
{
        const int size = 32;
        bitset<size> bSet(n);
        for(int i=0; i < size/2; i++){
            int temp = bSet[i];
            bSet[i] = bSet[size - i - 1];
            bSet[size - i -1] = temp;
        }
        return (uint32_t) bSet.to_ulong();
}

//4 Lookup Table with O(1)
uint32_t reverseBits(uint32_t n)
{
    int rbitTable[256] = {
        #define R2(n) n, n + 2*64, n + 1 * 64, n + 3 * 64
        #define R4(n) R2(n), R2(n + 2 * 16), R2(n + 1 * 16), R2(n + 3 * 16)
        #define R6(n) R4(n), R4(n + 2 * 4), R4(n + 1 * 4), R4(n + 3 * 4)
        R6(0), R6(2), R6(1), R6(3)
    };
    uint32_t v;
    unsigned char *p = (unsigned char*)&n;
    unsigned char *q = (unsigned char*)&v;
    q[0] = rbitTable[p[3]];
    q[1] = rbitTable[p[2]];
    q[2] = rbitTable[p[1]];
    q[3] = rbitTable[p[0]];
    return v;
}

//5
uint32_t reverseBits(uint32_t n)
{
        n = (n >> 16) | (n << 16);
        n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8);
        n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4);
        n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);
        n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1);
        return n;
}


//6
//cache
private final Map<Byte, Integer> cache = new HashMap<Byte, Integer>();
public int reverseBits(int n) {
    byte[] bytes = new byte[4];
    for (int i = 0; i < 4; i++) // convert int into 4 bytes
        bytes[i] = (byte)((n >>> 8*i) & 0xFF);
    int result = 0;
    for (int i = 0; i < 4; i++) {
        result += reverseByte(bytes[i]); // reverse per byte
        if (i < 3)
            result <<= 8;
    }
    return result;
}

private int reverseByte(byte b) {
    Integer value = cache.get(b); // first look up from cache
    if (value != null)
        return value;
    value = 0;
    // reverse by bit
    for (int i = 0; i < 8; i++) {
        value += ((b >>> i) & 1);
        if (i < 7)
            value <<= 1;
    }
    cache.put(b, value);
    return value;
}
```

SingleNumber

Given an array of integers, every element appears twice except for one. Find that single one.

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

```
思路：将数组里面的元素异或一遍 相同的两个数会异或为0，只留下出现一次的那个数

int singleNumber(int A[], int n)
{
        for (int i = 1; i < n; ++i)
                A[0] ^= A[i];
             return A[0];
}
```

SingleNumber2

Given an array of integers, every element appears three times except for one. Find that single one.

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

```
int singleNumber(vector<int>& nums)
{
        int three=0,two=0,one=0;

        for(auto num:nums)
        {
            three = two&num;
            two = two | one&num;
            one = one | num;

            two = two & ~three;  //把出现3次的 从2次和1次中去掉
            one = one & ~three;

        }
        return one;
}
```

SingleNumber3

Given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

For example:

Given nums = [1, 2, 1, 3, 2, 5], return [3, 5].

Note: The order of the result is not important. So in the above example, [5, 3] is also correct. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

```
思路：

先求得 a^b
找到a与b最低不相同的位
按照这位 值 的0或者1 分成两组
异或 一遍 即可求得a与b

vector<int> singleNumber(vector<int>& nums)
{
         int diff = accumulate(nums.begin(), nums.end(), 0, bit_xor<int>());
        diff &= -diff;//最低不相同的位对应的值
        vector<int> rets(2, 0);
        for (int num : nums)
            rets[!(num & diff)] ^= num;
        return rets;
}
```

MissingNumber

Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.

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

Note: Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

```
思路：
可参考题目FirstMissingPositive
对于数组中每个元素

若是  nums[i]>=n 跳过这个元素
若是  nums[i]<n  并且nums[i] != i 那么将 num[i] swap到第i个位置上， swap来的新元素若还是 ！=i 继

最后遍历一遍数组若是 nums[i] != i 返回i


int missingNumber(vector<int>& nums)
{

        int n = nums.size();
        for(int i=0;i<n;i++)
        {
            while(nums[i]<n && nums[i] != i)
                swap(nums[i],nums[nums[i]]);
        }

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

C语言 关于几个字符串处理的函数，面试中会有问到的可能性

```
求字符串的长度
int strlen(const char* str)
{
    assert(NULL != str);
    int len;
    while( (*str ++) != '\0') len++;
    return len;
}

比较两个字符串
int strcmp(const char* str1,const char* str2)
{
    assert(NULL != str1 && NULL!=str2);
    int ret =0;
    while(!(ret = *(unsigned char*)str1 - *(unsigned char*)str2) && *str1)
    {
        str1++;
        str2++;
    }
    if(ret<0) return -1;
    else if(ret>0) return 1;

    return ret;
}

拼接字符串 将strSrc拼接到strDest
char *strcat(char *strDest,const char* strSrc)
{
    char *address = strDest;
    assert(NULL!=strDest && NULL != strSrc);
    while(*strDest) strDest++;
    while(*strDest++ = *strSrc++);
    return address;
}

字符串复制 将strSrc的内容复制到以strDest为起始位置的地方
char* strcpy(char* strDest,const char* strSrc )
{
    assert(NULL!=strDest && NULL != strSrc);
    char *strD = strDest;
    while( (*strDest++ = *strSrc++) !='\0' );
    return strD;
}
```

分治

图

- CloneGraph
- CourseSchedule
- CourseScheduleII

CloneGraph

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

OJ's undirected graph serialization: Nodes are labeled uniquely.

We use # as a separator for each node, and , as a separator for node label and each neighbor of the node. As an example, consider the serialized graph {0,1,2#1,2#2,2}.

The graph has a total of three nodes, and therefore contains three parts as separated by #.

First node is labeled as 0. Connect node 0 to both nodes 1 and 2. Second node is labeled as 1. Connect node 1 to node 2. Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle. Visually, the graph looks like the following:

```
       1
      / \
     /   \
    0 --- 2
         / \
         \_/
```

一下代码来自leetcode的讨论区

```
方法1：DFS
一个全局hashmap hash[node]; 用来保存已经生成的新的节点
unordered_map<UndirectedGraphNode *,UndirectedGraphNode *> hash;

UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node)
{
        if(!node)
            return NULL;

        if(hash.find(node) == hash.end())
        {
            hash[node]= new UndirectedGraphNode(node->label);

            for(auto child: node->neighbors)
            {
                UndirectedGraphNode *nchild = cloneGraph(child);
                hash[node]->neighbors.push_back(nchild);
            }
        }

        return hash[node];
}


方法2：BFS
```

```
UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node)
{
        unordered_map<UndirectedGraphNode*, UndirectedGraphNode*> record;
        if(node == NULL)
            return node;

        deque<UndirectedGraphNode*> queue;
        queue.push_back(node);

        while(!queue.empty()) {
            UndirectedGraphNode *nextNode = queue.front();
            queue.pop_front();

            if(!record.count(nextNode)) {
                UndirectedGraphNode *newNode = new UndirectedGraphNode(nextNode->label);
                record[nextNode] = newNode;
            }
            for(int i = 0; i < nextNode->neighbors.size() ; i ++) {
                UndirectedGraphNode *childNode = nextNode->neighbors[i];
                if(!record.count(childNode)) {
                    UndirectedGraphNode *newNode = new UndirectedGraphNode(childNode->lab
                    record[childNode] = newNode;
                    queue.push_back(childNode);
                }
                record[nextNode]->neighbors.push_back(record[childNode]);
            }
        }
        return record[node];
    }
```

CourseSchedule

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

For example:

2, [[1,0]] There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

2, [[1,0],[0,1]] There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

```
思路：
整个题目就是判断图是否有正确的拓扑路径的问题

求拓扑路径有两种方式

1. DFS
2. BFS (Kahn算法)


方法1：DFS

DFS过程中若是产生了回边 那么就是产生了环路，那么对应课程就是不能完成的

bool DFS(vector<unordered_set<int>>& table,vector<int> &color,int num)
{
        color[num] =-1; //正在被访问还没有访问完事
         for(auto  i: table[num])
        {
            if(color[i] ==-1)
                return false;
            if(color[i]==0 && !DFS(table,color,i))
                return false;
        }
        color[num] =1; //访问完成
        return true;
}


bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites) {

        vector<unordered_set<int>> table(numCourses);

        vector<int> color(numCourses, 0);
        for(auto s:prerequisites)//构造图的过程
           if(table[s.second].find(s.first) == table[s.second].end())
            {
                table[s.second].insert(s.first);
            }
        }

        for(int i =0;i<numCourses ;i++)
        {
            if(color[i] ==0)
                if(!DFS(table,color,i))
                    return false;
        }

        return true;
}
```

方法2：BFS
不断找当前入度数为0的节点，将该节点放到拓扑队列的末尾（同时可以求出整个拓扑队列）
最终队列里节点的数目和所有节点的数目相同的话那么就是可以完成课程。

```
bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites)
{
        vector<unordered_set<int>> table(numCourses);

        vector<int> de(numCourses, 0);

        for(auto s:prerequisites)
        {
           if(table[s.second].find(s.first) == table[s.second].end())
            {
                table[s.second].insert(s.first);
                de[s.first]++;
            }
        }

        queue<int> queue;
        for (int i = 0; i < numCourses; ++ i)
            if (de[i] == 0)
                queue.push(i);

        int count = numCourses;

        while(!queue.empty())
        {
            int cur = queue.front();
            queue.pop();
            count --;

            for(auto  i: table[cur])
            {
                de[i]--;
                if (de[i] == 0)
                    queue.push(i);
            }
        }
        return count ==0;

}
```

CourseScheduleII

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

For example:

2, [[1,0]] There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1]

4, [[1,0],[2,0],[3,1],[3,2]] There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is [0,1,2,3]. Another correct ordering is[0,2,1,3].

```
思路：
承接上题，此题就是把拓扑排序给求出来，用BFS比较直观，但DFS也可以实现求拓扑排序

下述方法用类似邻接表的结构存储图
用BFS类似的方法进行拓扑排序

vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites)
{
        vector<unordered_set<int>> table(numCourses);

        vector<int> de(numCourses,0);

        vector<int> result;

        for(auto pair :prerequisites )
        {
            if(table[pair.second].find(pair.first) == table[pair.second].end())
            {
                table[pair.second].insert(pair.first);
                de[pair.first]++;
            }
        }

        queue<int> queue;
        for(int i=0;i<numCourses;i++)
        {
           if(de[i] ==0)
            queue.push(i);
        }
        int count = numCourses;

        while(!queue.empty())
        {
            int cur = queue.front();
            queue.pop();
            result.push_back(cur);
            count--;
            for(auto s:table[cur])
            {
                de[s]--;
                if(de[s] == 0)
                    queue.push(s);
            }
        }

        if(count ==0)
            return result;
        else{
            vector<int> eresult;
            return eresult;
        }
}
```

贪心

- Best Time to Buy and Sell Stock II
- Candy
- GasStation
- JumpGame
- JumpGame2

Best Time to Buy and Sell StockII

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

```
思路：
典型的贪心算法题目，能够进行尽可能多的交易次数大大降低了这个题目的难度
我们只有看某一天比上一天的股票价格高我们就能够卖掉赚钱...

int maxProfit(vector<int>& prices)
{
        int result =0;
        int n =prices.size();
        if(n<=0)
            return result;
        int min =prices[0];

        for(int i=1;i<n;i++)
        {
            if(prices[i]>min)
                result += prices[i]-min; //只要涨了就卖掉
            min = prices[i]; //重新设置低价
        }
        return result;
}



或者更简单粗暴的代码
直接把相邻的正差值  全部加起来 就行
public int maxProfit(int[] prices)
{
        int total = 0;
        for (int i=0; i< prices.length-1; i++) {
            if (prices[i+1]>prices[i]) total += prices[i+1]-prices[i];
        }

        return total;
}
```

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:

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

```
以下代码摘自leetcode讨论区
方法1：
int candy(vector<int>& ratings) {
        int nCandyCnt = 0;///Total candies
    int nSeqLen = 0;  /// Continuous ratings descending sequence length
    int nPreCanCnt = 1; /// Previous child's candy count
    int nMaxCntInSeq = nPreCanCnt;
    if(ratings.begin() != ratings.end())
    {
        nCandyCnt++;//Counting the first child's candy.
        for(vector<int>::iterator i = ratings.begin()+1; i!= ratings.end(); i++)
        {
            // if r[k]>r[k+1]>r[k+2]...>r[k+n],r[k+n]<=r[k+n+1],
            // r[i] needs n-(i-k)+(Pre's) candies(k<i<k+n)
            // But if possible, we can allocate one candy to the child,
            // and with the sequence extends, add the child's candy by one
            // until the child's candy reaches that of the prev's.
            // Then increase the pre's candy as well.

            // if r[k] < r[k+1], r[k+1] needs one more candy than r[k]
            //
            if(*i < *(i-1))
            {
                //Now we are in a sequence
                nSeqLen++;
                if(nMaxCntInSeq == nSeqLen)
                {
                    //The first child in the sequence has the same candy as the prev
                    //The prev should be included in the sequence.
                    nSeqLen++;
                }
                nCandyCnt+= nSeqLen;
                nPreCanCnt = 1;
            }
            else
            {
                if(*i > *(i-1))
                {
                    nPreCanCnt++;
                }
                else
                {
                    nPreCanCnt = 1;
                }
                nCandyCnt += nPreCanCnt;
                nSeqLen = 0;
                nMaxCntInSeq = nPreCanCnt;
            }
        }
    }
    return nCandyCnt;
}

方法2：

int candy(vector<int> &ratings) {

    int size=ratings.size();
     if(size<=1)
         return size;
     vector<int> num(size,1);
     for (int i = 1; i < size; i++)
     {
         if(ratings[i]>ratings[i-1])
             num[i]=num[i-1]+1;
     }
     for (int i= size-1; i>0 ; i--)
     {
         if(ratings[i-1]>ratings[i])
             num[i-1]=max(num[i]+1,num[i-1]);
     }
     int result=0;
     for (int i = 0; i < size; i++)
     {
         result+=num[i];
     }
     return result;
    }
```

GasStation

There are N gas stations along a circular route, where the amount of gas at station i is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

```
方法1：
思路：需要从每个点出发来判断是否能围着circle绕一圈,每个点花费时间O(N),总时间复杂度为O(N^2)
这显然不是我们希望的算法时间复杂度
有一个规律我们可以利用： 若是一个station i出发，但是在
x之前停了下来，那么说明了从i+1....x-1之间点 都不能到达x点。
为什么？ 若是 从i可以到达i+1...x-1之间的某一个点k,并且从k能够到达x, 那么说明i可以到达x点. 矛盾了

因此只能说明i+1....x-1之间点 都不能到达x点。
那么点i直接跳到了点x而不是i+1,从而整个算法额时间复杂度为O(N)

一下代码摘自leetcode讨论区
int canCompleteCircuit(vector<int>& gas, vector<int>& cost)
{
        int i, j, n = gas.size();


        // start from station i
        for (i = 0; i < n; i += j) {
            int gas_left = 0;
            // forward j stations
            for (j = 1; j <= n; j++) {
                int k = (i + j - 1) % n;
                gas_left += gas[k] - cost[k];
                if (gas_left < 0)
                    break;
            }
            if (j > n)
                return i;
        }
        return -1;
}


方法2：

int canCompleteCircuit(vector<int> &gas, vector<int> &cost)
{
        int start = gas.size()-1;
        int end = 0;
       int sum = gas[start] - cost[start];
       while (start > end) {
          if (sum >= 0) {
             sum += gas[end] - cost[end];
             ++end;
          }
```

```
            else {
               --start;
               sum += gas[start] - cost[start];
            }
        }
        return sum >= 0 ? start : -1;
}

方法3：
可转换成求循环数组的最大子数组和的问题
```

JumpGame

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.

Determine if you are able to reach the last index.

For example:

```
A = [2,3,1,1,4], return true.

A = [3,2,1,0,4], return false.
```

```
方法基本都是贪心的思想

方法1
 bool canJump(int A[], int n) {
        if(n==0||n==1){
            return true;
        }
        int maxstep=A[0];
        for(int i=1;i<n;i++){
            if(maxstep==0) return false;
            maxstep--;
            if(maxstep<A[i]){
                maxstep=A[i];
            }
            if(maxstep+i>=n-1){
                return true;
            }
        }
    }

方法2

bool canJump(vector<int>& nums)
{
        int truepos=nums.size()-1;//the lowest starting point that you can reach the end
        for(int i=nums.size()-2;i>=0;i--)
            truepos=(i+nums[i])>=truepos?i:truepos;
        return(truepos==0);
}

方法3
bool canJump(vector<int>& nums)
{
        if(nums.size()<=1) return true;

        int maxreach =0;
        for(int i=0;i<nums.size() && maxreach<nums.size()-1 && i<=maxreach;i++)
        {
            maxreach =max(i+nums[i],maxreach);
        }

        if(maxreach <nums.size()-1)
            return false;
        else
            return true;
}
```

JumpGame2

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.)

```
承接上题，这题是要求出能达到数目末尾需要的最少的步数

方法1
每次找到i点所能到达范围内每个点 能达到最远距离的那个点(index)

int jump(vector<int>& nums)
{
        int n = nums.size();
        if(n ==1) return 0;
        int ret=0;
        int j=1;
        int max_index=0;
        for(int i=0;i<n-1 && i+nums[i] <n-1;)
        {
            max_index =j;
            for(;j<=i+nums[i];j++)
            {
                max_index = max_index+nums[max_index] > j+nums[j] ? max_index:j;
            }

            i= max_index;
            ret++;
        }

        return ++ret;
}

方法二
每次判断一个新的index是否在 curRch范围内
curRch表示每一步所能走到的最远的地方
int jump(int A[], int n)
{
         int ret = 0;
        int curMax = 0;
        int curRch = 0;
        for(int i = 0; i < n; i ++)
        {
            if(curRch < i)
            {
                ret ++;
                curRch = curMax;
            }
            curMax = max(curMax, A[i]+i);
        }
        return ret;
}
```

哈希

IsomorphicStrings

Given two strings s and t, determine if they are isomorphic.

Two strings are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

For example, Given "egg", "add", return true.

Given "foo", "bar", return false.

Given "paper", "title", return true.

```
bool isIsomorphic(string s, string t)
{
        char map_s[128] = { 0 };
        char map_t[128] = { 0 };
        int len = s.size();
        for (int i = 0; i < len; ++i)
        {
            if (map_s[s[i]]!=map_t[t[i]]) return false;
            map_s[s[i]] = i+1;
            map_t[t[i]] = i+1;
        }
        return true;
}
```

SubstringwithConcatenationofAllWords

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).
```

```
 vector<int> findSubstring(string S, vector<string> &L)
{
         vector<int> ans;
        int n = S.size(), cnt = L.size();
        if (n <= 0 || cnt <= 0) return ans;

        // init word occurence
        unordered_map<string, int> dict;
        for (int i = 0; i < cnt; ++i) dict[L[i]]++;

        // travel all sub string combinations
        int wl = L[0].size();
        for (int i = 0; i < wl; ++i) {
            int left = i, count = 0;
            unordered_map<string, int> tdict;
            for (int j = i; j <= n - wl; j += wl) {
                string str = S.substr(j, wl);
                // a valid word, accumulate results
                if (dict.count(str)) {
                    tdict[str]++;
                    if (tdict[str] <= dict[str])
                        count++;
                    else {
                        // a more word, advance the window left side possiablly
                        while (tdict[str] > dict[str]) {
                            string str1 = S.substr(left, wl);
                            tdict[str1]--;
                            if (tdict[str1] < dict[str1]) count--;
                            left += wl;
                        }
                    }
                    // come to a result
                    if (count == cnt) {
                        ans.push_back(left);
                        // advance one word
                        tdict[S.substr(left, wl)]--;
                        count--;
                        left += wl;
                    }
                }
                // not a valid word, reset all vars
                else {
                    tdict.clear();
                    count = 0;
                    left = j + wl;
                }
            }
        }

        return ans;
}
```

TwoSum

Given an array of integers, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

Input: numbers={2, 7, 11, 15}, target=9 Output: index1=1, index2=2

思路：编程之美上的一道题目,利用HASH可以保证时间复杂度在O(N)

```
vector<int> twoSum(vector<int>& nums, int target)
{
        vector<int> result;
        unordered_map<int,int> hash;

        for(int i=0;i<nums.size();i++)
        {
            int findN = target -nums[i];
            if(hash.find(findN) != hash.end())
            {
                result.push_back(hash[findN]);
                result.push_back(i+1);
                return result;
            }
            hash[nums[i]] = i+1;
        }

        return result;
}
```

RepeatedDNASequences

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

```
For example,

Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",

Return:
["AAAAACCCCC", "CCCCCAAAAA"].
```

```
思路：要找到所有长度为10的出现两次以上的DNA串
观察ATCG字符的二进制
A:01000001
T:01010100
C:01000011
G:01000111

根据最后三位就可以辨别出来，这样一段段字符串是用最后三位拼接起来

利用 s[i] & 7 得到字符的最后三位,再利用hahsmap来保存已出现过的字符串


vector<string> findRepeatedDnaSequences(string s) {
    unordered_map<int, int> m;
    vector<string> r;
    for (int t = 0, i = 0; i < s.size(); i++)
        if (m[t = t << 3 & 0x3FFFFFFF | s[i] & 7]++ == 1)
            r.push_back(s.substr(i - 9, 10));
    return r;
}
```

ContainsDuplicate

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

```
方法1：
bool containsDuplicate(vector<int>& nums) {
        return nums.size() > set<int>(nums.begin(), nums.end()).size();
    }

方法2：
bool containsDuplicate(vector<int>& nums)
{
        int size=nums.size();
        sort(nums.begin(),nums.end());
        nums.erase(unique(nums.begin(),nums.end()),nums.end());
        return (size!=nums.size());
}

上述两个方法都是利用了STL里面的方法，常规方法用hashmap来保存每一个数是否出现过也是可以的
```

ContainsDuplicate2

Given an array of integers and an integer k, find out whether there there are two distinct indices i and j in the array such that nums[i] = nums[j] and the difference between i and j is at most k.

```
思路：
距离为k之内的 数据不允许重复
大于K之后把之前的数据从set里面删除掉
bool containsNearbyDuplicate(vector<int>& nums, int k)
{
        unordered_set<int> s;

       if (k <= 0) return false;
       if (k >= nums.size()) k = nums.size() - 1;

       for (int i = 0; i < nums.size(); i++)
       {
           if (i > k) s.erase(nums[i - k - 1]);
           if (s.find(nums[i]) != s.end()) return true;
           s.insert(nums[i]);
       }

       return false;
}
```

ValidSudo

Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules.

The Sudoku board could be partially filled, where empty cells are filled with the character '.'.

A partially filled sudoku which is valid.

Note: A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

```
思路：
只是简单的检查一下数独板上的数字是否冲突
used1 表示 对应的数字是否在某行出现过
used2 表示 对应的数字是否在某列出现过
used3 表示 对应的数字是否在某小九宫格出现过

bool isValidSudoku(vector<vector<char> > &board)
{
      int used1[9][9] = {0}, used2[9][9] = {0}, used3[9][9] = {0};

      for(int i = 0; i < board.size(); ++ i)
            for(int j = 0; j < board[i].size(); ++ j)
                if(board[i][j] != '.')
                {
                    int num = board[i][j] - '0' - 1, k = i / 3 * 3 + j / 3;
                    if(used1[i][num] || used2[j][num] || used3[k][num])
                        return false;
                    used1[i][num] = used2[j][num] = used3[k][num] = 1;
                }

      return true;
}
```

Anagrams

Given an array of strings, return all groups of strings that are anagrams.

Note: All inputs will be in lower-case.

```
先对每个字符串进行排序，存到map中 然后找size>2的数字

vector<string> anagrams(vector<string>& strs) {
        vector<string> result;
        unordered_map<string,vector<string>> hash;

        for(auto &str : strs)
        {
            string temp = str;
            sort(str.begin(),str.end());
            hash[str].push_back(temp);
        }

        for(auto map : hash)
        {
            if(map.second.size()>1)
                result.insert(result.end(),map.second.begin(),map.second.end());
        }

        return result;
    }
```

Valid Anagram

Given two strings s and t, write a function to determine if t is an anagram of s.

```
For example,
s = "anagram", t = "nagaram", return true.
s = "rat", t = "car", return false.
```

Note: You may assume the string contains only lowercase alphabets.

```
bool isAnagram(string s, string t)
{
        if (s.length() != t.length()) return false;
        int n = s.length();
        int counts[26] = {0};
        for (int i = 0; i < n; i++) {
            counts[s[i] - 'a']++;
            counts[t[i] - 'a']--;
        }
        for (int i = 0; i < 26; i++)
            if (counts[i]) return false;
        return true;
}
```

堆

- MergeKSortList

MergeKSortedList

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

```
方法1：朴素方法  O(n*k) ?\\O(n*k*k)
ListNode *mergeKLists(vector<ListNode *> &lists) {
        if(lists.empty()){
            return nullptr;
        }
        while(lists.size() > 1){
            lists.push_back(mergeTwoLists(lists[0], lists[1]));
            lists.erase(lists.begin());
            lists.erase(lists.begin());
        }
        return lists.front();
    }

ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        if(l1 == nullptr){
            return l2;
        }
        if(l2 == nullptr){
            return l1;
        }
        if(l1->val <= l2->val){
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        }
        else{
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
}


方法2：优先队列  O(n*k*lg(n))

public ListNode mergeKLists(List<ListNode> lists)
{
        if (lists==null||lists.size()==0) return null;

        PriorityQueue<ListNode> queue= new PriorityQueue<ListNode>(lists.size(),new Compa
            @Override
            public int compare(ListNode o1,ListNode o2){
                if (o1.val<o2.val)
                    return -1;
                else if (o1.val==o2.val)
                    return 0;
                else
                    return 1;
            }
        });

        ListNode dummy = new ListNode(0);
        ListNode tail=dummy;

        for (ListNode node:lists)
            if (node!=null)
                queue.add(node);

        while (!queue.isEmpty()){
            tail.next=queue.poll();
            tail=tail.next;

            if (tail.next!=null)
                queue.add(tail.next);
        }
        return dummy.next;
}
```

字符串

TextJusttification

Given an array of words and a length L, format the text such that each line has exactly L characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly L characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

```
For example,
words: ["This", "is", "an", "example", "of", "text", "justification."]
L: 16.

Return the formatted lines as:
[
   "This    is    an",
   "example  of text",
   "justification.  "
]
```

Note: Each word is guaranteed not to exceed L in length.

```
vector<string> fullJustify(vector<string>& words, int L) {
         vector<string> res;
        for(int i = 0, k, l; i < words.size(); i += k) {
            for(k = l = 0; i + k < words.size() and l + words[i+k].size() <= L - k; k++)
                l += words[i+k].size();
            }
            string tmp = words[i];
            for(int j = 0; j < k - 1; j++) {
                if(i + k >= words.size()) tmp += " ";
                else tmp += string((L - l) / (k - 1) + (j < (L - l) % (k - 1)), ' ');
                tmp += words[i+j+1];
            }
            tmp += string(L - tmp.size(), ' ');
            res.push_back(tmp);
        }
        return res;
}
```

CountandSay

The count-and-say sequence is the sequence of integers beginning as follows: 1, 11, 21, 1211, 111221, ...

1 is read off as "one 1" or 11. 11 is read off as "two 1s" or 21. 21 is read off as "one 2, then one 1" or 1211. Given an integer n, generate the nth sequence.

Note: The sequence of integers will be represented as a string.

```
思路：
可以利用stringstream进行字符串处理

string convert(string say)
    {
        stringstream ss;
        int count =0;
        char pre =say[0];

        for(int i=0;i<=say.size();i++)
        {
            if(say[i]==pre)
                count++;
            else{
                ss<<count<<pre;
                pre = say[i];
                count =1;
            }
        }
        return ss.str();
    }


string countAndSay(int n)
{
        if(n<=0) return "";
        string result ="1";

        for(int i=1;i<n;i++)
        {
            result = convert(result);
        }
        return result;
}
```

ZigZagConversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P   A   H   N
A P L S I I G
Y   I   R
And then read line by line: "PAHNAPLSIIGYIR"
Write the code that will take a string and make this conversion given a number of rows:

string convert(string text, int nRows);
convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".
```

```
以下两种方法都摘自leetcode讨论区

方法1：
string convert(string s, int numRows)
{
        if (s == "" || numRows == 1) return s;
        vector<string> vecstr(numRows);
        string res;
        int bounce = 0, direct = 1;
        for (int i = 0; i<=s.size()-1; ++i) {
            vecstr[bounce].push_back(s[i]);
            if (bounce == numRows-1) direct = -1;
            else if (bounce == 0) direct = 1;
            bounce += direct;
        }
        for (int i = 0; i<=numRows-1; ++i) res += vecstr[i];
        return res;
}

方法2：

nRows:2
1,   3,  5   step: 2
2,   4,  6   step: 2

nRows:3
1,       5,      9   step:   4
2,   4,  6,  8,  10  step:2      2
3,       7,      11  step:   4

nRows:4
1,           7,          13  step:   6
2,       6,  8,      12, 14  step:4      2
3,   5       9,  11,     15  step:2      4
4,           10,         16  step:   6

nRows:5
```

```
1,                9,               17  step:   8
2,           8,  10,         16, 18  step:6      2
3,       7,      11,     15,     19  step:4      4
4,   6,          12, 14,         20  step:2      6
5,               13,             21  step:    8

class Solution {
public:
    string convert(string s, int nRows) {
        size_t len = s.size();
        if (nRows <= 1 || nRows >= len) { // Handle Special Case.
            return s;
        }
        const char* str = s.c_str();
        string* retval = new string();
        size_t maxStep = (nRows - 1) * 2;
        for (int r = 0; r < nRows; ++r) {
            size_t pos = r;
            if (0 == r || nRows - 1 == r) { // First Row And Last Row.
                while (pos < len) {
                    retval->push_back(str[pos]);
                    pos += maxStep;
                }
            } else { // Rows In The Middle.
                size_t step = 2 * r;
                while (pos < len) {
                    retval->push_back(str[pos]);
                    step = maxStep - step;
                    pos += step;
                }
            }
        }
        return *retval;
    }
};
```

LengthofLastWord

Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example, Given s = "Hello World", return 5.

```
int lengthOfLastWord(string s)
{
      int i;int size = s.length();
      while(s[size-1]==' ')   s.resize(--size);
      i = s.find_last_of(' ');
      return (i==-1)?size:size-i-1;
}
```

CompareVersionNumber

Compare two version numbers version1 and version2. If version1 > version2 return 1, if version1 < version2 return -1, otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the . character. The . character does not represent a decimal point and is used to separate number sequences. For instance, 2.5 is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

Here is an example of version numbers ordering:

0.1 < 1.1 < 1.2 < 13.37

```
思路：
先判断小数点前面的大小，若相等在判断小数点后面的大小

int compareVersion(string version1, string version2)
{
        int i = 0;
    int j = 0;
    int n1 = version1.size();
    int n2 = version2.size();

    int num1 = 0;
    int num2 = 0;
    while(i<n1 || j<n2)
    {
        while(i<n1 && version1[i]!='.'){
            num1 = num1*10+(version1[i]-'0');
            i++;
        }

        while(j<n2 && version2[j]!='.'){
            num2 = num2*10+(version2[j]-'0');;
            j++;
        }

        if(num1>num2) return 1;
        else if(num1 < num2) return -1;

        num1 = 0;
        num2 = 0;
        i++;
        j++;
    }

    return 0;
}
```

LongestPalindromicSubstring

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

```
思路：求最长回文子串是个经典的题目，有三种常见的解法
可详看文章：

[最长回文子串](http://blog.csdn.net/zhaoyunfullmetal/article/details/46696917)

方法一 O(n^2)的算法 粗暴法

string longestPalindrome(string s)
{
        int len = s.length(), max = 1, ss = 0, tt = 0;
        bool flag[len][len];
        for (int i = 0; i < len; i++)
            for (int j = 0; j < len; j++)
                if (i >= j)
                    flag[i][j] = true;
                else flag[i][j] = false;
        for (int j = 1; j < len; j++)
            for (int i = 0; i < j; i++)
            {
                if (s[i] == s[j])
                {
                    flag[i][j] = flag[i+1][j-1];
                    if (flag[i][j] == true && j - i + 1 > max)
                    {
                        max = j - i + 1;
                        ss = i;
                        tt = j;
                    }
                }
                else flag[i][j] = false;
            }
        return s.substr(ss, max);
}

方法二

 string longestPalindrome(string s)
{
        if (s.empty()) return "";
        if (s.size() == 1) return s;
        int min_start = 0, max_len = 1;
        for (int i = 0; i < s.size();)
        {
              if (s.size() - i <= max_len / 2) break;
              int j = i, k = i;
              while (k < s.size()-1 && s[k+1] == s[k]) ++k; // Skip duplicate characters.
              i = k+1;
              while (k < s.size()-1 && j > 0 && s[k + 1] == s[j - 1]) { ++k; --j; } // Ex
              int new_len = k - j + 1;
```

```
                if (new_len > max_len) { min_start = j; max_len = new_len; }
            }
            return s.substr(min_start, max_len);
    }


方法三 Manacher's algorithm

string longestPalindrome(string s)
{
        string t;
    for(int i=0;i<s.size();i++)
        t=t+"#"+s[i];
    t.push_back('#');

    vector<int> P(t.size(),0);
    int center=0,boundary=0,maxLen=0,resCenter=0;

    for(int i=1;i<t.size()-1;i++)
    {
        int j= 2*center -i;

        if(boundary>i)
        {
            P[i] = min(P[j],boundary-i);
        }else
        P[i] =0;

        while(i-1-P[i]>=0 && i+1+P[i]<t.size() && t[i+1+P[i]] ==  t[i-1-P[i]])
            P[i]++;

        if(i+P[i]>boundary)
        { // update center and boundary
                center = i;
                boundary = i+P[i];
        }

        if(P[i]>maxLen)
        { // update result
                maxLen = P[i];
                resCenter = i;
        }
    }
    return s.substr((resCenter - maxLen)/2, maxLen);
}
```

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".

```
方法1:
KMP算法 将s反转之后拼到后面，算拼接后的字符串中 既是前缀子串又是后缀子串中的最长串

即是 最长的不需要变动的部分
string shortestPalindrome(string s)
{
        string rev_s = s;
        reverse(rev_s.begin(), rev_s.end());
        string l = s + "#" + rev_s;

        vector<int> p(l.size(), 0);
        for (int i = 1; i < l.size(); i++) {
            int j = p[i - 1];
            while (j > 0 && l[i] != l[j])
                j = p[j - 1];
            p[i] = (j += l[i] == l[j]);
        }

        return rev_s.substr(0, s.size() - p[l.size() - 1]) + s;
}




方法2
Manacher's 算法
已总结过

string preProcess(string s) {
  int n = s.length();
  if (n == 0) return "^$";
  string ret(2*n+3, '#');
  ret[0] = '^';ret[2*n+1] = '```;
  for (int i = 1; i <= n; i++)  ret[2*i]=s[i-1];

  return ret;
}

string shortestPalindrome(string s) {

  int len = s.size();
  if(len<=1) return s;
```

```
    string T = preProcess(s);
    const int n = T.length();
    int P[n], i_mirror;
    int C = 0, R = 0;

    for (int i = 1; i < n-1; i++) {
      i_mirror = 2*C-i; // equals to i' = C - (i-C)

      P[i] = (R > i) ? min(R-i, P[i_mirror]) : 0;

      // Attempt to expand palindrome centered at i
      while (T[i + 1 + P[i]] == T[i - 1 - P[i]])
        P[i]++;

      // If palindrome centered at i expand past R,
      // adjust center based on expanded palindrome.
      if (i + P[i] > R) {
        C = i;
        R = i + P[i];
      }
    }

    // Just changed this part,
    int maxLen = 0;
    int centerIndex = 0;
    for (int i = 1; i < n-1; i++) {
      if (1==i-P[i]) maxLen = P[i];
    }
    string temp = s.substr(maxLen);
    reverse(temp.begin(),temp.end());
    return temp+s;

  }
```

LongestCommonPrefix

Write a function to find the longest common prefix string amongst an array of strings.

```
string longestCommonPrefix(vector<string>& strs)
{
        if (strs.empty()) return "";
        for (int pos = 0; pos < strs[0].length(); pos++)
            for (int i = 1; i < strs.size(); i++)
                if (pos >= strs[i].length() || strs[i][pos] != strs[0][pos])
                    return strs[0].substr(0, pos);
        return strs[0];
}
```

树

BinaryTreeMaximumPathSum

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

```
For example:
Given the below binary tree,

       1
      / \
     2   3
Return 6.
```

```
int maxPathSum(TreeNode* root)
{
        int maxPath = INT_MIN;
        dfsMaxPath(root, maxPath);
        return maxPath;
}


int dfsMaxPath(TreeNode *root, int &maxPath)
{
        if (!root) return 0;
        int l = max(0, dfsMaxPath(root->left, maxPath));
        int r = max(0, dfsMaxPath(root->right, maxPath));
        maxPath = max(maxPath, l + r + root->val);
        return root->val + max(l, r);
}
```

Balanced Binary Tree

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

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

```
bool isBalanced(TreeNode* root)
{
        if(!root)
            return true;
        bool isbalance =true;
        isBan(root,isbalance);

        return isbalance;
}

int isBan(TreeNode* root, bool& isbalance)
{
    if(root == NULL) return 0;
    if(!isbalance) return 0;

    int leftDepth = isBan(root->left,isbalance);
    int rightDepth = isBan(root->right,isbalance);

    if(!isbalance) return 0;

    isbalance = abs(leftDepth-rightDepth) <= 1;
    return max(leftDepth, rightDepth) + 1;

}
```

SumRoottoLeafNumbers

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

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

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

For example,

```
    1
   / \
  2   3
```

The root-to-leaf path 1->2 represents the number 12. The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

```
思路：
一个深度优先搜索的过程..

int sumNumbers(TreeNode* root)
{
        int result =0;
        int cur=0;
        getNumbers(root,result,cur);

        return result;
}

void getNumbers(TreeNode* root,int& sum,int cursum)
{
        if(!root)
            return;
        cursum = cursum*10 + root->val;

        if(!root->left && !root->right)
        {
            sum += cursum;
        }

        if(root->left)
            getNumbers(root->left,sum,cursum);
        if(root->right)
            getNumbers(root->right,sum,cursum);

}
```

Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

```
For example,
Given

         1
        / \
       2   5
      / \   \
     3   4   6
The flattened tree should look like:
   1
    \
     2
      \
       3
        \
         4
          \
           5
            \
             6
```

Hints: If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.

```
方法1：保留prev 自底向上的生成
private TreeNode prev = null;

public void flatten(TreeNode root) {
    if (root == null)
        return;
    flatten(root.right);
    flatten(root.left);
    root.right = prev;
    root.left = null;
    prev = root;
}


方法2：
保存上一个访问的的元素lastVisited自上而下的构造

private static TreeNode lastVisited = null;

public static void flatten(TreeNode root)
{
        if(root == null)
            return;
        TreeNode savedRight = root.right;
        if(lastVisited != null) {
            lastVisited.left = null;
            lastVisited.right = root;
        }
        lastVisited = root;

        flatten(root.left);
        flatten(savedRight);
}


方法3：迭代的方法
void flatten(TreeNode *root)
{
        if(root == NULL) return;
        while(root){
            if(root->left){
                TreeNode *pre = root->left;
                while(pre->right)
                    pre = pre->right;
                pre->right = root->right;
                root->right = root->left;
                root->left = NULL;
            }
            root = root->right;
        }
}
```

Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key. The right subtree of a node contains only nodes with keys greater than the node's key. Both the left and right subtrees must also be binary search trees.

```
方法1:
class Solution {
    bool first = true;
    int prev = 0;
public:
    bool isValidBST(TreeNode *root) {
        if(root == NULL) return true;

        return (
            isValidBST(root->left)
            && check(root->val)
            && isValidBST(root->right));
    }

    bool check(int val)
    {
        if(first)
        {
            first = false;
            prev = val;
            return true;
        }

        if(prev >= val) return false;

        prev = val;
        return true;
    }
};


方法2 :
和方法1基本一样 用一个变量来保存 prev
bool isValidBST(TreeNode* root)
{
        TreeNode* prev =NULL;
        return validate(root,prev);
}

bool validate(TreeNode* node, TreeNode* &prev)
{
        if(node == NULL) return true;
        if (!validate(node->left, prev)) return false;
        if (prev != NULL && prev->val >= node->val) return false;
        prev = node;
        return validate(node->right, prev);
}


方法3
bool isValidBST(TreeNode *root)
{
        return isValidBSTHelper(root, LONG_MIN, LONG_MAX);
}


bool isValidBSTHelper(TreeNode *root, long lower, long upper)
{
    if (!root)  return true;

    long val = (long)root->val;
    return (val > lower && val < upper && isValidBSTHelper(root->left, lower, val) && isV
}
```

BinarySearchTreeIterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

Note: next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

```
思路：
用一个栈来辅助保存二叉搜索树的信息,栈顶保存的是当前二叉搜索树中最小的元素的节点, 每次弹出最小节点后, 掉用p

代码摘自leetcode讨论区...

    stack<TreeNode *> s;
    BSTIterator(TreeNode *root) {
        pushLeft(root);
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !s.empty();
    }

    /** @return the next smallest number */
    int next() {
        TreeNode* top = s.top();
        s.pop();
        pushLeft(top->right);
        return top->val;
    }

    void pushLeft(TreeNode* root)
    {
        if(root != NULL)
        {
            s.push(root);
            TreeNode* cur = root;
            while(cur->left)
            {
                s.push(cur->left);
                cur = cur->left;
            }
        }
    }
```

Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note: A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

```
1. 给定一个串 {1, 4, 3, 7, 9}, 只找到 4(!<=)3, 那么只需要swap这两个节点就能recover整个树.
2. 给定一个串 {1, 9, 4, 5, 3, 10},找到 9(!<=)4 和 5(!<=)3,
swap  9和3 可recover整个树.
我们通过两个指针one,two来记录 两个 调换的node,当发生以下情况是,就记录指针
    if(one && two)
    {
            int temp = one->val;
            one->val = two->val;
            two->val = temp;
    }
因此 one 指针 只被修改1次, two 指针 可能 被修改两次 或者1次

void recoverTree(TreeNode* root) {
        if(root ==NULL)
            return;
        TreeNode *one =NULL;
        TreeNode *two =NULL;
        TreeNode p(numeric_limits<int>::min());
        TreeNode *pre =&p;

        recover(root,one,two,pre);

        if(one && two)
        {
            int temp = one->val;
            one->val = two->val;
            two->val = temp;
        }
    }

    void recover(TreeNode* root,TreeNode* &one,TreeNode* &two,TreeNode* &pre)
    {
        if(root)
        {
            recover(root->left,one,two,pre);

            if(root->val < pre->val)
            {
                if(one ==NULL)
                    one =pre;
                two = root;
            }
            pre = root;
            recover(root->right,one,two,pre);
        }
    }
```

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?

Hint:

Try to utilize the property of a BST. What if you could modify the BST node's structure? The optimal runtime complexity is O(height of BST).

```
思路：类似中序遍历 时间复杂度为O(k)

int kthSmallest(TreeNode* root, int k)
{
        stack<TreeNode*> st;
        TreeNode* p = root;
        while(p || !st.empty())
        {
            while(p)
            {
                st.push(p);
                p = p->left;
            }
            p = st.top();
             st.pop();
            if(--k == 0)
             return p->val;

            p = p->right;

        }
}
```

Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

```
思路：不断的找中点，时间复杂度为O(N)

TreeNode *sortedArrayToBST(vector<int> &num)
{
    return BuildTree(num,0,num.size()-1);
}


TreeNode *BuildTree(vector<int> &num, int start, int end)
{
        if(start > end) return NULL;
        if(start == end) return new TreeNode(num[start]);
        int mid = (start+end)/2;
        TreeNode *treenode = new TreeNode(num[mid]);
        treenode->left = BuildTree(num,start,mid-1);
        treenode->right = BuildTree(num,mid+1,end);
        return treenode;
}
```

Convert Sorted List to Binary Search Tree

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

```
方法1思路：找中间节点,时间复杂度为O(nlgn)
TreeNode* sortedListToBST(ListNode* head)
{
         return sortedListToBST( head, NULL );
}

TreeNode *sortedListToBST(ListNode *head, ListNode *tail)
{
        if( head == tail )
            return NULL;
        if( head->next == tail )    //
        {
            TreeNode *root = new TreeNode( head->val );
            return root;
        }
        ListNode *mid = head, *temp = head;
        while( temp != tail && temp->next != tail )    // 寻找中间节点
        {
            mid = mid->next;
            temp = temp->next->next;
        }
        TreeNode *root = new TreeNode( mid->val );
        root->left = sortedListToBST( head, mid );
        root->right = sortedListToBST( mid->next, tail );
        return root;
}


方法2 :利用中序遍历的思路,时间复杂度为 O(N).
ListNode *list;
int count(ListNode *node){
        int size = 0;
        while (node) {
            ++size;
            node = node->next;
        }
        return size;
}

TreeNode *generate(int n)
{
        if (n == 0)
            return NULL;
        TreeNode *node = new TreeNode(0);
        node->left = generate(n / 2);
        node->val = list->val;
        list = list->next;
        node->right = generate(n - n / 2 - 1);
        return node;
}

TreeNode *sortedListToBST(ListNode *head)
{
        this->list = head;
        return generate(count(head));
}
```

Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

Note: You may assume that duplicates do not exist in the tree.

方法1：递归的方法

思路：postorder中最后一个元素即为根节点，根据此节点把inorder的元素分为左子树

```
TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        if(inorder.size() == 0)
            return nullptr;
        return make(inorder.begin(),inorder.end(),postorder.begin(),postorder.end());
    }

    template<typename Iter>
    TreeNode* make(Iter ibegin,Iter iend,Iter pobegin,Iter poend)
    {
        if(ibegin == iend) return nullptr;
        if(pobegin == poend) return nullptr;

        int val = *(poend-1);
        TreeNode* node = new TreeNode(val);

        auto  idx= find(ibegin,iend,val);
        int leftLength = idx-ibegin;

        node->left = make(ibegin,idx+1,pobegin,pobegin+leftLength);
        node->right =make(idx+1,iend,pobegin+leftLength,poend-1);

        return node;
    }


方法2：非递归的方法,利用栈

TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder)
{
        if(inorder.size() == 0)return NULL;
        TreeNode *p;
        TreeNode *root;
        stack<TreeNode *> stn;

        root = new TreeNode(postorder.back());
        stn.push(root);
        postorder.pop_back();

        while(true)
        {
            if(inorder.back() == stn.top()->val)
            {
                p = stn.top();
                stn.pop();
                inorder.pop_back();
                if(inorder.size() == 0) break;
                if(stn.size() && inorder.back() == stn.top()->val)
                    continue;
                p->left = new TreeNode(postorder.back());
                postorder.pop_back();
                stn.push(p->left);
            }
            else
            {
                p = new TreeNode(postorder.back());
                postorder.pop_back();
                stn.top()->right = p;
                stn.push(p);
            }
        }
        return root;
}
```

Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

Note: You may assume that duplicates do not exist in the tree.

```
方法1：递归的方法
思路：先序遍历的第一个节点就是根节点，再找到中序遍历中的该根节点，将中序遍历可以划分成该根节点的左右两个子

代码摘自leetcode 讨论区

TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder)
{
        if(preorder.size()==0)
            return nullptr;
        return make(preorder.begin(),preorder.end(),inorder.begin(),inorder.end());
}

template<typename Iter>
 TreeNode* make(Iter pFirst , Iter pLast , Iter iFirst , Iter iLast)
{
         if(pFirst == pLast) return nullptr;
         if(iFirst == iLast) return nullptr;
         int val = *pFirst;

         TreeNode* node = new TreeNode(val);

         auto ival= find(iFirst,iLast,val);

         int leftLength = ival-iFirst;

         node->left = make(pFirst+1,pFirst+leftLength+1,iFirst,ival);
         node->right = make(pFirst+leftLength+1,pLast,ival+1,iLast);

         return node;
}



方法2：迭代的方法

TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder)
{
        if(preorder.size()==0)
            return NULL;

        stack<TreeNode *> st;
        TreeNode *t,*r,*root;
        int i,j,f;

        f=i=j=0;
        root = new TreeNode(preorder[i]);
        st.push(root);
        t = root;
        i++;

        while(i<preorder.size())
        {
            if(!st.empty() && st.top()->val==inorder[j])
            {
                t = st.top();
                st.pop();
                f = 1;
                j++;
            }
            else
            {
                if(f==0)
                {
                    t -> left = new TreeNode(preorder[i]);
                    t = t -> left;
                    st.push(t);
                    i++;
                }
                else
                {
                    f = 0;
                    t -> right = new TreeNode(preorder[i]);
                    t = t -> right;
                    st.push(t);
                    i++;
                }
            }
        }

        return root;
}
```

Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

```
思路：根据性质来判断，时间复杂度为O(lgN)

TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
{
        TreeNode* cur = root;
        while (true) {
            if (p -> val < cur -> val && q -> val < cur -> val)
                cur = cur -> left;
            else if (p -> val > cur -> val && q -> val > cur -> val)
                cur = cur -> right;
            else return cur;
        }
    }
```

Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

三种方法详解请看： [LCA-最小公共父节点]
(http://blog.csdn.net/zhaoyunfullmetal/article/details/46924629)

```
思路：
求两个节点的最小公共父节点有三种常用的方法

此处只列上递归的方法

TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
{
        if(!root)
            return NULL;

        if(root == p || root == q)
            return root;

        TreeNode* left = lowestCommonAncestor(root->left,p,q);
        TreeNode* right = lowestCommonAncestor(root->right,p,q);

        if(left == NULL) return right;
        if(right==NULL) return left;

        return root;
}
```

UniqueBinarySearchTree

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

For example,

```
Given n = 3, there are a total of 5 unique BST's.

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

```
方法1：
动态规划方法

int numTrees(int n)
{
      int dp[n+1];
      dp[0] = dp[1] = 1;
      for (int i=2; i<=n; i++)
      {
            dp[i] = 0;
            for (int j=1; j<=i; j++) {
            dp[i] += dp[j-1] * dp[i-j];
            }
      }
      return dp[n];
}



方法2:
利用卡特兰数

int numTrees(int n) {
    //cantalan树
    //C(2n,n)/(n+1)
    long long ans =1;
    for(int i=n+1;i<=2*n;i++){
        ans = ans*i/(i-n);
    }
    return ans/(n+1);
}
```

UniqueBinarySearchTree2

Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example, Given n = 3, your program should return all 5 unique BST's shown below.

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

```
方法1：递归的方法
vector<TreeNode*> generateTrees(int n) {
        vector<TreeNode*> result;
        if(n==0)
        {
            result = getTree(1,0);
            return result;
        }
        result = getTree(1,n);

        return result;
    }

vector<TreeNode*> getTree(int from,int to)
    {
        vector<TreeNode*> result;
        if(from - to>0)
            result.push_back(NULL);
        if(from -to == 0)
            result.push_back(new TreeNode(from));

        if(from <to)
        {
            for(int i= from ;i<= to; i++)
            {
                vector<TreeNode*> left = getTree(from,i-1);
                vector<TreeNode*> right = getTree(i+1,to);

                for(int j=0 ;j<left.size();j++)
                for(int k=0 ;k<right.size() ;k++)
                {
                    TreeNode* temp = new TreeNode(i);
                    temp->left = left[j];
                    temp->right = right[k];

                    result.push_back(temp);
                }
            }
        }
        return result;
}



方法2：迭代的方法
代码摘自leetcode discuss.....

vector<TreeNode *> generateTrees(int n) {
    vector<TreeNode *> tmp;
    vector<TreeNode *> ret;
    tmp.push_back(NULL);
    ret.push_back(new TreeNode(1));
    if (!n) return tmp;

    /* insert the largeset number into previously contructed trees */
    for (int i = 2; i <= n; i++) {
        tmp.clear();
        for (int j = 0; j < ret.size(); j++) {
            /* firstly, put the largest number on the top of tree */
            TreeNode *orgTree = ret[j];
            TreeNode *newNode = new TreeNode(i);
            newNode->left = copy(orgTree);
            tmp.push_back(newNode);

            /* traverse thru the right-most branch,
             * insert the largest number one position after another */
            TreeNode *orgRunner = orgTree;
            while (orgRunner) {
                newNode = new TreeNode(i);
                newNode->left = orgRunner->right;
                orgRunner->right = newNode;
                tmp.push_back(copy(orgTree));

                /* recover the original tree */
                orgRunner->right = orgRunner->right->left;

                /* for the next loop */
                orgRunner = orgRunner->right;
            }
        }
        ret =  tmp;
    }
    return ret;
}

TreeNode *copy (TreeNode *root) {
    TreeNode *ret = NULL;
    if (root) {
        ret = new TreeNode(root->val);
        ret->left = copy(root->left);
        ret->right = copy(root->right);
    }
    return ret;
}
```

BinaryTreeRightSideView

Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

For example:

```
Given the following binary tree,
   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
You should return [1, 3, 4].
```

```
class Solution {

int level = 0;//已经看到的元素数目
    vector<int> answer;
public:
    vector<int> rightSideView(TreeNode *root) {
        if (root!=NULL) travDown(root,0);
        return answer;
    }
    void travDown(TreeNode* nd, int N){//travel down, always go right first
        if (N>=level) {
            answer.push_back(nd->val);
            level++;
        }
        ++N;
        if (nd->right!=NULL) travDown(nd->right,N);
        if (nd->left!=NULL) travDown(nd->left,N);
    }

};
```

Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

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

```
思路：easy级别的题目，可用DFS和BFS两种方法实现

方法1：DFS
int maxDepth(TreeNode* root)
{
        if(root)
        {
            int lmax = maxDepth(root->left);
            int rmax = maxDepth(root->right);
            return max(lmax,rmax)+1;
        }
        return 0;
}

方法2：BFS
int maxDepth(TreeNode *root)
{
    if(root == NULL)
        return 0;

    int res = 0;
    queue<TreeNode *> q;
    q.push(root);
    while(!q.empty())
    {
        ++ res;
        for(int i = 0, n = q.size(); i < n; ++ i)
        {
            TreeNode *p = q.front();
            q.pop();

            if(p -> left != NULL)
                q.push(p -> left);
            if(p -> right != NULL)
                q.push(p -> right);
        }
    }

    return res;
}
```

Same Tree

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

```
bool isSameTree(TreeNode* p, TreeNode* q)
{
        if(p == NULL && q== NULL)
            return true;
        if(p == NULL || q== NULL)
            return false;
        if(p->val != q->val)
            return false;
        if( isSameTree(p->left,q->left) && isSameTree(p->right,q->right) )
            return true;
        else
            return false;
}
```

Populating Next Right Pointers in Each Node

Given a binary tree

```
    struct TreeLinkNode {
      TreeLinkNode *left;
      TreeLinkNode *right;
      TreeLinkNode *next;
    }
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Note:

You may only use constant extra space. You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,

```
Given the following perfect binary tree,
         1
       /  \
      2    3
     / \  / \
    4  5  6  7
After calling your function, the tree should look like:
         1 -> NULL
       /  \
      2 -> 3 -> NULL
     / \  / \
    4->5->6->7 -> NULL
```

```
void connect(TreeLinkNode *root)
{
        if (root == NULL) return;
        TreeLinkNode *pre = root;
        TreeLinkNode *cur = NULL;
        while(pre->left) {
            cur = pre;
                while(cur)
                {
                    cur->left->next = cur->right;
                    if(cur->next) cur->right->next = cur->next->left;
                    cur = cur->next;
                }
            pre = pre->left;
        }
}
```

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
```

```
void connect(TreeLinkNode *root)
{
        if(!root)
            return;
        TreeLinkNode* tempChild = new TreeLinkNode(0);
        TreeLinkNode* currentChild;
        while(root)
        {
            currentChild = tempChild;
            while(root)
            {
                if(root->left) {currentChild->next = root->left; currentChild  = currentC
                if(root->right) {currentChild->next = root->right;currentChild  = current
                root = root->next;
            }
            root = tempChild->next;
            tempChild->next = NULL;
        }
}
```

InvertBinaryTree

Invert a binary tree.

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

Trivia: This problem was inspired by this original tweet by Max Howell: Google: 90% of our engineers use the software you wrote (Homebrew), but you can't invert a binary tree on a whiteboard so fuck off.

```
方法1：递归的方法
TreeNode* invertTree(TreeNode* root)
{
        if(root)
        {
            TreeNode* newleft =NULL;
            TreeNode* newright =NULL;
            if(root->left)
                newleft = invertTree(root->left);
            if(root->right)
                newright = invertTree(root->right);
            root->left = newright;
            root->right = newleft;
        }

        return root;
}

方法2：迭代的方法
TreeNode* invertTree(TreeNode* root)
{
        if (!root) return NULL;
        queue<TreeNode*> level;
        level.push(root);
        while (!level.empty()) {
            TreeNode* node = level.front();
            level.pop();
            swap(node -> left, node -> right);
            if (node -> left) level.push(node -> left);
            if (node -> right) level.push(node -> right);
        }
        return root;
}
```

SymmetricTree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

```
For example, this binary tree is symmetric:

    1
   / \
  2   2
 / \ / \
3  4 4  3
But the following is not:
    1
   / \
  2   2
   \   \
   3    3
```

```
bool isSymmetric(TreeNode* root)
{
        if(root)
        {
            if(bSys(root->left,root->right))
                return true;
            else
                return false;
        }

        return true;
}

bool bSys(TreeNode* left,TreeNode* right){
        if(left && right)
        {
            if(left->val == right->val)
            {
                if(bSys(left->left,right->right) && bSys(left->right,right->left))
                    return true;
                else
                    return false;
            }else
                return false;
        }else if( left ==NULL && right ==NULL)
            return true;
        else return false;
}
```

Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example:

```
Given binary tree {3,9,20,#,#,15,7},
    3
   / \
  9  20
    /  \
   15   7
return its level order traversal as:
[
  [3],
  [9,20],
  [15,7]
]
```

思路：完成二叉树的层次遍历是一个老题目，可用递归和非递归的方式来实现

方法1 递归方法

```
vector<vector<int>> levelOrder(TreeNode* root)
{
        vector<vector<int>> result;
        level(root,0,result);
        return result;
}


void level(TreeNode* root,int depth,vector<vector<int>> &result)
{
        if(root)
        {
            if(depth>= result.size())
            {
                vector<int> vect;

                result.push_back(vect);
            }

            result[depth].push_back(root->val);
            level(root->left,depth+1,result);
            level(root->right,depth+1,result);
        }

}
```

方法2 BFS方法，即迭代的方法

```
vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if(!root) return result;

        queue<TreeNode*> list;
        queue<int> level; //存放每个节点所属的level

        list.push(root);
        level.push(0);


        while(!list.empty())
        {

            TreeNode* temp = list.front();
            list.pop();

            int qlevel = level.front();
            level.pop();

            if(qlevel >= result.size())
            {
                vector<int> ele;
                result.push_back(ele);
            }

            result[qlevel].push_back(temp->val);

            if(temp->left)
            {
                list.push(temp->left);
                level.push(qlevel+1);
            }

            if(temp->right)
            {
                list.push(temp->right);
                level.push(qlevel+1);
            }
        }


        return result;
}
```

Binary Tree Level Order Traversal II

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example:

```
Given binary tree {3,9,20,#,#,15,7},
    3
   / \
  9  20
    /  \
   15   7
return its bottom-up level order traversal as:
[
  [15,7],
  [9,20],
  [3]
]
```

```
思路：
同样的方法 同 题目Binary Tree Level Order Traversal
层次遍历完成之后再把元素reverse一下即可

方法1：递归 DFS

vector<vector<int>> levelOrderBottom(TreeNode* root)
{
        vector<vector<int>> result;
        level(root,0,result);
        reverse(result.begin(),result.end()); // this
        return result;
}

void level(TreeNode* root,int depth,vector<vector<int>> &result)
{
        if(root)
        {
            if(depth>= result.size())
            {
                vector<int> vect;

                result.push_back(vect);
            }

            result[depth].push_back(root->val);
            level(root->left,depth+1,result);
            level(root->right,depth+1,result);
        }

}

方法2：BFS
```

```
vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if(!root) return result;

        queue<TreeNode*> list;
        queue<int> level; //存放每个节点所属的level

        list.push(root);
        level.push(0);


        while(!list.empty())
        {

            TreeNode* temp = list.front();
            list.pop();

            int qlevel = level.front();
            level.pop();

            if(qlevel >= result.size())
            {
                vector<int> ele;
                result.push_back(ele);
            }

            result[qlevel].push_back(temp->val);

            if(temp->left)
            {
                list.push(temp->left);
                level.push(qlevel+1);
            }

            if(temp->right)
            {
                list.push(temp->right);
                level.push(qlevel+1);
            }
        }

        reverse(result.begin(),result.end()); // this
        return result;
}
```

BinaryTreePreorderTaversal

Given a binary tree, return the preorder traversal of its nodes' values.

For example:

```
Given binary tree {1,#,2,3},
   1
    \
     2
    /
   3
return [1,2,3].
```

Note: Recursive solution is trivial, could you do it iteratively?

```
题意：完成二叉树的先序遍历

方法1：递归方式

void preorder(TreeNode *root, vector<int> &result)
{
      if(root)
      {
            result.push_back(root->val);
            if(root->left)
                preorder(root->left,result);//递归左子树
            if(root->right)
                preorder(root->right,result);//递归右子树
      }
}


vector<int> preorderTraversal(TreeNode *root)
{
        vector<int> result;
        preorder(root,result);
        return result;
}

方法2：非递归方法

vector<int> preorderTraversal(TreeNode *root)
{
        vector<int> result;
        if(!root) return result;

        stack<TreeNode*> s;
        s.push(root);

        while(!s.empty())
        {
            TreeNode* n = s.top();
            result.push_back(n->val);
            s.pop();
            //根据栈后进先出的特点先存入右子树，再存入左子树，
            确保了每次都先访问节点的左子树，然后是右子树
            if(n->right) s.push(n->right);
            if(n->left) s.push(n->left);

        }

        return result;
}
```

BinaryTreeInorderTraversal

Given a binary tree, return the inorder traversal of its nodes' values.

For example:

```
Given binary tree {1,#,2,3},
   1
    \
     2
    /
   3
return [1,3,2].
```

Note: Recursive solution is trivial, could you do it iteratively?

```
题意：完成二叉树的中序遍历

方法1:递归的方法
vector<int> inorderTraversal(TreeNode* root)
{
        vector<int> result;
        if(root == NULL)
            return result;

        inorder(root,result);
        return result;
}


void inorder(TreeNode *root,vector<int> &ret)
{
    if(root)
    {
        inorder(root->left,ret);
        ret.push_back(root->val);
        inorder(root->right,ret);
    }
}


方法2：非递归的方法

vector<int> inorderTraversal(TreeNode* root)
{
        vector<int> result;
        stack<TreeNode*> sta;
        TreeNode* p =root;
        do
        {
            while(p != NULL)
            {
                sta.push(p);
                p = p->left;
            }

            if(!sta.empty())
            {
                p = sta.top();
                result.push_back(p->val);
                sta.pop();
                p = p->right;
            }

        }while(!sta.empty() || p!=NULL);

        return result;
}
```

BinaryTreePostorderTraversal

Given a binary tree, return the postorder traversal of its nodes' values.

```
For example:
Given binary tree {1,#,2,3},
   1
    \
     2
    /
   3
return [3,2,1].
```

Note: Recursive solution is trivial, could you do it iteratively?

```
题意：完成二叉树的后序遍历

方法1：迭代法

vector<int> postorderTraversal(TreeNode *root)
{
        vector<int> result;
        if(!root) return result;

        stack<TreeNode*> stack;
        stack.push(root);

        while(!stack.empty())
        {
            TreeNode* node = stack.top();
            result.push_back(node->val);
            stack.pop();

            if(node->left) stack.push(node->left);
            if(node->right) stack.push(node->right);
        }
        reverse(result.begin(), result.end());

        return result;
}

方法2：递归方法

void postorder(TreeNode *root,vector<int> &result)
{
        if(root)
        {
            postorder(root->left,result);
            postorder(root->right,result);
            result.push_back(root->val);
        }
}
```

```
vector<int> postorderTraversal(TreeNode *root)
{
        vector<int> result;
        postorder(root,result);
        return result;
}
```

Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:

```
Given binary tree {3,9,20,#,#,15,7},
    3
   / \
  9  20
    /  \
   15   7
return its zigzag level order traversal as:
[
  [3],
  [20,9],
  [15,7]
]
```

```
思路：跟二叉树的层次遍历接近，加入一个辅助变量来判断每层节点加入的方向
vector<vector<int>> zigzagLevelOrder(TreeNode* root)
{
       vector<vector<int>> result;
       if(!root) return result;

       queue<TreeNode*> list;
       list.push(root);

       bool left = true;

       vector<int> cur;

       while(!list.empty())
       {
           int count = list.size();

           for(int i=0;i <count ;i++)
           {
               TreeNode* temp = list.front();
               list.pop();
               cur.push_back(temp->val);
               if(temp->left)
                  list.push(temp->left);
               if(temp->right)
                  list.push(temp->right);
           }

           if(!left)
           {
               reverse(cur.begin(), cur.end());
               left =true;
```

```
            }else
                left =false;
            result.push_back(cur);
            cur.clear();
        }

        return result;
}
```

CountCompleteTreeNodes

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

Definition of a complete binary tree from Wikipedia:

```
In a complete binary tree every level, except
possibly the last, is completely filled, and all
nodes in the last level are as far left as
possible.
```

It can have between 1 and 2h nodes inclusive at the last level h.

思路：先算两边的树的深度若一样 直接返回；若不一样，递归分治

```
int countNodes(TreeNode* root) {
        if (!root)
            return 0;
        int leftDepth = 0, rightDepth= 0;
        for(TreeNode* p=root; p; p=p->left) ++leftDepth;
        for(TreeNode* p=root; p; p=p->right) ++rightDepth;
        if (leftDepth==rightDepth) {
            return (1<< leftDepth) - 1 ;
        }
        else {
            return countNodes(root->left) + countNodes(root->right) + 1 ;
        }
    }
```

PathSum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example:

```
Given the below binary tree and sum = 22,
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \      \
        7    2      1
return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.
```

思路：使用DFS,递归的过程记录sum值，到叶节点的时候判断sum是否和给定的值相等

```
bool hasPathSum(TreeNode* root, int sum) {
        if (root == NULL)
            return false;

        if(!root->left && !root->right && sum==root->val)
            return true;

        if(root->left)
            if(hasPathSum(root->left,sum-root->val))
                return true;
        if(root->right)
            if(hasPathSum(root->right,sum-root->val))
                return true;

        return false;
    }
```

PathSum2

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:

```
Given the below binary tree and sum = 22,
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \    / \
        7    2  5   1
return
[
   [5,4,11,2],
   [5,8,4,5]
]
```

```
思路：典型的 DFS 求 全部路径的问题
vector<vector<int>> pathSum(TreeNode* root, int sum)
{
        vector<vector<int>> result;
        vector<int> cur;
        getpathSum(result,sum,0,cur,root);
        return result;
}

void  getpathSum(vector<vector<int>>& result, int sum,int cursum,vector<int> cur,TreeNode
{
        if(!node) return;

        cur.push_back(node->val);

        if(!node->left && !node->right && node->val+cursum == sum)
        {
            result.push_back(cur);
            return;
        }

        if(node->left)
            getpathSum(result,sum,cursum+node->val,cur,node->left);
        if(node->right)
            getpathSum(result,sum,cursum+node->val,cur,node->right);
}
```