1、三角形最小路径c和（动态规划）

给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。

相邻的结点 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。
链接：https://leetcode-cn.com/problems/triangle

错:

比较下一层，选取最小的，然后继续比较下一层。。。实际上，可能存在其他路径，某一层可能较大，但整体最小

```
#include "pch.h"
#include <iostream>
#include <vector>

using namespace std;

// 使用递归进行查找
class Solution {
public:

    void Moving(int index, int i,vector<vector<int>>& triangle,int& sum)
    {
        if (i >= triangle.size()) return;

        if (triangle[i][index] > triangle[i][index + 1])
        {
            sum = sum + triangle[i][index + 1];
            Moving(++index, ++i, triangle, sum);
        }
        else
        {
            sum = sum + triangle[i][index];
            Moving(index, ++i, triangle, sum);
        }
    }
    int minimumTotal(vector<vector<int>>& triangle)
    {
        int sum = 0;
        if(triangle.size()>1)
            Moving(0, 1, triangle,sum);
        return sum+triangle[0][0];
    }

};

int main()
{
    vector<vector<int>> triangle = {{ 2 }, {3, 4 }, {6, 5, 7 }, {4, 1, 8, 3 } };
    // vector<vector<int>> triangle = {{ -1 }, {2, 3 }, {6, 5, 71, -1, -3 }};  测试错误，结果应为-1
    Solution s;
    cout<<s.minimumTotal(triangle);
}
```

利用动态规划解题：需要找到状态变量（Dp），写出状态方程。

```
class Solution {

public:

  int minimumTotal(vector<vector<int>>& triangle)

  {

    int n = triangle.size();

    vector<vector<int>> result(n,vector<int>(n,0));  // 二维数组用来存放某点到头的路径

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

    {

      for (int j = 0; j < triangle[i].size(); j ++ )

      {

        if (i == 0) result[i][j] = triangle[i][j];

        if ((j == 0) && (i)) result[i][j] = result[i-1][j]+ triangle[i][j];

        if(i == j && i) result[i][j] = result[i - 1][j-1] + triangle[i][j];

        if(j<i && (j>0)) result[i][j] = min(result[i - 1][j - 1], result[i - 1][j]) +
triangle[i][j];

      }

    }

    return *min_element(result[n-1].begin(),result[n-1].end());

  }

};
```

三步问题

三步问题。有个小孩正在上楼梯，楼梯有n阶台阶，小孩一次可以上1阶、2阶或3阶。实现一种方法，计算小孩有多少种上楼梯的方式。结果可能很大，你需要对结果模1000000007。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/three-steps-problem-lcci
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

```
 class Solution {

public:

  int waysToStep(int n)
```

```
  {
    vector<int> f(n);
    for (int i = 1; i <= n; i++)
    {
      switch(i)
      {
      case 1:f[0] = 1; break;
      case 2:f[1] = 2; break;
      case 3:f[2] = 4; break;
      default:
        f[i-1] = ((f[i - 2] + f[i - 3]) % 1000000007+ f[i - 4])% 1000000007;
      }
    }
    return f[n-1];
  }
};
```

最小路径和

给定一个包含非负整数的 *m* x *n* 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

假设点（x，y）时的状态为c(x,y)，则向前推可得状态方程：f（x,y）=min(f(x,y-1),f(x-1,y))+c（x，y）

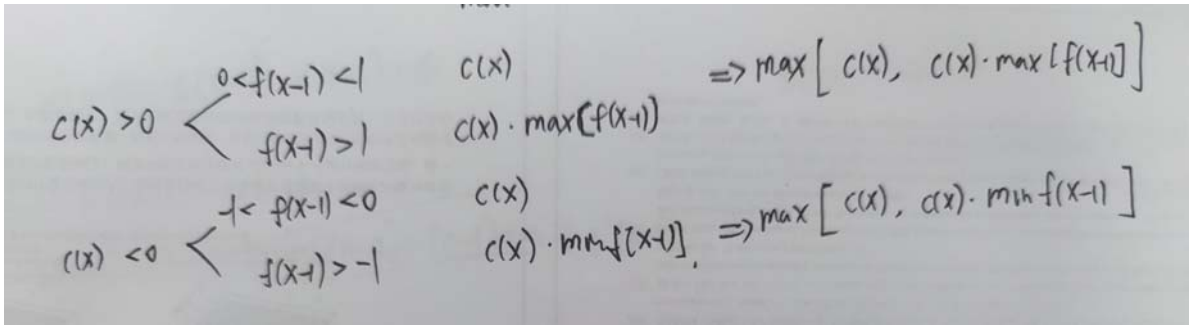
但需考虑边界问题（x=0 or y=0时，无需进行min判断，直接相加即可。）

```
class Solution {
public:
  int minPathSum(vector<vector<int>>& grid)
  {
    int row = grid.size();  // 取行数
    vector<vector<int>> result(row, vector<int>(grid[0].size())); // 存取某点到右上角的路径和

// 分别求每一点到右上角的最小路径和
```

```
    for (int i = 0; i < row; i++)
    {
      for (int j = 0; j < grid[0].size();j++)
      {
        if ((i == 0) && (j == 0)) result[i][j] = grid[i][j];
        else if (i == 0) result[i][j] = result[i][j - 1] + grid[i][j];
        else if (j == 0) result[i][j] = result[i-1][j] + grid[i][j];
        else result[i][j] = min(result[i - 1][j], result[i][j-1]) + grid[i][j];
      }
    }
    return result[row-1][grid[0].size()-1];    // 注意返回的是什么，本题要求到右下角，故返回最后一个


  }
};
```

乘积最大子数组

给你一个整数数组 nums ，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。




```
class Solution {
public:
  int maxProduct(vector<int>& nums)
  {
    int n = nums.size();
    vector<int> result_max(n);
    vector<int> result_min(n);
```

```
    for (int i = 0; i < n; i++)
    {
      if (i == 0)
      {
        result_max[i] = nums[i];
        result_min[i] = nums[i];
      }
      else if (nums[i] >= 0)
      {
        result_max[i] = max(nums[i], nums[i]*result_max[i-1]);
        result_min[i] = min(nums[i], nums[i]*result_min[i - 1]);
      }
      else
      {
        result_max[i] = max(nums[i], nums[i] * result_min[i - 1]);
        result_min[i] = min(nums[i], nums[i] * result_max[i - 1]);
      }
    }
    return *max_element(result_max.begin(),result_max.end());
  }
};
```

上述代码和绘图没错，但是忽略了一点，即提干中说明是“整数”，及不存在元素大于-1小于1的情形。那么代码可进一步优化。

```
class Solution {
public:
  int maxProduct(vector<int>& nums)
  {
    int n = nums.size();
    vector<int> result_max(n,nums[0]);
    vector<int> result_min(n,nums[0]);
```

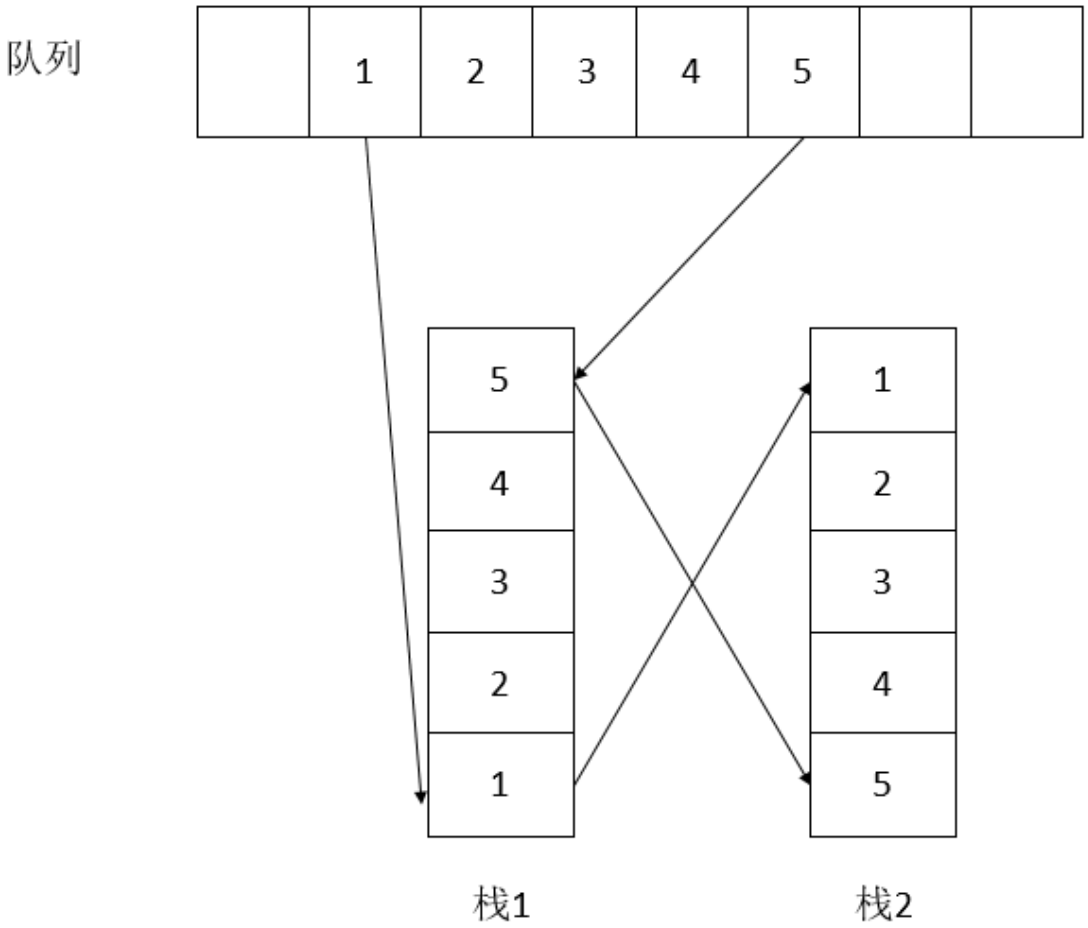
```
•
•     for (int i = 1; i < n; i++)
•     {
•
•       result_max[i] = max(nums[i] * result_min[i - 1],max(nums[i], nums[i] *
result_max[i - 1]));
•       result_min[i] = min(nums[i] * result_max[i - 1],min(nums[i], nums[i] *
result_min[i - 1]));
•
•     }
•     return *max_element(result_max.begin(),result_max.end());
  }
};
```

2、栈（剑指offer）

剑指 Offer 09. 用两个栈实现队列

题干：用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 appendTail 和 deleteHead ，分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素，deleteHead 操作返回 -1)

分析：队列为先入先出型，栈为先入后出型。若使用两个栈，则某一元素先入后出（第一个栈），再转移到另一个栈。此时，栈顶元素即为第一个进第一个栈的元素。



代码：

```
class CQueue {

public:

  CQueue() {



  }



  void appendTail(int value) {

•    s1.push(value);

  }



  int deleteHead() {

•    int output;



•    if (s2.empty())  // 此操作非常重要。假设插入插入删除 插入插入删除（删除时应该先将第一次插入的删除完，再进行元素填充）
```

```
    {
      while (!s1.empty())
      {
        s2.push(s1.top());
        s1.pop();
      }
    }

    if (s2.empty())
      output = -1;
    else
    {
      output = s2.top();
      s2.pop();
    }
      return output;

  }
private:
  stack<int> s1,s2;
};
```

注意：1、删除操作中，if (s2.empty())不可缺。。（只有当12345全部delete之后，才可往s2中加元素，不然，5可能最后才回pop）

2、 while (!s1.empty())，重复循环直到将s1清空。

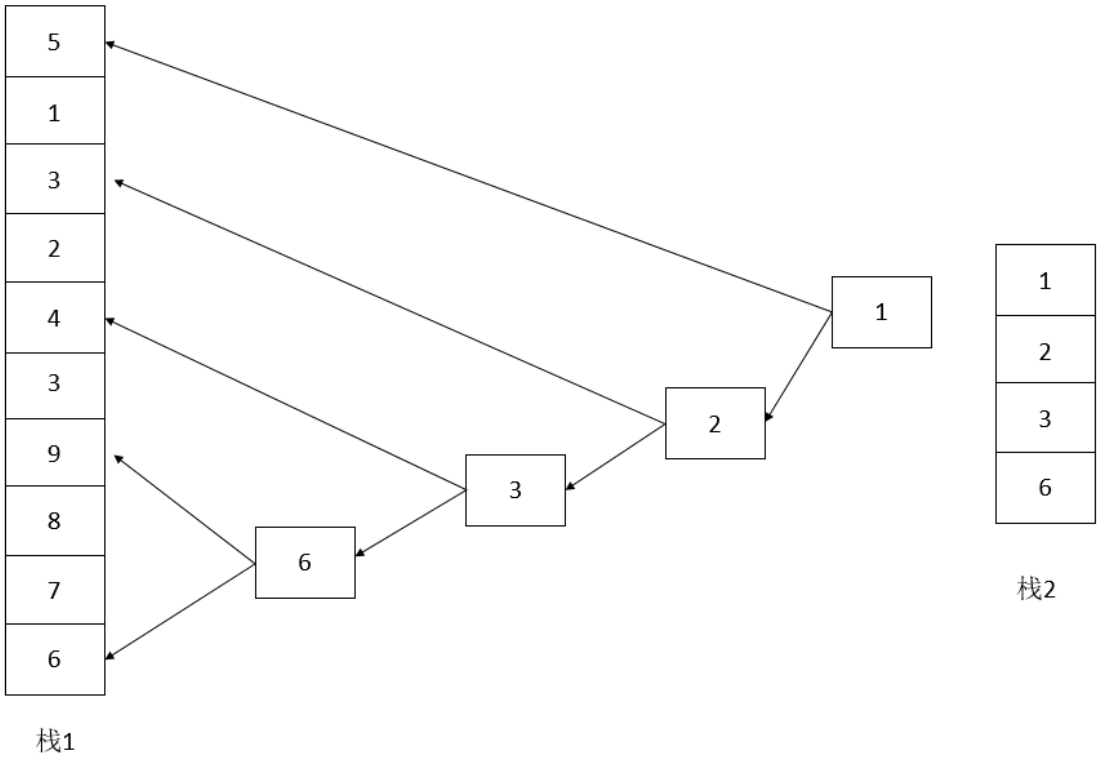
剑指 Offer 30. 包含min函数的栈

题干：定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。

分析：1、在定义的类型中创建一个属性，即min。push时，判断min与push元素的大小，并将小的放入min中。pop时，许判断min与pop元素是否一致，若一致，，则。。

2、上述分析在pop时无法实现的原因是因为，pop了当前最小元素后，无法更新新的最小元素。若能在一个地方，存储最小元素组，pop完最小，则出现次小元素，以此类推。

3、可采用另一个栈存最小元素。栈的优势在于先入后出，最小元素栈中，从栈顶到栈底从小到大排序，每一个元素都对应原栈的一段空间，且为该段空间的最小元素。



代码：

```
class MinStack {
public:
    /** initialize your data structure here. */
    MinStack() {

    }
// push时需判断s2是否为空，若空则将元素push。若为空，则需比较top与x
    void push(int x) {
        s1.push(x);
        if (s2.empty()) s2.push(x);
        else if (s2.top() >= x) s2.push(x);   // =非常重要，因为pop时即使相等也得pop

    }

    void pop() {
        if (!s1.empty())   // pop时，一定要判断
        {
            if (s2.top() == s1.top()) s2.pop();
            s1.pop();

        }
        ;
    }

    int top() {
        if(!s1.empty())
            return s1.top();
        return -1;
    }

    int min() {
        if(!s2.empty())
            return s2.top();
        return -1;
    }
```

```
private:
    stack<int> s1, s2;
};
```

剑指 Offer 59 - II. 队列的最大值

题干：请定义一个队列并实现函数 max_value 得到队列里的最大值，要求函数max_value、push_back 和 pop_front 的均摊时间复杂度都是O(1)。

若队列为空，pop_front 和 max_value 需要返回 -1

分析：利用两个栈定义队列，再用一个栈存放最大值。（x）

进一步分析：队列的特点是先进先出，若要判断其max或min，不能直接采用栈的方式。假设队列元素为942353637。若采用栈的方式第一个元素9低于个进，同样也会第一个出去。由于后面数均小于9，辅助栈中也不存在其他元素，也就是说，9一出去，辅助栈为空，无法判断。。。。。。那么需要利用一个辅助队列，辅助队列不仅要留最大数，还得留小于最大数的数。

按照下图进一步分析：

队列2中第一个进入的元素是9，若只保留最大数，则9出去后，辅助队列不再存在其他元素，无法求最大值，故需将7也放入队列2中，即步骤2-1。

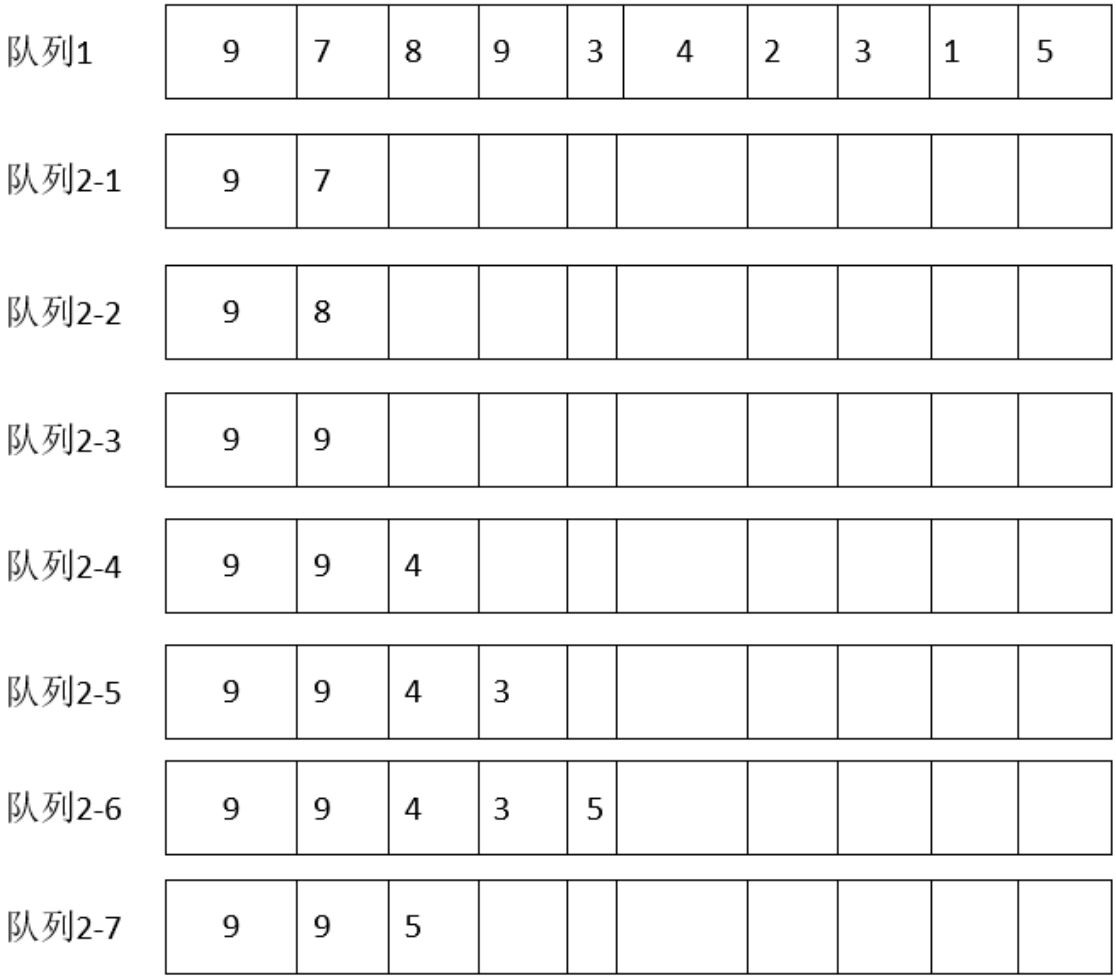
之后再进8，由于8大于7，9出队列之后、8出队列之前，最大值均是8，7无作用，故可删除7，进8，（步骤2-2）

同样，9仅队列后，8也不发挥作用，故删除8（步骤2-3）

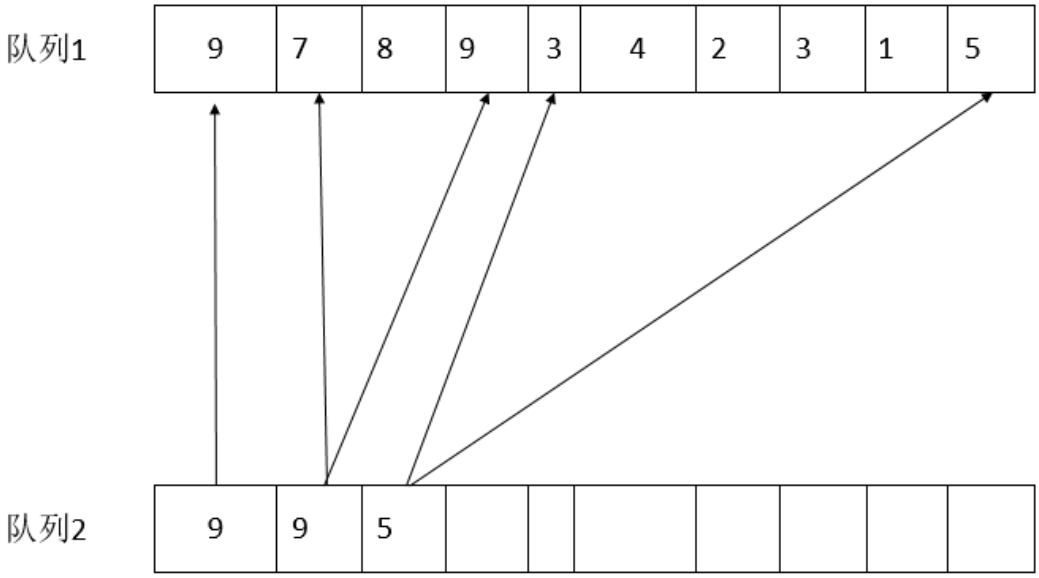
之后进入3，再进入4，4>3，故保留4即可。

以此类推

直到最后一个元素5进去，5大于3，故3删除，5大于4，4删除。辅助队列仅存995.



总结起来，辅助队列中存在元素的作用为：辅助队列元素能掌控的范围为队列中其位置之前的元素。第2个9的范围为1-3.5的范围为4-9.（可将辅助队列看做是将军，即队列1中的小兵一定得被辅助队列的将军所掌管）




```
class MaxQueue {

public:

  MaxQueue() {
```

```
  }

  int max_value() {
    if (q2.empty()) return -1;
    return q2.front();
  }

  void push_back(int value) {
    // q1 push
    q1.push_back(value);
    // q2 push
    while (!q2.empty() && q2.back() < value)
    {
      q2.pop_back();
    }
    q2.push_back(value);
  }

  int pop_front() {
    if (q1.empty()) return -1;
    else
    {
      if (q1.front() == q2.front()) q2.pop_front();
      int popvalue = q1.front();
      q1.pop_front();
      return popvalue;
    }
  }
private:
  deque<int> q1, q2;
```

```
};


/**
 \* Your MaxQueue object will be instantiated and called as such:
 \* MaxQueue* obj = new MaxQueue();
 \* int param_1 = obj->max_value();
 \* obj->push_back(value);
 \* int param_3 = obj->pop_front();
 */
```

3、堆（剑指offer）

剑指 Offer 40. 最小的k个数

题干：输入整数数组 arr ，找出其中最小的 k 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

分析：1、最直观的方法就是用sort进行排序，然后取k个最小值。nlogn

2、堆排序法，放进去再拿出来。但本题并没有要求全部排序，而是k个排序，故可仅创建一个k长度大小的堆：若k+1元素小于堆顶（大顶堆），则弹出堆顶并push该元素，以此类推直到结束。。。最后再将k个堆移到数组中。

3、与2类似，没必要对所有元素进行快速排序，只需对k个元素排序即可。

```
class Solution {
public:
  vector<int> getLeastNumbers(vector<int>& arr, int k) {
    priority_queue<int> q;
    vector<int> a(k);
    if(k == 0) return a;   // 需注意

    // 构建大顶堆
    for (int i = 0; i < k; i++)
      q.push(arr[i]);
```

```
    // 循环比较.若元素小于堆顶元素，则替换
    for (int j = k; j < arr.size(); j++)
    {
      if (q.top() > arr[j])
      {
        q.pop();
        q.push(arr[j]);
      }
    }

    // 将大顶堆中的元素放入数组中，并返回值
    for (int m = 0; m < k; m++)
    {
      a[m] = q.top();
      q.pop();
    }
    return a;
  }
};
```

4、排序

快速排序

指针交换法：

首先选定基准元素Pivot，并且设置两个指针left和right，指向数列的最左和最右两个元素：

接下来是**第一次循环**，从right指针开始，把指针所指向的元素和基准元素做比较。如果**大于等于**pivot，则指针向**左**移动；如果**小于**pivot，则right指针停止移动，切换到**left**指针。

在当前数列中，1<4，所以right直接停止移动，换到left指针，进行下一步行动。

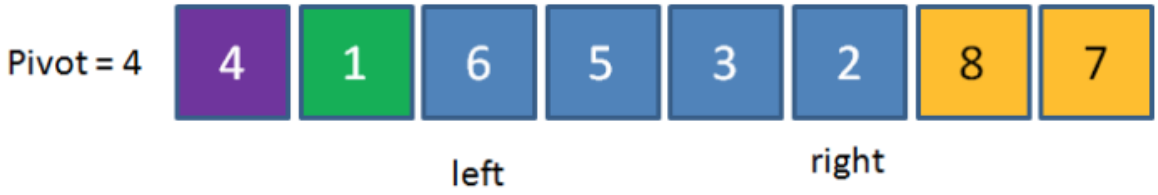
轮到left指针行动，把指针所指向的元素和基准元素做比较。如果**小于等于**pivot，则指针向**右**移动；如果**大于**pivot，则left指针停止移动。

由于left一开始指向的是基准元素，判断肯定相等，所以left右移一位。

由于7 > 4，left指针在元素7的位置停下。这时候，我们让**left和right指向的元素进行交换。**

接下来，我们进入**第二次循环**，重新切换到right向左移动。right先移动到8，8>2，继续左移。由于2<8，停止在2的位置。

切换到left，6>4，停止在6的位置。



当left和right指针重合之时，我们让pivot元素和left与right重合点的元素进行交换。此时数列左边的元素都小于4，数列右边的元素都大于4，这一轮交换终告结束。


```
#include "pch.h"
#include <iostream>

using namespace std;
```

```
void swap1(int& a, int& b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
void QSort(int *arr, int len, int *left,int *right)
{
    int pivot = *left;
    int *phead = left;   // 因为要迭代，
    int *ptail = right;
    while (left != right)
    {
        if (*right >= pivot) right--;
        else
        {
            if (*left <= pivot) left++;
            else swap1(*left, *right);
        }
    }
    swap1(*left, *phead);
    if (phead == ptail) return;  //  迭代退出
    QSort(arr, len, phead,left);
    QSort(arr, len, right+1,ptail);

}
void QuickSort(int *arr,int len)
{
    if (len < 1) return;
    QSort(arr, len, &arr[0],&arr[len-1]);
}

void Display(int *arr,int len)
{
    for (int i = 0; i < len; i ++ )
        cout << arr[i] << " ";
    cout << endl;
}
int main()
{
    int arr[] = { 1,3,2,8,6,0,33,64,22 };
    QuickSort(arr, sizeof(arr) / sizeof(arr[0]));
    Display(arr, sizeof(arr) / sizeof(arr[0]));

}
```

剑指 Offer 45. 把数组排成最小的数

题干：输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

分析：假设数字为5，38.，组合后385才为最小数字，，故采用直接拼接的方法组合。。转化为字符串

```
class Solution {

public:

  string minNumber(vector<int>& nums) {
```

```
    vector<string> str;
    string ans;
    // 转化为字符串数组
    for (int i = 0; i < nums.size(); i++)
    {
      str.push_back(to_string(nums[i]));
    }

    // 排序比较，利用sort函数
    sort(str.begin(), str.end(), [](string a, string b)-> bool {return a + b < b + a; });

    // 连接
    for (int i = 0; i < str.size(); i++)
      ans += str[i];
    return ans;

  }
};
```

5、位运算

剑指 Offer 15. 二进制中1的个数

题干：请实现一个函数，输入一个整数，输出该数二进制表示中 1 的个数。例如，把 9 表示成二进制是 1001，有 2 位是 1。因此，如果输入 9，则该函数输出 2。

```
class Solution {
public:
  int hammingWeight(uint32_t n) {
    int res = 0;
    while (n)
```

```
    {
      if(n & 1)
        res++;
      n >>= 1;
    }
    return res;
  }
};
```

```
class Solution {
public:
    int hammingWeight(uint32_t n) {
        int res = 0;
        while (n)
        {
            res++;
            n &= n - 1;
        }
        return res;
    }
};
```

剑指 Offer 39. 数组中出现次数超过一半的数字

题干：数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

分析：1、对数组直接进行排序，处在中间位置的即为数字。（nlogn）

2、摩尔投票法：核心理念为 **“正负抵消”** ，即假设众数为1，其余数为0，总体相加一定大于0.。那么在实际编程时，用一个votes来计数。可先假设第一个数字为众数，若第二个数字不是该数，则抵消；若是该数，则加1。以此类推直至循环遍历完，（时间复杂度N，空间1）

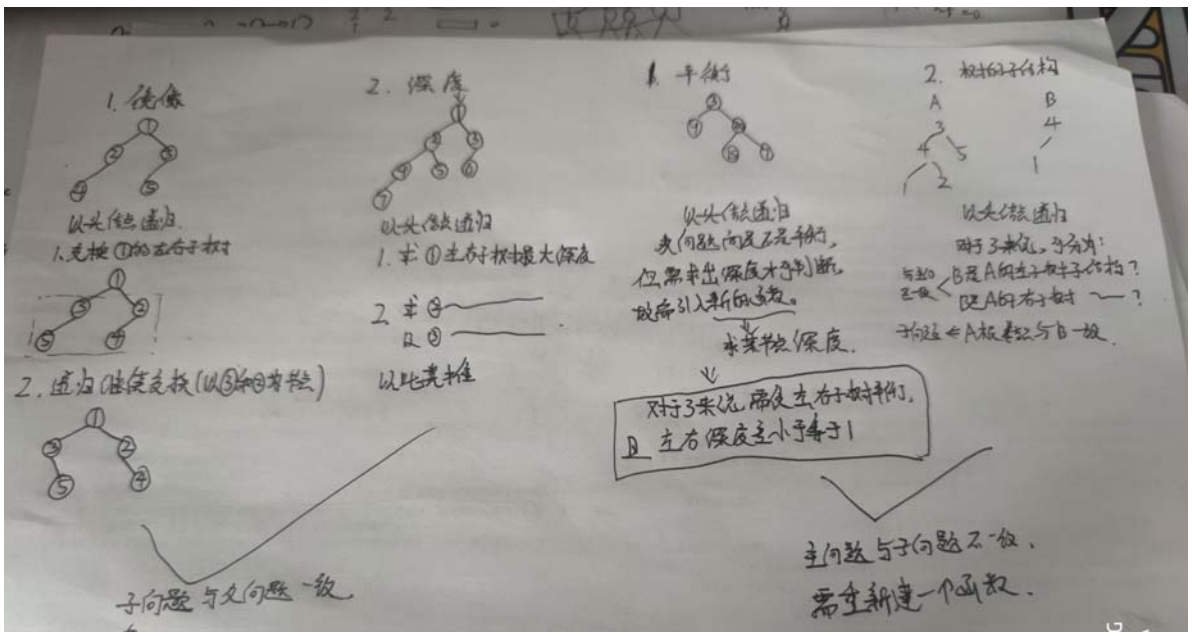
```
class Solution {
public:
  int majorityElement(vector<int>& nums) {
    sort(nums.begin(),nums.end());
    return nums[nums.size()/2];
  }
};
```

```
class Solution {
public:
```

```
  int majorityElement(vector<int>& nums) {
    int votes = 1;
    int x;
    x = nums[0];
    for (int i = 1; i < nums.size(); i++)
    {
      if (!votes) x = nums[i];
      if (x == nums[i]) votes++;
      else votes--;
    }
    return x;
  }
};
```

5、树

1、**对于普通树，可采用递归**：按头节点进行递归（即参数为头节点）。对于27题：由于要求镜像，对于节点root来说，需要将root->left 与root->right交换一下，并以此类推。对于55，要求求最大深度，对于root的最大深度为左子树与右子树的最大深度，故直接递归就可。



2、对于普通树，有时需要辅助函数。

3、普通树：递归+遍历。68题，需要采用后序遍历（先左、再右、后根），先寻找左右结点是否符合要求，再将结果返回根节点。

4、普通树，按照层来遍历。32题层序遍历，利用队列的先入先出特性；28题，判断是否对称，对称可通过一层一层比较，故需层递归。

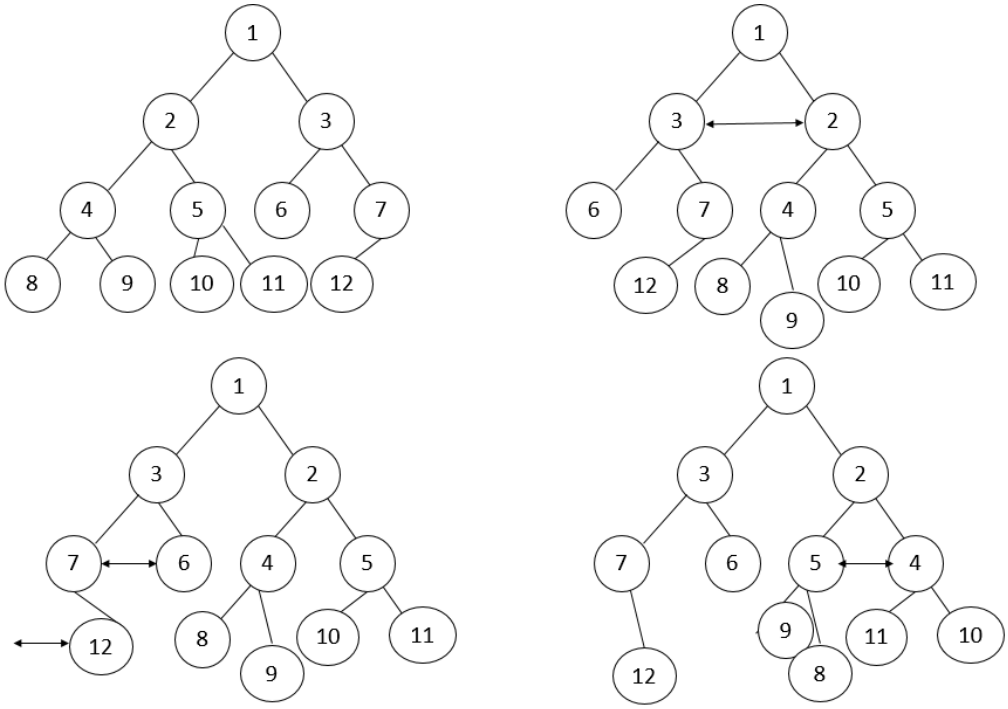
5、回溯：34题

6、二叉搜索树特点：中序遍历为从小到大

剑指 Offer 27. 二叉树的镜像

题干：请完成一个函数，输入一个二叉树，该函数输出它的镜像。

分析：考虑到树是由结点构成，其优势在于便于断链和重构，所以要往链方面考虑。采用递归的方法，每次将左右节点交换，如下图所示。




```
// 链表的特性
class Solution {
public:

    TreeNode* mirrorTree(TreeNode* root) {
        if(!root) return NULL;

        TreeNode *tmp = root->left;
        root->left = root->right;
        root->right = tmp;
        mirrorTree(root->left);
        mirrorTree(root->right);
        return root;
    }
};

// 遍历+递归
class Solution {
public:

    TreeNode* mirrorTree(TreeNode* root) {
        if(!root) return NULL;
```

```
        TreeNode *tmp = root->left;
        root->left = mirrorTree(root->right);
        root->right = mirrorTree(tmp);
        return root;
    }
};
```

剑指 Offer 55 - I. 二叉树的深度

题干：输入一棵二叉树的根节点，求该树的深度。从根节点到叶节点依次经过的节点（含根、叶节点）形成树的一条路径，最长路径的长度为树的深度。

分析：因为只给了根节点，只能通过遍历进行求深度

```
class Solution {

public:

  int maxDepth(TreeNode* root) {

•     if(!root) return 0;

•     return max(maxDepth(root->left),maxDepth(root->right))+1;

  }

}; // 双百

class Solution {
public:

    TreeNode* mirrorTree(TreeNode* root) {
        if(!root) return NULL;

        TreeNode *tmp = root->left;
        root->left = mirrorTree(root->right);
        root->right = mirrorTree(tmp);
        return root;
    }
}; // 后序遍历
```

剑指 Offer 54. 二叉搜索树的第k大节点

题干：给定一棵二叉搜索树，请找出其中第k大的节点。

分析：二叉搜索树的特点为： 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉搜索树。

根据上述特点，可以按右、中、左的顺序遍历（反中序遍历）。

```
class Solution {

public:

  int kthLargest(TreeNode* root, int k) {

•     if(!root) return 0;
```

```
•
•     kthLargest(root->right, k);
•     if(result.size()<k)
•        result.push(root->val);
•     kthLargest(root->left, k);
•
•     return result.top();
  }
private:
•     stack<int> result;


};
```

剑指 Offer 32 - II. 从上到下打印二叉树 II

题干：从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行

分析：利用层序遍历，唯一麻烦的是需要分层打印。可多加一层循环

```
class Solution {
public:
  vector<vector<int>> levelOrder(TreeNode* root) {
•
•     vector<vector<int>> result; // 存储结果
•     TreeNode *current = root;
•     queue<TreeNode *> q; // 通过队列实现层序排列
•     int len_q = 1;  // 当前层节点个数
•
•     while (current)  // 判断是否为空
•     {
•       vector<int> level;
•       for (int i = 0; i < len_q; i++)
•       {
•         if (current->left) q.push(current->left);
```

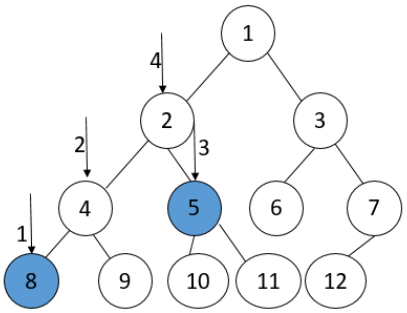
```
        if (current->right) q.push(current->right);
        level.push_back(current->val);
        if(q.empty()) //
        {
          result.push_back(level);
          return result;
        }
        current = q.front();     // 注意：应该先top，再pop。不然如果为空，top会报错
        q.pop();
      }
      result.push_back(level);
      len_q = q.size()+1;

    }
    return result;
  }
};
```

面试题68 - II. 二叉树的最近公共祖先

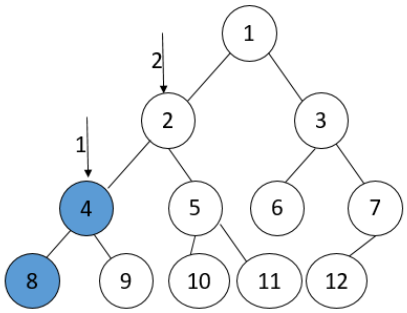
题干：给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”**说明:**

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

分析：公共祖先共有三种情况：1、p为两者的公共祖先（即q为p的子树)；2、q为二者公共祖先；3、pq不为祖先。若为1，则左或右return值为p，另一return值为空；2与1类似。若为3，则公共祖先处的两返回值分别为p、q。下图进一步分析



后序：
4->left == 8,所以返回8。4->right !=5，return NULL
4接收到8和NULL之后，继续向上返回8
等等
2->right == 5,return 5。
2既有左返回值，也有有返回值，故return2
再往上，1->right = NULL（遍历后）,故最终return2



后序：
2->left == 4,所以返回4。2->right !=8，return NULL
2接收到4和NULL之后，继续向上返回4
1->right== NULL(遍历后)，故最终return 4；

注：图为方便解释，将值用了进去，实际编程时，返回的是指向该值的指针。

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        TreeNode *left, *right;
        // 1、既可表示空、也可表示root为NUll
        if (!root) return NULL;
        // 1、若root等于p或q
        if (root == p || root == q) return root;

        left = lowestCommonAncestor(root->left, p, q);
        right = lowestCommonAncestor(root->right, p, q);

        // 2、若左右均有
        if (left && right) return root;
        // 2、若左有或右有
        if (left && !right) return left;
        if (!left && right) return right;
        // 2、先假设左右均为NULL，则
        return NULL;
    }
};
```

剑指 Offer 55 - II. 平衡二叉树

题干：输入一棵二叉树的根节点，判断该树是不是平衡二叉树。如果某二叉树中任意节点的左右子树的深度相差不超过1，那么它就是一棵平衡二叉树。

```
class Solution {

public:

  int GetDepth(TreeNode *root)

  {

•    if(!root) return 0;



•    return max(GetDepth(root->left),GetDepth(root->right))+1;
```

```
    }

    bool isBalanced(TreeNode* root) {

•       if(!root) return 1;  // 如果结点为空，则说明该节点的数为平衡树

•       return abs(GetDepth(root->left)-GetDepth(root->right))<2 && isBalanced(root-
>left) && isBalanced(root->right);

    }

};
```

```
class Solution {

public:

    int GetDepth(TreeNode *root)

    {

•       return !root? 0:max(GetDepth(root->left),GetDepth(root->right))+1;

    }

    bool isBalanced(TreeNode* root) {

•       // 如果结点为空，则说明该节点的数为平衡树

•       return !root?1:abs(GetDepth(root->left)-GetDepth(root->right))<2 &&
isBalanced(root->left) && isBalanced(root->right);

    }

};
```

剑指 Offer 28. 对称的二叉树

题干：请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

分析：1、镜像对称的中序排列也是对称的。所以可先中序排列，再双指针比较。（错，中序对称的不一定为镜像树，例如：**[1,2,2,2,null,2]** ）

代码如下：

```
class Solution {
public:
    void dfs(TreeNode *root)
    {
        if(!root) return ;

        dfs(root->left);
        nums.push_back(root->val);
        dfs(root->right);
    }
```

```
    bool isSymmetric(TreeNode* root) {
        dfs(root);

        int left =0;
        int right = nums.size()-1;
        while(left<right)
        {
            if(nums[left] != nums[right]) return 0;
            left++;
            right--;
        }
        return 1;
    }

private:
    vector<int> nums;

};
```

2、按层进行递归。

```
class Solution {

public:

  bool helper(TreeNode *Left, TreeNode *Right)

  {

    // 按层迭代

    // 迭代结束条件

    if (!Left && !Right) return 1;

    if ((!Left && Right) || (Left && !Right)) return 0;



    if (Left->val != Right->val) return 0;



    // 进入下一层判断

    return helper(Left->left, Right->right) && helper(Left->right, Right->left);



  }

  bool isSymmetric(TreeNode* root) {

    if (!root) return 1;

    return helper(root->left, root->right);

  }

};
```

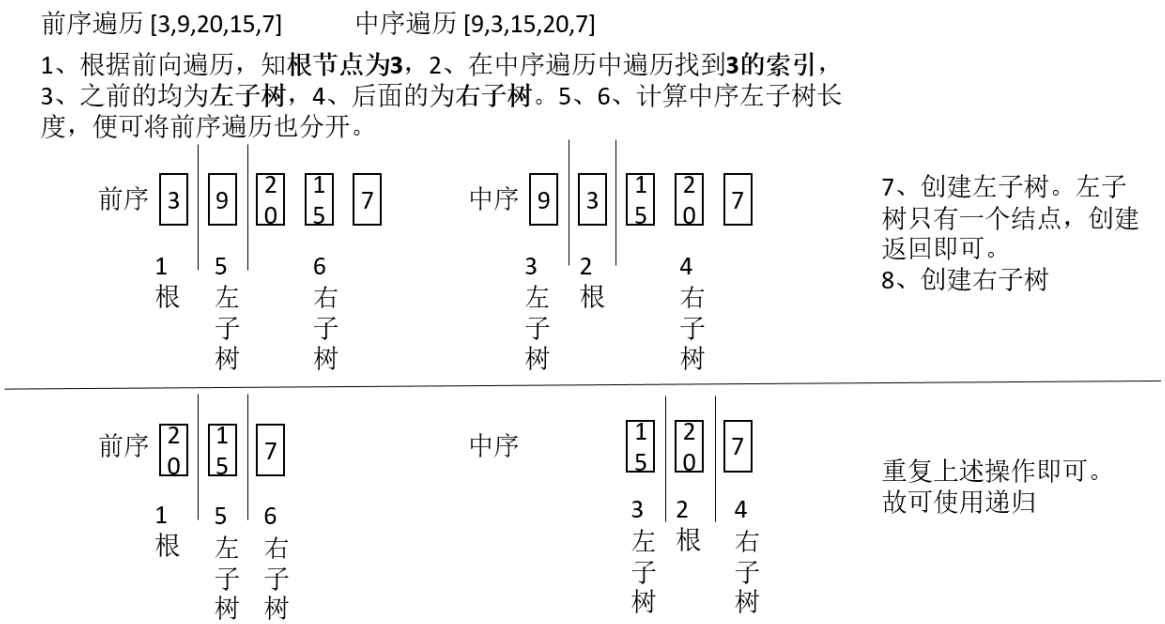
剑指 Offer 07. 重建二叉树

题干：输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。

例如，给出

```
前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]
```

分析：




```
class Solution {

public:



  // 3、递归

  TreeNode *helper(vector<int>& preorder, int pre_start, int pre_end, vector<int>&
inorder, int in_start, int in_end)

  {

•    // 递归出口(因为前序和中序类似，故只用一个判断即可)

•    if (pre_start > pre_end) return NULL;

•    // 递归操作:建节点，算参数

•    TreeNode *root = new TreeNode(preorder[pre_start]);

•    if (pre_start == pre_end ) return root;



•    int root_inindex = hm[preorder[pre_start]];
```

```
    int left_nodes = root_inindex - in_start;
    int right_nodes = in_end - root_inindex;


    root->left = helper(preorder, pre_start + 1, pre_start + left_nodes, inorder, in_start, root_inindex - 1);
    //root->right = helper(preorder, pre_start + left_nodes + 1, pre_end, inorder, root_inindex+1, in_end);
    root->right = helper(preorder, pre_end - right_nodes + 1, pre_end, inorder, root_inindex + 1, in_end);
    return root;
  }
  TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    if (preorder.size() == 0) return NULL;
    // 1、将数组移到map中
    for (int i = 0; i < inorder.size(); i++)
    {
      hm[inorder[i]] = i;
    }



    // 2、准备递归
    return helper(preorder, 0, preorder.size() - 1, inorder, 0, inorder.size() - 1);
  }
private:
  unordered_map<int, int> hm;


};
```

剑指 Offer 32 - I. 从上到下打印二叉树

题干：从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印

分析：层序排列，比32-II简单

```
class Solution {
```

```
public:
  vector<int> levelOrder(TreeNode* root) {
    vector<int> nums;
    if (!root) return nums ;
    queue<TreeNode*> q;
    TreeNode *current = root;
    while (current)
    {
      if (current->left) q.push(current->left);
      if (current->right) q.push(current->right);

      nums.push_back(current->val);
      if (q.empty()) return nums;
      current = q.front();
      q.pop();
    }
    return nums;
  }
};
```

剑指 Offer 32 - III. 从上到下打印二叉树 III

题干：请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。

分析：在32-ii的基础上修改即可。。之前是利用了先进先出队列，本题可用双端队列，奇数偶数层分开，若从后面取，则push在前面，，从前面取，push在后面。

```
/**
 \* Definition for a binary tree node.
 \* struct TreeNode {
 \*   int val;
 \*   TreeNode *left;
 \*   TreeNode *right;
 \*   TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 \* };
```

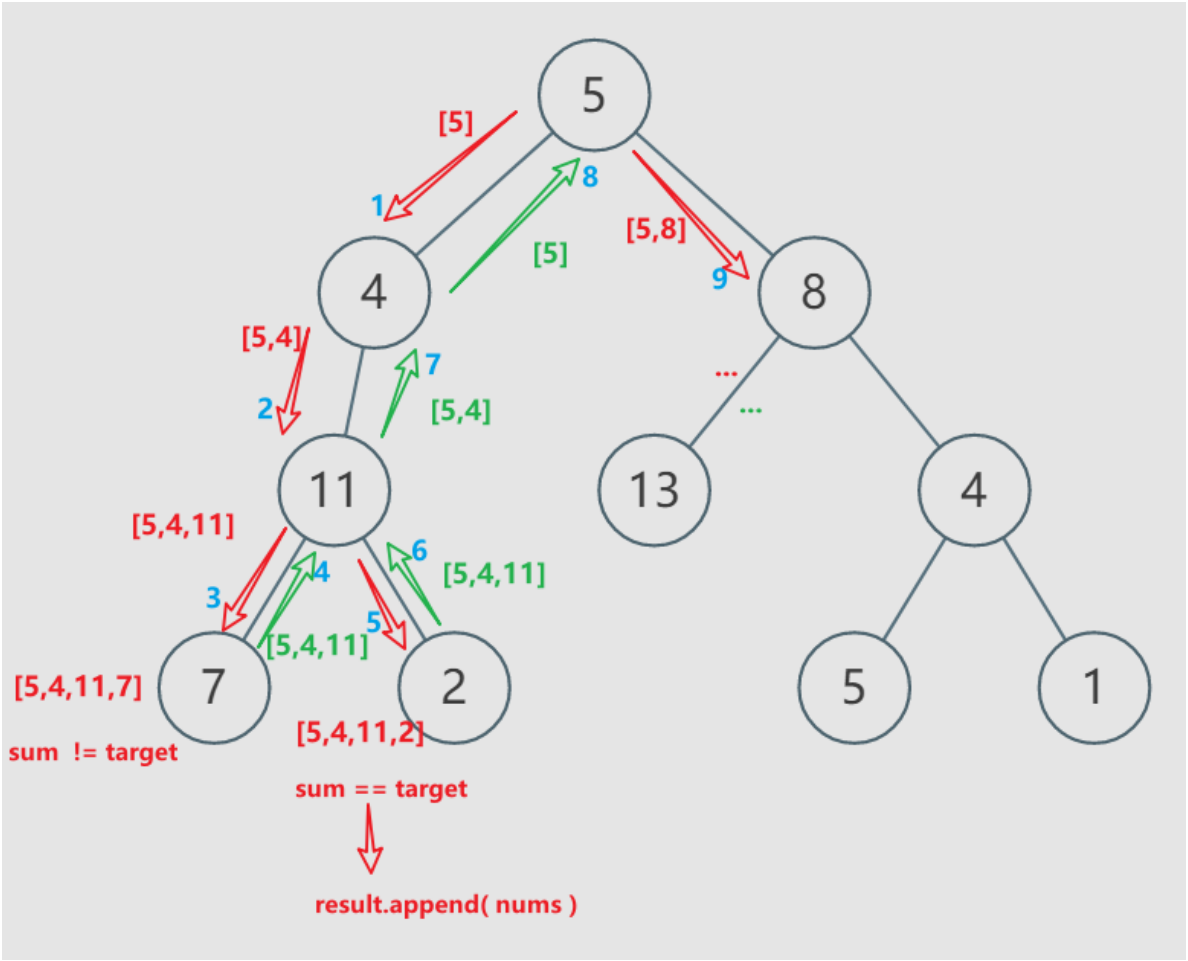
```
 */
class Solution {
public:
  vector<vector<int>> levelOrder(TreeNode* root) {
    // 队列
    deque<TreeNode *> d;
    // 存放数据的容器
    vector<vector<int> > result;
    bool flag = true;
    //
    d.push_back(root);
    if(!root) return result;


    while (!d.empty())
    {
      vector<int> level;
      int times = d.size();
      for (int i = 0; i < times; i++)
      {
        TreeNode * current;
        if (flag)
        {
          current = d.front();
          d.pop_front();
          if (current->left) d.push_back(current->left);
          if (current->right) d.push_back(current->right);
        }
        else
        {
          current = d.back();
          d.pop_back();
```

```
            if (current->right) d.push_front(current->right);
            if (current->left) d.push_front(current->left);
          }
          level.push_back(current->val);
        }
        result.push_back(level);
        flag = !flag;
      }
      return result;
    }
};
```

剑指 Offer 34. 二叉树中和为某一值的路径

题干：输入一棵二叉树和一个整数，打印出二叉树中节点值的和为输入整数的所有路径。从树的根节点开始往下一直到叶节点所经过的节点形成一条路径。

分析：看到题首先想到的是递归，即每次输入节点和值，看能不能走到叶节点。但本文需要返回走过的路径，即可以把题目看成：1、先找到所有从根节点 到叶节点的路径　2、若该路径符合要求，则push一下。3、由于一嗖用递归。根-左-右，故为前序。



```
class Solution {
public:
    vector<vector<int> > result;    // 存放符合条件的路径
    vector<int> path;  // 存放每次走的路径

    void dfs(TreeNode* root, int sum)
    {
        // 递归出口
        if (!root) return;

        // 前序，与根有关的操作
        path.push_back(root->val);
        sum = sum - root->val;

        dfs(root->left, sum );
        if (!sum && !root->left && !root->right) result.push_back(path);
        dfs(root->right, sum );

        // 回溯，每返回上一层，需将path pop一次
        path.pop_back();
    }
    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        dfs(root, sum);
        return result;
    }
};
```

剑指 Offer 26. 树的子结构

题干：输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)B是A的子结构， 即 A中有出现和B相同的结构和节点值。

与剑指 Offer 55 - II. 平衡二叉树类似

若B是A的子结构，则有：1、A、B结点一致　2、B是A的左子树的子结构　3、B是A的右子树的子结构。

```
class Solution {

public:

  bool helper(TreeNode* A, TreeNode* B)

  {

    if(B == NULL) return true;

    if(A == NULL) return false; // A无，B有，则B一定不是A的子结构



    if(A->val != B->val) return false;



    return helper(A->left,B->left) && helper(A->right,B->right);

  }
```

```
  bool isSubStructure(TreeNode* A, TreeNode* B) {
    if(A == NULL || B == NULL) return false;
    return helper(A,B) ||isSubStructure(A->left, B) || isSubStructure(A->right, B);
  }
};
```

6、数组

若是排序数组，则首先考虑二分法。普通数组可考虑哈希表。

剑指 Offer 04. 二维数组中的查找

题干：在一个 n * m 的二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数

分析：针对下面对矩阵，可以想办法减小搜索量。。如果搜索20，则可先让20和对角线比较，大于17，小于30，则可将范围缩小在倒数第一行和倒数第一列。以此递归。

现有矩阵 matrix 如下：

```
[
  [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]
]
```

该法不易实现，较为麻烦。

从二维数组的右上角开始查找。如果当前元素等于目标值，则返回 `true`。如果当前元素大于目标值，则移到左边一列。如果当前元素小于目标值，则移到下边一行。

```
class Solution {
public:
  bool findNumberIn2DArray(vector<vector<int>>& matrix, int target) {
    if(matrix.empty()|| matrix[0].empty()){    // 注：1、判断二维容器时hi，应同时判断两个
      return false;
    }
    int line = matrix[0].size() ;
    int row = matrix.size();
    int l_c = line-1;
    int r_c = 0;
    while (l_c >= 0 && r_c < row)
    {
      if (matrix[r_c][l_c] > target) l_c--;
      else if (matrix[r_c][l_c] < target) r_c++;
      else return 1;
    }
    return 0;
```

```
    }
};
```

剑指 Offer 53 - II. 0～n-1中缺失的数字

题干：一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围0～n-1之内。在范围0～n-1内的n个数字中有且只有一个数字不在该数组中，请找出这个数字。

分析：首先想到的是快慢指针，一个指针在中间，一个指针在开头，然后移动判断。但二分法效率更高。

```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int start = 0;
        int end = nums.size() - 1;
        while (start <= end)    // 注意：1、需要等于
        {
            int middle =(start+end) / 2;
            if (nums[middle] != middle) end = middle - 1;
            else start = middle + 1;
        }
        return start;       //注意：2、由于索引的特殊情况，可等于这个
    }
};
```

剑指 Offer 29. 顺时针打印矩阵

题干：输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。

分析：设置上下左右四个边界，先将一圈存储进去，一圈存储完毕后，将边界缩小，重新顺时针循环。

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {

        vector<int> result;
        if(matrix.empty() || matrix[0].empty()) return result;
        int count = 0;
        int left = 0;
        int top = 0;
        int bottom = matrix.size();
        int right = matrix[0].size();

        int sum = matrix[0].size() * matrix.size();
        while (count < sum)
        {
            for (int i = left; i < right && count< sum; i++)
            {
                result.push_back(matrix[top][i]);
                count++;
            }
            for (int j = top+1; j < bottom && count< sum; j++)
            {
                result.push_back(matrix[j][right-1]);
                count++;
            }
            for (int i = right - 2; i >= left&&count< sum; i--)
            {
```

```
        result.push_back(matrix[bottom-1][i]);
        count++;
      }
      for (int j = bottom - 2; j > top&&count< sum; j--)
      {
        result.push_back(matrix[j][left]);
        count++;
      }
      left++;
      top++;
      right--;
      bottom--;
    }
    return result;
  }
};
```

剑指 Offer 53 - I. 在排序数组中查找数字 I

题干：统计一个数字在排序数组中出现的次数。

分析：利用二分查找找到目标数字，然后向两边延伸。

```
class Solution {
public:
  int search(vector<int>& nums, int target) {
    int start = 0;
    int end = nums.size()-1;
    int count = 0;
    int change;
    int middle;
    bool flag = 0;
    if(nums.empty()) return 0;
    while(start <= end)
    {
      middle = (start+end)/2;
      if(nums[middle] < target) start = middle +1;
      else if(nums[middle] >target) end = middle-1;
      else
      {
        flag = 1;
        break;
      }
    }
    if(flag)
    {
      change = middle;
      while(change>=0 && change <nums.size() &&nums[change] == target)
      {
        change++;
        count++;
      }
      change = middle-1;
      while(change>=0 && change <nums.size() &&nums[change] == target)
      {
        change--;
        count++;
      }
    }
// another way
```

```
if(!flag) return 0;
int left,right;

left=middle-1;
right = middle;
while(right<nums.size() && nums[right] == target) right++;
while(left >= 0 && nums[left] == target) left--;
return right - left - 1;
•    return count;

  }
};
#
```

剑指 Offer 03. 数组中重复的数字

题干：找出数组中重复的数字。在一个长度为 n 的数组 nums 里的所有数字都在 0 ~ n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

分析：利用哈希表，键值为数组元素，value为次数

```
class Solution {
public:
  int findRepeatNumber(vector<int>& nums) {
•    unordered_map<int, int>hm;
•    for (auto num : nums)
•    {
•      if (hm[num]++) return num;
•    }
•    return -1;
  }
};
```

7、哈希

剑指 Offer 50. 第一个只出现一次的字符

题干：在字符串 s 中找出第一个只出现一次的字符。如果没有，返回一个单空格。 s 只包含小写字母。

分析：1、对于无序字符串，要找到次数为1的字符，，题目与3差不多，均需采用哈希表，时间复杂度为O(N)，即一定需要全部遍历完所有字符，才能知道谁出现次数小于2。。

2、优化的关键问题在于遍历完一次后，如何快速找到第一个字符，，想到可以用变量，，变量过程中若次数 == 1，则变量赋值。。。。这样的问题是，变量保存的是最后一个出现为一次的字符。（错，若是aabc，该思路不对）

3、有序哈希表：　　Map<Character, Boolean> dic = new LinkedHashMap<>();

```
class Solution {
public:
  // map 与 unorderedmap
  // map:
  /*优点:
```

```
    有序性，这是map结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作
    红黑树，内部实现一个红黑书使得map的很多操作在lgn的时间复杂度下就可以实现，因此效率非常的高
    缺点： 空间占用率高，因为map内部实现了红黑树，虽然提高了运行效率，但是因为每一个节点都需要额外保存父节点、孩子节点和红 / 黑性质，使得每一个节点都占用大量的空间
    适用处：对于那些有顺序要求的问题，用map会更高效一些
  */
  // unordered_map:
    /*优点： 因为内部实现了哈希表，因此其查找速度非常的快
    缺点： 哈希表的建立比较耗费时间
    适用处：对于查找问题，unordered_map会更加高效一些，因此遇到查找问题，常会考虑一下用unordered_map
    总结:
    */
  char firstUniqChar(string s) {
    map<char, int> m;
    for (auto s_sub : s)
    {
      m[s_sub] ++;
    }
    for (auto s_su : s)
    {
      if (m[s_su] == 1) return s_su;
    }
    return ' ';
  }
};
```

剑指 Offer 48. 最长不含重复字符的子字符串

题干：请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度。

分析：动态规划+hashmap。。。假设第i个字符最长子字符串为f(i) ，若f(i)不在f(i-1)中，则f(i) = f(i-1)+1.else f(i)=1;分析不完整，，当为ABCAD时，按上述分析，最长为3（ABC），实际上位4（BCAD）。故不能直接判断f(i)在不在f(i-1)中

```
class Solution {
public:
  // 利用哈希表存储，作用为快速查找。
  // 利用动态规划求得第i个字符所在的最大字符串
  int lengthOfLongestSubstring(string s) {
    unordered_map<char, int> hm;
    vector<int> result(s.size()+1);
    result[0] = 0;
    if(s.size() == 0) return 0;
    for (int i = 1; i <= s.size(); i++)
    {
      if (hm.find(s[i-1]) != hm.end()) // 找到相同元素
      {
        hm.clear();
        hm[s[i-1]] = i;
        result[i] = 1;
      }
      else
      {
        hm[s[i-1]] = i;
        result[i] = result[i - 1] + 1;
      }
    }
    return *max_element(result.begin(), result.end());
```

```
  }
};
```

进一步分析：考虑到上述问题，若有重复，则需找到重复值的索引i。比较当前最长长度len与索引差(j-i)，若j-i>len,说明当前长度中无重合数字，加1即可。。。若j-i<len，则需要将当前长度更改为j-i。

```
class Solution {
public:
  int lengthOfLongestSubstring(string s) {
    unordered_map<char, int> hm;
    int res = 0;  // 存储最大长度
    int cur = 0;  // 存储当前最大长度
    for (int i = 0; i < s.size(); i++)
    {
     // 有重复
      if (hm.find(s[i]) != hm.end()) i - hm[s[i]] > cur ? cur++ : cur = i - hm[s[i]];
      else cur++;

      hm[s[i]] = i;
      res = max(res, cur);
    }
    return res;
  }
};
#
```

8、链表

剑指 Offer 22. 链表中倒数第k个节点

题干：输入一个链表，输出该链表中倒数第k个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。例如，一个链表有6个节点，从头节点开始，它们的值依次是1、2、3、4、5、6。这个链表的倒数第3个节点是值为4的节点。

```
class Solution {
public:
  ListNode* getKthFromEnd(ListNode* head, int k) {
    if (!head) return NULL;
    if (!head->next && k > 1) return NULL;
    if (!head->next && k == 1) return head;
    ListNode *pslow = head;
    ListNode *pfast = head;
    for (int i = 1; i < k; i++)
    {
      if (!pfast) return NULL;  // 防止k过大
      pfast = pfast->next;
    }
    while (pfast->next)
    {
      pslow = pslow->next;
      pfast = pfast->next;
    }
```

```
    return pslow;
  }
}
```

剑指 Offer 06. 从尾到头打印链表

题干：输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

分析：考虑到要从后往前打印，且链表不能通过地址寻找，故利用栈的先入后出原则。

```
class Solution {
public:
  vector<int> reversePrint(ListNode* head) {
    stack<int> s;
    vector<int> result;
    if(!head) return result;
    // 入栈
    s.push(head->val);
    while (head->next)
    {
      head = head->next;
      s.push(head->val);
    }
    // 出栈放在数组。
    while (!s.empty())
    {
      result.push_back(s.top());
      s.pop();
    }
    return result;
  }
};
```

剑指 Offer 24. 反转链表

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

```
class Solution {
public:
  ListNode* reverseList(ListNode* head) {
    if(!head) return NULL;
    ListNode *pfront = head;
    ListNode *pback = head->next;
    while(pback)
    {
      ListNode *tmp = pback;
      pback = pback->next;
      tmp->next = pfront;
      pfront = tmp;
    }
    head->next = NULL;
    return pfront;
  }
};
```

剑指 Offer 52. 两个链表的第一个公共节点

题干：输入两个链表，找出它们的第一个公共节点。

分析：1、利用哈希表，表的key为结点地址，value为结点值，，当find到一样的值后，返回value

```
class Solution {
public:
  ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    if (!headA || !headB) return NULL;

    unordered_map<ListNode*, int> hm;
    while (headA)
    {
      hm[headA] = headA->val;
      headA = headA->next;
    }
    while (headB)
    {
      if (hm.find(headB) != hm.end()) return headB;
      else headB = headB->next; // HEADb无需存入
    }
    return NULL;
  }
};
```

2、走同样长的路，看能不能相遇。我们使用两个指针 node1，node2 分别指向两个链表 headA，headB 的头结点，然后同时分别逐结点遍历，当 node1 到达链表 headA 的末尾时，重新定位到链表 headB 的头结点；当 node2 到达链表 headB 的末尾时，重新定位到链表 headA 的头结点。这样，当它们相遇时，所指向的结点就是第一个公共结点。

```
class Solution {
public:
  ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    ListNode *nodeA = headA;
    ListNode *nodeB = headB;

    while (nodeA != nodeB)
    {
      nodeA = !nodeA ? headB : nodeA->next;     // 注：不能有！ nodeA->next，，node为空时，没有next
      nodeB = !nodeB ? headA : nodeB->next;     // 退出循环的条件为，两者都为NULL。或者交点。。若用                                                  //  !nodeA->next 则假设无交点，则永远不会相等。
    }
    return nodeA;
  }
};
#
```

剑指 Offer 18. 删除链表的节点

题干：给定单向链表的头指针和一个要删除的**节点的值**，定义一个函数删除该节点。返回删除后的链表的头节点。

分析：

```
class Solution {
public:
  ListNode* deleteNode(ListNode* head, int val) {
    if (!head) return NULL; // 若空，返回NULL
    ListNode *pfirst = new ListNode; // 创建伪头节点，
```

```
    pfirst ->next = head;
    ListNode *pchange = head;
    ListNode *pfront = pfirst;
    while (pchange)
    {
      if (pchange->val != val) pchange = pchange->next;
      else
      {
        pfront->next = pchange->next;
        return pfirst->next;
      }
      pfront = pfront->next; // 防止删除head
    }
    return NULL;
  }
};
#
```

剑指 Offer 35. 复杂链表的复制

请实现 copyRandomList 函数，复制一个复杂链表。在复杂链表中，每个节点除了有一个 next 指针指向下一个节点，还有一个 random 指针指向链表中的任意节点或者 null。

剑指 Offer 49. 丑数

题干：我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 n 个丑数。

分析：

设已知长度为 n 的丑数序列 $x_1, x_2, \cdots, x_n$ ，求第 $n+1$ 个丑数 x_{n+1} 。根根据递推性质，丑数 x_{n+1} 只可能是以下三种情况其中之一（索引 a, b, c 为未知数）：

$$x_{n+1} = \begin{cases} x_a \times 2 & , a \in [1, n] \\ x_b \times 3 & , b \in [1, n] \\ x_c \times 5 & , c \in [1, n] \end{cases}$$

a、b、c分别为距离最近的满足要求的丑数，因此关键在于如何找到xa、xb、xc。由于数组是递增的，所以可知，如果2=2*1，下次用到2乘的数一定要是2乘2，即4.。3乘5乘也类似，与他们相乘的丑数也是会递增的。因此，

```
class Solution {
public:
  int nthUglyNumber(int n) {
    vector<int> dp(n+1,0);
    dp[0] = 1;
    int a=0;int b=0;int c = 0;
    for(int i =1;i< n;i++)
    {
      dp[i] = min(min(dp[a]*2,dp[b]*3),dp[c]*5);
      if(dp[i] == dp[a]*2) a++;
      if(dp[i] == dp[b]*3) b++;
      if(dp[i] == dp[c]*5) c++;
    }
    return dp[n-1];
  }
};
```

未分组

剑指 Offer 16. 数值的整数次方

题干：实现函数double Power(double base, int exponent)，求base的exponent次方。不得使用库函数，同时不需要考虑大数问题。

分析：首先想到的是使用递归进行计算。但是，指数幂容易产生指数爆炸，需要一种方法避免指数爆炸。本题中采用的是快速幂算法。详细讲解可见，非常易懂https://blog.csdn.net/qq_19782019/article/details/85621386

可一步一步分析：产生指数爆炸的主要问题在于指数幂过大，因此需要想办法减小幂指数。以2^10为例，2^10 = (2*2)^5，这样就将原来的10变为了5（可当作二分）；以此类推，4^5 = 4 x 4^4=4 x 16^2 = 4 x 32^1 = 4 x 32。上述操作可以分为：1、当指数幂为偶数时，底数平方，指数减半； 2、指数幂为奇数时，分离出一个底数之后指数变为偶数，继续1的操作。。到最后可以总结出规律，即2^10实际上为指数为奇数的底数乘积（4 x 32；二者指数均为1）。那么代码可写为：

```
class Solution {

public:

  double myPow(double x, int n) {

•    double result = 1;  // 用来存放指数为奇数的底数乘积

•    long b1 = n; // 主要是解决 n = -2147483648时, n = -n超出界限

•    if(n < 0)   // n<0时，需修改一下才能适用于下面代码

•    {

•      x = 1/x;

•      b1 = -b1;

•    }

•    while(b1 > 0 )

•    {

•      if(b1 % 2) // 指数为奇数时，分离出一个底数，再将底数平方，指数减半

•      {

•        result = result * x;

•      }

•      x = x*x;

•      b1 = b1/2;
```

```
    }
    return result;
  }
};
```

可进一步修改：

1、奇偶数判断：原先用b1 % 2,现用b1 & 1（结果为0，偶数；结果为1，奇数）

2、除2操作，b1>>1；左移一位相当于除以2.

```
class Solution {
public:
  double myPow(double x, int n) {
    double result = 1;  // 用来存放指数为奇数的底数乘积
    long b1 = n; // 主要是解决 n = -2147483648时, n = -n超出界限
    if(n < 0)   // n<0时，需修改一下才能适用于下面代码
    {
      x = 1/x;
      b1 = -b1;
    }
    while(b1 > 0 )
    {
      if(b1 & 1) // 指数为奇数时，分离出一个底数，再将底数平方，指数减半
      {
        result = result * x;
      }
      x = x*x;
      b1 >>= 1;
    }
    return result;
  }
};
```

剑指 Offer 59 - I. 滑动窗口的最大值

题干：给定一个数组 nums 和滑动窗口的大小 k，请找出所有滑动窗口里的最大值。

分析：此题和59题 队列最大值类似，都需要通过队列先入先出的原则进行求解。每次需要在窗口中找到最大值，故可以考虑将最大值放入队列，，但若窗口最大值出去，则无法判断剩下当中谁是次大值，需要重新计算最大值。。。。因此参考59的解题方式，在窗口移动时，比较即将进入的值与已有值得大小，，若队列中有值小于该值，则pop。。这一步可以使队列中处于不递增的状态，，能保证队列开头为最大值。。

```
class Solution {
public:
  vector<int> maxSlidingWindow(vector<int>& nums, int k) {
•    vector<int> result; // 存放窗口最大值
•    deque<int> d;  // 双端队列，便于队尾pop
•    if(nums.empty()) return result;    // 注：2、 为判断空数组，出错
•    // 构建第一个窗口
•    d.push_back(nums[0]);
•    for (int i = 1; i < k; i++)
•    {
•      while (!d.empty() && nums[i] > d.back() ) d.pop_back();   // 注：3、pop前未判断对列是否为空
•      d.push_back(nums[i]);
•    }
•    result.push_back(d.front()); // 第一个窗口的最大值
•    // 开始循环
•    for (int j = k; j < nums.size(); j++)
•    {
•      while (!d.empty() && nums[j] > d.back()) d.pop_back();  // 若新进元素大于之前队列元素，则pop
•      if (!d.empty() && nums[j - k] == d.front()) d.pop_front(); // 若准备出去的元素为队列最大元素，则pop
•      d.push_back(nums[j]);   // 注：1、忘记push，结果出错
•      result.push_back(d.front());
•    }
•    return result;
  }
};
```

剑指 Offer 58 - I. 翻转单词顺序

题干：输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。为简单起见，标点符号和普通字母一样处理。例如输入字符串"I am a student. "，则输出"student. a am I"。

分析：1、利用栈辅助逆序：出现空格判断第一个栈是否为空，若不为空，则将栈1送入栈2，并在最后入一空格。

```
class Solution {
public:
    void EraseSpace(string &s)
    {
        //ch可换成其他字符
        const char ch = ' ';
        s.erase(s.find_last_not_of(" ") + 1);
        s.erase(0, s.find_first_not_of(" "));
    }
    string reverseWords(string s) {
        string result;
        bool flag = 1;
        EraseSpace(s);
        // 若不为空格，则入栈1，遇到空格，判断栈1是否为空，不空则将栈1数据入栈2
```

```
        for (auto c : s)
        {
            if (c != ' ')
            {
                s1.push(c);
            }
            else
            {
                while (!s1.empty())
                {
                    s2.push(s1.top());
                    s1.pop();
                    if(s1.empty()) s2.push(' ');
                }
            }
        }
        while (!s1.empty())
        {
            s2.push(s1.top());
            s1.pop();
        }
        while (!s2.empty())
        {
            result.push_back(s2.top());
            s2.pop();
        }
        return result;
    }
private:
    stack<char> s1;
    stack<char> s2;
};
```

2、双指针：从后往前，当遇到空格，则将两个指针之间的内容

```
class Solution {
public:
    string reverseWords(string s) {
        string res;
        int front = s.size() - 1;
        int end = s.size() - 1;

        while(front >= 0 )
        {
            if (s[front] != ' ') front--;
            else
            {
                res.append(s,front + 1, end - front);
                res += ' ';
                while (s[front] == ' ') front--;
                end = front;
            }
        }
        res.append(s, front + 1, end - front);
        return res;
    }
};
```