3、数组中重复的数字

```
// class Solution {
// public:
//     int findRepeatNumber(vector<int>& nums) {
//        for(int i=0;i<nums.size();i++){
//            while(nums[i]!=i){
//                if(nums[i]==nums[nums[i]]){return nums[i];}
//                swap(nums[i],nums[nums[i]]);
//            }
//        }
//        return -1;
//     }
// };

// class Solution {
// public:
//     int findRepeatNumber(vector<int>& nums) {
//         sort(nums.begin(),nums.end());
//         for(int i = 0; i < nums.size()-1; ++i)
//         {
//             if(nums[i] == nums[i+1])    return nums[i];
//         }
//         return -1;
//     }
// };

class Solution{
    public:
    int findRepeatNumber(vector<int>& nums){
        unordered_set<int> m; //unordered_map<int,int> m;
        for(int i : nums){    // for(int num:nums)
          if(m.count(i)!=0) return i;else m.insert(i);
                            // if(++m[num]>1) return num;
        }
        return -1;

    }
};
```

4、二维数组的查找

```
class Solution {//暴力查找
public:
    bool findNumberIn2DArray(vector<vector<int>>& matrix, int target) {
    int i= matrix.size()-1,j=0;
    while(i>=0 && j<matrix[0].size()){
        if(matrix[i][j]>target) i--;
        else if(matrix[i][j]<target) j++;
             else return true;
    }
    return false;
    }
};
```

5、替换空格

```
class Solution {
public:
    string replaceSpace(string s) {
        int count=0, len=s.size();
        //统计空格数量
        for(char c : s){
            if(c==' '){count++;}//''中间有空格
        }
        //修改s长度
        s.resize(len+2*count);
        //倒序遍历修改
        for(int i= len-1,j=s.size()-1;i>=0;i--,j--){
            if(s[i]!=' '){
                s[j]=s[i];
            }
            else{
                s[j]='0';
                s[j-1]='2';
                s[j-2]='%';
                j=j-2;
            }
        }
        return s; //所以判别式可以为i<j
    }
};

//遍历方式学习
//字符串修改大小
```

6、从尾到头打印链表

```
class Solution {
public:
    vector<int> res;
    vector<int> reversePrint(ListNode* head) {
        if(head!=NULL){
            if(head->next!=NULL){
                reversePrint(head->next);
            }
            res.push_back(head->val);
        }
        return res;
    }
};
```

7、重建二叉树

```
/**
 * 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 {
    int index = 0;
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
         TreeNode* head;
         int left = 0;
         int right = inorder.size();
         head = rebuild(preorder, inorder, left, right);
         return head;
    }

    TreeNode* rebuild(vector<int>& preorder, vector<int>& inorder, int left, int
right){
        if (index == preorder.size()||left == right) return NULL;
        TreeNode *head = NULL;
        for (int i=left;i<right;i++){
            if (preorder[index]==inorder[i]){
                head = new TreeNode(preorder[index]);
                index++;
                head->left = rebuild(preorder, inorder, left,i);
                head->right = rebuild(preorder,inorder,i+1,right);
                break;
            }
        }
        return head;
    }
};
```

9、用两个栈实现队列

```
class CQueue {
    stack<int> stack1,stack2;
public:
    CQueue() {
       while(!stack1.empty()){
           stack1.pop();
       }
       while(!stack2.empty()){
           stack2.pop();
       }
    }//初始化

    void appendTail(int value) {
        stack1.push(value);
    }

    int deleteHead() {
        if(stack2.empty()){
            while(!stack1.empty()){
                stack2.push(stack1.top());
                stack1.pop();
            }
        }
        if(stack2.empty()){
            return -1;
        }else{
            int deleteelem = stack2.top();
            stack2.pop();
            return deleteelem;
        }
    }
};
```

10、斐波那契数列

```
class Solution{
    public:
    int fib(int n){
    int result[2]={0,1};
    long long fib0 = 0 , fib1 =1 ,fibN=0;
    if(n<=1) {
        return result[n];
    }

    else{
        for(int i=2;i<=n;i++){
           fibN = (fib0 + fib1)%1000000007; //取模 题目要求
           fib0 = fib1;
           fib1 = fibN;
        }
    return fibN;}
    }
};

//青蛙跳台阶
class Solution {
```

```
public:
    int numWays(int n) {
      int result[2]={1,1};
      int fib0=1, fib1=1,fibN=0;
      if(n<=1){return result[n];}
      else{
          for(int i=2;i<=n;i++){
              fibN = (fib0+fib1)%1000000007;
              fib0 = fib1;
              fib1 = fibN;
           }
        return fibN;}
    }
};
```

11、旋转数组的最小数字

```
class Solution {
public:
    int minArray(vector<int>& numbers) {
        int left=0,right=numbers.size()-1;
        if (right-left==1){
            if(numbers[left]<=numbers[right]){right = left;}
        }//[1,2][2,1]
        else{
            while(right-left>1){
            int mid = (left+right)/2;
            if (numbers[left]<numbers[right]){
                right = left; break;
            }//[1,2,3,4][3,1,3]
            if(numbers[left]==numbers[right] && numbers[left]==numbers[mid]){
                right -= 1;
            }
            else if(numbers[left]<=numbers[mid]){
                left = mid;
            }
            else if(numbers[right]>=numbers[mid]){
                right = mid;
            }
        }
        }
       return numbers[right];
    }
};
```

12、矩阵中的路径

```
class Solution {
public:
    bool exist(vector<vector<char>>& board, string word) {
        rows = board.size();
        cols = board[0].size();
        for(int i=0;i<=rows-1;i++){
            for(int j=0;j<=cols-1;j++){
                if(back(board,word,i,j,0)) return true;//public里要返回结果
```

```
            }
        }
        return false;
    }
private:
    int rows,cols;
    bool back(vector<vector<char>>& board, string word, int i, int j, int k){
        if(i>=rows || i<0 || j<0 || j>=cols || board[i][j]!=word[k]) return
false;
        if(k==word.size()-1) return true;
        board[i][j]='\0';//改变board了//走过的路径用空字符'\0'表示
        bool res = back(board,word,i+1,j,k+1) || back(board,word,i,j+1,k+1) ||
back(board,word,i-1,j,k+1) || back(board,word,i,j-1,k+1);
        board[i][j]=word[k];
        return res;
    }
};
```

13、机器人运动范围

```
// class Solution {
// public:
//     int movingCount(int m, int n, int k) {
//         vector<vector<bool>> visited(m,vector<bool>(n,0));//辅助矩阵
//         return dfs(0,0,visited, m, n ,k);
//     }
// private:
//     int dfs(int i, int j, vector<vector<bool>> &visited, int m, int n, int k)
{
//         if(i>=m || j>=n || sums(i)+sums(j)>k || visited[i][j]) return 0;
//         visited[i][j]= true;
//         return 1+dfs(i+1,j,visited,m,n,k)+dfs(i,j+1,visited,m,n,k);//计数和寻找
路径不一样
//     }
//     int sums(int x){
//         int sum=0;
//         while(x>0){
//             sum += x%10;
//             x = x/10;
//         }
//         return sum;
//     }
// }; ///DFS

class Solution{
    public:
        int movingCount(int m, int n, int k){
            vector<vector<bool>> visited(m,vector<bool>(n,0));
            int res = 0;
            queue<vector<int>> que;
            que.push({0,0,0,0});
            while(que.size()>0){
                vector<int> x = que.front();
                que.pop();
                int i = x[0], j=x[1], si=x[2], sj=x[3];
                if(i>=m||j>=n||k<si+sj||visited[i][j]) continue;
```

```
                visited[i][j]=1;
                res++;
                que.push({i+1,j,(i+1)%10!=0?si+1:si-8,sj});
                que.push({i,j+1,si, (j+1)%10!=0?sj+1:sj-8});
            }
            return res;
    }
};
```

14、剪绳子

```
class Solution {
public:
    int cuttingRope(int n) {
        if(n<2) return 0;
        if(n==2) return 1;
        if(n==3) return 2; ///少于3的时候必须减return的是必剪之后的最大值  不剪的话是最大值//因此要分类讨论

        int* products = new int[n+1];
        products[0]=0;
        products[1]=1;
        products[2]=2;
        products[3]=3;
        int max=0;
        for(int i=4;i<=n;++i){
            max =0 ;
            for(int j=1;j<= i/2;++j){
                int product = products[j]*products[i-j];
                if(max<product){
                    max = product;
                }
                products[i]=max;
            }
        }
        max = products[n];
        delete[] products;
        return max;
    }
};

class Solution {
public:
    int cuttingRope(int n) {
        if(n<=3) return n-1;
        if(n==4) return 4;
        long res = 1;
        while(n>4){
            res = res*3%1000000007;
            n -=3;
        }
        return res*n%1000000007;
    }
};//贪婪算法
```

15、二进制中1的个数

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

16、数值的整数次方

```
class Solution {
public:
    double myPow(double x, int n) {
     long nums = n;
     if(n==0 || x==0 ||x==1) return 1;
     if(n<0) {
         x=1/x;
         nums=-nums;      ///直接转换会溢出
     }
     double ans = 1;

     while(nums){  //快速幂
         if(nums & 1) ans*=x;
         x*=x;
         nums = nums>>1;
     }
    return ans;
    }
};
```

17、打印从1到最大的n位数

```
class Solution {
public:
    vector<int> ans;

    vector<int> printNumbers(int n) {
         string s(n,'0');
    if(n<=0) return vector<int>(0);
        while(!overflow(s))
            input(s);
        return ans;
    }

    bool overflow(string& s){
```

```
        bool flag = false;
        int carry = 0;
        for(int i=s.size()-1;i>=0;i--){
           int current = s[i]-'0'+carry;
           if(i==s.size()-1) current++;
           if(current>=10){
               if(i==0) {flag= true;}
               else{
                    carry = 1;
                    s[i]=current-10+'0';
               }
           }else{s[i]=current+'0';break;}
        }
        return flag;
    }

    void input(string& s){
        bool flagzero = false;
        string tmp = "";
        for(int i=0;i<s.size();i++){
            if(!flagzero && s[i]!='0'){flagzero = true;}
            if(flagzero) tmp = tmp + s[i];
        }
        ans.push_back(stoi(tmp));
    }
};
```

18. 删除链表的节点

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* deleteNode(ListNode* head, int val) {
    if(!head) return NULL;
    if(head->val == val) return head->next;
    ListNode* ans=head;
    while(ans->next!=NULL && ans->next->val != val){
        ans = ans->next;
    }

    if(ans->next!=NULL){
        ans->next = ans->next->next;
    }
    return head;///不是return ans;

    }
};
```

20、表示数值的字符串

```
class Solution {
public:
    bool isNumber(string s) {
        //A[.[B]][e|EC]//.B[e|EC]
        //A带符号的整数；B无符号整数，e后面必有一个（有符号）整数，小数点前后必有数字，空格只能出现在字符串首尾；
        int flag=0;//用来标记是否检测到数字
        //空，直接返回
        if(s=="") return false;
        //检测字符串之前是否有空格
        while(s[0]==' ') s=s.substr(1);
        //遇到正负号，向后移
        if(s[0]=='+'||s[0]=='-') s=s.substr(1);
        //检测小数点前是否有数字，有的话后移，并标记
        while(((s[0]-'0')>=0) && ((s[0]-'0')<=9)){
            s=s.substr(1);flag=1;
        }
        //如果后面是.,那就向后移，有.就要判断后面有没有数字
        if(s[0]=='.'){
            s=s.substr(1);
            while(((s[0]-'0')>=0) && ((s[0]-'0')<=9)){
                s=s.substr(1);flag=1;
            }
        }
        //判断前半部分有没有数字
        if(flag==0) return false;
        flag=0;
        //接下来判断是有e|E;与上面是串联关系，所以用if不用else
        //如果存在e|E，那就一定要判断，它后面是否跟了数字
        if(s[0]=='e'||s[0]=='E'){
            s=s.substr(1);
            //判断有没有正负号
            if(s[0]=='+'||s[0]=='-') s=s.substr(1);
            //判断整数
            while(((s[0]-'0')>=0) && ((s[0]-'0')<=9)){
                s=s.substr(1);flag=1;
            }
            //如果有e没有数字，那就出错了
            if(flag==0) return false;
        }
        //判断空格结尾
        while(s[0]==' ') s=s.substr(1);
        //如果结束了那就是true，如果还有其他字母，那就false
        if(s=="") return true;
        return false;
    }
};
```

思路：首先字符串只能是两种形式A[.[B]][e|EC]//.B[e|EC]
明确一些情况：
A带符号的整数；B无符号整数，
小数点前后必有数字，e后面必有一个（有符号）整数，空格只能出现在字符串首尾；

21、调整数组顺序使奇数位于偶数前面

```
class Solution {
public:
    vector<int> exchange(vector<int>& nums) {
        int low = 0, fast = 0;
        while (fast < nums.size()) {
            if (nums[fast] & 1) {
                swap(nums[low], nums[fast]);
                low ++;
            }
            fast ++;
        }
        return nums;
    }
};//快慢双指针

//首尾双指针
class Solution {
public:
    vector<int> exchange(vector<int>& nums) {
        int left = 0, right = nums.size() - 1;
        while (left < right) {
            if ((nums[left] & 1) != 0) {
                left ++;
                continue;
            }
            if ((nums[right] & 1) != 1) {
                right --;
                continue;
            }
            swap(nums[left++], nums[right--]);
        }
        return nums;
    }
};
```

22、链表总结

```
class Solution {
public:
    ListNode* getKthFromEnd(ListNode* head, int k) {
        ListNode *p = head, *q = head; //初始化
        while(k--) {   //将 p指针移动 k 次
            p = p->next;
        }
        while(p != nullptr) {//同时移动，直到 p == nullptr
            p = p->next;
            q = q->next;
        }
        return q;
    }
};
//中间元素
class Solution {
```

```
public:
    ListNode* middleNode(ListNode* head) {
        ListNode *p = head, *q = head;
        while(q != nullptr && q->next != nullptr) {
            p = p->next;
            q = q->next->next;
        }
        return p;
    }
};

//是否为环
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *slow = head;
        ListNode *fast = head;
        while(fast != nullptr) {
            fast = fast->next;
            if(fast != nullptr) {
                fast = fast->next;
            }
            if(fast == slow) {
                return true;
            }
            slow = slow->next;
        }
        return nullptr;
    }
};
```

24、反转链表

```
///递归
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if( head==NULL || head->next == NULL){return head;};
        ListNode* ret=reverseList(head->next);
        head->next->next= head;
        head->next=NULL;
        return ret;
    }
};

//双指针
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
     ListNode* cur = NULL , *pre = head;
     while(pre!=NULL){
         ListNode* t = pre->next; //保存结点
         pre->next =cur; //翻转
         // 更新两个指针
         cur = pre;
```

```
            pre = t;
        }
        return cur;
    }
};
```

25、合并两个排序链表

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    if(l1==NULL) {return l2;
    }else if(l2==NULL) {return l1;
    }else if(l1->val < l2->val){
        l1->next = mergeTwoLists(l1->next ,l2);
        return l1;
    }else{
        l2->next = mergeTwoLists(l1, l2->next);
        return l2;
    }

    }
};///迭代

class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {

    ListNode* mergedhead = new ListNode(-1);
    ListNode* prev = mergedhead;

    while(l1 != NULL && l2 != NULL){
       if(l1->val < l2->val){
        prev -> next = l1;
        l1 = l1->next;
       }else{
        prev -> next = l2;
        l2 = l2->next;
       }
       prev = prev -> next;
    }

    prev-> next = l1 == NULL? l2 : l1; ///l1和l2长度不一样
    return mergedhead -> next;

    }
};  //递归
```

26、数的子结构

*每次使用指针时，判断一下有没有可能是nullptr

```
class Solution {
public:
    bool isSubStructure(TreeNode* A, TreeNode* B) {
```

```
        if(!A||!B) return false;
        bool result = false;
        // if(A!=NULL && B!=NULL){
            if(A->val == B->val){
                result = issubtree(A,B);
            }

            if(!result) result = isSubStructure(A->left,B);//result=
            if(!result) result = isSubStructure(A->right,B);
        // }
        return result;
    }

    bool issubtree(TreeNode *A1, TreeNode *B1){
        if(!B1) return true;//b已经遍历完了，true 先判断b遍历完了嘛，在判断a //终止条件
        if(!A1) return false;
        if(A1->val != B1->val) return false;
        //相等的话才考虑左右结点
        return issubtree(A1->left,B1->left) && issubtree(A1->right,B1->right)  ;
    }
};
```

27、交换镜像

```
class Solution {
public:
    TreeNode* mirrorTree(TreeNode* root) {
        if(!root || (!root -> left && !root -> right)) return root;
        swapNode(root);  //先头部交换
        mirrorTree(root -> left);//
        mirrorTree(root -> right);//
        return root;
    }
    void swapNode(TreeNode *node)
    {
        if(!node || (!node -> left && !node -> right)) return;
        TreeNode *temp = node -> left;
        node -> left = node -> right;
        node -> right = temp;
    }
};
```

28、判断对称二叉树

```
class Solution {
public:
    bool compare(TreeNode* left, TreeNode* right) {

        if (left == NULL && right == NULL) return true;///终止条件
        if (left == NULL || right == NULL) return false;
        if (left->val != right->val) return false;

        // 此时就是：左右节点都不为空，且数值相同的情况
        // 此时才做递归，做下一层的判断
```

```
        bool outside = compare(left->left, right->right);   // 左子树：左、 右子树：右
        bool inside = compare(left->right, right->left);    // 左子树：右、 右子树：左
        bool isSame = outside && inside;
        return isSame;
    }

    bool isSymmetric(TreeNode* root) {
        if (root == NULL) return true;
        return compare(root->left, root->right);
    }
};
```

29、顺时针打印矩阵

```
class Solution {
private:
    static constexpr int directions[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};//用来表示方向的
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        if (matrix.size() == 0 || matrix[0].size() == 0) {
            return {};
        }

        int rows = matrix.size(), columns = matrix[0].size();
        vector<vector<bool>> visited(rows, vector<bool>(columns));
        int total = rows * columns;
        vector<int> order(total);

        int row = 0, column = 0;
        int directionIndex = 0;
        for (int i = 0; i < total; i++) {
            order[i] = matrix[row][column];
            visited[row][column] = true;
            int nextRow = row + directions[directionIndex][0], nextColumn = column + directions[directionIndex][1];
            if (nextRow < 0 || nextRow >= rows || nextColumn < 0 || nextColumn >= columns || visited[nextRow][nextColumn]) {
                directionIndex = (directionIndex + 1) % 4;
            }///这个地方很巧妙啊
            row += directions[directionIndex][0];
            column += directions[directionIndex][1];
        }
        return order;
    }
};

class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        if (matrix.size() == 0 || matrix[0].size() == 0) {
            return {};
        }
```

```
        int rows = matrix.size(), columns = matrix[0].size();
        vector<int> order;
        int left = 0, right = columns - 1, top = 0, bottom = rows - 1;
        while (left <= right && top <= bottom) {
            for (int column = left; column <= right; column++) {
                order.push_back(matrix[top][column]);
            }
            for (int row = top + 1; row <= bottom; row++) {
                order.push_back(matrix[row][right]);
            }
            if (left < right && top < bottom) {
                for (int column = right - 1; column > left; column--) {
                    order.push_back(matrix[bottom][column]);
                }
                for (int row = bottom; row > top; row--) {
                    order.push_back(matrix[row][left]);
                }
            }
            left++;
            right--;
            top++;
            bottom--;
        }
        return order;
    }
};///一圈一圈打印
```

30、 包含min函数的栈

```
class MinStack {
public:
    stack<int> stk_data;
    stack<int> stk_min;
    /** initialize your data structure here. */
    MinStack() {

    }

    void push(int x) {
        stk_data.push(x);
        if(stk_min.empty()||x<stk_min.top()){
            stk_min.push(x);
        }else { stk_min.push(stk_min.top());}

    }

    void pop() {
        stk_data.pop();
        stk_min.pop();

    }

    int top() {
        return stk_data.top();

    }
```

```
    int min() {
        return stk_min.top();

    }
};
```

31、 栈的压入、弹出序列

```
class Solution {
public:
    stack<int> stk;
    bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
        int index = 0; //pop的index
        for(int i=0; i<pushed.size();i++){///for(auto n: pushed)
            stk.push(pushed[i]);
            while(!stk.empty() && index < popped.size() &&
stk.top()==popped[index]){
                stk.pop(); index++;
            }
        }
        return stk.empty();
    }
};
```

32、从上到下打印二叉树

```
class Solution {
public:
    vector<int> levelOrder(TreeNode* root) {
        vector<int> res; //存结果
        if(!root) return res;
        queue<TreeNode*> q;//存结点
        q.push(root);
        while(!q.empty()){
            int len = q.size();
            for(int i=0;i<len;i++){ ////i是从0开始的
                TreeNode* node = q.front();
                q.pop();
                res.push_back(node->val);
                if(node->left) q.push(node->left);
                if(node->right) q.push(node->right);
            }
        }
    return res;
    }
};

class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res;
        if(!root) return res;
```

```
        queue<TreeNode*> q;//存结点
        q.push(root);
        int level=0;
        while(!q.empty()){
            vector<int> tmp;//临时储存值
            int len=q.size();
            for(int i=0;i<len;i++){ ////i是从0开始的
                TreeNode* node = q.front();
                q.pop();
                tmp.push_back(node->val);
                if(node->left) q.push(node->left);
                if(node->right) q.push(node->right);
            }
           if(level%2==1){
                reverse(tmp.begin(),tmp.end());
            }
           res.push_back(tmp);
           level++;
        }
    return res;
    }
};//按层遍历
```

33、二叉树的后序遍历

```
class Solution {
public:
    bool verifyPostorder(vector<int>& postorder) {
        if(postorder.empty()) return true;
        int len = postorder.size();
        return recursion(postorder,0,len-1);
    }

    bool recursion(vector<int>& postorder, int left, int right){
        if (left >= right){
            return true;}//终止条件
     int root = postorder[right];

     int i=left; //i值保存下来
     for(;i<right;i++){
         if(postorder[i] > root)
          break;
     }

     for(int j=i;j<right;j++){
         if(postorder[j] < root)
           return false;
     }

     bool ans = (recursion(postorder,left,i-1) && recursion(postorder,i,right-
1));
     return ans;
    }
};
// 前序
// class Solution {
```

```
// public:
//     vector<int> preorderTraversal(TreeNode* root) {
//         vector<int> res;
//         preorder(root,res);
//         return res;
//     }
//     void preorder(TreeNode* root, vector<int> &res){
//         if(root==nullptr){
//             return;
//         }
//         res.push_back(root->val);
//         preorder(root->left,res);
//         preorder(root->right,res);
//     }
// }; ///递归法
///迭代法，用栈实现递归 ///栈显示出来
class Solution{
    public:
    vector<int> preorderTraversal(TreeNode* root){
       vector<int> res;
       if(root==nullptr){
           return res;
       }

       stack<TreeNode*> stk;
    //    TreeNode* node=root;
    //    while(!stk.empty() || node!=nullptr){
    //        while(node!=nullptr){
    //            res.emplace_back(node->val);
    //            stk.emplace(node);
    //            node=node->left;
    //        }
    //        node=stk.top();
    //        stk.pop();
    //        node=node->right;
    //    }
       while(!stk.empty() || root!=nullptr){
           while(root!=nullptr){
               res.emplace_back(root->val);
               stk.emplace(root);
               root=root->left;
           }
           root=stk.top();
           stk.pop();
           root=root->right;
       }
    return res;
    }
};
```

35、复杂链表的复制

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
```

```
    if(head == nullptr) return nullptr;
    Node* cur = head;
    while(cur!=NULL){
        Node* tmp = new Node(cur->val);
        tmp->next = cur->next;
        cur->next = tmp;
        cur = tmp->next;//++
    }///step1: 链接在原始的后面

    cur = head;
    while(cur!=NULL){
        if(cur->random!=NULL){
            cur->next->random = cur->random->next;
        }
        cur = cur->next->next;//++
    }///step2: 链接random

    Node* cloned = head->next;
    Node* pre = head;
    Node* res = head->next;
    while(cloned->next!=NULL){
        pre->next = pre->next->next;
        cloned->next = cloned->next->next;
        pre = pre->next;//++
        cloned = cloned->next;//++
    }
    pre->next = nullptr;//单独处理原链表
    return res;

    }
};
```

36、二叉搜索树与双向链表

```
class Solution {
public:
    Node* treeToDoublyList(Node* root) {
        if(root == nullptr) return nullptr;
        inorder(root);
        head->left = pre;
        pre->right = head;
        return head;
    }
private:
    Node *pre, *head;
    void inorder(Node* cur) {
        if(cur == nullptr) return;

        inorder(cur->left);//左

        if(!pre) head = cur; ///pre从nullptr开始
        else pre->right = cur;
        cur->left = pre;
        pre = cur;
```

```
        inorder(cur->right);//右
    }
};
```

37、序列化二叉树

```
class Codec {
public:

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        string data;
        queue<TreeNode*> que;
        if(root) que.push(root);

        while(!que.empty()){
            auto cur = que.front();
            que.pop();

            if(cur){
                data += to_string(cur->val) +',';
                que.push(cur->left);
                que.push(cur->right);
            }else{data += "null,";}
        }

        if(!data.empty()) data.pop_back();//去掉最后一个逗号
        return data;
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        if(data.empty()) {return NULL;}
        istringstream iss(data);
        string tmp="";
        getline(iss,tmp,',');
        TreeNode *root = new TreeNode(stoi(tmp));
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()){
            TreeNode *node = q.front();
            q.pop();
            getline(iss,tmp,',');///左子树
            if(tmp=="null"){
                node->left = NULL;
            }else{
                node->left = new TreeNode(stoi(tmp));
                q.push(node->left);
            }
            tmp = ""; ////右子树
            getline(iss,tmp,',');
            if(tmp=="null"){
                node->right = NULL;
            }else{
                node->right = new TreeNode(stoi(tmp));
```

```
                q.push(node->right);
            }
        }
        return root;
    }
};
```

38、 字符串排列

```
class Solution {
    string s_copy;
    vector<string> ans;
public:
    vector<string> permutation(string s) {
        if(s.empty()) return{};
        s_copy = s;
        perm(0);
        return ans;
    }

    void perm(int pos){
        set<int> st;
        if(pos == s_copy.size()-1) {ans.push_back(s_copy);}
        for(int i=pos;i<s_copy.size();i++){
            if(st.find(s_copy[i]) != st.end()) continue; // 重复，因此剪枝 特例"aab"
            st.insert(s_copy[i]);//相同的话交换还是同一个
            swap(s_copy[i],s_copy[pos]);
            perm(pos+1);
            swap(s_copy[i],s_copy[pos]);
        }
    }

};
```

39、数组中次数超过一半的数

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        // sort(nums.begin(),nums.end());
        // return nums[nums.size()/2];  //排序取中位数//也可通过partition函数

        // unordered_map<int,int> hash;
        // int res , len = nums.size();
        // for(int n:nums){
        //     hash[n]++;
        //     if (hash[n] > len/2)
        //        {res = n;break;}
        // }
        // return res;//建立hash表

        int res = 0, count =0;
        for(int i=0;i<nums.size();i++){
            if(count==0){
                res = nums[i];
```

```
                count++;
            }else{
                res == nums[i]? count++ : count--;
            }
        }
        return res;//摩尔投票法//若数组的前a个数字的票数和 = 0 ，则 数组剩余(n-a)个数字的票数和一定仍>0，即后(n-a)个数字的众数仍为 x
    }
};
```

40、最小的k个数

```
class Solution {
public:
    vector<int> getLeastNumbers(vector<int>& arr, int k) {
      vector<int> ans;
      if(k==0) return ans;//默认为空

      priority_queue<int> bin;
      for(int i=0;i<k;i++){
           bin.push(arr[i]);
      }
      for(int i=k; i<arr.size();i++){
          if(arr[i] < bin.top()){
              bin.pop();
              bin.push(arr[i]);
          }
      }
      for(int i=0;i<k;i++){
          ans.push_back(bin.top());
          bin.pop();
      }
      return ans;
    }///大根堆 o(nlogk)
};

class Solution {
public:
    vector<int> getLeastNumbers(vector<int>& arr, int k) {
        if (k >= arr.size()) return arr;
        return quickSort(arr, k, 0, arr.size() - 1);
    }
private:
    vector<int> quickSort(vector<int>& arr, int k, int l, int r) {
        int i = l, j = r;
        while (i < j) {
            while (i < j && arr[j] >= arr[l]) j--;
            while (i < j && arr[i] <= arr[l]) i++;
            swap(arr[i], arr[j]); ///哨兵是arr[i];
        }
        swap(arr[i], arr[l]);
        if (i > k) return quickSort(arr, k, l, i - 1);
        if (i < k) return quickSort(arr, k, i + 1, r);
        vector<int> res;
        res.assign(arr.begin(), arr.begin() + k);
```

```
        return res;
    }
};//基于partition
```

41、数据流的中位数

```
class MedianFinder {
public:
    priority_queue<int, vector<int>, less<int> > maxheap; //默认是大根堆
    priority_queue<int, vector<int>, greater<int> > minheap;//小根堆
    /** initialize your data structure here. */
    MedianFinder() {}

    void addNum(int num) {
        if(maxheap.size()==minheap.size()){
            maxheap.push(num);
            minheap.push(maxheap.top());
            maxheap.pop();
        }else{
            minheap.push(num);
            maxheap.push(minheap.top());
            minheap.pop();
        }
    }

    double findMedian() {
        int maxSize = maxheap.size(), minSize = minheap.size();
        int mid1 = maxheap.top(), mid2 = minheap.top();
        return maxSize == minSize ? ((mid1 + mid2) * 0.5) : mid2;
    }
};
```

42、连续子数组的最大和

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int res = nums[0];
        int n = nums.size();
        for( int i=1; i<n;i++){
            nums[i] += max(nums[i-1],0);
            res = max(nums[i],res);
        }
        return res; //没用辅助内存，直接在nums上改
    }
};

class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        vector<int> dp(nums.size(), nums[0]);
        int ans = dp[0];
        for(int i = 1; i < nums.size(); i ++){
```

```
            dp[i] = max(dp[i - 1] + nums[i], nums[i]);
            ans = max(ans, dp[i]);
        }
        return ans;
    }
};//辅助内存
```

43、1~n整数中1出现的次数

```
class Solution {
public:
    int countDigitOne(int n) {
       long digit = 1;
       int high = n/10, low = 0, cur = n%10;
       int res = 0;
       while(cur!=0 || high!=0){
           if(cur==0){
              res += high*digit;
           }else if(cur==1){
               res+= high*digit + low + 1;
           }else{
               res+= (high+1)*digit;
           }///判断不同数位出现1的次数

           low = low + cur*digit;
           cur = high%10;
           high = high/10;
           digit = digit*10;
       }
       return res;
    }
};
```

44、 数字序列中的某个数

```
class Solution {
public:
    int findNthDigit(int n) {
       if(n<0) return -1;
       int digits = 1;

       while(true){
          long length = lengthofdigit(digits); //长度//放循环里
          if(n<length){
           break;///不能在循环里return
        }else{
           n-= length;
           digits++;
        }
       }

       return countNum(n,digits);
    }
```

```
    long lengthofdigit(int digits){
        if(digits==1) return 10;
        else return 9*pow(10,digits-1)*digits;
    }

    int countNum(int n, int digits){
        int base = 0; //初始值
        if(digits!=1) base = pow(10,digits-1);
        int number = base + n/digits;

        //找对应的n%digits位
        int reindex = digits - n % digits;
        for(int i=1;i<reindex;i++){
            number = number/10;
        }
        return number%10;
    }
};
```

45、把数组排成最小的数

```
class Solution {
public:
    string minNumber(vector<int>& nums) {
        vector<string> strs;
        for(int i=0; i<nums.size();i++){
             strs.push_back(to_string(nums[i]));
        }
        sort(strs,0,strs.size()-1);
        //sort(strs.begin(), strs.end(), [](string& x, string& y){ return x + y
< y + x; });
        //sort(str_arr.begin(), str_arr.end(), Cmp());
        string res;
        for(string str : strs){
            res.append(str);
        }
        return res;
    }

    void sort(vector<string>& strs, int l, int r){
        if(l >= r) return;//终止条件
        int i = l, j = r;
        while(i<j){
            while(strs[j]+strs[l] >= strs[l]+strs[j] && i<j) j--;//先这一步
            while(strs[i]+strs[l] <= strs[l]+strs[i] && i<j) i++;//和轴点比
            swap(strs[i],strs[j]);
        }
        swap(strs[i],strs[l]);
        sort(strs,l,i-1);
        sort(strs,i+1,r);
    }
};//快排 mn<nm

struct Cmp {
```

```
    bool operator() (const string& s1, const string& s2) {
        if(s1[0] != s2[0]) return s1[0] < s2[0];
        return (s1+s2).compare(s2+s1) < 0;
        //string add1 = s1 + s2;string add2 = s2 + s1;
        //return add1 < add2;
    }
};//自己定义
```

46、把数字翻译成字符串

```
class Solution {
public:
    int translateNum(int num) {
        string vec = to_string(num);
        int p =0 ,q=0, r=1;
        for(int i=0; i < vec.size(); i++){
            p = q;
            q = r;
            r = 0;
            r += q;
            if(i==0){continue;}
            string pre = vec.substr(i-1,2);
            if(pre>="10" && pre<="25") {
                r+= p;
            }
        }
        return r;
    }
};//dp //p,q,r压缩内存

class Solution {
public:
    int translateNum(int num) {

        string nums = to_string(num);
        return recursive(nums,1,1,nums.size()-2);
    }
    int recursive(string nums,int a,int b,int n){
        if(n < 0){
            return b;
        }
        string tmp = nums.substr(n,2);//首位与长度
        int c = tmp.compare("10") >= 0 && tmp.compare("25") <= 0 ? a+b : b;
        a = b;
        b = c;
        return recursive(nums,a,b,n-1);
    }//递归
    //class Solution {
    //  public:
    //    int translateNum(int num) {
    //       string nums = to_string(num);
    //       int a = 1;
    //       int b = 1;
    //       int c;
    //       int len = nums.length();
    //        for(int i=2;i<=len;i++){
```

```
    //            string tmp = nums.substr(i-2,2);//首位与长度
    //            c = tmp.compare("10") >= 0 && tmp.compare("25") <= 0 ? a+b : b;
    //        //比较的函数返回的是int型
    //            a = b;
    //            b = c;
    //    }
    //    return b;
    //}
    //};

};
```

47、礼物的最大价值

```
class Solution {
public:
    int maxValue(vector<vector<int>>& grid) {
        if(grid.size()==0 && grid[0].size()==0){
            return 0;
        }
        int rows = grid.size(), columns = grid[0].size();
        vector<vector<int>> dp(rows, vector<int>(columns,0));
        for(int i=0; i<rows;i++){
            for(int j=0; j<columns;j++){
                int left=0, up=0;
                if(i>0) up=dp[i-1][j];
                if(j>0) left=dp[i][j-1];
                dp[i][j] = max(up,left)+grid[i][j];//当i,j为零时，已初始化边界
            }
        }
        return dp[rows-1][columns-1];
    }
};
```

48、最长不重复子字符串

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
       int maxlen = 0;
       unordered_set<char> chars;
       for(int i=0;i<s.size();i++){//每个元素开始的无重复子字符串
           int len = 1;
           chars.clear();
           chars.insert(s[i]);
           for(int j=i+1;j<s.size();j++){
               if(chars.count(s[j])){
                   break;
               }else{
                   chars.insert(s[j]);
                   len++;
               }
           }
           maxlen = max(maxlen,len);
```

```
        }
        return maxlen;
    }
};//两次遍历

class Solution{
public:
    int lengthOfLongestSubstring(string s) {//先移右再移左
      int maxlen = 0;
      if(s.size()==0) return 0;
      unordered_set<char> chars;
      int right = 0;
      for(int left=0; left<s.size();left++){
          while(right < s.size() && !chars.count(s[right])){
              chars.insert(s[right]);
              right++;
          }
          maxlen = max(maxlen,right-left);
          if(right == s.size()) break;
          chars.erase(s[left]);
      }
      return maxlen;
    }
};///双指针//一次遍历


//思路：以i结尾的不重复子串 //滑动窗口
class Solution{
public:
    int lengthOfLongestSubstring(string s) {//先移右再移左
      int maxlen = 0;
      if(s.size()==0) return 0;
      vector<int> m(128,0);
      int left = -1;
      for(int i=0; i<s.size();i++){
          left = max(left,m[s[i]]);
          m[s[i]]=i;//未来左指针能及时跳到重复的位置
          maxlen = max(maxlen,i-left);
      }
      return maxlen;
    }
};
```

49、丑数

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。

```
class Solution {
public:
    int nthUglyNumber(int n) {
        int dp[n];
        int a=0,b=0,c=0;
        dp[0]=1;
        for(int i=1;i<n;i++){
            int n2=dp[a]*2, n3=dp[b]*3, n5=dp[c]*5;
            dp[i]= min(min(n2,n3),n5);
```

```
            if(dp[i]==n2) a++;
            if(dp[i]==n3) b++;
            if(dp[i]==n5) c++;
        }
        return dp[n-1];
    }
};
```

50、第一个只出现一次的字符

```
class Solution {
public:
    char firstUniqChar(string s) {
         char ans=' ';
         if(s.size()==0) return ans;
         vector<int> hash(256,0);//不是vector<char>
         for(int i=0;i<s.size();i++){
            hash[s[i]]++;
         }
         for(int i=0;i<s.size();i++){
             if(hash[s[i]]==1){
                 ans = s[i];
                 break;
             }
         }
         return ans;
    }
};//自己写的
```

51、数组逆序对（归并排序）

```
class Solution {
public:
    int reversePairs(vector<int>& nums) {
      int len = nums.size();
      vector<int> tmp(len);
      return mergedsort(nums,tmp,0,len-1);
    }
    int mergedsort(vector<int>& nums, vector<int>& tmp, int l, int r){//tmp是全局
的
        if(l>=r) return 0;//中止条件
        int m = (l+r) / 2;
        int ans = mergedsort(nums,tmp,l,m) + mergedsort(nums,tmp,m+1,r);
        for(int k=l; k<=r; k++){
            tmp[k] = nums[k];
        }
        int i = l, j= m+1;
        for(int k=l;k<=r; k++){//重排l-r的序列
            if(i==m+1)
                nums[k]=tmp[j++];//i开始的短
            else if(j==r+1 || tmp[i]<=tmp[j])//j开始的短或者tmp[i]<=tmp[j]
                nums[k]=tmp[i++];
            else{
                nums[k]=tmp[j++];
```

```
                ans += m-i+1;//i后面的数都比tmp[j]大
            }
        }
        return ans;
    }
};
```

```
class Solution {
public:
    int reversePairs(vector<int>& nums) {
        if( nums.size() == 0 ) return 0;
        tmp.resize(nums.size());   // 辅助数组
        mergeSort(nums, 0, nums.size()-1);
        return res;
    }
private:
    vector<int> tmp; // 辅助数组
    int res;    // 记录逆序数
    void merge( vector<int>& nums, int start, int mid, int end ){
        if( start >= end ) return ;
        //printf("---%d, %d, %d---\n", start, mid, end);
        //merge:合并两个有序数组 nums[start, mid] 和 nums[mid+1, end]
        for( int i = start; i <= end; ++i ){
            tmp[i] = nums[i];
        }
        int i = start, j = mid+1;
        int k = start;
        for( ; i <= mid && j <= end; k++ ){
            if( tmp[i] <= tmp[j] )
                nums[k] = tmp[i++];
            else{   // tmp[i] > tmp[j], 此时逆序数为[i,mid]之间的元素, 个数mid-i+1
                nums[k] = tmp[j++];
                res += mid - i + 1;
            }

        }
        while(i <= mid){
            nums[k++] = tmp[i++];
        }
        while(j <= end){
            nums[k++] = tmp[j++];
        }
    }

    void mergeSort( vector<int>& nums, int start, int end )
    {
        if( start >= end ) return;
        int mid = start + ((end - start) >> 1);
        mergeSort( nums, start, mid );
        mergeSort( nums, mid+1, end );
        merge(nums, start, mid, end );
    }

};
```

52、两个链表的第一个公共结点

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        ListNode *A = headA, *B = headB;
        while(A!=B){
            A = A !=NULL ? A->next : headB;
            B = B !=NULL ? B->next : headA;
        }
        return A;
    }
};

class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        unordered_map<ListNode*,int> m;
        while(headA){
            m[headA]++;
            headA = headA->next;
        }
        while(headB){
            if(m[headB])
                return headB;
            headB = headB->next;
        }
        return NULL;
    }
};
```

53、在排序数组中查找数字

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0, right = nums.size();
        int count = 0;
        while(left < right){///[ )
           int mid = (left + right)/2;
           if(nums[mid]==target) right = mid;
           else if(nums[mid] > target) right = mid;
           else if(nums[mid] < target) left = mid + 1;// (2+3)/2 = 2
        }//找左边界
        while(left < nums.size() && nums[left]==target){
            count++;
            left++;
        }
        return count;
    }
//     int getright(vector<int>& nums, int target) {
//         int left = 0, right = num.size();
//         while(left < right){
//         int mid = left + (right - left)/2;
//         if(nums[mid] == target) left = mid + 1;
```

```
//         else if(nums[mid] > target) right = mid;
//         else if(nums[mid] < target) left = mid + 1;
//     }
//     return left - 1;
// }
    unordered_map<int,int> map;
        for(int num : nums){
            map[num]++;
        }
        return map[target];///solution2
};
```

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。

```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int left = 0, right = nums.size();//左闭右开right不减1//左闭右闭的话加一些判断
        while(left < right){
            int mid = (right-left)/2 + left;
            if(nums[mid] != mid) right = mid;
            if(nums[mid] == mid) left = mid + 1;
        }
        // if(left == nums.size()-1 && nums[left]==left){
        //    left++;
    //}
        return left;
    }
};
```

54、二叉搜索树的第k大节点

```
/**
 * 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:
    int kthLargest(TreeNode* root, int k) {
        vector<int> res;
        inorder(root,res);
        return res[res.size()-k];
    }
    void inorder(TreeNode* root, vector<int>& res){
        if(root == NULL) {return;}
        inorder(root->left,res);
        res.push_back(root->val);
        inorder(root->right,res);
    }
```

```
    int kthLargest(TreeNode* root, int k) {
        vector<int> res;
        if(root == NULL) {return 0;}
        stack<TreeNode*> stk;
        while(!stk.empty() || root!=NULL){
            while(root!=NULL){
                stk.emplace(root);
                root = root->left;
            }
            root = stk.top();
            stk.pop();
            res.emplace_back(root->val);
            root = root->right;
        }
        return res[res.size()-k];
    }///递推
};
```

55、二叉树最大深度

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(root == NULL) return 0;
        int left = maxDepth(root->left);
        int right = maxDepth(root->right);
        return (right > left) ? right+1 : left+1;
    }
};

class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(root == NULL)    return 0;
        queue<TreeNode* >q;
        q.push(root);
        int depth = 0;
        while(!q.empty()){
            ++depth;
            int count = q.size();   //保存每一层元素个数
            while(count--){
                TreeNode* temp = q.front();
                q.pop();

                if(temp->left)   q.push(temp->left);
                if(temp->right)  q.push(temp->right);
            }
        }
        return depth;
    }
};
```

55-II 平衡二叉树

```cpp
class Solution {
public:
    bool isBalanced(TreeNode* root) {
        bool res = true;
        getdepth(root, 0, res);
        return res;
    }
    int getdepth(TreeNode* root, int depth, bool& res){
        if(root==NULL) return depth;
        depth++;
        int tmp1 = getdepth(root->left,depth, res);
        int tmp2 = getdepth(root->right,depth,res);
        if(abs(tmp1-tmp2)> 1){
            res = false;
        }
        return max(tmp1,tmp2);//每层保留的是从根开始沿当前分支的最大深度
    }
};

///别人的解法，思路是一样的
class Solution {
public:
    bool ret = true;
    int get_height(TreeNode* root){
        if(!root || !ret) return 0;
        int l = get_height(root->left) + 1,
            r = get_height(root->right) + 1;
        if(abs(l - r) > 1) ret = false;
        return max(l, r);
    }

    bool isBalanced(TreeNode* root) {
        if(!root) return true;
        return abs(get_height(root->left) - get_height(root->right)) <= 1 &&
ret;
    }
};
```

56、数组中数字出现的次数- I

```cpp
///只出现一次,其他出现两次
///异或(任何一个数字异或自己等于0) 4^2^4=2  2^4^3^4 = 2^3
class Solution {
public:
    vector<int> singleNumbers(vector<int>& nums) {
        int tmp = 0;
        for(int n:nums){
            tmp = tmp ^ n;
        }
        int div = 1;
        while((tmp & div) == 0){//==优先级更高，要加括号
            div =  div << 1;
        }
        int a =0, b=0;
        for(int n:nums){
            if(n & div) { //判断某一位是否为1
```

```
                a ^= n;
            }
            else{
                b ^= n;
            }
        }
        return vector<int>{a,b};
    }
};

///用Hashmap做
class Solution {
public:
    vector<int> singleNumber(vector<int>& nums) {
        map<int, int> count;
        for (int n : nums) count[n] ++;
        vector<int> res;
        for (auto p : count)
            if (p.second == 1)
                res.push_back(p.first);
        return res;
    }
};
```

56-II 其他数出现3次

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        vector<int> bits(32,0);
        for(int num : nums){
            for(int i=0;i<32;i++){
                bits[i] += num & 1;
                num >>= 1;
            }
        }
        int ans=0;
        for(int i=31;i>=0;i--){
            ans <<= 1;
            ans += bits[i] % 3;
        }
        return ans;
    }
};///位运算

class Solution {
public:
    int singleNumber(vector<int>& nums) {
        unordered_map<int, int> store;
        for (int i : nums) {
            store[i]++;
        }
        //unordered_map<int, int>::const_iterator itr;
        for (int i : nums) {
            auto itr = store.find(i);
            if (itr -> second == 1) {
```

```
                return itr -> first;
            }
        }
        return -1;
    }
};
```

57、两数之和为S

输入一个递增排序的数组和一个数字S，在数组中查找两个数，使得他们的和正好是S，如果有多对数字的和等于S，输出两个数的乘积最小的

```
//思路:用两个指针，头尾指针向中间靠拢，
//当两数之和大于S时，说明最大值太大了（数组是递增的），所以右指针向左移动
//两数之和小于S时，说明太小了，左指针向右移动
//两数之和等于S就是所要求的
//乘积最小使迷惑的，当第一个符合条件就是最小的
class Solution {
public:
    vector<int> FindNumbersWithSum(vector<int> array,int sum) {
        vector<int> result;

        int left = 0, right = array.size()-1;  // 双指针
        while(left < right)
        {
            if (array[left]+array[right] == sum)
            {
                result.push_back(array[left]);
                result.push_back(array[right]);
                break;
            }
            else if(array[left]+array[right] > sum) // 说明右边的太大，
                right--;
            else
                left++;
        }
        return result;
    }
};

/// Hashmap
class Solution {
public:
    unordered_set<int> ss;
    vector<int> twoSum(vector<int>& nums, int target) {
        for (int i = 0; i < nums.size(); i++) {
            int find = target - nums[i];
            if (find < 0) return vector<int>();
            if (ss.count(find) != 0) return vector<int>{nums[i], find};
            ss.insert(nums[i]);
        }

        return vector<int>();
    }
};
```

57-II 打印出所有和为s的连续正数序列

```
class Solution {
public:
    vector<vector<int>> findContinuousSequence(int target) {
        vector<vector<int>> ans;
        if(target<3) return ans;
        int left = 1, right = 2;
        int limit = (target-1)/2;
        int cursum = left + right;
        while(left <= limit){
            if(cursum == target){
                vector<int> tmp;
                for(int i=left;i<=right;i++){
                    tmp.push_back(i);
                }
                ans.push_back(tmp);
            }
            while(cursum > target && left <=limit){
                cursum -= left;
                left++;
                if(cursum == target){
                    vector<int> tmp;
                    for(int i=left;i<=right;i++){
                    tmp.push_back(i);
                    }
                    ans.push_back(tmp);
                }
            }
            right++;
            cursum += right;
        }
        return ans;
    }
};
```

58、翻转字符串

```
class Solution {
public:
    string reverseWords(string s) {
        istringstream iss(s);
        string str;
        string ans="";
        while(iss >> str){///getline(iss,str,' ')//要是有多个空格会有问题
            if(ans ==""){
                ans = str;
            }
            else{
                ans = str + " " + ans ;
            }
        }
        return ans;
    }
};
```

59、队列的最大值

```
class MaxQueue {
    vector<int> q;
    int left = 0 , right = 0;
public:
    MaxQueue() {

    }

    int max_value() {
        int ans = -1;
        for(int i= left; i !=right ; ++i){
            ans = max(ans, q[i]);
        }
        return ans;
    }

    void push_back(int value) {
        q.push_back(value);
        right++;
    }

    int pop_front() {
        if(left == right){//不能用q.size()判别
            return -1;
        }
        return q[left++];
    }
};

class MaxQueue {
    queue<int> q;
    deque<int> d;
public:
    MaxQueue() {
    }

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

    void push_back(int value) {
        while (!d.empty() && d.back() < value) {
            d.pop_back();
        }
        d.push_back(value);
        q.push(value);
    }

    int pop_front() {
        if (q.empty())
            return -1;
```

```
        int ans = q.front();
        if (ans == d.front()) {
            d.pop_front();
        }
        q.pop();
        return ans;
    }
};
```

60、 n个骰子的点数

```
class Solution {
public:
    vector<double> dicesProbability(int n) {
        vector<double> dp(6, 1.0/6.0);///1.0/6.0 和 1/6是有区别的
        for(int i=2; i<=n; i++){
            vector<double> tmp(5*i+1,0);
            for(int j=0;j<dp.size();j++){
                  for(int k=0; k<6; k++){
                    tmp[j+k] += dp[j]/6.0;
                }
            }
            dp = tmp;
        }
        return dp;
    }
};
```

61、扑克牌中的顺子

```
class Solution {
public:
    bool isStraight(vector<int>& nums) {
        sort(nums.begin(),nums.end());
        int index = 0;
        for(int num:nums){
            if(num==0){
                index++;
            }
        }
        int guard = 0;
        for(int j =index; j<nums.size()-1;j++){
            if(nums[j]==nums[j+1]){
                return false;
            }else{
                guard += (nums[j+1]-nums[j]-1);
            }
        }
        if(index >= guard){
            return true;
        }else return false;
    }
};
```

62、圆圈中最后剩下的数字

```
class Solution {
public:
    int lastRemaining(int n, int m) {
        int f = 0;
        for (int i = 2; i != n + 1; ++i) {
            f = (m + f) % i;
        }
        return f;
    }
};
```

63、股票利润

Ⅰ、只能选择 某一天 买入这只股票，并选择在未来的某一个不同的日子卖出该股票

给定一个数组 prices ，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。你只能选择 **某一天** 买入这只股票，并选择在未来的**某一个不同的日子**卖出该股票。设计一个算法来计算你所能获取的最大利润。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if(prices.size()<=1) return 0;
        int minpro = prices[0], maxpro=0;
        for (int i = 0; i < prices.size(); ++i){
            maxpro = max(maxpro, prices[i]-minpro);
            minpro = min(minpro, prices[i]);
        }
        return maxpro;
    }
}; ///一次遍历 //暴力解法O(N^2) 时间过长
```

Ⅱ、你可以尽可能地完成更多的交易（多次买卖一支股票）。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
    int ans = 0;
    for(int i=1;i < prices.size();i++){
        if(prices[i]>prices[i-1]){
            ans += (prices[i]-prices[i-1]);
        }
        //ans += max(0, prices[i] - prices[i - 1]);
    }
    return ans;
    }
}; //贪心算法 //时间复杂度O(n),空间复杂度O(1)
```

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        int dp[n][2]; //定义状态dp[i][0]. dp[i][0] 表示第i天交易完后手里没有股票的最大利润，dp[i][1] 表示第i天交易完后手里持有一支股票的最大利润
        dp[0][0] = 0, dp[0][1] = -prices[0];
        for (int i = 1; i < n; ++i) { //状态转移方程
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
        }
        return dp[n - 1][0];

//         int dp0 = 0, dp1 = -prices[0];
//         for (int i = 1; i < n; ++i) {
//             int newDp0 = max(dp0, dp1 + prices[i]);
//             int newDp1 = max(dp1, dp0 - prices[i]);
//             dp0 = newDp0;
//             dp1 = newDp1;
//         }
//         return dp0;

    }
}; //时间复杂度：O(n)，其中n为数组的长度。一共有2n个状态，每次状态转移的时间复杂度为O(1)，因此时间复杂度为 O(2n)=O(n)； 空间复杂度：O(n)。我们需要开辟O(n)空间存储动态规划中的所有状态。如果使用空间优化(只要存我要的数)，空间复杂度可以优化至 O(1)。
```

Ⅲ、你最多可以完成 两笔 交易。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
    vector<vector<int>> dp(prices.size(),vector<int>(5,0));
    dp[0][1] = -prices[0]; dp[0][3] = -prices[0];
    for(int i=1;i < prices.size();i++){ //从1开始
        dp[i][0] = dp[i-1][0];
        dp[i][1] = max(dp[i-1][0]-prices[i],dp[i-1][1]);
        dp[i][2] = max(dp[i-1][1]+prices[i],dp[i-1][2]);
        dp[i][3] = max(dp[i-1][2]-prices[i],dp[i-1][3]);
        dp[i][4] = max(dp[i-1][3]+prices[i],dp[i-1][4]);
    }
    return dp[prices.size()-1][4];
    }
};
//节省空间写法
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.size() == 0) return 0;
        vector<int> dp(5, 0);
        dp[1] = -prices[0];
        dp[3] = -prices[0];
        for (int i = 1; i < prices.size(); i++) {
            dp[1] = max(dp[1], dp[0] - prices[i]);//dp[1]取dp[1]，即保持买入股票的状态
            dp[2] = max(dp[2], dp[1] + prices[i]);
```

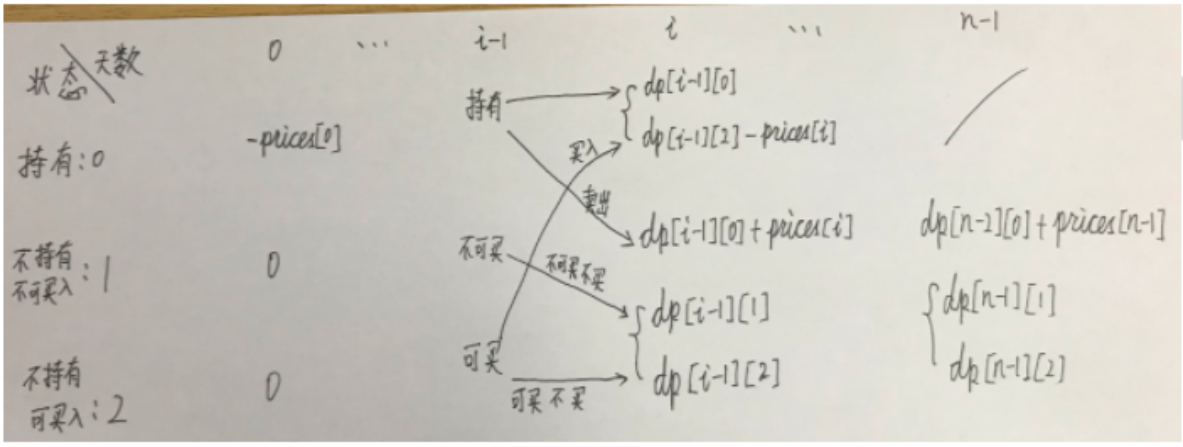
```
            dp[3] = max(dp[3], dp[2] - prices[i]);
            dp[4] = max(dp[4], dp[3] + prices[i]);
        }
        return dp[4];
    }
};
```

IV、你最多可以完成 k 笔交易。

```
class Solution {
public:
    int maxProfit(int k, vector<int>& prices) {

    // int n = prices.size();
    // k = min(k, n / 2); ///一次or两次

    if(prices.size()==0) return 0;
    vector<vector<int>> dp(prices.size(),vector<int>(2*k+1,0));
    for(int j=1; j<2*k;j+=2){ //从1开始 //初始值
        dp[0][j] = -prices[0];
    }

    for(int i=1;i<prices.size();i++){
        for(int j=0;j<2*k-1;j+=2){ ///注意这是2k-1 (2k+1)-2
            dp[i][j+1] = max(dp[i-1][j]-prices[i],dp[i-1][j+1]);
            dp[i][j+2] = max(dp[i-1][j+1]+prices[i],dp[i-1][j+2]);
        }
    }
    return dp[prices.size()-1][2*k];
    }
};
```

V、尽可能地完成更多的交易（多次买卖一支股票）：卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        if (n == 0) return 0;
        vector<vector<int>> dp(n, vector<int>(3, 0));
        dp[0][0] -= prices[0]; // 持股票
        for (int i = 1; i < n; i++) {
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] - prices[i]);//持有股票后的最多现金
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][2]);//不持有股票（能购买）的最多现金
            dp[i][2] = dp[i - 1][0] + prices[i];//不持有股票（冷冻期）的最多现金
        }
        return max(dp[n - 1][1], dp[n - 1][2]);
    }
};
```




```
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length < 2) return 0;
        int[][] dp = new int[prices.length][3];
        dp[0][0] = -prices[0];
        for (int i = 1; i < prices.length; i++) {
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][2] - prices[i]);
            dp[i][1] = dp[i - 1][0] + prices[i];
            dp[i][2] = Math.max(dp[i - 1][1], dp[i - 1][2]);
        }
        int maxProfit = Math.max(dp[prices.length - 1][1], dp[prices.length - 1]
[2]);
        return maxProfit;
    }
}
//优化
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length < 2) return 0;
        int dp0 = -prices[0], dp1 = 0, dp2 = 0;
        for (int i = 1; i < prices.length; i++) {
            int temp0 = Math.max(dp0, dp2 - prices[i]);
            int tmep1 = dp0 + prices[i];
            int temp2 = Math.max(dp1, dp2);
            dp0 = temp0;
            dp1 = tmep1;
            dp2 = temp2;
        }
        int maxProfit = Math.max(dp1, dp2);
        return maxProfit;
    }
}
```

VI、需要手续费

```
class Solution {
public:
    int maxProfit(vector<int>& prices, int fee) {
        // dp[i][1]第i天持有的最多现金
        // dp[i][0]第i天持有股票所剩的最多现金
        int n = prices.size();
        vector<vector<int>> dp(n, vector<int>(2, 0));
        dp[0][0] -= prices[0]; // 持股票
```

```
        for (int i = 1; i < n; i++) {
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] - prices[i]);
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] + prices[i] - fee);
        }
        return max(dp[n - 1][0], dp[n - 1][1]);
    }
};
```

64、求1+2+3+...+n

```
class Solution {
public:
    int sumNums(int n) {
        int ans = n;
        ans && (ans+= sumNums(ans-1));
        return ans;
    }
};

// class test{
// public:
//     test(){n++;sum+=n;}
//     void reset(){n=0;sum=0;}
//     int getnum(){return sum;}
// private:
//     int n;
//     int sum;
// }
// int sum_solution(int n){
//     test::reset;
//     test *a = test[n];
//     delete []a;
//     a=null;
//     return test::getnum();
// } //构造函数

// typedef int (*fun)(int);
// int (*solution_end)(int n){
//    return 0;
// }
// int (*sum_solution)(int n){
//    fun f[2]={solution_end,sum_solution};
//    return n + f[!!n](n-1);
// }///函数指针

// template<int n>struct sum_solution{
//     enum value{ N = sum_solution<n-1>::N+n};
// };
// template<>struct sum_solution<1>{
//     enum value{ N=1 };
// };
```

65、不用加减乘除做加法

////不用加法的加法运算 = 不进位加法 + 只进位加法 = 异或运算 + 与运算后向左进一位。
///又为了消除上式中的加号，需要用一个while循环来判断，当不再进位时跳出循环。

```
class Solution {
public:
    int add(int a, int b) {
        while(b != 0){
            int not_carry = a ^ b;
            int carry = ((unsigned int)(a & b) << 1);
            a = not_carry;
            b = carry;
        }
        return a;
    }
};
```

66、构建乘积数组

```
class Solution {
public:
    vector<int> constructArr(vector<int>& a) {
        if (!a.size()){
            return a;
        }
        //vector<int> ans(a.size(),0);
        int n = a.size();
        vector<int> c(n,0);
        //vector<int> d(a.size(),0);
        c[0] = 1; //这样赋值是不对的//万一a是空的，赋值就越界了//所以需要预先判断
        for(int i=1; i<n; ++i){
            c[i] = c[i-1]*a[i-1];
        }
        // d[a.size()-1] = 1;
        // ans[a.size()-1] = c[a.size()-1];//优化内存
        int tmp = 1;
        for(int i=n-2; i>=0; --i){
            tmp *= a[i+1];
            //ans[i]= c[i]*d[i];//直接把c作为ans减少不必要的内存消耗
            c[i] *= tmp;
        }
        return c;
    }
};
```

67、把字符串转换为整数

```
class Solution {
public:
    int strToInt(string str) {
        if(str.empty()) return 0;
        int trimSpace = 0;
        while (str[trimSpace] == ' ') {
            trimSpace++;// 将前面的空格都去掉
            if (trimSpace >= str.length()) {
```

```
                return 0;
            }
        }
        str = str.substr(trimSpace); // str 从第一个不是空格的字符开始

        int flag = 1;//判断正负号
        int i = 0;
        long long res = 0;
        if(str[i]=='-'){
            flag = -1; i++;
        }else if(str[i]=='+'){i++;}

        while(i<str.size() && isdigit(str[i])){
            int digit = str[i] - '0';
            res = res*10 + digit;
            if( (flag==1 && res > INT_MAX) || (flag==-1 && -res < INT_MIN)){
                return flag > 0 ? INT_MAX : INT_MIN;
            }
            //处理溢出
            i++;
        }
        // while(i<str.size() && isdigit(str[i])){ ///可以不使用long long
        //     int digit = str[i] - '0';
        //     //处理溢出
        //     if(res > INT_MAX/10 || (res==INT_MAX/10 && digit>7)){
        //         return flag > 0 ? INT_MAX : INT_MIN;
        //     }
        //     res = res*10 + digit;
        //     i++;
        // }
        return flag == 1 ? res : -res;
    }
};
```

68、二叉搜索数的最低公共祖先

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root == NULL) return NULL;
        if(root->val > p->val && root->val > q->val){
            return lowestCommonAncestor(root->left, p, q);
        }
        if(root->val < p->val && root->val < q->val){
            return lowestCommonAncestor(root->right, p, q);
        }else{
            return root;
        }
    }
};
```

68-II 二叉树的公共祖先

```
class Solution {
public:
```

```
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root || !p || !q || p == root || q==root) {return root;}

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

        if(left != NULL && right != NULL)
            return root;
        return left == NULL ? right : left;// 左边没找到右边找到了//左边找到了右边没找
到
    }
};///递归法

class Solution {
public:
    unordered_map<int, TreeNode*> nodes;
    unordered_map<int, bool> vis;
    void dfs(TreeNode* root){
        if (root->left != nullptr) {
            nodes[root->left->val] = root;
            dfs(root->left);
        }
        if (root->right != nullptr) {
            nodes[root->right->val] = root;
            dfs(root->right);
        }
    }
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        nodes[root->val] = nullptr;
        dfs(root);
        while (p != nullptr) {
            vis[p->val] = true;
            p = nodes[p->val];
        }
        while (q != nullptr) {
            if (vis[q->val]) return q;
            q = nodes[q->val];
        }
        return nullptr;
    }
};///我们可以用哈希表存储所有节点的父节点，然后我们就可以利用节点的父节点信息从 p 结点开始不断
往上跳，并记录已经访问过的节点，再从 q 节点开始不断往上跳，如果碰到已经访问过的节点，那么这个节
点就是我们要找的最近公共祖先。
```