offer++

1. 排序算法总结

1.1. 冒泡排序 O(n^2) 稳定

```
void bubbleSort(int a[], int n)
{
    for (int i = n - 1; i > 0; i--)
    {
        for (int j = 0; j < i; j++)
        {
            if (a[j] > a[j + 1]) swap(a[j], a[j + 1]);
        }
    }
}
```

1.2. 快速排序 O(nlogn)

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

    int i = l, j = r, tmp = a[l];

    while (i < j)
    {
        while (i < j && a[j] >= tmp) j--;
        if (i < j) a[i++] = a[j];
        while (i < j && a[i] <= tmp) i++;
        if (i < j) a[j--] = a[i];
    }
    a[i] = tmp;
    quickSort(a, l, i - 1);
    quickSort(a, i + 1, r);
}
```

1.3. 插入排序 O(n^2) 稳定

```
void insertSort(int a[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int tmp = a[i], j;
        for (j = i; j > 0 && tmp < a[j - 1]; j--)
            a[j] = a[j - 1];
        a[j] = tmp;
    }
}
```

1.4. 选择排序 O(n^2)

```
void selectSort(int a[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (a[i] > a[j]) swap(a[i], a[j]);
        }
    }
}
```

1.5. 归并排序 O(nlogn) 稳定

```
void mergeSort(int a[], int l, int r)
{
    if (l >= r) return;

    int tmp[r - l + 1];

    int mid = l + r >> 1;
    mergeSort(a, l, mid), mergeSort(a, mid + 1, r);

    int i = l, j = mid + 1, k = 0;

    while (i <= mid && j <= r)
        if (a[i] <= a[j]) tmp[k++] = a[i++];
        else tmp[k++] = a[j++];

    while (i <= mid) tmp[k++] = a[i++];
    while (j <= r) tmp[k++] = a[j++];

    for (int i = l, j = 0; i <= r; i++, j++) a[i] = tmp[j];
}
```

1.6. 堆排序 O(nlogn)

```
void adjust_heap(int a[], int x, int n)
{
    int l = x * 2 + 1;
    int r = x * 2 + 2;
    int max = x;

    if (l < n && a[l] > a[max]) max = l;
    if (r < n && a[r] > a[max]) max = r;

    if (max != x)
    {
        swap(a[x], a[max]);
        adjust_heap(a, max, n);
    }
}

void heapSort(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        adjust_heap(a, i, n);

    for (int i = n - 1; i > 0; i--)
    {
        swap(a[0], a[i]);
        adjust_heap(a, 0, i);
```

```
    }
}
```

2. 剑指 offer

2.1. 找出数组中重复数字

给定一个长度为 n 的整数数组 nums，数组中所有的数字都在 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。 <>

```
// 给定 nums = [2, 3, 5, 4, 3, 2, 6, 7]。
// 返回 2 或 3。
class Solution {
public:
    int duplicateInArray(vector<int>& nums) {
        int n = nums.size();
        for(int i = 0; i < n; i++){
            if(nums[i] < 0 || nums[i] >n-1)
            return -1;
        }
        for(int i = 0; i < n; i++){
            // 原地交换
            while(i != nums[i]){
                //  把nums[i]换到正确的位置
                if(nums[nums[i]] == nums[i]) return nums[i];
                swap(nums[i], nums[nums[i]]);
            }
        }
        return -1;
    }
};
```

2.2. 不修改数组找出重复的数字

给定一个长度为 n+1 的数组nums，数组中所有的数均在 1~n 的范围内，其中 n≥1。请找出数组中任意一个重复的数，但不能修改输入的数组。https://www.acwing.com/problem/content/description/15/

```
class Solution {
public:
    int duplicateInArray(vector<int>& nums) {
        int l = 1, r = nums.size() - 1;
        while(l < r){
            int mid = r + l >> 1;
            int s = 0;
            for(auto x: nums) if(x >= l && x <= mid) s++;
            if(s > mid - l + 1) r = mid;
            else l = mid +1;
         }
         return r;
```

```
    }
};
```

2.3. 二维数组查找

在一个二维数组中，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。

```
class Solution {
public:
    bool findNumberIn2DArray(vector<vector<int>>& matrix, int target) {
        // 从右上角开始遍历
        if(matrix.size()==0) return false;
        int i = 0, j = matrix[0].size()-1;
        while(i<matrix.size() && j>=0){
            if(matrix[i][j] == target) return true;
            else if (matrix[i][j] < target) i++;
            else j--;
        }
        return false;

    }
};
```

2.4. 替换空格为%20

请实现一个函数，把字符串中的每个空格替换成"%20"。

```
class Solution {
public:
    string replaceSpaces(string &str) {
        int l = str.size()-1;
        // 不开新的数组
        for(auto c: str){
            if(c == ' '){
                str += "00";
            }
        }
        int l2 = str.size() - 1;
        for(int i = l; i >= 0; i--){
            if(str[i] == ' '){
                str[l2--] = '0';
                str[l2--] = '2';
                str[l2--] = '%';
            }
            else{
                str[l2--] = str[i];
            }
        }
```

```
        return str;
    }
};
```

2.5. 从尾到头打印链表

输入一个链表的头结点，按照 从尾到头 的顺序返回节点的值。返回的结果用数组存储。

```
class Solution {
public:
    vector<int> printListReversingly(ListNode* head) {
        vector<int> ans;
        while(head){
            ans.push_back(head->val);
            head = head->next;
        }
        return vector<int>(ans.rbegin(), ans.rend());
    }
};
```

2.6. 前序和中序遍历重建二叉树

输入一棵二叉树前序遍历和中序遍历的结果，请重建该二叉树。

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        return dfs(preorder, inorder, 0, preorder.size()-1, 0, inorder.size()-1);
    }

    TreeNode* dfs(vector<int>& preorder, vector<int>& inorder, int ps, int pend,
int is, int iend){
        if(ps > pend) return NULL;
        TreeNode* root = new TreeNode(preorder[ps]);
        int l = is;
        while(inorder[l] != preorder[ps]) l++;
        int left=l-is;   //左子树的长度
        int right=iend-l;  //右子树长度
        root->left = dfs(preorder, inorder, ps + 1, ps + left, is, l - 1);
        root->right = dfs(preorder, inorder, ps + 1 + left, pend, l+1, iend);
        return root;
    }
};
```

2.7. 二叉树的下一个结点（给定father结点）

给定一棵二叉树的其中一个节点，请找出中序遍历序列的下一个节点。（给定father结点）

```
class Solution {
public:
    TreeNode* inorderSuccessor(TreeNode* p) {
        // 有无右子树讨论
        if (p->right) {
            p = p->right;
            while (p->left) p = p->left;
            return p;
        }
        // 如果p是father的右儿子，继续往上找
        while (p->father && p == p->father->right) p = p->father;
        return p->father;
    }
};
```

2.8. 两个栈实现一个队列

```
class CQueue {
public:
    stack<int> s1, s2;
    CQueue() {
    }

    void appendTail(int value) {
        s1.push(value);
    }

    int deleteHead() {
        if(s2.empty()){
            while(!s1.empty()){
            int temp = s1.top();
            s2.push(temp);
            s1.pop();
            }
        }
        if(s2.empty()) return -1;
        int temp = s2.top();
        s2.pop();
        return temp;
    }
};
```

2.9. 斐波那契数列

假定从0开始，第0项为0。(n<=39)

```
class Solution {
public:
```

```
    int Fibonacci(int n) {
        int a = 0, b = 1;
        if(n == 0) return 0;
        while(--n){
            int c = a + b;
            a = b;
            b = c;
        }
        return b;
    }
};
```

2.10. 旋转数组的最小数字（二分查找）

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个升序（非降序）的数组的一个旋转，输出旋转数组的最小元素。 https://www.acwing.com/solution/content/727/

```
class Solution {
public:
    int minArray(vector<int>& nums) {
        int n = nums.size() - 1;
        if (n < 0) return -1;
        while (n > 0 && nums[n] == nums[0]) n -- ;
        if (nums[n] >= nums[0]) return nums[0];
        int l = 0, r = n;
        while (l < r) {
            int mid = l + r >> 1;       // [l, mid], [mid + 1, r]
            if (nums[mid] < nums[0]) r = mid;
            else l = mid + 1;
        }
        return nums[r];
    }
};
```

2.11. 矩阵中的路径（DFS路径）

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一格开始，每一步可以在矩阵中向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。例如，在下面的3×4的矩阵中包含一条字符串“bfce”的路径（路径中的字母用加粗标出）。 [["a","b","c","e"], ["s","f","c","s"], ["a","d","e","e"]]

```
class Solution {
public:
    bool exist(vector<vector<char>>& matrix, string w) {
        int n = matrix.size(), m = matrix[0].size();
        for(int i = 0; i < n; i++){
            for(int j =0; j < m; j++){
                if(dfs(matrix, w, 0, i, j)){
```

```
                    return true;
                }
            }
        }
        return false;
    }

    bool dfs(vector<vector<char>>& matrix, string& w, int u, int i, int j){
        if (i < 0 || i >= matrix.size() || j < 0 || j >= matrix[0].size() ||
matrix[i][j] != w[u]){
            return false;
        }
        if(u == w.size()-1) return true;
        char t = matrix[i][j];
        // 回溯
        matrix[i][j] = '*';
        bool ans = dfs(matrix, w, u+1, i-1, j)||
                dfs(matrix, w, u+1, i+1, j)||
                dfs(matrix, w, u+1, i, j-1)||
                dfs(matrix, w, u+1, i, j+1);
        matrix[i][j] = t;
        return ans;
    }
};
```

2.12. 机器人的运动范围（bfs搜索）

地上有一个m行n列的方格，从坐标 [0,0] 到坐标 [m-1,n-1] 。一个机器人从坐标 [0, 0] 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于k的格子。例如，当k为18时，机器人能够进入方格 [35, 37] ，因为3+5+3+7=18。但它不能进入方格 [35, 38]，因为3+5+3+8=19。请问该机器人能够到达多少个格子？

```
class Solution {
public:
    int get_num(int x, int y){
        int ans = 0;
        while(x){
            ans += x % 10;
            x /= 10;
        }
        while(y){
            ans += y % 10;
            y /= 10;
        }
        return ans;
    }

    int movingCount(int n, int m, int k) {
        int ans = 0;
        // 标记数组
        vector<vector<bool>> st(n, vector<bool>(m));
```

```
        queue<pair<int, int>> q;
        q.push({0,0});
        int dx[4] = {0, 1, 0, -1}, dy[4] = {-1, 0, 1, 0};
        // BFS
        while(!q.empty()){
            auto x = q.front();
            q.pop();
            if(get_num(x.first, x.second) <= k  && st[x.first][x.second] == false)
{
                ans ++;
                st[x.first][x.second] = true;
                for(int i =0; i < 4; i++){
                    if(x.first+dx[i] >= 0 && x.first+dx[i] < n &&  x.second+ dy[i]
>=0 && x.second+ dy[i] < m){
                        q.push({x.first+dx[i], x.second+ dy[i]});
                    }
                }
            }
        }
        return ans;
    }
};
```

2.13. 剪绳子（分段最大乘积）

给你一根长度为 n 绳子，请把绳子剪成 m 段（m、n 都是整数，2≤n≤58 并且 m≥2）。每段的绳子的长度记为 k[0]、k[1]、……、k[m]。k[0]k[1] ... k[m] 可能的最大乘积是多少？例如当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到最大的乘积18。

```
class Solution {
public:
    int maxProductAfterCutting(int length) {
        if (length <= 3) return 1 * (length-1);
        if(length % 3 == 0) return pow(3, length / 3);
        if(length % 3 == 1) return pow(3, length / 3 - 1) * 4;
        if(length % 3 == 2) return pow(3, length / 3) * 2;
    }
};
```

2.14. 二进制中1的个数（unsigned int n = _n;）

输入一个32位整数，输出该数二进制表示中1的个数。 注意：负数在计算机中用其绝对值的补码来表示。 补码：如果我们指定了这个数据是unsigned类型的，意思就是说不将这个数据以补码的形式来读取。而是以纯二进制来读取。

```
class Solution {
public:
    int NumberOf1(int _n) {
```

```
        int ans = 0;
        // 如果是负数，右移高位补１，则死循环，而无符号整数在高位补０。
        unsigned int n = _n;
        while(n){
            ans += (n & 1) != 0;
            n >>= 1;
        }
        return ans;
    }
};
```

2.15. 实现数值的整数次方，即pow()

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。

```
class Solution {
public:
    double Power(double base, int exponent) {
        double ans = 1.0;
        int n = abs(exponent);
        while(n){
            if(n & 1) ans *= base;
            base *= base;
            n >>= 1;
        }
        if(exponent < 0) return 1 / ans;
        return ans;
    }
};
```

2.16. 在O(1)时间删除链表结点

给定单向链表的一个节点指针，定义一个函数在O(1)时间删除该结点。假设链表一定存在，并且该节点一定不是尾节点。

```
class Solution {
public:
    void deleteNode(ListNode* node) {
        node->val = node->next->val;
        ListNode *t = node->next;
        node->next = node->next->next;
        delete t;
    }
};
```

2.17. 删除链表中重复的节点

在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留。(一个都不留) 输入：1->2->3->3->4->4->5 输出：1->2->5

```
class Solution {
public:
    ListNode* deleteDuplication(ListNode* head) {
        auto dummy = new ListNode(-1);
        dummy->next = head;

        auto p = dummy;
        while (p->next) {
            auto q = p->next;
            while (q && p->next->val == q->val) q = q->next;

            if (p->next->next == q) p = p->next;
            else p->next = q;
        }

        return dummy->next;
    }
};
```

2.18. 正则表达式匹配

请实现一个函数用来匹配包括'.'和'*'*的正则表达式。模式中的字符*'.'*表示任意一个字符，而*'*表示它前面的字符可以出现任意次（含0次）。在本题中，匹配是指字符串的所有字符匹配整个模式。 例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，但是与"aa.a"和"ab*a"均不匹配。

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int n = s.size(), m = p.size();
        s = ' ' +s; p = ' ' + p;
        vector<vector<bool>> dp(n+1, vector<bool>(m+1));
        dp[0][0] = true;
        for(int i = 0; i <= n; i++){
            for(int j =1; j <= m; j++){
                if(j + 1 <= m && p[j+1] == '*') continue;
                if(i && p[j] != '*'){
                    dp[i][j] = dp[i-1][j-1] && (s[i] == p[j] || p[j] =='.');
                }
                else if (p[j] == '*'){
                    dp[i][j] = dp[i][j-2] || i && dp[i-1][j] && (s[i] == p[j-1] ||
p[j-1] == '.');
                }
            }
        }
        return dp[n][m];
    }
};
```

2.19. 表示数值的字符串

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。 例如，字符串"+100","5e2","-123","3.1416"和"-1E-16"都表示数值。但是"12e","1a3.14","1.2.3","+-5"和"12e+4.3"都不是。

```
class Solution {
public:
    bool isNumber(string s) {
        int i = 0;
        while (i < s.size() && s[i] == ' ') i ++ ;
        int j = s.size() - 1;
        while (j >= 0 && s[j] == ' ') j -- ;
        if (i > j) return false;
        s = s.substr(i, j - i + 1);

        if (s[0] == '-' || s[0] == '+') s = s.substr(1);
        if (s.empty() || s[0] == '.' && s.size() == 1) return false;

        int dot = 0, e = 0;
        for (int i = 0; i < s.size(); i ++ )
        {
            if (s[i] >= '0' && s[i] <= '9');
            else if (s[i] == '.')
            {
                dot ++ ;
                if (e || dot > 1) return false;
            }
            else if (s[i] == 'e' || s[i] == 'E')
            {
                e ++ ;
                if (i + 1 == s.size() || !i || e > 1 || i == 1 && s[0] == '.')
return false;
                if (s[i + 1] == '+' || s[i + 1] == '-')
                {
                    if (i + 2 == s.size()) return false;
                    i ++ ;
                }
            }
            else return false;
        }
        return true;
    }
};
```

```
# python代码
class Solution(object):
    def isNumber(self, s):
```

```
        """
        :type s: str
        :rtype: bool
        """
        try:
            float(s)
            return True
        except:
            return False
```

2.20. 调整数组顺序使奇数位于偶数前面

输入一个整数数组，实现一个函数来调整该数组中数字的顺序。使得所有的奇数位于数组的前半部分，所有的偶数位于数组的后半部分。 样例 输入：[1,2,3,4,5] 输出: [1,3,5,2,4]

```
class Solution {
public:
    void reOrderArray(vector<int> &array) {
         int left = 0, right = array.size() - 1;
         while(left < right){
            while(left<right && array[left] % 2 == 1) left++;
            while(left<right && array[right] % 2 == 0) right--;
            if(left < right) swap(array[left], array[right]);
            left++;
            right--;
        }
    }
};
```

2.21. 链表中倒数第k个节点

输入一个链表，输出该链表中倒数第k个结点。

注意： k >= 0; 如果k大于链表长度，则返回 NULL;

```
class Solution {
public:
    ListNode* findKthToTail(ListNode* pListHead, int k) {
        ListNode *p = pListHead;
        int llen = 0;
        while(p){
            llen++;
            p = p->next;
        }
        if(k>llen) return NULL;
        p = pListHead;
        int t = llen -k;
        while(t--){
            p = p->next;
```

```
        }
        return p;
    }
};
```

2.22. 寻找环形链表入口

```
/*
用两个指针 first,second 分别从起点开始走，first 每次走一步，second 每次走两步。如果过程中 second 走到null，则说明不存在环。否则当 first 和 second 相遇后，让 first 返回起点，second 待在原地不动，然后两个指针每次分别走一步，当相遇时，相遇点就是环的入口。
*/
class Solution {
public:
    ListNode *entryNodeOfLoop(ListNode *head) {
        ListNode *first = head, *second = head;
        while(first && second){
            first = first->next;
            if(second->next->next) second = second->next->next;
            else return NULL;
            if(first == second) break;
        }
        first = head;
        while(first != second){
            first = first->next;
            second = second->next;
        }
        return first;
    }
};
```

2.23. 翻转链表

2.23.1. (1)迭代

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *pre = NULL, *p = head;
        while(p){
            ListNode *t = NULL;
            if(p->next) t = p->next;
            p->next = pre;
            pre = p;
            p = t;
        }
        return pre;
    }
};
```

2.23.2. (2) 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;
    }
};
```

2.24. 合并两个排序的链表

输入两个递增排序的链表，合并这两个链表并使新链表中的结点仍然是按照递增排序的。

```
class Solution {
public:
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode(-1);
        auto p = dummy;
        while(l1 && l2){
            if(l1 ->val < l2->val){
                p->next = l1;
                l1 = l1->next;
            }
            else{
                p->next = l2;
                l2 = l2->next;
            }
            p = p->next;
        }
        if(l1) p->next = l1; else p->next = l2;
        return dummy->next;
    }
};
```

2.25. 树的子结构(判断B是不是A的子结构)

输入两棵二叉树A，B，判断B是不是A的子结构。我们规定空树不是任何树的子结构。

```
class Solution {
public:
    bool hasSubtree(TreeNode* pRoot1, TreeNode* pRoot2) {
        if (!pRoot1 || !pRoot2) return false;
        if (isSame(pRoot1, pRoot2)) return true;
        return hasSubtree(pRoot1->left, pRoot2) || hasSubtree(pRoot1->right,
pRoot2);
    }

    bool isSame(TreeNode* pRoot1, TreeNode* pRoot2) {
        if (!pRoot2) return true;
        if (!pRoot1 || pRoot1->val != pRoot2->val) return false;
        return isSame(pRoot1->left, pRoot2->left) && isSame(pRoot1->right, pRoot2-
>right);
    }
};
```

2.26. 二叉树的镜像

```
class Solution {
public:
    void mirror(TreeNode* root) {
        if(!root) return;
        TreeNode *t = root->left;
        root->left = root->right;
        root->right = t;
        mirror(root->left);
        mirror(root->right);
    }
};
```

2.27. 判断对称（镜像）的二叉树

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。

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        return dfs(root->left, root->right);
    }
    bool dfs(TreeNode *q, TreeNode *p){
        // 搜索到没有最底部
        if(!q && !p) return true;
        if(!q || !p) return false;
        if(q->val != p->val) return false;
        return dfs(q->left, p->right) && dfs(q->right, p->left);
```

```
    }
};
```

2.28. 顺时针打印矩阵

```
/*
输入:
[
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9,10,11,12]
]
输出: [1,2,3,4,8,12,11,10,9,5,6,7]
*/

class Solution {
public:
    vector<int> printMatrix(vector<vector<int>>& matrix) {
        vector<int> res;
        if (matrix.empty()) return res;
        int n = matrix.size(), m = matrix[0].size();
        vector<vector<bool>> st(n, vector<bool>(m, false));
        int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
        int x = 0, y = 0, d = 1;
        for (int k = 0; k < n * m; k ++ )
        {
            res.push_back(matrix[x][y]);
            st[x][y] = true;

            int a = x + dx[d], b = y + dy[d];
            // 碰壁就改变方向;
            if (x + dx[d] < 0 || x + dx[d] >= n || y + dy[d] < 0 || y + dy[d] >= m
|| st[a][b]) d = (d + 1) % 4;
            x = x + dx[d], y = y + dy[d];
        }
        return res;
    }
};
```

2.29. 包含min函数的栈

设计一个支持push，pop，top等操作并且可以在O(1)时间内检索出最小元素的堆栈。
push(x)–将元素x插入栈中
pop()–移除栈顶元素
top()–得到栈顶元素
getMin()–得到栈中最小元素

```

```

```
class MinStack {
public:
    /** initialize your data structure here. */
    // 维护一个单调栈s2;
    stack<int> s1, s2;
    MinStack() {
    }

    void push(int x) {
        if(s2.empty() || x<=s2.top()) s2.push(x);
        s1.push(x);
    }

    void pop() {
        if(s1.top() == s2.top()) s2.pop();
        s1.pop();
    }

    int top() {
        return s1.top();
    }

    int getMin() {
        return s2.top();
    }
};
```

2.30. 栈的压入、弹出序列

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否可能为该栈的弹出顺序。假设压入栈的所有数字均不相等。 例如序列1,2,3,4,5是某栈的压入顺序，序列4,5,3,2,1是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。
注意：若两个序列长度不等则视为并不是一个栈的压入、弹出序列。若两个序列都为空，则视为是一个栈的压入、弹出序列。

```
class Solution {
public:
    bool isPopOrder(vector<int> pushV,vector<int> popV) {
        stack<int> s;
        if(pushV.size() != popV.size()) return false;
        int j = 0;
        for(int i : pushV){
            s.push(i);
            while(!s.empty() && s.top() == popV[j]){
                s.pop();
                j++;
            }

        }
        return s.empty();
```

```
    }
};
```

2.31. 不分行从上往下打印二叉树(层次遍历)

从上往下打印出二叉树的每个结点，同一层的结点按照从左到右的顺序打印。

```
class Solution {
public:
    vector<int> printFromTopToBottom(TreeNode* root) {
        vector<int> ans;
        queue<TreeNode*> q;
        if(!root) return ans;
        q.push(root);
        while(!q.empty()){
            if(q.front()->left) q.push(q.front()->left);
            if(q.front()->right) q.push(q.front()->right);
            ans.push_back(q.front()->val);
            q.pop();
        }
        return ans;
    }
};
```

2.32. 分行从上往下打印二叉树

从上到下按层打印二叉树，同一层的结点按从左到右的顺序打印，每一层打印到一行。

```
class Solution {
public:
    vector<vector<int>> printFromTopToBottom(TreeNode* root) {
        vector<vector<int>> ans;
        if(!root) return ans;
        queue <TreeNode*> q;
        q.push(root);
        while(!q.empty()){
            int q_long = q.size();
            vector<int> temp;
            while(q_long--){
                if(q.front()->left) q.push(q.front()->left);
                if(q.front()->right) q.push(q.front()->right);
                temp.push_back(q.front()->val);
                q.pop();
            }
            ans.push_back(temp);
        }
        return ans;
    }
};
```

2.33. 之字形打印二叉树

请实现一个函数按照之字形顺序从上向下打印二叉树。
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。

```
class Solution {
public:
    vector<vector<int>> printFromTopToBottom(TreeNode* root) {
        vector<vector<int>> ans;
        if(!root) return ans;
        // 主要认识双端队列
        // 单端: queue  双端: deque
        deque <TreeNode*> q;
        q.push_back(root);
        int lr = -1;
        while(!q.empty()){
            int q_long = q.size();
            vector<int> temp;
            lr *= -1;
            while(q_long--){
                if(lr == 1){
                    if(q.front()->left) q.push_back(q.front()->left);
                    if(q.front()->right) q.push_back(q.front()->right);
                    temp.push_back(q.front()->val);
                    q.pop_front();
                }
                else{
                    if(q.back()->right) q.push_front(q.back()->right);
                    if(q.back()->left) q.push_front(q.back()->left);
                    temp.push_back(q.back()->val);
                    q.pop_back();
                }
            }
            ans.push_back(temp);
        }
        return ans;
    }
};
```

2.34. 二叉树中和为某一值的路径(回溯)

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。

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

```
    vector<int> tem;
    int cur_sum = 0;
    void dfs(TreeNode* root, int sum){
        if(!root) return;
        tem.push_back(root->val);
        if(root->val == sum && !root->left && !root->right) {ans.push_back(tem);}
        dfs(root->left, sum-root->val);
        dfs(root->right, sum-root->val);

        tem.pop_back();
    }
public:
    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        dfs(root, sum);
        return ans;
    }
};
```

2.35. 二叉搜索树的后序遍历序列

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。 如果是则返回true，否则返回false。假设输入的数组的任意两个数字都互不相同。

```
class Solution {
public:
    bool verifySequenceOfBST(vector<int> sequence) {
        return verify(sequence, 0, sequence.size()-1);
    }
    bool verify(vector<int>& sequence, int s, int e){
        if(s >= e) return true;
        // e是根节点，判断根节点把数组分成左右两部分.
        int t = sequence[e];
        int i = s;
        while(sequence[i] < t) i++;
        int j = i;
        while(sequence[i] > t) i++;
        if(i!= e) return false;
        return verify(sequence, s, j-1) && verify(sequence, j, e-1);
    }
};
```

2.36. 二叉树中和为某一值的路径

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

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

```
    vector<vector<int>> findPath(TreeNode* root, int sum) {
        vector<int> temp;
        if(!root) return ans;
        findone(root, sum, temp);
        // 若不要求从根节点开始找，就加上这两行;
        // findPath(root->left, sum);
        // findPath(root->left, sum);
        return ans;
    }
    void findone(TreeNode* root, int sum, vector<int>&temp){
        if(!root) return;
        if(sum-root->val == 0 && !root->left && !root->right) {
            temp.push_back(root->val);
            ans.push_back(temp);
        }
        else{
            sum -= root->val;
            temp.push_back(root->val);
            findone(root->left, sum, temp);
            findone(root->right, sum, temp);
        }
        temp.pop_back();
    }
};
```

2.37. 复杂链表的复刻

请实现一个函数可以复制一个复杂链表。 在复杂链表中，每个结点除了有一个指针指向下一个结点外，还有一个额外的指针指向链表中的任意结点或者null。

```
/**
 * Definition for singly-linked list with a random pointer.
 * struct ListNode {
 *     int val;
 *     ListNode *next, *random;
 *     ListNode(int x) : val(x), next(NULL), random(NULL) {}
 * };
 */
```

2.38. 字符串转数字

忽略所有行首空格，找到第一个非空格字符，可以是‘+/−’表示是正数或者负数，紧随其后找到最长的一串连续数字，将其解析成一个整数； 整数后可能有任意非数字字符，请将其忽略； 如果整数长度为0，则返回0；如果整数大于INT_MAX(2^31 − 1)，请返回INT_MAX；如果整数小于INT_MIN(−2^31)，请返回INT_MIN；

```
class Solution {
public:
```

```
    int strToInt(string str) {
        int k = 0;
        //去空格
        while (k < str.size() && str[k] == ' ') k++;
        bool is_minus = false;
        long long num = 0;
        //判正负
        if (str[k] == '+') k++;
        else if (str[k] == '-') k++, is_minus = true;
        //字符变数字
        while (k < str.size() && str[k] >= '0' && str[k] <= '9') {
            num = num * 10 + str[k] - '0';
            k++;
        }
        //处理特例
        if (is_minus) num *= -1;
        if (num > INT_MAX) num = INT_MAX;
        if (num < INT_MIN) num = INT_MIN;
        return (int)num;
    }
};
```

2.39. 约瑟夫坏（圆圈中最后剩下的）

2.39.1. 暴力模拟

```
class Solution {
public:
    int lastRemaining(int n, int m) {
        vector<int> ve;
        for (int i = 0; i < n; i++) ve.push_back(i);
        int t = 0;
        if (ve.size() < 1) return 0;
        while (ve.size() != 1) {
            for (int i = 0; i < m - 1; i++) {
                t++;
                if (t == ve.size()) t = 0;
            }
            ve.erase(ve.begin() + t);
            if (t == ve.size()) t = 0;
        }
        return ve[0];

    }
};
```

2.39.2. 递推

```
class Solution {
public:
    int lastRemaining(int n, int m) {
        if (n == 1)
            return 0;
        else
            return (lastRemaining(n - 1, m) + m) % n;
    }
};
```

2.40. 扑克牌顺子

```
class Solution {
public:
    bool isContinuous(vector<int> nums) {
        unordered_set<int> se;
        if (nums.size() < 5) return false;
        int mint = INT_MAX, maxt = INT_MIN;
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] == 0) continue;
            mint = min(mint, nums[i]);
            maxt = max(maxt, nums[i]);

            if (se.count(nums[i])) return false;
            else se.insert(nums[i]);
        }
        return maxt - mint <= 4;
    }
};
```

2.41. 一排路由器可以覆盖的信号

一条直线上等距离放置了n台路由器。路由器自左向右从1到n编号。第i台路由器到第j台路由器的距离为| i-j |。每台路由器都有自己的信号强度，第i台路由器的信号强度为ai。所有与第i台路由器距离不超过ai的路由器可以收到第i台路由器的信号 （注意，每台路由器都能收到自己的信号）。问一共有多少台路由器可以收到至少k台不同路由器的信号。https://www.nowcoder.com/profile/1334434/codeBookDetail?submissionId=86144859

```
#include<iostream>
#include<vector>
using namespace std;
int main() {
    int n, k;
    cin >> n >> k;
    vector<int> ve(n);
    for (int i = 0; i < n; i++) {
        cin >> ve[i];
    }
    vector<int> anst(n, 0);
```

```
    for (int i = 0; i < n; i++) {
        int l = i - ve[i], r = i + ve[i];
        if (l >= 0) anst[l]++; else anst[0]++;
        if (r >= n - 1)continue; else anst[r + 1]--;
    }
    int temp = 0, ans = 0;
    for (auto i : anst) {
        temp += i;
        if (temp >= k) ans++;
    }
    cout << ans;
    return 0;
}
```

2.42. 滑动窗口最大值

给定一个数组和滑动窗口的大小，请找出所有滑动窗口里的最大值。 例如，如果输入数组[2, 3, 4, 2, 6, 2, 5, 1]及滑动窗口的大小3, 那么一共存在6个滑动窗口，它们的最大值分别为[4, 4, 6, 6, 6, 5]。

```
#include <iostream>
#include <vector>
#include <stack>
#include <deque>
using namespace std;
int main()
{
    vector<int> nums = { 2, 3, 4, 2, 6, 2, 5, 1 };
    int k = 3;
    deque<int> q;
    vector<int> ans;
    //记录下标
    for (int i = 0; i < nums.size(); i++) {
        //当前最大值坐标不在范围里，移除
        if (q.size() && q.front() < i - k + 1) q.pop_front();
        // 单调队列
        while (q.size() && nums[i] > nums[q.back()]) q.pop_back();
        q.push_back(i);
        if (i >= k-1) ans.push_back(nums[q.front()]);
    }

    for (int i = 0; i < ans.size(); i++)
        cout << ans[i] << ',';
    return 0;
}
```

股票最大利润 假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票一次可能获得的最大利润是多少?

```
class Solution {
public:
    int maxDiff(vector<int>& nums) {
        if (nums.size() < 2) return 0;
        int mint = INT_MAX;
        int ans = INT_MIN;
        for (int i = 0; i < nums.size(); i++) {
            mint = min(mint, nums[i]);
            ans = max(ans, nums[i] - mint);
        }
        return ans;
    }
};
```

2.43. 乘积数组 B[i]=A[0]×A[1]...×A[n-1]

```
class Solution {
public:
    vector<int> multiply(const vector<int>& A) {
        vector<int> ans;
        int t = 1;
        if (A.empty()) return ans;
        for (int i = 0; i < A.size(); i++) {
            t *= A[i];
            ans.push_back(t);

        }
        t = 1;
        for (int i = ans.size() - 1; i > 0; i--) {
            ans[i] = t * ans[i - 1];
            t *= A[i];
        }
        ans[0] = t;
        return ans;
    }
};
```

2.44. 分裂二叉树最大乘积

给你一棵二叉树，它的根为 root 。请你删除 1 条边，使二叉树分裂成两棵子树，且它们子树和的乘积尽可能大。
由于答案可能会很大，请你将结果对 10 ^ 9 + 7 取模后再返回。

```
class Solution {
public:
    const int mod = 1e9 + 7;
    vector<int> temp;
    long long dfs(TreeNode* root) {
```

```
        if (!root) return 0;
        long long res = root->val + dfs(root->left) + dfs(root->right);
        temp.push_back(res);
        return res;
    }
    int maxProduct(TreeNode* root) {
        long long ans = 0;
        int v = dfs(root);
        for (long long t : temp) {
            // cout<<t<<' ';
            ans = max(ans, t * (v - t));
        }
        return (long long)ans % (int)(1e9 + 7);
    }
};
```

2.45. 二叉树最低公共祖先

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

2.46. 大数相乘

```
string BigMutiple(string num1, string num2) {
    string res = "";
    //两个数的位数
    int m = num1.size(), n = num2.size();
    //一个i位数乘以一个j位数，结果至少是i+j-1位数
    vector<long long> tmp(m + n - 1);
    //每一位进行笛卡尔乘法
    for (int i = 0; i < m; i++) {
        int a = num1[i] - '0';
        for (int j = 0; j < n; j++) {
            int b = num2[j] - '0';
            tmp[i + j] += a * b;
        }
    }
    //进行进位处理，注意左侧是大右侧是小
    int carry = 0;
```

```
    for (int i = tmp.size() - 1; i >= 0; i--) {
        int t = tmp[i] + carry;
        tmp[i] = t % 10;
        carry = t / 10;
    }
    //若遍历完仍然有进位
    while (carry != 0) {
        int t = carry % 10;
        carry /= 10;
        tmp.insert(tmp.begin(), t);
    }
    //将结果存入到返回值中
    for (auto a : tmp) {
        res = res + to_string(a);
    }
    if (res.size() > 0 && res[0] == '0')return "0";
    return res;
}

//测试函数
int main() {
    string num1, num2;
    while (cin >> num1 >> num2) {
        cout << BigMutiple(num1, num2) << endl;
    }
    return 0;
}
```

2.47. 大数相加

```
string add(const string& a, const string& b) {
    const int n = a.size(), m = b.size();
    if(n < m) return add(b, a);

    string c;
    vector<int> tem;
    // 数位和，两个加数对应的数位都加到 sum 上
    // 0 <= sum <= 19
    int sum = 0;
    for(int i = 0; i < n; i++) {
        sum += a[i] - '0';
        if(i < m) sum += b[i] - '0';
        tem.push_back(sum % 10); // 获取该数位的数字
        sum /= 10;              // 获取进位信息
    }
    if(sum) tem.push_back(sum);  // 最高位的进位处理
    for (auto a : tem) {
        c = c + to_string(a);
    }
    return c;
}
```

```
int main() {
    string num1, num2;

    while (cin >> num1 >> num2) {
        reverse(num1.begin(), num1.end());
        reverse(num2.begin(), num2.end());
        string anst = add(num1, num2);
        string ans = string(anst.rbegin(), anst.rend());
        cout << ans << endl;
    }
    return 0;
}
```

2.48. 不用加减乘除做加法

A + B 分为2个部分，A^B是不进位加法，(A&B) << 1是进位，二者相加就起到了相同的作用。 因为A + B = A^B + ((A&B) << 1)，所以说 还是会用到加号+，对此我们的解决方案是 使用一个while()循环， 不断迭代赋值，将 异或的结果和进位的结果分别变成a和b，因为b不断左移，所以总有一天会变成0，这时候while就跳出来。 答案一直存储在a里面，也就是异或(不进位加法)中，最后进位b=0，a没有必要进位了，答案就是最后的a。

```
class Solution {
public:
    int add(int a, int b) {
        while (b)
        {
            int sum = a ^ b;
            int carry = (a & b) << 1;
            a = sum;
            b = carry;
        }

        return a;
    }
};
```

3. LeetCode

3.1. 1.两数之和

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。题目website

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> heap;
        for(int i = 0; i<nums.size(); i++){
```

```
            if(heap.count(target-nums[i])) return {i, heap[target-nums[i]]};
            heap[nums[i]] = i;
        }
        return {};
    }
};
```

3.2. 2. 两数相加

给出两个非空的链表用来表示两个非负的整数。其中，它们各自的位数是按照逆序的方式存储的，并且它们的每个节点只能存储一位数字。website

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        // 记录该位的和
        int sum = 0;
        ListNode* dummy = new ListNode(0);
        ListNode *cur = dummy;
        while(l1 || l2 || sum)
        {
            if(l1) {sum += l1->val; l1 = l1->next;}
            if(l2) {sum += l2->val; l2 = l2->next;}
            auto temp = new ListNode(sum % 10);
            sum /=10;
            cur = cur->next = temp;
        }
        return dummy->next;
    }
};
```

3.3. 3. 无重复字符的最长子串

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。website

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_set<char> heap;
        int right = 0, ans = 0;
        for(int i = 0; i<s.size(); i++){
            while(right < s.size() && !heap.count(s[right])){
                heap.insert(s[right]);
                ans = max(ans, right-i+1);
                right++;
            }
            heap.erase(s[i]);
        }
        return ans;
```

```
    }
};
```

3.4. 4. 寻找两个正序数组的中位数

给定两个大小为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。请你找出这两个正序数组的中位数，并且要求算法的时间复杂度为 O(log(m + n))。website

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int n1 = 0, n2 = 0;
        vector<int> heap;
        for(int i = 0; i < nums1.size()+nums2.size(); i++){
            if(n1>=nums1.size()){
                heap.push_back(nums2[n2]);
                n2++;
            }
            else if(n2>=nums2.size()){
                heap.push_back(nums1[n1]);
                n1++;
            }
            else if(nums1[n1] < nums2[n2]){
                heap.push_back(nums1[n1]);
                n1++;
            }
            else{
                heap.push_back(nums2[n2]);
                n2++;
            }
        }
        if((n1 + n2)%2 == 1) return heap[(n1+n2)/2];
        else return (heap[(n1+n2)/2-1] + heap[(n1+n2)/2])/2.0;
    }
};
```

3.5. 5. 最长回文子串

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。website

```
class Solution {

public:
    int ans[2] = {0};
    void help(string &s, int i, int j){
        // 如果合理，计算;
        while(i>=0 && i<s.size()&&j>=0&&j<s.size()&&s[i] == s[j]){
            if((j-i) > ans[1] -ans[0]){
                ans[0] = i;
```

```
                ans[1] = j;
            }
            i--;
            j++;
        }
        return;
    }
    string longestPalindrome(string s) {
        for(int i =0; i<s.size(); i++){
            help(s, i, i);
            help(s, i, i+1);
        }
        return s.substr(ans[0], ans[1]-ans[0]+1);
    }
};
```

3.6. 6. Z 字形变换

将一个给定字符串根据给定的行数，以从上往下、从左到右进行 Z 字形排列。website L C I R E T O E S I I G E D H N

```
\\ 找规律
class Solution {
public:
    string convert(string s, int numRows) {
        string ans;
        if(numRows == 1) return s;
        for(int i = 0; i<numRows; i++){
            if(i==0 || i== numRows-1){
                for(int j = i; j<s.size(); j += 2*numRows-2){
                    ans+=s[j];
                }
            }
            else{
                for(int j = i, z = 2*numRows-2-i; j<s.size()||z<s.size();
j+=2*numRows-2, z+= numRows*2-2){
                    if(j<s.size())ans+=s[j];
                    if(z<s.size())ans+=s[z];
                }
            }
        }
        return ans;
    }
};
```

3.7. 7. 整数反转

给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。其数值范围为 [−231, 231 − 1]。请根据这个假设，如果反转后整数溢出那么就返回 0。website

```
\\转换为字符串（to_string->atoi）或者：
class Solution {
public:
    int reverse(int x) {
        long long ans = 0;
        while(x){
            ans *= 10;
            ans += x%10;
            x /= 10;
        }
        if(ans<INT_MIN || ans >INT_MAX) return 0;
        return ans;
    }
};
```

3.8. 8. 字符串转换整数 (atoi)

请你来实现一个 atoi 函数，使其能将字符串转换成整数。该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止.website

```
class Solution {
public:
    int myAtoi(string s) {
        int k =0;
        long long ans = 0;
        while(k<s.size() && s[k]==' ') k++;

        int flag = 1;
        if(s[k] == '-') flag = -1, k++;
        else if(s[k] == '+') k++;

        while(k<s.size()&&(s[k]>='0' && s[k] <= '9')){
            ans*=10;
            ans+=(s[k]-'0');
            k++;
            if(ans>INT_MAX && flag==1) return INT_MAX;
            if(ans>INT_MAX && flag==-1) return INT_MIN;
        }
        return flag * ans;
    }
};
```

3.9. 9. 回文数

判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。website

```
class Solution {
public:
```

```
    bool isPalindrome(int x) {
        string s1 = to_string(x);
        // string s2 = to_string(x);
        // reverse(s1.begin(), s1.end());
        string s2 = string(s1.rbegin(), s1.rend());
        return s1 == s2;
    }
};
```

3.10. 10. 正则表达式匹配

给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int n = s.size(), m = p.size();
        s = ' ' +s; p = ' ' + p;
        vector<vector<bool>> dp(n+1, vector<bool>(m+1));
        dp[0][0] = true;
        for(int i = 0; i <= n; i++){
            for(int j =1; j <= m; j++){
                if(j + 1 <= m && p[j+1] == '*') continue;
                if(i && p[j] != '*'){
                    dp[i][j] = dp[i-1][j-1] && (s[i] == p[j] || p[j] =='.');
                }
                else if (p[j] == '*'){
                    dp[i][j] = dp[i][j-2] || i && dp[i-1][j] && (s[i] == p[j-1] ||
p[j-1] == '.');
                }
            }
        }
        return dp[n][m];
    }
};
```

4. 岛屿问题

4.1. 岛屿数量

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。 岛屿总是被水包围，并且每座岛屿只能由水平方向或竖直方向上相邻的陆地连接形成。 此外，你可以假设该网格的四条边均被水包围。

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int n = grid.size();
        int ans = 0;
```

```
        if(n==0) return ans;
        int m = grid[0].size();
        for(int i = 0; i<n; i++){
            for(int j =0; j<m; j++){
                if(grid[i][j] == '1'){
                    dfs(grid, n, m, i ,j);
                    ans++;
                }
            }
        }
        return ans;
    }

    void dfs(vector<vector<char>>& grid, int n, int m, int r, int c){
        if(r<0||r>=n||c<0||c>=m||grid[r][c]=='0') return;
        grid[r][c] = '0';
        dfs(grid, n, m, r-1, c);
        dfs(grid, n, m, r+1, c);
        dfs(grid, n, m, r, c-1);
        dfs(grid, n, m, r, c+1);
    }
};
```

4.2. 岛屿的最大面积

给定一个包含了一些 0 和 1 的非空二维数组 grid 。一个 岛屿 是由一些相邻的 1 (代表土地) 构成的组合，这里的「相邻」要求两个 1 必须在水平或者竖直方向上相邻。你可以假设 grid 的四个边缘都被 0（代表水）包围着。 找到给定的二维数组中最大的岛屿面积。(如果没有岛屿，则返回面积为 0 。)

```
class Solution {
public:
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int ans = 0;
        for(int i =0; i<grid.size(); i++){
            for(int j=0; j<grid[0].size(); j++){
                int temp = 0;
                dfs(grid, i, j, temp);
                ans = max(ans, temp);
            }
        }
        return ans;
    }

    void dfs(vector<vector<int>>& grid, int i , int j, int &temp){
        if(i<0||i>=grid.size()||j<0||j>=grid[0].size()||grid[i][j]==0) return;
        else{grid[i][j]=0; temp++;}
        dfs(grid, i-1, j, temp);
        dfs(grid, i+1, j, temp);
        dfs(grid, i, j-1, temp);
        dfs(grid, i, j+1, temp);
```

```
    }
};
```

4.3. 岛屿的周长

给定一个包含 0 和 1 的二维网格地图，其中 1 表示陆地 0 表示水域。 网格中的格子水平和垂直方向相连（对角线方向不相连）。整个网格被水完全包围，但其中恰好有一个岛屿（或者说，一个或多个表示陆地的格子相连组成的岛屿）。 岛屿中没有“湖”（“湖” 指水域在岛屿内部且不和岛屿周围的水相连）。格子是边长为 1 的正方形。网格为长方形，且宽度和高度均不超过 100 。计算这个岛屿的周长。

```
class Solution {
public:
    int islandPerimeter(vector<vector<int>>& grid) {
        int n = grid.size();
        if(n == 0) return 0;
        int m = grid[0].size();
        int ans = 0;
        for(int i = 0; i<n; i++){
            for(int j =0; j<m; j++){
                if(grid[i][j] == 1) {
                    ans += 4;
                    if(i-1>=0 && grid[i-1][j] == 1) ans -= 2;
                    if(j-1>=0 && grid[i][j-1] == 1) ans -= 2;
                }
            }
        }s
        return ans;
    }
};
```