offer++

1. 排序算法总结

1.1. 冒泡排序 bubbleSort 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 quickSort 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 insertSort 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 selectSort 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 mergeSort 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 heapSort 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. 二分查找模板 bsTemplate

二分模板
1.循环必须是l < r
2.if判断条件看是不是不满足条件， 然后修改上下界
3.若if else后是r = mid - 1，则前面mid 语句要加1
4.出循环一定是l == r，所以l和r用哪个都可以

二分只有下面两种情况
1：找满足某个条件的第一个数
2：找满足某个条件的最后一个数

二分的流程：
1. 确定二分边界
2. 设计一个check（性质）
3. 判断一下区间如何更新
4. 如果更新方式是l=mid，r=mid-1 那么就在算mid是加 1。

```
// 判断条件很复杂时用check函数，否则if后直接写条件即可
bool check(int mid) {
    ...
    return ...;
}
// 能二分的题一定是满足某种性质，分成左右两部分
// if的判断条件是让mid落在满足你想要结果的区间内
// 找满足某个条件的第一个数  即右半段
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;
        else l = mid + 1;
    }
    return l;
}

// 找满足某个条件的最后一个数  即左半段
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

2.1. 在排序数组中查找元素的第一个和最后一个位置 searchRange

给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 O(log n) 级别。 如果数组中不存在目标值，返回 [-1, -1]。

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

    int rbs(vector<int>& nums, int target){
        int l = 0, r = nums.size() - 1;
        while(l < r){
            int mid = (l + r + 1) >> 1;
            if(nums[mid] <= target) l = mid;
            else r = mid - 1;
        }
        return r;
    }
    vector<int> searchRange(vector<int>& nums, int target) {
        if(nums.size() == 0) return {-1, -1};
        int left = lbs(nums, target);
        int right = rbs(nums, target);
        if(nums[left] != target) return {-1, -1};
        return {left, right};
    }
};
```

2.2. x的平方根 sqrtofx

快速求sqrt(x)

```
class Solution {
public:
    int mySqrt(int x) {
        int l = 0, r = x;
        while (l < r) {
            // 两个int相加减会溢出 中间加个长整型常量
            // 少用乘法，用除法可以防止溢出
            int mid = l + 1ll + r >> 1;
```

```
            if (mid  <= x / mid) l = mid;
            else r = mid - 1;
        }
        return l;
    }
};
```

3. LeetCode

3.1. 1.两数之和 twoSum

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 addTwoNumbers

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 lengthOfLongestSubstring

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 findMedianSortedArrays

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 longestPalindrome

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 zConvert

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 intReverse

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 isPalindrome1

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 isMatch q

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];
    }
};
```

3.11. 11. 成水最多的容器 maxWaterArea

给你 n 个非负整数 a1，a2，...，an，每个数代表坐标中的一个点 (i, ai) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, ai) 和 (i, 0)。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

```
class Solution {
public:
    int maxArea(vector<int>& height) {
       int ans = 0, l =0, r = height.size()-1;
       while(l<r){
           ans = max(ans, (r-l)*min(height[r], height[l]));
           if(height[l] < height[r]) l++;
           else r--;
       }
       return ans;
    }
};
```

3.12. 12. 整数转罗马数字 intToRoman

```
class Solution {
public:
    string intToRoman(int num) {
        int values[] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
        string reps[] = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX",
"V", "IV", "I"};
        string res;
        // 贪心
        for (int i = 0; i < 13; i ++ )
            while(num >= values[i])
            {
                num -= values[i];
                res += reps[i];
            }
        return res;
    }
};
```

3.13. 13. 罗马数字转整数 romanToInt

https://leetcode-cn.com/problems/roman-to-integer/

```
class Solution {
public:
    int romanToInt(string s) {
        int result=0;
        map<char,int> luomab={
            {'I',1},
            {'V',5},
            {'X',10},
            {'L',50},
            {'C',100},
            {'D', 500},
            {'M', 1000}
        };//初始化哈希表
        for(int i=0;i<s.length();i++)
        {
            if(luomab[s[i]] < luomab[s[i+1]])
                result -= luomab[s[i]];
            else
            {
                result += luomab[s[i]];
            }
        }
        return result;
    }
};
```

3.14. 14. 最长公共前缀 longestCommonPrefix

编写一个函数来查找字符串数组中的最长公共前缀。 如果不存在公共前缀，返回空字符串 ""。

```
class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        string ans;
        if(strs.size()==0) return ans;
        for(int j = 0; j<strs[0].size(); j++){
            char c = strs[0][j];
            bool flag = true;
            for(int i = 0; i < strs.size(); i++){
                if(strs[i][j] != c) {flag = false; return ans;}
            }
            if(flag) ans += c;
        }
        return ans;
    }
};
```

3.15. 15. 三数之和 等于0 threeSum

给你一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？请你找出所有满足条件且不重复的三元组。 注意：答案中不可以包含重复的三元组。

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> ans;
        if(nums.size()<3) return ans;
        sort(nums.begin(), nums.end());
        // for循环+双指针
        for(int i = 0; i<nums.size(); i++){
            // 与前一个数字相同的话，去重。
            if(i>0 && nums[i]==nums[i-1]) continue;
            int l = i+1, r = nums.size() - 1;
            while(l < r){
                // 当答案正确时去重，不能直接去重，如[0, 0, 0, 0];
                int sumt = nums[i] + nums[l] + nums[r];
                if(sumt == 0){
                    while(l<r && nums[l] == nums[l+1]) l++;
                    while(l<r && nums[r] == nums[r-1]) r--;
                    ans.push_back({nums[i], nums[l], nums[r]});
                    l++; r--;
                }
                else if(sumt < 0) l++;
                else r--;
            }
        }
        return ans;
    }
};
```

3.16. 16. 最接近的三数之和 threeSumClosest

给定一个包括 n 个整数的数组 nums 和 一个目标值 target。找出 nums 中的三个整数，使得它们的和与 target 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

```
class Solution {
public:
    int threeSumClosest(vector<int>& nums, int target) {
        sort(nums.begin(), nums.end());
        int maxt = INT_MAX;
        int ans, l, r, sumt;
        for(int i = 0; i < nums.size(); i++){
            if(i > 0 && nums[i] == nums[i-1]) continue;
            l = i + 1, r = nums.size() - 1;
            while(l < r){
                sumt = nums[i] + nums[l] + nums[r];
                if(abs(sumt - target) < maxt){
                    ans = sumt;
```

```
                    maxt = abs(sumt - target);
                }
                if(sumt == target) return sumt;
                else if(sumt > target) r--;
                else l++;
            }
        }
        return ans;
    }
};
```

3.17. 17. 电话号码的字母组合

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。 给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。https://leetcode-cn.com/problems/letter-combinations-of-a-phone-number/

```
class Solution {
public:
    vector<string> nums = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs",
"tuv", "wxyz"};
    vector<string> ans;
    string tem;
    void dfs(string &digits, int ind){
        if(ind == digits.size()){
            ans.push_back(tem);
            return;
        }
        for(int j = 0; j < nums[digits[ind] -'0'].size(); j++){
            tem += nums[digits[ind]-'0'][j];
            dfs(digits, ind + 1);
            tem.pop_back();
        }
    }
    vector<string> letterCombinations(string digits) {
        if(digits.size() == 0)
        return ans;
        dfs(digits, 0);
        return ans;
    }
};
```

3.18. 18. 四数之和

给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a，b，c 和 d ，使得 a + b + c + d 的值与 target 相等？找出所有满足条件且不重复的四元组。

注意：答案中不可以包含重复的四元组。

```
class Solution{
    public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        sort(nums.begin(),nums.end());
        vector<vector<int> > res;
        if(nums.size() < 4)
        return res;
        int a, b, c, d;
        for(a = 0;a < nums.size(); a++){
            if(a > 0 && nums[a] == nums[a-1]) continue;      //确保nums[a] 改变了
            for(b = a + 1; b < nums.size(); b++){
                if(b > a+1 && nums[b] == nums[b-1])continue;   //确保nums[b] 改变了
                c = b + 1, d = nums.size() - 1;
                while(c < d){
                    if(nums[a] + nums[b] + nums[c] + nums[d] < target)
                        c++;
                    else if(nums[a]+nums[b]+nums[c]+nums[d]>target)
                        d--;
                    else{
                        res.push_back({nums[a],nums[b],nums[c],nums[d]});
                        while(c < d && nums[c + 1] == nums[c]) c++;    //确保
nums[c] 改变了
                        while(c < d && nums[d - 1] == nums[d]) d--;      //确保
nums[d] 改变了
                        c++;
                        d--;
                    }
                }
            }
        }
        return res;
    }
};
```

3.19. 19. 删除链表倒数第n个结点

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode *dummy = new ListNode(0); dummy->next = head;
        ListNode *l = dummy, *r = dummy;
        while(n--) r = r->next;
        while(r->next){
            r = r -> next;
            l = l -> next;
        }
        l->next = l->next->next;
        return dummy->next;
    }
};
```

3.20. 20. 有效的括号 kuoisValid

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足： 左括号必须用相同类型的右括号闭合。 左括号必须以正确的顺序闭合。

```
class Solution {
public:
    bool isValid(string s) {
        stack<char> st;
        unordered_map <char, char> mp = {{'}','{'}, {')','('}, {']','['}};
        for(char c: s){
            if(mp.count(c) == 0) st.push(c);
            else{
                if(st.empty() || mp[c] != st.top()) return false;
                else st.pop();
                // bool flag = false;
                // if(c=='}' && st.top()=='{') {st.pop(); flag = true;}
                // if(c==')' && st.top()=='(') {st.pop(); flag = true;}
                // if(c==']' && st.top()=='[') {st.pop(); flag = true;}
                // if(!flag) return false;
            }
        }
        return st.empty();
    }
};
```

3.21. 21. 合并两个有序链表 mergeTwoLists

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode(0);
        ListNode *p = dummy;
        while(l1 && l2){
            if(l1->val < l2->val){
                p->next = l1;
                p = p->next;
                l1 = l1->next;
            }
            else{
                p->next = l2;
                p = p->next;
                l2 = l2->next;
            }
        }
        if(l1) p->next = l1;
        else p->next = l2;
        return dummy->next;
```

```
    }
};
```

3.22. 22. 括号生成 generateParenthesis

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

```
class Solution {
public:
    vector<string> ans;
    string tem;
    int left = 0, right = 0;
    void dfs(int left, int right, int n){
        if(tem.size() == 2*n) ans.push_back(tem);
        // 最多方 n 个左括号
        if (left < n) {
            tem.push_back('(');
            dfs(left + 1, right, n);
            tem.pop_back();
        }
        // 左括号不能比左括号多
        if (right < left) {
            tem.push_back(')');
            dfs(left, right + 1, n);
            tem.pop_back();
        }
    }
    vector<string> generateParenthesis(int n) {
        dfs(0, 0 ,n);
        return ans;
    }
};
```

3.23. 23. 合并k个升序链表 mergeklist

连接

3.24. 24. 两两交换链表中的节点 swapPairs

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例: 给定 1->2->3->4, 你应该返回 2->1->4->3.

```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        ListNode *dummy = new ListNode(0);
        dummy ->next = head;
        ListNode *p = dummy;
```

```
        while(p->next && p->next->next){
            ListNode *a = p->next, *b = a->next;
            a->next = b->next;
            b->next = a;
            p->next = b;
            p = a;
        }
        return dummy->next;
    }
};
```

3.25. 25. 每 K 个一组，翻转链表 reverseKGroup

给你链表的头节点 head ，每 k 个节点一组进行翻转，请你返回修改后的链表。

k 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

```
ListNode* reverseKGroup(ListNode* head, int k) {
    ListNode *dummy=new ListNode(0);
    dummy->next = head;
    ListNode *p = dummy;
    while(head){
        int i=0;
        ListNode *start = head, *end = head;
        while(i < k-1){
            if(end==nullptr) break;
            end = end->next;
            i++;
        }
        if(end==nullptr) break;
        ListNode * t1 = end->next;
        p->next = end;
        ListNode *t2 = t1, *t3;
        while(start != t1){
            t3 = start->next;
            start->next = t2;
            t2 = start;
            start = t3;
        }
        p=head;
        head=t1;
    }
    return dummy->next;
}
```

3.26. 26. 删除排序数组中的重复项 removeDuplicates (快慢指针)

给定一个排序数组，你需要在 原地 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在 原地 修改输入数组 并在使用 O(1) 额外空间的条件下完成。

```
int removeDuplicates(vector<int>& nums) {
    int left = 0;
    for(int right = 0; right < nums.size(); right++){
        if(nums[right] != nums[left]){
            left++;
            nums[left] = nums[right];
        }
    }
    return left+1;
}
```

3.27. 27. 移除数组指定值元素 removeElement

给你一个数组 nums 和一个值 val，你需要 原地 移除所有数值等于 val 的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并 原地 修改输入数组。 元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int left = 0;
        for(int right = 0; right<nums.size(); right++){
            if(nums[right] != val)
                nums[left++] = nums[right];
        }
        return left;
    }
};
```

3.28. 28. 找出字符串中第一个匹配项的下标 strStr

给你两个字符串 haystack 和 needle ，请你在 haystack 字符串中找出 needle 字符串的第一个匹配项的下标（下标从 0 开始）。如果 needle 不是 haystack 的一部分，则返回 -1 。

链接

```
int strStr(string haystack, string needle) {
    if(haystack.find(needle) != string::npos)
        return haystack.find(needle);
    else
        return -1;
}
```

3.29. 29. 两数相除，不用除法 divide-two-integers

给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用乘法、除法和 mod 运算符。

返回被除数 dividend 除以除数 divisor 得到的商。

链接

3.30. 30. 串联所有单词的子串 findallSubstring

给定一个字符串 s 和一个字符串数组 words。 words 中所有字符串 长度相同。s 中的 串联子串 是指一个包含 words 中所有字符串以任意顺序排列连接起来的子串。

例如，如果 words = ["ab","cd","ef"]， 那么 "abcdef"， "abefcd"，"cdabef"， "cdefab"，"efabcd"，和 "efcdab" 都是串联子串。 "acdbef" 不是串联子串，因为他不是任何 words 排列的连接。 返回所有串联字串在 s 中的开始索引。你可以以 任意顺序 返回答案。

(链接)[https://leetcode.cn/problems/substring-with-concatenation-of-all-words/]

3.31. 31. 下一个排列 nextPermutation

整数数组的一个 排列 就是将其所有成员以序列或线性顺序排列。

例如，arr = [1,2,3] ，以下这些都可以视作 arr 的排列：[1,2,3]、[1,3,2]、[3,1,2]、[2,3,1] 。 整数数组的 下一个排列 是指其整数的下一个字典序更大的排列。

```
void nextPermutation(vector<int>& nums) {
        int i = nums.size() - 2;
        while (i >= 0 && nums[i] >= nums[i + 1]) {
            i--;
        }
        if (i >= 0) {
            int j = nums.size() - 1;
            while (j >= 0 && nums[i] >= nums[j]) {
                j--;
            }
            swap(nums[i], nums[j]);
        }
        reverse(nums.begin() + i + 1, nums.end());
    }
```

3.32. 32. 最长有效括号长度 longestValidParentheses

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

输入：s = ")()())"
输出：4
解释：最长有效括号子串是 "()()"

```
int longestValidParentheses(string s) {
    stack<int> st;
    int ans = 0;
```

```
    // 假设-1位置为第一个未匹配位置
    st.push(-1);
    for(int i = 0; i<s.size(); i++){
        if(s[i]=='(') st.push(i);
        else{
            st.pop();
            // 保留最后一个未匹配的右括号索引
            // 因为左括号会被弹出去
            if(st.empty()) st.push(i);
            else {
                ans = max(ans, i-st.top());
            }
        }
    }
    return ans;
}
```

3.33. 42. 接雨水 trap

链接

```
int trap(vector<int>& height) {
    int res = 0;
    vector<int> max_left(height.size(), 0);
    vector<int> max_right(height.size(), 0);
    for(int i = 1; i<height.size()-1; i++)  max_left[i] = max(max_left[i-1],
height[i-1]);
    for(int i = height.size()-2; i>0; i--)  max_right[i] = max(max_right[i+1],
height[i+1]);
    //最两端的列不用考虑，因为一定不会有水
    for (int i = 1; i < height.size() - 1; i++) {
        //找出两端较小的
        int min_lr = min(max_left[i], max_right[i]);

        //只有较小的一段大于当前列的高度才会有水
        if (min_lr > height[i]) {
            res = res + (min_lr - height[i]);
        }
    }
    return res;
}
```

3.34. 86. 分隔链表

给定一个链表和一个特定值 x，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。 你应当保留两个分区中每个节点的初始相对位置。

```
class Solution {
public:
```

```
    ListNode* partition(ListNode* head, int x) {
        ListNode *dummy1 = new ListNode(0), *dummy2 = new ListNode(0);
        ListNode *p1 = dummy1, *p2 = dummy2;
        while(head){
            if(head->val < x){
                p1->next = head;
                p1 = p1->next;
            }
            else {
                p2->next = head;
                p2 = p2->next;
            }
            head = head->next;
        }
        p2->next = NULL;
        p1->next = dummy2->next;
        return dummy1->next;
    }
};
```

3.35. 79. 单词搜索 （二维dfs） existpath

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。 示例:

board =
[
['A','B','C','E'],
['S','F','C','S'],
['A','D','E','E']
]

给定 word = "ABCCED", 返回 true
给定 word = "SEE", 返回 true
给定 word = "ABCB", 返回 false

```
class Solution {
public:
    bool dfs(vector<vector<char>>& board, string &word, int ind, int i, int j){
        if(i<0 || i>=board.size() || j<0 || j>=board[0].size() || board[i][j] !=
word[ind]) return false;
        if(ind == word.size()-1) return true;
        char tem = board[i][j];
        board[i][j] = '*';
        if( dfs(board, word, ind + 1, i-1, j) ||
            dfs(board, word, ind + 1, i+1, j) ||
            dfs(board, word, ind + 1, i, j-1) ||
            dfs(board, word, ind + 1, i, j+1)
            )
            return true;
```

```
            board[i][j] = tem;;
            return false;
        }
        bool exist(vector<vector<char>>& board, string word) {
            if(board.size() == 0) return false;
            for(int i = 0; i<board.size(); i++){
                for(int j = 0; j<board[0].size(); j++){
                    if(dfs(board, word, 0, i, j))
                        return true;
                }
            }
            return false;
        }
    };
```

3.36. 迷路的机器人 pathWithObstacles

设想有个机器人坐在一个网格的左上角，网格 r 行 c 列。机器人只能向下或向右移动，但不能走到一些被禁止的网格（有障碍物）。设计一种算法，寻找机器人从左上角移动到右下角的路径。 https://leetcode-cn.com/problems/robot-in-a-grid-lcci/ 输入:
[
[0,0,0],
[0,1,0],
[0,0,0]
]
输出: [[0,0],[0,1],[0,2],[1,2],[2,2]]

```
class Solution {
public:
    vector<vector<int>> ans;
    bool dfs(vector<vector<int>>& a, int i, int j){
        if(i<0||i>=a.size()||j<0||j>=a[0].size()||a[i][j]==1) return false;
        ans.push_back({i, j});
        if(i == a.size()-1 && j == a[0].size()-1) return true;
        if(dfs(a, i+1, j) || dfs(a, i, j+1)) return true;
        // 这道题其实也是一道很典型的DFS题目。
        // 刚开始是用DFS加回溯来做的，然后超时了，但是后来发现这道题目完全没必要回溯，
        // 因为如果一个坐标返回false，那么就意味着这个坐标是无法到达终点的，那么直接去掉坐标就行了。
        a[i][j] = 1;
        ans.pop_back();
        return false;
    }
    vector<vector<int>> pathWithObstacles(vector<vector<int>>& obstacleGrid) {
        dfs(obstacleGrid, 0, 0);
        return ans;
    }
};
```

3.37. 91. 解码方法 1-26 to a-z

给定一个只包含数字的非空字符串，请计算解码方法的总数。 输入: "226"
输出: 3
解释: 它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6) 。

```
class Solution {
public:
    int numDecodings(string s) {
        int pre1 = 1, pre2 = 1;
        if(s[0] == '0') return 0;
        int p;
        for(int i = 0; i<s.size(); i++){
            p = 0;
            string tem;
            if(i>0){
                tem += s[i-1];
                tem += s[i];
            }
            if(tem>="10" && tem<="26")
                p += pre1;
            if(s[i] != '0')
                p += pre2;
            pre1 = pre2; pre2 = p;
        }
        return p;
    }
};
```

3.38. 反转链表 reverseList1

```
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };

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

3.39. 92. 反转链表 II 反转区间链表 reverseBetween

反转从位置 m 到 n 的链表。请使用一趟扫描完成反转。

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int m, int n) {
        if(n == 1 || !head) return head; // 特殊情况处理（当然不写也行，写是为了找找感觉）

        ListNode *prev = NULL, *curr = head; // 定义prev和curr指针，用于翻转链表
        while(m > 1)
        {
            prev = curr;
            curr = curr -> next; // 将prev与curr定位到需要翻转的初始位置
            m --;
            n --;
        }
        ListNode *before = prev, *after = curr; // 定义prev前面一位和curr后面一位的指针，方便翻转完链表后重新连起来
        while(n > 0)
        {
            // 翻转链表四步大法
            ListNode* nextptr = curr -> next;
            curr -> next = prev;
            prev = curr;
            curr = nextptr;
            n --;
        }
        if(before) before -> next = prev; // 将链表重新连起来
        else head = prev; // 将链表重新连起来
        after -> next = curr; // 将链表重新连起来
        return head;
    }
};
```

3.40. 215 topk

```
class Solution {
public:
    void quickSort(vector<int> &a, int l, int r, int k, int &ans){
        // 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;
        if(i==a.size()-k) {ans = a[i]; return;}

        if(i > a.size()-k)
            quickSort(a, l, i - 1, k, ans);
        else
            quickSort(a, i + 1, r, k, ans);
    }

    int findKthLargest(vector<int>& nums, int k) {
        int ans = 0;
        quickSort(nums, 0, nums.size()-1, k, ans);
        return ans;
    }
};
```

3.41. 221. 最大正方形 maximal-square

在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。https://leetcode-cn.com/problems/maximal-square/

```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        int n = matrix.size();
        if(n == 0) return 0;
        int m = matrix[0].size();
        vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
        int ans = 0;

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (matrix[i][j] == '0')
                    dp[i + 1][j + 1] = 0;
                else
                    dp[i + 1][j + 1] = min(min(dp[i][j + 1], dp[i + 1][j]), dp[i]
[j]) + 1;
                ans = max(ans, dp[i + 1][j + 1]);
            }
        }
        return ans * ans;
    }
};
```

3.42. 322. 零钱兑换

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

```
class Solution {
public:
    const int MAXN = 1e8;
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp(amount+1, MAXN);
        dp[0] = 0;
        for(int i = 0; i<=amount; i++){
            for(auto c : coins){
                if(i-c < 0) continue;
                dp[i] = min(dp[i], dp[i-c] + 1);
            }
        }
        return dp[amount] == MAXN ? -1 : dp[amount];
    }
};
```

4. 岛屿问题 land problem

4.1. 岛屿数量 numIslands

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 maxAreaOfIsland

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 islandPerimeter

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;
    }
};
```

5. 子集组合排列问题 sbuset permute prpblem

//递归思想：
//①画出递归树，找到状态变量(回溯函数的参数)，这一步非常重要※
//②根据题意，确立结束条件
//③找准选择列表(与函数参数相关)，与第一步紧密关联※
//④判断是否需要剪枝
//⑤作出选择，递归调用，进入下一层
//⑥撤销选择

5.1. 全排列 permute

给定一个 没有重复 数字的序列，返回其所有可能的全排列。

```
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;
public:
    void dfs(vector<int>& nums, vector<bool>& vis) {
        if (path.size() == nums.size()) {
            ans.push_back(path);
            return;
        }
        for (int i = 0; i < nums.size(); i++) {
            if (vis[i]) continue;
            vis[i] = true;
            path.push_back(nums[i]);
            dfs(nums, vis);
            vis[i] = false;
            path.pop_back();
        }
    }

    vector<vector<int>> permute(vector<int>& nums) {
        vector<bool> vis(nums.size(), false);
```

```
        dfs(nums, vis);
        return ans;
    }
};
```

5.2. 全排列 结果无重复 permuteUnique

给定一个可包含重复数字的序列，返回所有不重复的全排列。

```
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;
public:
    void dfs(vector<int>& nums, vector<bool>& vis) {
        if (path.size() == nums.size()) {
            ans.push_back(path);
            return;
        }
        for (int i = 0; i < nums.size(); i++) {
            if (vis[i]) continue;
            // 剪枝
            if (i > 0 && !vis[i - 1] && nums[i] == nums[i - 1]) continue;
            vis[i] = true;
            path.push_back(nums[i]);
            dfs(nums, vis);
            vis[i] = false;
            path.pop_back();
        }
    }

    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<bool> vis(nums.size(), false);
        // 先排序
        sort(nums.begin(), nums.end());
        dfs(nums, vis);
        return ans;
    }
};
```

5.3. 组合 combine77

给定两个整数 n 和 k，返回 1 ... n 中所有可能的 k 个数的组合。 输入: n = 4, k = 2 输出: [[2,4], [3,4], [2,3], [1,2], [1,3], [1,4],]

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

```
    void dfs(vector<int> &nums, int &k, int ind){
        if(tem.size() == k) {ans.push_back(tem); return;}
        for(int i = ind; i< nums.size(); i++){
            tem.push_back(nums[i]);
            dfs(nums, k, i+1);
            tem.pop_back();
        }
    }
    vector<vector<int>> combine(int n, int k) {
        // vector<bool> vis(n, false);
        if(n<k) return ans;
        vector<int> nums;
        for(int i = 0; i<n; i++) nums.push_back(i+1);
        dfs(nums, k, 0);
        return ans;
    }
};
```

5.4. 数组总和 combinationSum

给定一个无重复元素的数组 candidates 和一个目标数 target ， 找出 candidates 中所有可以使数字和为 target 的组合。candidates 中的数字可以无限制重复被选取。

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> tem;
    int sumt = 0;
    void dfs(int start, vector<int> &candidates, int t){
        if(sumt > t) return;
        if(t == sumt) {ans.push_back(tem);}
        for(int i = start; i<candidates.size(); i++){

            tem.push_back(candidates[i]);
            sumt += candidates[i];
            dfs(i, candidates, t);
            sumt -= candidates[i];
            tem.pop_back();
        }

    }
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        //sort(candidates.begin(), candidates.end());
        dfs(0, candidates, target);
        return ans;
    }
};
```

5.5. 数组总和 结果无重复 combinationSum2

给定一个数组 candidates 和一个目标数 target ， 找出 candidates 中所有可以使数字和为 target 的组合。candidates 中的每个数字在每个组合中只能使用一次。

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

    void dfs(int start, vector<int>& candidates, int t) {
        if (t < 0) return;
        if (t == 0) { ans.push_back(tem); }
        for (int i = start; i < candidates.size(); i++) {
            if (i > start && candidates[i] == candidates[i - 1]) continue;
            tem.push_back(candidates[i]);
            dfs(i + 1, candidates, t - candidates[i]);
            tem.pop_back();
        }

    }
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(), candidates.end());
        dfs(0, candidates, target);
        return ans;
    }
};
```

5.6. 216. 组合总和 III combinationSum3

找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> tem;
    int sumt = 0;

    void dfs(int n, int k, int ind){
        if(9 - ind + 1 < k - tem.size()) return;
        if(tem.size() == k && sumt == n) {ans.push_back(tem); return;}
        for(int i = ind; i<=9; i++){
            tem.push_back(i);
            sumt += i;
            dfs(n, k, i + 1);
            sumt -= i;
            tem.pop_back();
        }
    }
    vector<vector<int>> combinationSum3(int k, int n) {
        dfs(n, k, 1);
```

```
        return ans;
    }
};
```

5.7. 子集 结果无重复 subsetsWithDup

```
class Solution_subset2 {
public:
    vector<vector<int>> ans;
    vector<int> tem;
    void dfs(int start, vector<int>& nums) {
        ans.push_back(tem);
        for (int i = start; i < nums.size(); i++) {
            if (i > start && nums[i] == nums[i - 1]) continue;
            tem.push_back(nums[i]);
            dfs(i + 1, nums);
            tem.pop_back();
        }
    }
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        dfs(0, nums);
        return ans;
    }
};
```

5.8. 子集 subsets1

给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

```
void subsets(int start, vector<int> &input) {
    ans.push_back(tem);
    for (int i = start; i < input.size(); i++) {
        tem.push_back(input[i]);
        subsets(i + 1, input);
        tem.pop_back();
    }
}
int main() {
    vector<int> input = { 1,2,3 };
    subsets(0, input);
    for (int i = 0; i < ans.size(); i++) {
        for (int j = 0; j < ans[i].size(); j++) {
            cout << ans[i][j] << ',';
        }
        cout << endl;
    }
}
```

5.9. 字符串排列 结果无重复 stringpermutation

输入一个字符串，打印出该字符串中字符的所有排列。 你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。
输入：s = "abc" 输出：["abc","acb","bac","bca","cab","cba"]

```
class Solution {
public:
    vector<string> ans;
    string path;
    void dfs(vector<bool>& vis, string& s) {
        if (path.size() == s.size()) {
            ans.push_back(path);
            return;
        }
        for (int i = 0; i < s.size(); i++) {
            if (!vis[i]) {
                if (i > 0 && !vis[i - 1] && s[i] == s[i - 1]) continue;
                path.push_back(s[i]);
                vis[i] = true;
                dfs(vis, s);
                vis[i] = false;
                path.pop_back();
            }
        }
    }
    vector<string> permutation(string s) {
        vector<bool> vis(s.size());
        sort(s.begin(), s.end());
        dfs(vis, s);
        return ans;
    }
};
```

6. 二叉树的题　all_bt

6.1. 二叉树前中后遍历（非递归实现） prein

```
// 前序
vector<int> preorderTraversal(TreeNode* root) {
    stack<TreeNode*> st;
    vector<int> v;
    while (root || st.size()) {
        while (root) {
            st.push(root->right);
            v.push_back(root->val);
            root = root->left;
        }
        root = st.top(); st.pop();
    }
```

```
    return v;
}
```

```
// leetcode 94 中序
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        stack<TreeNode*> st;
        vector<int> v;
        while(root || st.size()){
            while(root){
                st.push(root);
                root = root->left;
            }
            root = st.top(); st.pop();
            v.push_back(root->val);
            root = root->right;
        }
        return v;
    }
};
```

```
// 后序
vector<int> postorderTraversal(TreeNode* root) {
    stack<TreeNode*> st;
    vector<int> v;
    while (root || st.size()) {
        while (root) {
            st.push(root->left);
            v.push_back(root->val);
            root = root->right;
        }
        root = st.top(); st.pop();
    }
    reverse(v.begin(), v.end());
    return v;
}
```

6.2. 二叉树的直径 diameterOfBinaryTree

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

```
class Solution {
    int ans;
    // 最长的直径肯定是以某个结点为根节点的子树的左右子树高度之和。只需要深搜遍历即可。
    int depth(TreeNode* root){
```

```
        if (root == NULL) {
            return 0;
        }
        int L = depth(root->left);
        int R = depth(root->right);
        ans = max(ans, L + R + 1); // 计算d_node即L+R+1 并更新ans
        return max(L, R) + 1; // 返回该节点为根的子树的深度
    }
public:
    int diameterOfBinaryTree(TreeNode* root) {
        ans = 1;
        depth(root);
        return ans - 1;
    }
};
```

6.3. 验证平衡二叉树 isBalancedtree

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

```
class Solution {
public:
    bool isBalanced(TreeNode* root) {
        if(!root) return true;
        if(abs(get_depth(root->left) - get_depth(root->right))>1) return false;
        return isBalanced(root->left) && isBalanced(root->right);
    }

    int get_depth(TreeNode *root){
        if(!root) return 0;
        return max(get_depth(root->left), get_depth(root->right))+1;
    }
};
```

6.4. 前序和中序遍历重建二叉树 buildTree

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:
    unordered_map<int, int> mp;

    TreeNode* dfs(vector<int>& preorder, vector<int>& inorder, int pl, int pr, int
il, int ir) {
        if(pl>pr) return NULL;
        int k = mp[preorder[pl]] - il;
        TreeNode* root=new TreeNode(preorder[pl]);
        root->left = dfs(preorder, inorder, pl+1, pl+k, il, il+k-1);
```

```
        root->right = dfs(preorder, inorder, pl + k + 1, pr, il + k + 1, ir);
        return root;
    }

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {

        for(int i = 0; i<preorder.size(); i++){
            mp[inorder[i]] = i;
        }
        return dfs(preorder, inorder, 0, preorder.size()-1, 0, inorder.size()-1);
    }
};
```

6.5. 序列化二叉树 serializetree

请实现两个函数，分别用来序列化和反序列化二叉树。 示例: 你可以将以下二叉树：

```
  1
```

/
2 3 /
4 5

序列化为 "[1,2,3,null,null,4,5]"

```
class Codec {
public:

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        string ans;
        dfs_serialize(root , ans);
        // cout<<ans;
        return ans;
    }
    void dfs_serialize(TreeNode *root, string& ans){
        if(!root) {
            ans += "null ";
            return;
        }
        ans += to_string(root->val);
        ans += " ";
        dfs_serialize(root->left, ans);
        dfs_serialize(root->right, ans);
        return;
    }s

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
```

```
        int cur = 0;
        auto root = dfs_deserialize(data , cur);
        return root;
    }

    TreeNode* dfs_deserialize(string& data, int& cur){
        if(cur>=data.size()) return NULL;
        if(data[cur] == 'n'){
            cur += 5;
            return NULL;
        }
        int temp = 0, flag = 1;
        if(data[cur] == '-'){
            flag = -1;
            cur++;
        }
        while(data[cur] !=' '){
            temp *= 10;
            temp += (int)(data[cur]-'0');
            cur++;
        }
        cur++;
        TreeNode *root = new TreeNode(flag*temp);
        root->left = dfs_deserialize(data, cur);
        root->right = dfs_deserialize(data, cur);
        return root;
    }
};
```

6.6. 判断对称（镜像）的二叉树 isSymmetric

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);
    }
};
```

6.7. 输出二叉树的镜像 mirror

```
// 后序无返回值
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);
    }
};
// 前序有返回值
class Solution {
public:
    TreeNode* mirrorTree(TreeNode* root) {
        if(!root) return NULL;
        mirrorTree(root->left);
        mirrorTree(root->right);
        TreeNode *t = root->left;
        root->left = root->right;
        root->right = t;
        return root;
    }
};
```

6.8. 不分行从上往下打印二叉树(层次遍历) printFromTopToBottom1

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;
    }
};
```

6.9. 分行从上往下打印二叉树 printFromTopToBottom2

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;
    }
};
```

6.10. 之字形打印二叉树 printFromTopToBottom3

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;
    }
};
```

6.11. 二叉树最低公共祖先 lowestCommonAncestor1

```
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;
    }
};
```

6.12. 二叉树搜索树最低公共祖先

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if((root -> val - p->val)*(root -> val - q ->val) <= 0 ){
            return root;
        }
        if(root -> val > p -> val){
            return lowestCommonAncestor(root -> left, p, q);
        }
        if(root -> val < p -> val){
            return lowestCommonAncestor(root -> right, p, q);
        }
        return NULL;
    }
};
```

6.13. 二叉搜索树转换为双向循环链表 treeToDoublyList

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向.

```
class Solution {
public:
    Node* treeToDoublyList(Node* root) {
        if(!root) return nullptr;
        Node* head = nullptr, *pre = nullptr;
        helper(root, head, pre);
        head->left = pre;
        pre->right = head;
        return head;
    }
    void helper(Node* root, Node*& head, Node*& pre) {
        if(!root)  return;
        helper(root->left, head, pre);
        if(!head) {
            head =   root;   // 找到head
            pre = root;    // 对pre进行初始化
        } else {
            pre->right = root;
            root->left = pre;
            pre = root;
        }
        helper(root->right, head, pre);
    }
};
```

6.14. 二叉搜索树的第k大节点 treekthLargest

```
class Solution {
public:
    int cou = 0;
    int ans;
    int kthLargest(TreeNode* root, int k) {
        inorder(root, k);
        return ans;
    }
    void inorder(TreeNode * root, int k){
        if(!root) return;
        inorder(root->right, k);
        cou++;
        if(cou == k) {ans = root->val; return;}
        inorder(root->left, k);
    }
};
```

6.15. 二叉搜索树的后序遍历序列 verifySequenceOfBST

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);
    }
};
```

6.16. 二叉树中和为某一值的路径(回溯) treePathSum

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;
    }
};
```

6.17. 二叉树中和为某一值的路径 treeFindPath1

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();
    }
};
```

6.18. 合并二叉树 - 相加二叉树 mergeTrees

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。
你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为 NULL 的节点将直接作为新二叉树的节点。

```
class Solution {
public:
    TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
        if(!t1&&!t2) return NULL;
        TreeNode* root = new TreeNode(0);
        if(!t1){
            return t2;
        }
        else if(!t2){
            return t1;
        }
        else{
            root->val = t1->val+t2->val;
```

```
        }

        root->left = mergeTrees(t1->left,t2->left);
        root->right = mergeTrees(t1->right,t2->right);
        return root;
    }

};
```

6.19. 二叉树剪枝 (去掉全为0的子树) pruneTree

给定二叉树根结点 root ， 此外树的每个结点的值要么是 0，要么是 1。

返回移除了所有不包含 1 的子树的原二叉树。

```
// 后续遍历
class Solution {
public:
    TreeNode* pruneTree(TreeNode* root) {
        if(!root) return NULL;
        root->left = pruneTree(root->left);
        root->right = pruneTree(root->right);
        if(root->val == 0 && !root->left && ! root->right) return NULL;
        return root;
    }
};
// 前序判断
class Solution {
public:
    TreeNode* pruneTree(TreeNode* root) {
        if(!root) return NULL;
        if(has1(root))return NULL;
        root->left = pruneTree(root->left);
        root->right = pruneTree(root->right);
        return root;
    }
    bool has1(TreeNode *root){
        if(!root) return true;
        if(root->val == 1) return false;
        return has1(root->left) &&  has1(root->right);
    }
};
```

6.20. 翻转二叉树 (输出对称二叉树) invertTree1

翻转一棵二叉树。

```
class Solution {
public:
```

```
    TreeNode* invertTree(TreeNode* root) {
        if(!root) return NULL;
        TreeNode *tem = NULL;
        tem = root->left;
        root->left = root->right;
        root->right = tem;
        invertTree(root->left);
        invertTree(root->right);

        return root;
    }
};
```

6.21. 树的子结构(判断B是不是A的子结构) hasSubtree

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);
    }
};
```

6.22. 构造最大二叉树

给定一个不含重复元素的整数数组。一个以此数组构建的最大二叉树定义如下：

二叉树的根是数组中的最大元素。 左子树是通过数组中最大值左边部分构造出的最大二叉树。 右子树是通过数组中最大值右边部分构造出的最大二叉树。

```
class Solution {
public:
    TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
        return dfs(nums, 0, nums.size()-1);
    }
    TreeNode *dfs(vector<int>& nums, int l, int r){
        if(l>r) return NULL;
        int max_ind = l;
```

```
        for(int i = l; i<=r; i++) if(nums[i]>nums[max_ind]) max_ind = i;
        TreeNode *root = new TreeNode(nums[max_ind]);
        root->left = dfs(nums, l, max_ind-1);
        root->right = dfs(nums, max_ind+1, r);
        return root;
    }
};
```

6.23. 96. 二叉搜索树个数 numbTrees

给定一个整数 n，求以 1 ... n 为节点组成的二叉搜索树有多少种?

```
class Solution {
public:
    int numTrees(int n) {
        vector<int> dp(n+1, 0); dp[0] = 1; dp[1] = 1;
        for(int i = 2; i<=n; i++){
            // j为当前选中节点， 左边为j-1个，右边为i-j个。
            for(int j = 1; j <= i; j++)
                dp[i] += dp[j - 1] * dp[i - j];
        }
        return dp[n];
    }
};
```

6.24. 98. 验证二叉搜索树 isValidBST

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

节点的左子树只包含小于当前节点的数。
节点的右子树只包含大于当前节点的数。
所有左子树和右子树自身必须也是二叉搜索树。

```
class Solution {
public:
    vector<int> ans;
    void inorder(TreeNode *root){
        if(!root) return;
        inorder(root->left);
        ans.push_back(root->val);
        inorder(root->right);
    }
    bool isValidBST(TreeNode* root) {
        inorder(root);
        for(int i = 1; i<ans.size(); i++){
            if(ans[i] <= ans[i-1]) return false;
        }
```

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

6.25. 99. 恢复二叉搜索树 recoverTreeB

二叉搜索树中的两个节点被错误地交换。

请在不改变其结构的情况下，恢复这棵树.

```
class Solution {
public:
    vector<TreeNode *> in;
    void inorder(TreeNode* root){
        if(!root) return;
        inorder(root->left);
        in.push_back(root);
        inorder(root->right);
    }
    // 利用搜索二叉树中序遍历的性质
    void recoverTree(TreeNode* root) {
        TreeNode *p1 = nullptr, *p2 = nullptr;
        inorder(root);
        for(int i = 0; i<in.size() - 1; i++){
            if(in[i]->val>in[i+1]->val){
                // ***
                if(!p1) p1 = in[i];
                p2 = in[i+1];
            }
        }
        if(p1 && p2){
            int t = p1->val;
            p1->val = p2->val;
            p2->val = t;
        }
    }
};
```

6.26. 100. 相同的树 isSameTree

给定两个二叉树，编写一个函数来检验它们是否相同。
如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(!p && !q) return true;
        if(!p) return false;
        if(!q) return false;
```

```
        if(p->val != q->val) return false;
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```

6.27. 单值二叉树 isUnivalTree

如果二叉树每个节点都具有相同的值，那么该二叉树就是单值二叉树。
只有给定的树是单值二叉树时，才返回 true；否则返回 false。

```
class Solution {
public:
    bool isUnivalTree(TreeNode* root) {
        if(!root) return true;
        if(root->left && root->val != root->left->val) return false;
        if(root->right && root->val != root->right->val) return false;
        return isUnivalTree(root->left) && isUnivalTree(root->right);
    }
};
```

6.28. 修剪二叉搜索树 trimBST

给定一个二叉搜索树，同时给定最小边界L 和最大边界 R。通过修剪二叉搜索树，使得所有节点的值在[L, R]中 (R>=L) 。你可能需要改变树的根节点，所以结果应当返回修剪好的二叉搜索树的新的根节点。
https://leetcode-cn.com/problems/trim-a-binary-search-tree/

```
class Solution {
public:
    TreeNode* trimBST(TreeNode* root, int &L, int &R) {
        if (root == nullptr) return nullptr;
        if (root->val < L) return trimBST(root->right, L, R);
        if (root->val > R) return trimBST(root->left, L, R);

        root->left = trimBST(root->left, L, R);
        root->right = trimBST(root->right, L, R);
        return root;
    }
};
```

6.29. 翻转等价二叉树 (判断经过左右互换变为同一棵树) flipEquiv

我们可以为二叉树 T 定义一个翻转操作，如下所示：选择任意节点，然后交换它的左子树和右子树。
只要经过一定次数的翻转操作后，能使 X 等于 Y，我们就称二叉树 X 翻转等价于二叉树 Y。

```
class Solution {
public:
```

```
    bool flipEquiv(TreeNode* root1, TreeNode* root2) {
        if (root1 == root2) return true;
        if (!root1 || !root2 || root1->val != root2->val) return false;
        return flipEquiv(root1->left, root2->left) && flipEquiv(root1->right,
root2->right) ||
               flipEquiv(root1->left, root2->right) && flipEquiv(root1->right,
root2->left);
    }
};
```

二叉树的坡度（所有节点左右子树的差的和） findTilt 给定一个二叉树，计算整个树的坡度。
一个树的节点的坡度定义即为，该节点左子树的结点之和和右子树结点之和的差的绝对值。空结点的的坡度是0。
整个树的坡度就是其所有节点的坡度之和。

```
class Solution {
public:
    int ans = 0;
    int findTilt(TreeNode* root) {
        helper(root);
        return ans;
    }
    int helper(TreeNode *root){
        if(!root) return 0;
        int l = helper(root->left);
        int r = helper(root->right);
        ans += abs(l - r);
        return l+r+root->val;
    }
};
```

6.30. 填充二叉树的右侧节点指针 (层次遍历变形) connecttreenode

给定一个完美二叉树，其所有叶子节点都在同一层，每个父节点都有两个子节点。} 填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。
https://leetcode-cn.com/problems/populating-next-right-pointers-in-each-node/

```
class Solution {
public:
    Node* connect(Node* root) {
        if (root == NULL) return NULL;
        queue<Node*> Q;
        Q.push(root);
        while (!Q.empty()) {
            int size = Q.size();
            for(int i = 0; i < size; i++) {
                Node* node = Q.front();
                Q.pop();
```

```
                // 不是当前层最后一个
                if (i < size - 1) node->next = Q.front();
                if (node->left) Q.push(node->left);
                if (node->right) Q.push(node->right);
            }
        }
        return root;
    }
};
```

7. 剑指Offer

7.1. 数组中超过一半的数字 majorityElement

设置一个计数器count，每遇到一个和当前的数字相同的数字，就让count自增，遇到一个和当前数字不一样的数字，就让count--，当count < 0时，就将cur设置为当前遍历的数字。因为有一个数字出现次数超过数组长度的一半，最后得到的必然是该数字。

```
class Solution {
public:
    int moreThanHalfNum_Solution(vector<int>& nums) {
        int res=nums[0];
        int cnt = 1;
        for (int i = 1; i<nums.size(); i++){
            if(nums[i] == res) cnt++;
            else{
                cnt--;
                if(cnt==0){
                    res = nums[i];
                    cnt++;
                }
            }
        }
        return res;
    }
};
```

7.2. 找出数组中重复数字 duplicateInArray

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;
    }
};
```

7.3. 不修改数组找出重复的数字 duplicateInArray2

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;
    }
};
```

7.4. 二维数组查找 findNumberIn2DArray

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;

    }
};
```

7.5. 替换空格为%20 replaceSpaces

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;
    }
};
```

7.6. 从尾到头打印链表 （逆序打印链表） printListReversingly

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());
    }
};
```

7.6.1. 递归方式

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

7.7. 二叉树的下一个结点（给定father结点） inorderSuccessor

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;
    }
};
```

7.8. 两个栈实现一个队列 2stack2queue

```
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;
    }
};
```

7.9. 打印从1到最大的n位数 printNumbers 1-n

```
class Solution {
public:
    vector<int> printNumbers(int n) {
        vector<int> ans;
        for (int i =1; i<pow(10,n); i++){
            ans.push_back(i);
        }
        return ans;
    }
};
```

7.10. 斐波那契数列 Fibonacci

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;
    }
};
```

7.11. 旋转数组的最小数字（二分查找） minArray

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];
    }
};
```

7.12. 矩阵中的路径（DFS路径） existpath

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;
        }
    };
```

7.13. 机器人的运动范围（bfs搜索） movingCount

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;
    }
};
```

7.14. 剪绳子（分段最大乘积） maxProductAfterCutting

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;
    }
};
```

7.15. 二进制中1的个数（unsigned int n = _n;） NumberOf1

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;
    }
};
```

7.16. 实现数值的整数次方，即pow() Power

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;
    }
};
```

7.17. 删除链表的节点 deleteNodett

```
class Solution {
public:
    ListNode* deleteNode(ListNode* head, int val) {
        ListNode*pre, *p;
        if(head->val == val) {head = head->next; return head;}
        pre = head; p = head->next;
        while(p){
            if(p->val == val){
                pre->next = p->next;
                p=p->next;
            }
            else{
                pre = p;
                p = p->next;
            }
        }
        return head;
    }
};
```

7.18. 在O(1)时间删除链表结点 deleteNode

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;
    }
};
```

7.19. 删除链表中重复的节点 deleteDuplication

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;
    }
};
```

7.20. 正则表达式匹配 isMatch

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];
    }
};
```

7.21. 表示数值的字符串 isNumber

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
```

7.22. 调整数组顺序使奇数位于偶数前面 reOrderArray

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--;
        }
    }
};
```

7.22.1. 双指针解法2 reOrderArray2

```
class Solution {
public:
    vector<int> exchange(vector<int>& nums) {
        for(int i = 0,j=0;j<nums.size();j++){
            if(nums[j]%2!=0) swap(nums[i],nums[j]),i++;
        }
```

```
        return nums;
    }
};
```

7.23. 链表中倒数第k个节点 findKthToTail

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;
    }
};
```

7.24. 寻找环形链表入口 entryNodeOfLoop

```
/*
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;
    }
};
```

7.25. 翻转链表 reverseList

7.25.1. (1)迭代 r1

```
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;
    }
};
```

7.25.2. (2) 递归 r2

```
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;
    }
};
```

7.26. 合并两个排序的链表 merge

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;
    }
};
```

7.27. 顺时针打印矩阵 printMatrix

```
/*
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;
    }
};
```

7.28. 包含min函数的栈 MinStack

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();
    }
};
```

7.29. 栈的压入、弹出序列 isPopOrder

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();
    }
};
```

7.30. 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) {}
 * };
 */
```

7.31. 字符串转数字 strToInt

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;
    }
};
```

7.32. 约瑟夫坏（圆圈中最后剩下的） lastRemaining

7.32.1. 暴力模拟 I1

```
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];

    }
};
```

7.32.2. 递推 I2

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

```
            return (lastRemaining(n - 1, m) + m) % n;
    }
};
```

7.33. 扑克牌顺子 isContinuous

```
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;
    }
};
```

7.34. 一排路由器可以覆盖的信号 Router

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;
}
```

7.35. 滑动窗口最大值 slide

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 maxShares 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;
    }
};
```

7.36. 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;
    }
};
```

7.37. 分裂二叉树最大乘积 maxProduct

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);
    }
};
```

7.38. 大数相乘 BigMutiple

```
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;
}
```

7.39. 大数相加 bigAdd

```
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;
}
```

7.40. 不用加减乘除做加法 bitopAdd

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;
    }
};
```

8. 动态规划 dynamic programming

8.1. 最长上升子序列 lengthOfLIS

```
class Solution
{
public:
    int bs(vector<int> &dp, int target, int l, int r){
        while(l < r){
            int mid = (l + r) >> 1;
            if(dp[mid] >= target) r = mid;
            else l = mid + 1;
        }
        return r;
    }
    int lengthOfLIS(vector<int>& nums) {
        if(nums.size()==0) return 0;
        vector<int> dp;
        dp.push_back(nums[0]);
        for(int i =1; i<nums.size(); ++i){
            if(nums[i]>dp[dp.size()-1])
                dp.push_back(nums[i]);
            else{
                dp[bs(dp, nums[i], 0, dp.size()-1)] = nums[i];
            }
        }
        return dp.size();
    }
};
```

8.2. 最长公共子序列 longestCommonSubsequence

给定两个长度分别为N和M的字符串A和B，求既是A的子序列又是B的子序列的字符串长度最长是多少。
https://leetcode-cn.com/problems/longest-common-subsequence/submissions/

输入样例： 4 5 acbd abedc

```
#include <iostream>
using namespace std;
const int N = 1010;
int n, m;
char a[N], b[N];
int f[N][N];
int main() {
  cin >> n >> m >> a + 1 >> b + 1;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      if (a[i] == b[j]) {
        f[i][j] = f[i - 1][j - 1] + 1;
      } else {
        f[i][j] = max(f[i - 1][j], f[i][j - 1]);
      }
    }
  }
  cout << f[n][m] << '\n';
  return 0;
}、


class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int n = text1.size(), m = text2.size();
        vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
        for(int i = 1; i<=n; i++){
            for(int j = 1; j<=m; j++){
                if(text1[i-1] == text2[j-1]) dp[i][j] = dp[i-1][j-1] + 1;
                else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
        return dp[n][m];
    }
};
```

8.3. 三角形最小路径和 sanjiaominimumTotal

给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。 相邻的结点 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。

例如，给定三角形：

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

```
class Solution {
public:
    //不改变原数组， 额外n个空间
    int minimumTotal(vector<vector<int>>& triangle) {
        int n = triangle.size();
        vector<int> dp(n);
        dp[0] = triangle[0][0];
        for(int i = 1; i < n; i++){
            // 每一行最右侧的元素
            dp[i] = dp[i - 1] + triangle[i][i];
            for(int j = i-1; j > 0; j--)
                dp[j] = min(dp[j - 1], dp[j]) + triangle[i][j];
            // 每一行最左侧的元素
            dp[0] = dp[0] + triangle[i][0];
        }
        return *min_element(dp.begin(), dp.end());
    }
};

class Solution {
public:
    //  原地操作
    int minimumTotal(vector<vector<int>>& triangle) {
        int n = triangle.size();
        for (int i = 1; i < n; ++i) {
            // 每一行最左侧的元素
            triangle[i][0] = triangle[i - 1][0] + triangle[i][0];
            for (int j = 1; j < i; ++j)
                triangle[i][j] = min(triangle[i - 1][j - 1], triangle[i - 1][j]) +
triangle[i][j];
            // 每一行最右侧的元素
            triangle[i][i] = triangle[i - 1][i - 1] + triangle[i][i];
        }
        return *min_element(triangle[n - 1].begin(), triangle[n - 1].end());
    }
};
```

8.4. 按照频率将数组升序排序 frequencySort

```
#include<unordered_map>
#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;
vector<int> frequencySort(vector<int>& nums) {
    unordered_map<int ,int> mp;
    for(auto x: nums) mp[x]++;
    // lambda 表达式 [&]获取外部作用域中所有变量，并作为引用在函数体中使用
```

```
    sort(nums.begin(), nums.end(), [&](int a, int b){
        if(mp[a] != mp[b]) return mp[a]<mp[b];
        return a>b;
    });
    return nums;
}


int main(){
    vector<int> c = {1, 1, 1,2,3,3};
    frequencySort(c);
    for(int i = 0; i<c.size(); i++) cout<<c[i]<<' ';
    return 0;
}
```

9. C++ 刷题知识 Brush the question.

9.1. 不常见输入方式 nousuallyinput

输入：
a,c,bb
f,dddd
nowcoder

```
while (cin>>s){
    vector<string>a;
    string tmp;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] == ',') {
            a.push_back(tmp);
            tmp.clear();
        }
        else{
            tmp += s[i];
        }
    }
    a.push_back(tmp);
}
```

对输入的字符串进行排序后输出

输入
a c bb
f dddd
nowcoder

```
#include<vector>
#include<iostream>
```

```
#include<string>
#include<algorithm>
using namespace std;
int main(){
    string str;
    while(getline(cin, str)){
        //cout<<str;
        vector<string> ans;
        string tem;
        for(int i = 0; i<str.size(); i++){
            if(str[i] == ' ') {
                ans.push_back(tem);
                tem.clear();
            }
            else tem += str[i];
        }
        ans.push_back(tem);
        sort(ans.begin(), ans.end());
        for(int i = 0; i<ans.size()-1; i++) cout<<ans[i]<<' ';
        cout<<ans[ans.size()-1]<<'\n';
    }
    return 0;
}
```

9.2. vector(动态数组)

vector 是向量类型，它可以容纳许多类型的数据，如若干个整数，所以称其为容器。vector 是C++ STL的一个重要成员，使用它时需要包含头文件：#include;

9.2.1. vector初始化 init

```
    (1) vector<int> a(10); //定义了10个整型元素的向量（尖括号中为元素类型名，它可以是任何合法的数据类型），但没有给出初值，其值是不确定的。
    (2) vector<int> a(10,1); //定义了10个整型元素的向量,且给出每个元素的初值为1
    (3) vector<int> a(b); //用b向量来创建a向量，整体复制性赋值
    (4) vector<int> a(b.begin(),b.begin+3); //定义了a值为b中第0个到第2个（共3个）元素
    (5) int b[7]={1,2,3,4,5,9,8};
        vector<int> a(b,b+7); //从数组中获得初值
```

9.2.2. vector 重要操作 method

```
    (1) a.assign(b.begin(), b.begin()+3); //b为向量，将b的0~2个元素构成的向量赋给a
    (2) a.assign(4,2); //是a只含4个元素，且每个元素为2
    (3) a.back(); //返回a的最后一个元素
    (4) a.front(); //返回a的第一个元素
    (5) a[i]; //返回a的第i个元素，当且仅当a[i]存在2013-12-07
    (6) a.clear(); //清空a中的元素
    (7) a.empty(); //判断a是否为空，空则返回ture,不空则返回false
```

```
    (8) a.pop_back(); //删除a向量的最后一个元素
    (9) a.erase(a.begin()+1,a.begin()+3); //删除a中第1个（从第0个算起）到第2个元素，也就是说删除的元素从a.begin()+1算起（包括它）一直到a.begin()+3（不包括它）
    (10) a.push_back(5); //在a的最后一个向量后插入一个元素，其值为5
    (11) a.insert(a.begin()+1,5); //在a的第1个元素（从第0个算起）的位置插入数值5，如a为1,2,3,4，插入元素后为1,5,2,3,4
    (12) a.insert(a.begin()+1,3,5); //在a的第1个元素（从第0个算起）的位置插入3个数，其值都为5
    (13) a.insert(a.begin()+1,b+3,b+6); //b为数组，在a的第1个元素（从第0个算起）的位置插入b的第3个元素到第5个元素（不包括b+6），如b为1,2,3,4,5,9,8，插入元素后为1,4,5,9,2,3,4,5,9,8
    (14) a.size(); //返回a中元素的个数;
    (15) a.capacity(); //返回a在内存中总共可以容纳的元素个数
    (16) a.resize(10); //将a的现有元素个数调至10个，多则删，少则补，其值随机
    (17) a.resize(10,2); //将a的现有元素个数调至10个，多则删，少则补，其值为2
    (18) a.reserve(100); //将a的容量（capacity）扩充至100，也就是说现在测试a.capacity();的时候返回值是100.这种操作只有在需要给a添加大量数据的时候才显得有意义，因为这将避免内存多次容量扩充操作（当a的容量不足时电脑会自动扩容，当然这必然降低性能）
    (19) a.swap(b); //b为向量，将a中的元素和b中的元素进行整体性交换
    (20) a==b; //b为向量，向量的比较操作还有!=,>=,<=,>,<
```

9.2.3. vector 读写 readwrite

```
// 添加元素
int a[6]={1,2,3,4,5,6};
vector<int> b;
vector<int> c(a,a+4);
for(vector<int>::iterator it=c.begin();it<c.end();it++)
b.push_back(*it);

// 通过遍历器方式读取
int a[6]={1,2,3,4,5,6};
vector<int> b(a,a+4);
for(vector<int>::iterator it=b.begin();it!=b.end();it++)
    cout<<*it<<" ";
```

9.2.4. vector常用algorithm算法

```
#include<algorithm>
 (1) sort(a.begin(),a.end()); //对a中的从a.begin()（包括它）到a.end()（不包括它）的元素进行从小到大排列
 (2) reverse(a.begin(),a.end()); //对a中的从a.begin()（包括它）到a.end()（不包括它）的元素倒置，但不排列，如a中元素为1,3,2,4,倒置后为4,2,3,1
 (3) copy(a.begin(),a.end(),b.begin()+1); //把a中的从a.begin()（包括它）到a.end()（不包括它）的元素复制到b中，从b.begin()+1的位置（包括它）开始复制，覆盖掉原有元素
 (4) find(a.begin(),a.end(),10); //在a中的从a.begin()（包括它）到a.end()（不包括它）的元素中查找10，若存在返回其在向量中的位置
```

```
    vector<int>::iterator t = find(b.begin(), b.end(), 0);
    if (t != b.end()) cout << *t;
```

9.3. set集合

set翻译为集合，是一个内部自动有序且不含重复元素的容器。默认是升序。底层采用红黑树实现。
set的定义：set<'typename'> s，降序的定义方式为set<typename,greater> s。typename可以是任意类型包括STL容器。Set数组的定义方式为，set s[size].s[0]...s[size-1]都是set类型。迭代器的定义方式set::iterator it
set容器内元素的访问：set只能通过迭代器(iterator)访问。

9.3.1. set重要操作 method

set的常见用途： set最主要的作用是自动去重并且升序排序，因此碰到需要去重但不方便开数组的 时候，可以尝试用set解决。 注意：set中的元素是唯一的，如果需要处理不唯一的情况可以使用multiset。C++11中还增加了unordered_ set,以散列代替set内部的红黑树，unordered_set可以处理需要去重但是不需要排序的情况，速度比set快得多。Multiset和unordered_set的定义和常用函数和set类似。

```
 (1)  insert(x) 可将x插入set容器中，并且自动递增排序和去重，时间复杂度O(logN),其中N是set中元素的数量。
 (2)  find (x) 返回set中对应值为x的迭代器，时间复杂度O(logN),N为set内元素的个数。
 (3)  erase () , erase有两种用法：删除单个元素，删除一个区间内的所有元素。删除单个元素有两种方式，erase(it)删除该迭代器对应的元素，时间复杂度O(1),erase(x)删除该元素,时间复杂度O(logN).删除一个区间的元素，erase(st,ed),删除区间[st,ed)内的元素，时间复杂度O(ed-st)
 (5)  clear(),用来清空set中所有元素，复杂度O(N),其中N为set内元素的个数。
 (6)  count(x),返回set中x的数量
for (auto &ele : st) cout << ele << ' ';
    cout << st.size() << endl;
cout << st.count(4) << endl;//set去重了，返回只能是0或1
cout << *st.find(1) << endl;
st.erase(1);
st.erase(st.begin(), st.find(4));

puts("");
st.clear();
cout << st.size() << endl;
```

9.4. string 字符串

定义方式与基本数据类型相同，只需要在string后面跟上变量名称即可。
eg. string str;如果需要初始化，可以直接给string类型的变量赋值，string str = “hello”。

9.4.1. string method

```
// 输入输出
string str ;
cin >> str ;
cout << str << endl ;
```

```
printf("%s\n",str.c_str()) ;
return 0 ;
输入：hello
输出：hello
hello

// string和vector一样，支持直接对迭代器进行加减某个数字。
string str = "hello" ;
string::iterator it  ;
for(it = str.begin();it!=str.end();it++){
    cout << *it ;
}
 (1)  operator+=拼接
string str1 = "hello" ;
string str2 = " world" ;

 (2)  compare operator比较
两个string类型可以直接使用==,!=,<,<=,>,>=比较大小，比较规则是字典序。
 (3)  length()/size()取得大小
length()返回string的长度，即string存放的字符数，时间复杂度O(1)。size()和length()基本相同。
 (4)  insert () 插入（原字符串不会被覆盖）
insert()函数有很多种写法，这里列出几个常用的写法，时间复杂度度O(N)。
insert(pos,string)，在pos号位置插入string。
insert(it,it1,it2)，it为原字符串欲插入的位置，it2和it3为待插字符串的首尾迭代器，用来表示串[it1,it2)将被插在it的位置上。
 (5)  erase()删除
erase()有两种用法，删除单个元素，上出一个区间内所有元素。时间复杂度O(N)。
a.  erase(it)用于删除单个元素，it为需要删除的元素的迭代器。
b. 删除一个区间的元素有两种方法：
第一种是erase(st,ed),st，ed为string迭代器，表示删除区间[st,ed)之间的元素。
第二种是erase(pos,len)，其中pos为需要删除的起始位置，len为删除的字符个数
 (7)  substr()截取子串
substr(pos,len)返回的是以pos位开始长度为len的子串，时间复杂度O(len)。
 (8)  find()查找
str.find(str1),当str1是str的子串时，返回其在str中第一次出现的位置。如果str1不是str的子串，那么返回string::npos
str.find(str1,pos),从str的pos号位开始匹配str1,返回值与上面的相同，时间复杂度为O(nm),其中n和m分别为str和str1的长度。

与algorithm中find的区别：
find(a.begin(), a.end(), 'a');
此函数只能查找单个元素，找不到返回a.end()，找到返回"a"的迭代器，若取索引可以
find(a.begin(), a.end(), 'a') - a.begin();
 (9)  replace()
str.replace(pos,len,str1),把str从pos号位开始，长度为len的子串替换为str1。
str.replace(it1,it2,str1)把str的迭代器[it1,it2)替换为str1

string str1 = "hello world" ;
    string str2 = "kangkang" ;
    cout << str1.replace(6,5,str2) << endl ;
    cout << str1.replace(str1.begin()+6,str1.end(),str2) << endl ;
结果：hello kangkang
hello kangkang
```

9.5. map 映射

map翻译成映射，map可以将任何基本类型（包括STL容器）映射到任何基本类。（包括STL容器）。

```
map<string, int> mp;
mp["aa"] = 1;
mp.insert({"bb", 2});
cout << mp["bb"]; // 2
cout << mp["cc"]; // 0
mp["dd"]; // 声明之后就存在， value值为 0。
cout << (mp.find("dd") != mp.end()); // 1

// 遍历
for (auto &i : mp) {
        cout << i.first<<i.second<<',';
    }
    cout << endl;
for (map<string, int>::iterator it = mp.begin(); it != mp.end(); it++) {
    cout << it->first << ' ' << it->second;
}
迭代器本质是指针，所以使用 -> ， 若为set是，因为储存单个值，使用 *it 即可访问。

// erase(),erase()有两种用法：删除单个元素和删除一个区间内的元素。
// 删除单个元素时，可以接受迭代器和key值，删除区间元素智能接受迭代器
（1） 需要建立字符（或字符串）与整数之间映射的题目，使用map可以减少代码量。
（2） 判断大整数或者其他类型数据是否存在的题目，可以把map当成bool数组用。
（3） 字符串和字符串之间的映射。
补充：map和键和值都是唯一的
```

9.6. unordered_map——哈希表

unordered_map是C++中的哈希表，可以在任意类型与类型之间做映射。

1. 引用头文件(C++11)：#include <unordered_map>
2. 定义：unordered_map<int,int>、unordered_map<string, double> ...
3. 插入：例如将("ABC" -> 5.45) 插入unordered_map<string, double> hash中，hash["ABC"]=5.45
4. 查询：hash["ABC"]会返回5.45
5. 判断key是否存在：hash.count("ABC") != 0 或 hash.find("ABC") != hash.end()
6. 遍历

```
for (auto &item : hash)
{
    cout << item.first << ' ' << item.second << endl;
}
或者：
for (unordered_map<string, double>::iterator it = hash.begin(); it != hash.end();
it ++ )
{
```

```
    cout << it->first << ' ' << it->second << endl;
}
```

9.7. 由数据范围反推算法复杂度以及算法内容 datarange2algorithm

一般ACM或者笔试题的时间限制是1秒或2秒。 在这种情况下，C++代码中的操作次数控制在 107107 为最佳。

下面给出在不同数据范围下，代码的时间复杂度和算法该如何选择：

n≤30n≤30, 指数级别, dfs+剪枝，状态压缩dp

n≤100n≤100 => O(n3)O(n3)，floyd，dp

n≤1000n≤1000 => O(n2)O(n2)，O(n2logn)O(n2logn)，dp，二分，朴素版Dijkstra、朴素版Prim、Bellman-Ford

n≤10000n≤10000 => O(n∗n√)O(n∗n)，块状链表、分块、莫队

n≤100000n≤100000 => O(nlogn)O(nlogn) => 各种sort，线段树、树状数组、set/map、heap、拓扑排序、dijkstra+heap、prim+heap、spfa、求凸包、求半平面交、二分

n≤1000000n≤1000000 => O(n)O(n), 以及常数较小的 O(nlogn)O(nlogn) 算法 => hash、双指针扫描、并查集，kmp、AC自动机，常数比较小的 O(nlogn)O(nlogn) 的做法：sort、树状数组、heap、dijkstra、spfa

n≤10000000n≤10000000 => O(n)O(n)，双指针扫描、kmp、AC自动机、线性筛素数

n≤109n≤109 => O(n√)O(n)，判断质数

n≤1018n≤1018 => O(logn)O(logn)，最大公约数，快速幂

n≤101000n≤101000 => O((logn)2)O((logn)2)，高精度加减乘除

n≤10100000n≤10100000 => O(logn×loglogn)O(logn×loglogn)，高精度加减、FFT/NTT