一、线性表

0x00 二分法

查找>=key的第一个元素，描述为满足某种情况的最小的元素。

如 2 3 3 4 4 4 5查找 4 则返回 下标3 即最左边的4

```
    int l = 0, r = n-1;
        while(l < r){
            int mid = l + r >> 1;
            if(a[mid] >= x) r = mid;
            else l = mid+1;
        }
        if(a[l]!=x) printf("-1 -1\n");//查找失败
```

查找小于<=key的最后一个元素，描述为满足某种情况的最大的元素。

如 2 3 3 4 4 4 5查找 4 则返回 下标5 即最右边的4

```
        int l = 0, r = n-1;
            while(l < r){
                int mid = l+r+1>>1;
                if(a[mid] <= x) l = mid;
                else r = mid-1;
            }
            if(a[l]!=x) printf("-1 -1\n");//查找失败
```

##

0x01 回文链表

请判断一个链表是否为回文链表。

示例 1:

输入: 1->2 输出: false

示例 2:

输入: 1->2->2->1 输出: true

```
    //     s-慢指针 f-快指针
        //     循环结束时的状态如下:
        //     1 2 2 1 NULL 偶数长度 后半部分起点就是s
        //         s    f
        // else
```

```
        //     1 2 3 2 1 奇数长度  后半部分起点是s的下一个
        //         s   f
    bool isPalindrome(ListNode* head) {
        if(!head||!head->next) return true;//0个或1个数自然为真
        stack<int> stk;//存放前半个数
        auto f = head,s = head;
        while(f&&f->next){
            stk.push(s->val);
            s = s->next;
            f = f->next->next;
        }
        if(f) s = s->next;//后半部分起点
        while(s){
            if(s->val!=stk.top()) return false;
            stk.pop(),s = s->next;
        }
        return true;
    }
```

##

0x02 删除链表的倒数第n个节点

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：

给定一个链表: 1->2->3->4->5, 和 n = 2.

当删除了倒数第二个节点后，链表变为 1->2->3->5.

```
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode *L = new ListNode(0);
        L->next = head;
        ListNode *f = L,*s = L;
        while(n--&&f->next){f = f->next;}
        while(f->next){
            s = s->next;
            f = f->next;
        }
        s->next = s->next->next;
        return L->next;
    }
```

##

0x03 合并两个有序链表

将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例：

输入：1->2->4, 1->3->4

输出：1->1->2->3->4->4

```
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {//迭代版
        auto h = new ListNode(0), r = h;//r为尾结点
        while(l1&&l2){
            if(l1->val < l2->val) r->next = l1,l1 = l1->next,r = r->next;//把l1连到尾部
            else r->next = l2,l2 = l2->next,r = r->next;
        }
        if(!l1) r->next = l2;
        if(!l2) r->next = l1;
        return h->next;
    }
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {//递归版
        if(!l1) return l2;//空时
        if(!l2) return l1;
        if(l1->val < l2->val){
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        }
        else{
            l2->next = mergeTwoLists(l1,l2->next);
            return l2;
        }
    }
```

##

0x04 旋转链表

给定一个链表，旋转链表，将链表每个节点向右移动 k 个位置，其中 k 是非负数。

示例 1:

输入: 1->2->3->4->5->NULL, k = 2

输出: 4->5->1->2->3->NULL

解释:

向右旋转 1 步: 5->1->2->3->4->NULL

向右旋转 2 步: 4->5->1->2->3->NULL

```
   ListNode* rotateRight(ListNode* head, int k) {
        if(!head) return head;
        auto f = head, s = head;
        int n = 0;
        while(f){n++,f = f->next;}
        k = k%n;
        f = head;
        while(k-- && f->next) f = f->next;
        while(f->next) s = s->next, f = f->next;
        f->next = head;
        head = s->next;
        s->next = NULL;
        return head;
    }
```

##

0x05 反转链表

反转从位置 m 到 n 的链表。请使用一趟扫描完成反转。

说明: 1 ≤ m ≤ n ≤ 链表长度。

示例:

输入: 1->2->3->4->5->NULL, m = 2, n = 4

输出: 1->4->3->2->5->NULL

```
    ListNode* reverseBetween(ListNode* head, int m, int n) {
        if(m==n) return head;
        auto L = new ListNode(0);
        L->next = head;
        auto a = L,d = L->next;
        for(int len = n-m;len > 0;len--) d = d->next;
        while(m-->1){
            a = a->next;
            d = d->next;
        }
        auto c = a->next, b = d->next;
        for(auto p = c, q = c->next; q!=b;){
            auto r = q->next;
            q->next = p;
            p = q;
            q = r;
        }
        a->next = d;
        c->next = b;
        return L->next;

    }
```

##

0x06 有序链表转换二叉搜索树

给定一个单链表，其中的元素按升序排序，转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

```
    vector<int> vec;
    int mid;
    TreeNode* sortedListToBST(ListNode* head) {
        vec.clear();
        while(head){
            vec.push_back(head->val);
            head=head->next;
        }
       return buildTree(0,vec.size()-1);
```

```
    }
    TreeNode*buildTree(int l,int r){
       if(l>r) return NULL;
       int mid=(l+r)/2;
       TreeNode *p=new TreeNode(vec[mid]);
       p->left=buildTree(l,mid-1);
       p->right=buildTree(mid+1,r);
       return p;
     }
```

##

0x07 环形链表

给定一个链表，返回链表开始入环的第一个节点.如果链表无环，则返回 null。

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

##

0x08 链表的插入排序

输入: 4->2->1->3

输出: 1->2->3->4

```
    ListNode* insertionSortList(ListNode* head) {
        auto h = new ListNode(-1),pre = h,q=head;
        for(auto p = head; p; p=q){//把head的每个节点p插入到h链中
            for(pre = h; pre->next&&p->val>pre->next->val;pre = pre->next){}//找插入点
            q = p->next,p->next = pre->next, pre->next = p;//插入
        }
        return h->next;
    }
```

##

0x09 链表的归并排序

```
    ListNode *sortList(ListNode *head){
        if (!head || !head->next) return head;
        ListNode *s = head, *f = head->next;
        //找到链表的中间位置
        while (f&&f->next){        //快慢指针，注意必须前两步存在
            s = s->next;
            f = f->next->next;
        }
        ListNode *l1 = sortList(s->next);        //右链表
        s->next = NULL;        //将其断开，为两个链表
        ListNode *l2 = sortList(head);
        return merge(l1, l2);
    }
    ListNode *merge(ListNode *l1, ListNode *l2){
        auto h = new ListNode(0), r = h;//r为尾结点
        while(l1&&l2){
            if(l1->val < l2->val) r->next = l1,l1 = l1->next,r = r->next;//把l1连到尾部
            else r->next = l2,l2 = l2->next,r = r->next;
        }
        if(!l1) r->next = l2;
        if(!l2) r->next = l1;
        return h->next;
    }
```

##

0x0a 多项式加法和乘法

```
#include<bits/stdc++.h>
using namespace std;
typedef struct ListNode{
    int val,ex;//系数和指数
    ListNode *next;
    ListNode(int v, int e) : val(v),ex(e),next(NULL){ }
};
void display(ListNode *h){
    ListNode *p = h->next;
    if(!p){
        printf("0 0");//零多项式
        return;
    }
    else printf("%d %d",p->val,p->ex),p = p->next;//第一项单独输出
    while(p) printf(" %d %d",p->val,p->ex),p = p->next;
}
ListNode* add(ListNode *h1, ListNode *h2){
    ListNode *h = new ListNode(-1,-1), *r = h,*p = h1->next,*q = h2->next;
    while(p&&q){
        if(p->ex > q->ex) r->next = new ListNode(p->val,p->ex),r = r->next, p = p->next;
```

```
        else if(p->ex < q->ex) r->next = new ListNode(q->val,q->ex),r = r->next, q = q->next;
        else{
            if(q->val+p->val==0){//抵消时
                p = p->next,q = q->next;
                continue;
            }
            r->next = new ListNode(q->val+p->val,q->ex);
            r = r->next, p = p->next,q = q->next;
        }
    }
    if(!p) r->next = q;
    if(!q) r->next = p;
    return h;
}
ListNode* mult(ListNode *h1, ListNode *h2){
    ListNode *h = new ListNode(-1,-1);
    for(ListNode *p = h1->next; p; p = p->next){
        for(ListNode *q = h2->next; q; q = q->next){
            int val = p->val*q->val, ex = p->ex+q->ex;//一项乘积的值
            ListNode *pre = h;
            for(; pre->next&&pre->next->ex > ex; pre = pre->next){};//找到插入位置
            if(pre->next&&pre->next->ex==ex){
                pre->next->val += val;//指数相同,合并同类项
                if(pre->next->val==0){//合并后系数为零，则需要删除
                    ListNode *t = pre->next;
                    pre->next = pre->next->next;
                    delete t;
                }
            }
            else {
                ListNode *d = new ListNode(val,ex);//乘积项节点 待插入
                d->next = pre->next, pre->next = d;
            }
        }
    }
    return h;
}
int main(){
    ListNode *h1 = new ListNode(-1,-1), *h2 = new ListNode(-1,-1), *r1 = h1, *r2 = h2;//建立两个表达式的头结点
    int n, m, v, e;
    scanf("%d",&n);
    while(n--){//尾插法建立表达式链表
        scanf("%d %d",&v,&e);
        r1->next = new ListNode(v,e);
        r1 = r1->next;
    }
    scanf("%d",&m);
    while(m--){
        scanf("%d %d",&v,&e);
        r2->next = new ListNode(v,e);
        r2 = r2->next;
    }
    display(mult(h1,h2));
    printf("\n");
    display(add(h1,h2));
    return 0;
}
```

##

0x0b 相交链表

找到两个单链表的交点

A和B两个链表长度可能不同，但是A+B和B+A的长度是相同的，所以遍历A+B和遍历B+A一定是同时结束。如果相交的话，A和B有一段尾巴是相同的，所以两个指针一定会同时到达交点。

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    auto pa = headA,pb = headB;
    while(pa!=pb){
        pa = (pa ? pa->next:headB);
        pb = (pb ? pb->next:headA);
    }
    return pa;
}
```

##

0x0c 删除排序链表中的重复元素

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

输入: 1->1->2->3->3

输出: 1->2->3

```
ListNode* deleteDuplicates(ListNode* head) {
    auto f = head;
    while(f){
        if(f->next&&f->val==f->next->val) f->next = f->next->next;
        else f = f->next;
    }
    return head;
}
```

##

0x0d 删除排序链表中的重复元素 II

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 没有重复出现 的数字。

输入: 1->2->3->3->4->4->5

输出: 1->2->5

```
    ListNode* deleteDuplicates(ListNode* head) {
        auto h = new ListNode(-1);//添加头结点
        h->next = head;
        auto pre = h,p = pre->next;//pre指向无重复的最后一个数 p为遍历节点
        while(p){
            while(p->next&&p->val==p->next->val) p = p->next;//p与后不同时退出
            if(pre->next==p) pre = p, p = p->next;//表示p是无重复元素，更新无重复数
pre = p
            else p = p->next,pre->next = p;//p是重复元素,[pre->next,p]全重复 跳过这
段即可
        }
        return h->next;
    }
```

二、栈

0x00 最小栈

设计一个支持 push，pop，top 操作，并能在常数时间内检索到最小元素的栈。

push(x) -- 将元素 x 推入栈中。 pop() -- 删除栈顶的元素。 top() -- 获取栈顶元素。 getMin() -- 检索栈中的最小元素。

```
    stack<int> s,mins;//mins栈同步存储当前栈内元素的最小值
    MinStack() {

    }

    void push(int x) {
        s.push(x);
        if(mins.empty()) mins.push(x);
        else mins.push(min(mins.top(),x));
    }

    void pop() {
        s.pop();
        mins.pop();
    }

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

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

##

0x01 有效的括号

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

```
    bool isValid(string s) {
        stack<char> stk;
        for(auto v : s){
            if(v=='('||v=='{'||v=='[') stk.push(v);//是左括号则直接入栈
            else if(stk.empty()) return false;//只有右括号没有左括号时显然不匹配
            else{
                int x = stk.top();
                stk.pop();
                if(v==')'&&x!='('||v=='}'&&x!='{'||v==']'&&x!='[') return false;//
不匹配情况
            }
        }
        return stk.empty();//完整的匹配最后栈应该为空
    }
```

##

0x02 栈序列合法性

给定 pushed-进栈顺序 和 popped-给定的出栈顺序 两个序列，每个序列中的值都不重复，判断给出的出栈序列是否合法。

示例 1：

输入：pushed = [1,2,3,4,5], popped = [4,5,3,2,1]

输出：true 解释：我们可以按以下顺序执行：

push(1), push(2), push(3), push(4), pop() -> 4,

push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1

示例 2：

输入：pushed = [1,2,3,4,5], popped = [4,3,5,1,2]

输出：false

解释：1 不能在 2 之前弹出。

```
    bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
        stack<int> s;
        int n = pushed.size();
        for(int i = 0,j=0; i < n; i++){//模拟进栈
            s.push(pushed[i]);//先直接进栈
            //然后根据出栈序列决定是否出栈
            while(!s.empty()&&j<n&&s.top()==popped[j]) s.pop(),j++;//出栈条件
        }
        return s.empty() ? true:false;//若合法，则此时栈一定是空的
    }
```

##

0x03 单调栈

给定一个长度为N的整数数列，输出每个数左边第一个比它小的数，如果不存在则输出-1。

输入样例： 5 3 4 2 7 5

输出样例： -1 3 -1 2 2

```
#include<bits/stdc++.h>
using namespace std;
const int N = 100010;
int stk[N],tt,n,x;
int main(){
    scanf("%d",&n);
    for(int i = 0; i < n; i++){
        scanf("%d",&x);
        while(tt&&x<=stk[tt])tt--;
        //如果发现新来的x比栈顶元素更小 那么对于之后的数来说 栈内大于x的数便没有价值了
        //因为如果之后的数存在左侧更小值,那么会首先选择x,所以栈内大于x的没有价值了
        //
        if(tt==0) printf("-1 ");
        else printf("%d ",stk[tt]);
        stk[++tt] = x;
    }
    return 0;
}
```

##

0x04 逆波兰表达式求值

根据逆波兰表示法，求表达式的值。

有效的运算符包括 +, -, *, / 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

整数除法只保留整数部分。 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

输入: ["10", "6", "9", "3", "+", "-11", "", "/", "", "17", "+", "5", "+"]

输出: 22

解释:

((10 *(6 / ((9 + 3)* -11))) + 17) + 5

= ((10 *(6 / (12* -11))) + 17) + 5

= ((10 * (6 / -132)) + 17) + 5

= ((10 * 0) + 17) + 5

= (0 + 17) + 5

= 17 + 5

= 22

```
    int getv(string &s){
        int ans = 0;
        if(s[0]=='-'){
            for(int i = 1; i < s.size(); i++) ans = ans*10+s[i]-'0';
            return -ans;
        }
        else{
            for(auto v:s) ans = ans*10+v-'0';
            return ans;
        }
    }
    int calc(int a, int b, char op){
        if(op=='+') return a+b;
        else if(op=='-') return a-b;
        else if(op=='*') return a*b;
        else return a/b;
    }
    int evalRPN(vector<string>& t) {
        stack<int> stk;
        for(auto s : t){
            if(s[0]>='0'&&s[0]<='9'||s[0]=='-'&&s.size()>1) stk.push(getv(s));//读入数字 负数和减号需要特判
            else {//是运算符，则从栈顶弹出两个操作数 进行运算
                int b = stk.top();
                stk.pop();
                int a = stk.top();
                stk.pop();
                stk.push(calc(a,b,s[0]));
            }
        }
        return stk.top();
    }
```

##

0x05 表达式计算

给出一个表达式,其中运算符仅包含+,-,*,/,^（加 减 乘 整除 乘方）要求求出表达式的最终值。

数据可能会出现括号情况，还有可能出现多余括号情况。

数据保证不会出现大于或等于2^31的答案。

数据可能会出现负数情况。

输入格式 输入仅一行，即为表达式。

输出格式 输出仅一行，既为表达式算出的结果。

输入样例： (2+2)^(1+1)

输出样例： 16

```
#include<bits/stdc++.h>
using namespace std;
stack<char> ops;//运算符栈
```

```cpp
stack<int> nums;//运算数栈
int quick_mi(int a, int b){//快速幂
    int t = a,ans = 1;
    while(b){
        if(b&1) ans*=t;
        t = t*t;
        b>>=1;
    }
    return ans;
}
void calc(){//执行一次计算
    int b = nums.top();
    nums.pop();
    int a = nums.top();
    nums.pop();
    char c = ops.top();
    ops.pop();
    int d;//结果
    if(c=='+') d = a + b;
    else if(c=='-') d = a - b;
    else if(c=='*') d = a * b;
    else if(c=='/') d = a / b;
    else d = quick_mi(a,b);
    nums.push(d);
}
int main(){
    string str,left;
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin>>str;
    for(int i = 0; i < str.size(); i++) left += '(';
    str  = left+str+')';//在最左侧添加左括号，防止右括号过多
    for(int i = 0; i < str.size(); i++){
        if(str[i]>='0'&&str[i]<='9'){//读取正数
            int j = i,ta = 0;
            while(str[j]>='0'&&str[j]<='9') ta = ta*10+str[j]-'0',j++;//获得该数的值
            i = j-1;
            nums.push(ta);
        }
        else if(str[i]=='-'&&i&&!(str[i-1]>='0'&&str[i-1]<='9')&&str[i-1]!=')'){//读取负数 负号的判定，负号前如果是数字，则是减号,反之即可
            int j = i+1,ta = 0;
            while(str[j]>='0'&&str[j]<='9') ta = ta*10+str[j]-'0',j++;//获得该数的值
            i = j-1;
            nums.push(-ta);
        }
        else if(str[i]=='-'||str[i]=='+'){//+,-优先级最低 前面的可以先算了
            while(ops.top()!='(') calc();
            ops.push(str[i]);
        }
        else if((str[i]=='*'||str[i]=='/')){// * /时
            while(ops.top()=='*'||ops.top()=='/'||ops.top()=='^') calc();//前方可以算的条件
            ops.push(str[i]);
        }
        else if(str[i]=='^'){
            while(ops.top()=='^') calc();
            ops.push(str[i]);
        }
```

```
        else if(str[i]=='(') ops.push(str[i]);//左括号直接进
        else if(str[i]==')'){//括号内的此时一定是优先级递增 可以把括号里的都算了
            while(ops.top()!='(') calc();
            ops.pop();
        }
    }
    cout<<nums.top();
    return 0;
}
```

##

0x06 a进制数转为b进制数

编写一个程序，可以实现将一个数字由一个进制转换为另一个进制。

这里有62个不同数位{0-9,A-Z,a-z}

输入

62 2 abcdefghiz

输出

62 abcdefghiz

2 1101110000010001011111001001011001111100100110001101001000 1

解释: 即把62进制的数 转为 2进制 并输出原数和转化后的数

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n,a,b;
    cin>>n;
    while(n--){
        string a_line,b_line;//a进制的a_line转为b进制的b_line
        vector<int> nums,res;//存储a_line代表的十进制数字,存储b_line代表的十进制数字
        cin >> a >> b >> a_line;
        for(auto v : a_line){//获得每一位的十进制真值
            if(v>='0'&&v<='9') nums.push_back(v-'0');
            else if(v>='A'&&v<='Z') nums.push_back(v-'A'+10);
            else nums.push_back(v-'a'+36);
        }
        reverse(nums.begin(),nums.end());//反转，从低位开始,易于处理高位的进位和删除
        while(nums.size()){//商不为零时,模拟短除法 直接将a进制转化为b进制
            int r = 0;//余数
            for(int i = nums.size()-1; i >=0 ; i--){
                nums[i] += r*a;//计算当前位的真值(相当于该位的十进制真值)
                r = nums[i] % b;//当前位除b后的余数
                nums[i]/=b;//当前位的商
            }
            res.push_back(r);//余数
            while(nums.size()&&nums.back()==0) nums.pop_back();//去除除法后高位的前导零
        }
        reverse(res.begin(),res.end());
        for(v:res){//十进制数组转化为字符表达
```

```
            if(v<10) b_line.push_back(v+'0');
            else if(v>=10&&v<36) b_line.push_back(v-10+'A');
            else b_line.push_back(v-36+'a');
        }
        cout<<a<<" "<<a_line<<endl;
        cout<<b<<" "<<b_line<<endl;
        cout<<endl;
    }
}
```

三、队列

##

0x00 设计循环队列

设计你的循环队列实现。 循环队列是一种线性数据结构，其操作表现基于 FIFO（先进先出）原则并且队尾被连接在队首之后以形成一个循环。它也被称为“环形缓冲器”。

循环队列的一个好处是我们可以利用这个队列之前用过的空间。在一个普通队列里，一旦一个队列满了，我们就不能插入下一个元素，即使在队列前面仍有空间。但是使用循环队列，我们能使用这些空间去存储新的值。

```
    vector<int> data;
    int len,front,rear;
    MyCircularQueue(int k) {
        len = k+1;
        data = vector<int>(len);
        front = 0,rear = 0; //front指向队头 rear 指向队尾的下一个位置
    }

    /** Insert an element into the circular queue. Return true if the operation is successful. */
    bool enQueue(int value) {
        if((rear+1)%len==front) return false;//此时队列无法插入
        data[rear] = value,rear = (rear+1)%len;
        return true;
    }

    /** Delete an element from the circular queue. Return true if the operation is successful. */
    bool deQueue() {
        if(rear==front) return false;
        front = (front+1)%len;
        return true;
    }

    /** Get the front item from the queue. */
    int Front() {
        if(rear==front) return -1;
        return data[front];
    }
```

```
28
29      /** Get the last item from the queue. */
30      int Rear() {
31          if(rear==front) return -1;
32          return data[(rear-1+len)%len];
33      }
34
35      /** Checks whether the circular queue is empty or not. */
36      bool isEmpty() {
37          return rear==front;
38      }
39
40      /** Checks whether the circular queue is full or not. */
41      bool isFull() {
42          return (rear+1)%len==front;
43      }
```

##

0x01 单调队列-滑动窗口

有一个大小为k的滑动窗口，它从数组的最左边移动到最右边。

您只能在窗口中看到k个数字。

每次滑动窗口向右移动一个位置

您的任务是确定滑动窗口位于每个位置时，窗口中的最大值和最小值。

输入格式

输入包含两行。

第一行包含两个整数n和k，分别代表数组长度和滑动窗口的长度。

第二行有n个整数，代表数组的具体数值。

同行数据之间用空格隔开。

输出格式

输出包含两个。

第一行输出，从左至右，每个位置滑动窗口中的最小值。

第二行输出，从左至右，每个位置滑动窗口中的最大值。

输入样例：

8 3

1 3 -1 -3 5 3 6 7

输出样例：

-1 -3 -3 -3 3 3

3 3 5 5 6 7

```
1   #include<bits/stdc++.h>
```

```cpp
using namespace std;
const int N = 1000010;
int a[N];
int q[N],hh,tt=-1;//队列 队头 队尾 注意队列中存的是所代表的数的下标 为了判断是否超出滑动窗口左侧
int main(){
    int n,k;//l,r表示窗口范围表示窗口大小
    scanf("%d %d",&n,&k);
    for(int i = 0; i < n; i++) scanf("%d",&a[i]);
    for(int i = 0; i < n; i++){
       while(hh<=tt && a[i]<=a[q[tt]]) tt--;//如果发现有更小的数入队,那么之前比它大的数就没用了,因为之前的数会先出队,有小的撑着就行
       q[++tt] = i;//入队
       if(hh<=tt&&q[hh]<i-k+1) hh++;//如果发现当前最小值出界了(右边界i-窗口长度+1),那么就出队
       if(i>=k-1) printf("%d ",a[q[hh]]);//当窗口内有k个元素就输出答案了
    }
    puts("");
    hh = 0, tt = -1;
    for(int i = 0; i < n; i++){
       while(hh<=tt && a[i]>=a[q[tt]]) tt--;//去除之前的更小的数即可,留大的撑着
       q[++tt] = i;//入队
       if(hh<=tt&&q[hh]<i-k+1) hh++;
       if(i>=k-1) printf("%d ",a[q[hh]]);
    }
    return 0;
}
```

四、二叉树

##

0x00 二叉树的前序遍历

给定一个二叉树，返回它的前序遍历序列。

```cpp
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> ans;
        stack<TreeNode*> stk;
        auto p = root;//p为遍历点
        while(p||!stk.empty()){
            while(p){//只把左侧链全压入
                ans.push_back(p->val);//根
                stk.push(p);
                p = p->left;
            }
            p = stk.top(),stk.pop();
            p = p->right;
        }
        return ans;
    }
```

##

0x01 二叉树的中序遍历

```
vector<int> inorderTraversal(TreeNode* root) {
    vector<int> ans;
    auto p = root;//p为遍历点
    stack<TreeNode* > stk;
    while(p || !s.empty()){
        while(p){只把左侧链全压入
            stk.push(p);
            p = p->left;
        }
        p = s.top(),stk.pop();
        ans.push_back(p->val);//根
        p = p->right;//进入右子树
    }
    return ans;
}
```

##

0x02 二叉树的后序遍历

给定一个二叉树，返回它的 后序 遍历。

版本一: 由前序推后序

```
//考虑前序 根左右 想要得到后序 左右根 应该怎么做呢
//首先可以把前序调整一下 根右左 然后逆序即可得到 左右根 即为后序遍历结果
vector<int> postorderTraversal(TreeNode* root) {
    vector<int> ans;
    stack<TreeNode* > stk;
    auto p = root;//遍历点
    while(p||!stk.empty()){//根右左的前序遍历
        while(p){
            ans.push_back(p->val);
            stk.push(p->left);//此处与前序相反 变为右左
            p = p->right;
        }
        p = stk.top();
        stk.pop();
    }
    reverse(ans.begin(),ans.end());//结果逆序即可
    return ans;
}
```

版本二: 直接法，增设最近访问节点,用于判断是从左还是右返回的

```
vector<int> postorderTraversal(TreeNode* root) {
    vector<int> ans;
    stack<TreeNode* > stk;
    TreeNode *p = root,*vised = NULL;//遍历点 最近访问点
```

```
        while(p||!stk.empty()){
            while(p){//左侧链全部压入
                stk.push(p);
                p = p -> left;
            }
            p = stk.top();
            if(p->right&&p->right!=vised){//说明p的右边还未访问 需要进入右子树遍历
                p = p->right;
            }
            else {//说明p的右子树访问完毕了 可以输出根p了
                ans.push_back(p->val);//根
                vised = p;
                stk.pop();//该点访问完毕 弹出
                p = NULL;//下次的p将从栈中取得
            }
        }
        return ans;
    }
```

##

0x03 二叉树的层序遍历

给定一个二叉树，返回其按层次遍历的节点值（即逐层地，从左到右访问所有节点）。

```
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if(!root) return ans;
        queue<TreeNode* > q;
        q.push(root);
        while(!q.empty()){
            vector<int> levelAns;//保存一层的节点
            int len = q.size();//该层节点数量
            while(len--){
                auto p = q.front();
                q.pop();
                if(p->left) q.push(p->left);
                if(p->right) q.push(p->right);
                levelAns.push_back(p->val);
            }
            ans.push_back(levelAns);
        }
        return ans;
    }
```

##

0x04 从前序与中序遍历序列构造二叉树

根据一棵树的前序遍历与中序遍历构造二叉树。

注意: 你可以假设树中没有重复的元素。

例如，给出

前序遍历 preorder = [3,9,20,15,7]

中序遍历 inorder = [9,3,15,20,7]

返回对应的二叉树。

```
    unordered_map<int,int> pos;//哈希表 快速查找值的下标
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        int n = inorder.size();
        for(int i = 0; i < n; i++) pos[inorder[i]] = i;//记录值的下标
        return dfs(preorder,inorder,0,n-1,0,n-1);
    }
    TreeNode* dfs(vector<int>& preorder, vector<int>& inorder, int pl, int pr, int il, int ir){//前序序列的范围 中序序列的范围
        if(pl>pr) return NULL;
        int val = preorder[pl];
        auto rt = new TreeNode(val);
        int k = pos[preorder[pl]];//根节点在中序中的位置
        int len =  k - il;//左子树长度
        rt->left = dfs(preorder, inorder, pl+1, pl+len, il,k-1);
        rt->right = dfs(preorder,inorder, pl+len+1,pr,k+1,ir);
        return rt;
    }
```

##

0x05 从层序和中序遍历序列构造二叉树

根据一棵树的层序遍历与中序遍历构造二叉树。

注意: 你可以假设树中没有重复的元素。

例如，给出

层序遍历 preorder = [3,9,20,15,7]

中序遍历 inorder = [9,3,15,20,7]

返回对应的二叉树。

```
    int n;//序列长度
    unordered_map<int,int> pos;//哈希表 快速查找值的下标
    unordered_map<int, bool> vis;//值是否加入树中
    TreeNode* buildTree(vector<int>& lev, vector<int>& in) {
        n = in.size();
        vector<bool> vis(n);//标记层序节点是否用过
        for(int i = 0; i < n; i++) pos[in[i]] = i;//记录值的下标
        return dfs(lev,in,vis,0,n-1,0);
    }
    TreeNode* dfs(vector<int>& lev, vector<int>& in,vector<bool>& vis, int il, int ir,int levp){
                                                        //il 和 ir为中序区间 levp是层序开始下标
        if(ir<il) return NULL;
        int ret,p,flag = 0;//根的值及其在中序中的位置 成功标志
        for(int i = levp; i < n; i++){//找当前的根 顺序往后遍历层序节点
            p = pos[lev[i]],ret = in[p];//查找层序节点在中序中的位置
            if(ret!=lev[i]||p<il||p>ir||vis[ret]) continue;//查找失败
```

```
            vis[ret] = true,flag = 1;//查找成功 标记 跳出
            break;
        }
        if(!flag) return NULL;//查找失败
        TreeNode* r = new TreeNode(ret);
        r->left = dfs(lev,in,vis,il,p-1,levp+1);
        r->right = dfs(lev,in,vis,p+1,ir,levp+1);
        return r;
    }
```

##

0x06 二叉树的宽度

即求节点数最多的那一层的节点个数

```
    int widthOfBinaryTree(TreeNode* root) {
        if(!root) return 0;
        int width = 0;
        queue<TreeNode* > q;
        q.push(root);
        while(!q.empty()){
            int len = q.size();//获得该层节点数量
            width = max(width,len);//更新最大宽度
            while(len--){
                auto p = q.front();
                q.pop();
                if(p->left) q.push(p->left);
                if(p->right) q.push(p->right);
            }
        }
        return width;
    }
```

##

0x07 二叉树的最近公共祖先

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

输出: 3

解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

```
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        //后续遍历思想
        if(!root||root==p||root==q) return root;//p,q节点在根部时
        //p,q节点在子树中时
        auto left = lowestCommonAncestor(root->left,p,q);//左子树的最近祖先
        auto right = lowestCommonAncestor(root->right,p,q);//右
        if(!left){//左子树中不包含p和q，只能是右的返回值
            return right;
        }
        if(!right) return left;//同理
        return root;//此时说明p,q在左右两侧，root就是最近祖先
    }
```

##

0x08 对称二叉树

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的。

递归版

```
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        return dfs(root->left,root->right);
    }
    bool dfs(TreeNode* l, TreeNode* r){//判断左右子树l,r是否对称
        if(!l&&!r) return true;
        if(l&&!r||!l&&r) return false;
        if(l->val==r->val){
            return dfs(l->right,r->left)&&dfs(l->left,r->right);
            //左子树的右和右子树的左相同，左子树的左和右子树的右相同时 则对称
        }
        return false;
    }
```

迭代版

```
    bool isSymmetric(TreeNode* root) {//非递归
        //对左子树采取左中右的中序遍历，对右子树采取右中左的中序遍历，在遍历过程中比较即可
        if(!root) return true;
        stack<TreeNode* > s1,s2;//遍历左子树的栈和遍历右边的栈
        auto p1 = root->left, p2 = root->right;
        while(p1||s1.size()||p2||s2.size()){
            while(p1&&p2){
                s1.push(p1),p1 = p1->left;
                s2.push(p2),p2 = p2->right;
            }
            if(p1||p2) return false;//此时两个本应该都为空的
            p1 = s1.top(), s1.pop();
            p2 = s2.top(), s2.pop();
            if(p1->val!=p2->val) return false;
```

```
            p1 = p1->right,p2 = p2->left;
        }
        return true;
    }
```

##

0x09 验证二叉搜索树

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。

版本一:中序遍历

```
long long pre = -1e10;//中序遍历过程中的前驱节点的值
bool isValidBST(TreeNode* root) {
    if(!root) return true;
    bool t = isValidBST(root->left);
    if(!t||pre>=root->val) return false;//左子树非搜索树或者左子树不小于根
    pre = root->val;//准备遍历右子树 则当前根作为右子树的前驱节点了
    return isValidBST(root->right);
}
```

版本二:限定节点值的范围

```
bool isValidBST(TreeNode* root) {
    return dfs(root,INT_MIN,INT_MAX);
}
//限定节点值的范围即可
bool dfs(TreeNode* p, long long minv, long long maxv){
    if(!p) return true;
    if(p->val<minv||p->val>maxv) return false;//不满足范围则一定不是
    return dfs(p->left,minv,p->val-1ll)&&dfs(p->right,p->val+1ll,maxv);
}
```

##

0x0a 验证平衡二叉树

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过1。

示例 1:

给定二叉树 [3,9,20,null,null,15,7] 则为true

示例 2:

给定二叉树 [1,2,2,3,3,null,null,4,4] 则为false

```
    bool isBalanced(TreeNode* root) {
        int h = 0;//树的高度
        return dfs(root,h);
    }
    bool dfs(TreeNode* p, int &h){//求出以p为根的树的高度并返回是否平衡
        if(!p) return true;
        int hl = 0, hr = 0;//求出左右子树的高度
        bool lbal = dfs(p->left,hl);
        bool rbal = dfs(p->right,hr);
        h = max(hl,hr)+1;//以p为根的树的高度
        return lbal && rbal && abs(hl-hr)<=1;//左右均平衡且高度差<=1
    }
```

##

0x0b n皇后

n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。(攻击范围为所在的行、列、正反对角线上)。

给定一个整数 n，返回 n 皇后不同的解决方案的数量。

```
    int n,ans;//n皇后
    void dfs(int u, auto &col,auto &gd, auto &rgd){
        if(u==n){//搜索完毕
            ans++;
            return;
        }
        for(int j = 0; j < n; j++){//枚举u行放在j列时
            if(!col[j]&&!gd[j-u+n]&&!rgd[u+j]){//列,对角,反对角不冲突时
                col[j] = gd[j-u+n] = rgd[u+j] = 1;//占用
                dfs(u+1,col,gd,rgd);//进入下一行搜索
                col[j] = gd[j-u+n] = rgd[u+j] = 0;//恢复现场
            }
        }
    }
    int totalNQueens(int len) {
        n = len;
        vector<bool> col(n),gd(2*n),rgd(2*n);//列 对角线 反对角线
        dfs(0,col,gd,rgd);//从第0行开始搜索
        return ans;
    }
```

##

0x0c 迷宫问题

给定一个 n×n 的二维数组，如下所示：

int maze[5][5] = {

0, 1, 0, 0, 0,

0, 1, 0, 1, 0,

0, 0, 0, 0, 0,

0, 1, 1, 1, 0,

0, 0, 0, 1, 0,

}; 它表示一个迷宫，其中的1表示墙壁，0表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的最短路线。

数据保证至少存在一条从左上角走到右下角的路径。

对于上图则输出如下路径：

0 0

1 0

2 0

2 1

2 2

2 3

2 4

3 4

4 4

```
#include<bits/stdc++.h>
using namespace std;
typedef pair<int,int> PII;
const int N = 1010;
PII pre[N][N];//保存搜狗所过程中上一步的坐标
bool g[N][N];
int n;
void bfs(PII start){
    int dx[]={-1,0,1,0}, dy[]={0,1,0,-1};//四个方向
    queue<PII> q;
    q.push(start);
    memset(pre,-1,sizeof pre);//为-1表示此处还没搜过
    pre[n-1][n-1] = {n-1,n-1};
    while(q.size()){
        PII t = q.front();//当前位置
        q.pop();
        int tx = t.first, ty = t.second;
        for(int i = 0; i < 4; i++){//向四个方向探索下一步
            int x = tx + dx[i], y = ty + dy[i];
            if(x<0||x>=n||y<0||y>=n||g[x][y]||pre[x][y].first!=-1) continue;
            q.push({x,y}),pre[x][y] = {tx,ty};//保存该步信息
        }
    }
    PII end({0,0});
    while(true){
        int x = end.first, y = end.second;
        printf("%d %d\n",x,y);
        if(x==n-1&&y==n-1) break;
```

```
        end = pre[x][y];
    }
}
int main(){
    scanf("%d",&n);
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++) scanf("%d",&g[i][j]);
    PII end({n-1,n-1});
    bfs(end);//逆着搜(从终点向起点搜) 方便输出路径
    return 0;
}
```

##

0x0d 树的重心

给定一颗树，树中包含n个结点（编号1~n）和n-1条无向边。

请你找到树的重心，并输出将重心删除后，剩余各个连通块中点数的最大值。

重心定义：重心是指树中的一个结点，如果将这个点删除后，剩余各个连通块中点数的最大值最小，那么这个节点被称为树的重心。

输出一个整数，表示删除重心后,所有的子树中最大的子树的结点数目。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 100010, M = N*2;//因为是无向图 需要存双向边
int h[N],e[M],ne[M],idx;//邻接表
int n;
bool st[N];
void add(int a, int b){//插入一条边a->b
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}
int ans = 0x3f3f3f3f;//
int dfs(int u){//返回以u为根的树的大小
    st[u] = 1;
    int sum = 1,res = 0;//sum-以u为根的子树大小 res-删除u后的连通块内点的个数最大值
    for(int i = h[u]; i != -1; i = ne[i]){//遍历u的所有邻接点
        int j = e[i];//对于u->j这条边
        if(!st[j]){
            int subs = dfs(j);//获得以j为根的子树大小
            res = max(res, subs);
            sum += subs;
        }
    }
    res = max(n-sum,res);//求出删除u后的最大连通块内节点个数
    ans = min(ans,res);//结果值
    return sum;
}
int main(){
    cin>>n;
    memset(h,-1,sizeof h);
    for(int i = 0 ,a,b; i < n-1; i++){
        cin>>a>>b;
        add(a,b),add(b,a);
    }
    dfs(1);
```

```
    cout<<ans<<endl;
    return 0;
}
```

五、图

##

0x00 有向图的拓扑排序

给定一个n个点m条边的有向图，请输出任意一个该有向图的拓扑序列，如果拓扑序列不存在，则输出-1。

若一个由图中所有点构成的序列A满足：对于图中的每条边(x, y)，x在A中都出现在y之前，则称A是该图的一个拓扑序列。

输入样例：

3 3（指3个点3条边）

1 2

2 3

1 3

输出样例：

1 2 3

版本一:bfs 不断删除入度为0的点

```
bool topsort(){
    int hh = 0, tt = -1;
    for(int i = 1; i <= n; i++){//将所有入度为0的入队
        if(!ind[i]) q[++tt] = i;//ind[i]表示节点i的入度
    }
    while(hh<=tt){
        int t = q[hh++];
        for(int i = h[t]; i!=-1; i = ne[i]){
            int j = e[i];
            ind[j]--;
            if(ind[j]==0) q[++tt] = j;
        }
    }
    return tt==n-1;//如果节点全部入队 说明无环
    //若返回true,则可输出拓扑序列，即入队顺序
    //for(int i = 0; i < n; i++) cout<<q[i]<<" ";
}
```

版本二:dfs 利用dfs序 按dfs序从大到小输出点

```cpp
int dfstime = 0;//dfs序(即dfs过程中 搜索结束的时间)
vector<pair<int,int> > ans;
int st[N];// -1 表示搜索完毕 1表示该轮dfs还没结束
bool dfs(int u){//给出dfs序以及返回是否有环
    if(st[u]==-1) return false;//对于已经搜索完毕了点 无需搜
    if(st[u]==1) return true;//因为该点在当前轮搜到过 而现在又来了 第二次到达 说明存在环
    st[u] = 1;//当前轮开始访问
    dfstime++;//时间+1 到达u
    for(int i = h[u]; i != -1; i = ne[i]){
        int j = e[i];
        if(dfs(j)) return true;//邻接点存在环
    }
    st[u] = -1;//以u为起点的dfs访问完毕
    ans.push_back({-dfstime,u});//记录该点的退出时间(dfs序) 变为负数 方便从大到小排序
    return false;//不存在环
}
bool topsort_dfs(){
    for(int i = 1; i <= n; i++){
        if(st[i]!=-1){
            if(dfs(i)) return false;//发现该连通分量有环 拓扑失败
        }
    }
    sort(ans.begin(),ans.end());
    for(auto v : ans){
        cout<<v.second<<" ";
    }
    return true;
}
```

##

0x01 Dijkstra求最短路

给定一个n个点m条边的有向图, 请你求出1号点到n号点的最短距离，如果无法从1号点走到n号点，则输出-1。

```cpp
int g[N][N];//邻接矩阵
int d[N];//距离数组
bool st[N];//标记数组
int n,m;
int dijkstra(){
    memset(d,0x3f,sizeof d);//初始时每个点到起点距离为无穷
    d[1] = 0;//1为起点
    for(int i = 0; i < n; i++){
        int t = -1;//最小值点
        for(int j = 1; j <= n; j++){//寻找未加入最短路的距离最小的点(此处可用堆优化找最小值)
            if(!st[j]&&(t==-1||d[t]>d[j])) t = j;
        }
        st[t] = true;
        for(int j = 1; j <= n; j++){//用新加入的点更新其它点
            if(!st[j]&&d[j] > d[t] + g[t][j]) d[j] = d[t]+g[t][j];
        }
    }
    if(d[n] == 0x3f3f3f3f) return -1;
    else return d[n];
```

```
}
```

##

0x02 最小生成树

给定一张边带权的无向图G=(V, E)，其中V表示图中点的集合，E表示图中边的集合，n=|V|，m=|E|。

由V中的全部n个顶点和E中n-1条边构成的无向连通子图被称为G的一棵生成树，其中边的权值之和最小的生成树被称为无向图G的最小生成树。输出最小生成树的权重之和

版本一: prim算法

```
const int INF = 0x3f3f3f3f;
int n,m;
int g[N][N];
int d[N];//每个点距离生成树的距离
bool st[N];
int prim(){
    memset(d,0x3f,sizeof d);
    int res = 0;//最小生成树的权重之和
    for(int i = 0; i < n; i++){
        int t = -1;
        for(int j = 1; j <= n; j++)
            if(!st[j]&&(t==-1||d[t] > d[j])) t = j;
        if(i&&d[t]==INF) return -1;//说明不联通,则不存在
        st[t] = true;
        if(i) res += d[t];//纳入该边权重，除了起点
        for(int j = 1; j <= n; j++){
            if(!st[j]&&d[j] > g[t][j]) d[j] = g[t][j];
        }
    }
    return res;
}
```

版本二: Kruskal算法求最小生成树

```
struct Edge{//边集
    int a,b,w;
    bool operator < (const Edge &E) const{//按权重从小到大排序
        return w < E.w;
    }
}e[N];
int n,m;
int p[N/2];//并查集数组 若p[i]=j 则表示i的父节点为j
int find(int x){//并查集 寻找x的根
    if(x!=p[x]) p[x] = find(p[x]);//路径压缩版
    return p[x];
}
int kruskal(){
    sort(e,e+m);//把边按权重从小到大排序
    int res = 0,cnt = 0;//最小生成树的权重之和，选择的边数
    for(int i = 1; i <= n; i++) p[i] = i;//并查集初始化
    for(int i = 0; i < m; i++){//从小到大选边
        int x = find(e[i].a), y = find(e[i].b);
        if(x!=y){//若该条边不构成回路 则加入
```

```
            p[x] = y;
            cnt++;
            res += e[i].w;//纳入该边
        }
    }
    return cnt==n-1 ? res:0x3f3f3f3f;//最终要选够n-1条边
}
```

##

0x03 floyd求最短路

即求出所有点对的最短距离

```
int d[N][N];//任意两点间的最短距离
int n,m;
void floyd(){
    for(int k = 1; k <= n; k++){
        for(int i = 1; i <= n; i++){
            for(int j= 1; j <= n; j++){
                d[i][j] = min(d[i][j],d[i][k] + d[k][j]);
            }
        }
    }
}
```

六、查找

##

0x00 kmp字符串

给定一个模式串S，以及一个模板串P，所有字符串中只包含大小写英文字母以及阿拉伯数字。

模板串P在模式串S中多次作为子串出现。

求出模板串P在模式串S中所有出现的位置的起始下标。

输入样例：

3

aba(模板串P)

5

ababa(模式串S)

输出样例：

0 2(输出P在S中出现的起始下标)

```
#include<bits/stdc++.h>
using namespace std;
const int N = 10010, M = 100010;
char p[N], s[M];
int n,m;
int ne[N];//next数组 匹配失败时 模板串回退的位置
int main(){
    cin>>n>>p+1>>m>>s+1;//让下标从1开始
    for(int i = 2, j = 0; i <= n; i++){//求出next数组
        while(j&&p[j+1]!=p[i]) j = ne[j];//匹配j+1和i,若当前不匹配时 回退寻找匹配的位置
        if(p[j+1] == p[i]) j++;//匹配 往前移动
        ne[i] = j;//如果起点就不匹配 就是零 否则 就是当前匹配了的长度
    }
    for(int i = 1, j = 0; i <= m; i++){//匹配过程
        while(j&&p[j+1]!=s[i]) j = ne[j];
        if(p[j+1]==s[i]) j++;
        if(j==n){
            cout<<i - n<<" ";//匹配成功 输出下标
            j = ne[j];//为继续匹配，在成功为强行失败即可
        }
    }
    return 0;
}
```

##

0x01最短回文串

给定一个字符串 s，你可以通过在字符串前面添加字符将其转换为回文串。找到并返回可以用这种方式转换的最短回文串。

示例 1:

输入: "aacecaaa" 输出: "aaacecaaa"

示例 2:

输入: "abcd" 输出: "dcbabcd"

```
/*思路  如对于串 abcd 想要将其变为回文串
      那么先把它逆序 然后放在前面 自然是回文了
                                  abcd
                              dcba
                              dcbaabcd ->是回文
      但是我们发现根本没必要放这么多在前面 因为abcd的前缀和dcab的后缀有重合(如a) 所以为了只添加最少      的字符,我们在前方只需要添加不重复的即可
                                   abcd
                               dcba
                               dcbabcd ->依然是回文
     //为了添加的最少 我们就需要找到dcba的后缀和abcd的前缀重合的部分，且让重合部分最大即可
     //故而联想到kmp算法，它的next数组就是用来求一个串的前缀和后缀相同的长度的最大值
     //所以拼接起字符串 abcddcba 但是我们所求的前缀是不能超过中点的，因此用一个特殊字符隔开
```

```
13      //              即为 abcd#dcba 这样在匹配前后缀时，相同长度就一定不会超过#号了
14      //              这样问题就转化为了 求abcd#dcba的next数组 知道该串的最长前后缀相同时的
    最大长度为1
15                                          a       a 前后缀相同的最大长度
16                                          所以把后半部分除去重叠的部分拼接到前半部分即可
17                                  答案就是  dcbabcd
18                                          大功告成!
19
20      */
21      string shortestPalindrome(string s) {
22          string revs = s;
23          int tn = s.size();//终点处,#前面的位置
24          reverse(revs.begin(),revs.end());
25          s = ' '+ s + '#' + revs;//让下标从1开始
26          int n = s.size()-1;//实际长度
27          vector<int> ne(n+1);//next数组
28          for(int i = 2, j = 0; i <= n; i++){//求next数组
29              while(j&&s[i]!=s[j+1]) j = ne[j];
30              if(s[i]==s[j+1]) j++;
31              ne[i] = j;
32          }
33          return s.substr(tn+2,tn-ne[n])+s.substr(1,tn);//后半部分除去重叠后缀+前半部
    分
34      }
```

七、排序

##

0x00 直接插入排序

给定一个整数数组 nums，将该数组升序排列。

以第一个元素作为有序数组，其后的元素通过在这个已有序的数组中找到合适的位置并插入

空间：O(1)

最好：O(n)，初始为有序时

最坏：O(n^2)，初始为是逆序时

平均：O(n^2)

稳定性：是

```cpp
vector<int> sortArray(vector<int>& nums) {
    int n = nums.size();
    for(int i = 1,j; i < n; i++){//此时0~i-1已有序
        int v = nums[i];//i位置元素待插入
        for(j = i-1; j>=0&&v<nums[j]; j--){//向前寻找插入位置
            nums[j+1]=nums[j];
        }
        nums[j+1] = v;//插入
    }
    return nums;
}
```

##

0x01 折半插入排序

给定一个整数数组 nums，将该数组升序排列。

与直接插入排序的唯一区别就是 查找插入位置时 使用二分，复杂度不变

```cpp
vector<int> sortArray(vector<int>& nums) {
    int n = nums.size();
    for(int i = 1; i < n; i++){//此时0~i-1已有序
        int v = nums[i];//i位置元素待插入

        int l = 0, r = i-1;//二分查找插入位置
        while(l<r){
            int mid = l+r+1 >> 1;
            if(nums[mid]<=v) l = mid;
            else r = mid-1;
        }
        if(nums[l]>v) l--;//防止在单个元素中二分的情况
        //此时的l即为插入位置的前一个位置

        for(int j = i-1; j > l; j--) nums[j+1] = nums[j];//后移
        nums[l+1] = v;//插入
    }
    return nums;
}
```

##

0x02 希尔排序

给定一个整数数组 nums，将该数组升序排列。

类插入排序，只是向前移动的步数变成d，插入排序每次都只是向前移动1。

空间：O(1)

平均：当n在特定范围时为 O(n^1.3)

稳定性：否

```cpp
    vector<int> sortArray(vector<int>& nums) {
        int n = nums.size();
        for (int d = n / 2; d >= 1; d /= 2) {//d为增量 d=1时就是0x00的插入排序
            for (int i = d,j; i < n; i++) {
                int v = nums[i];//i位置元素待插入
                for (j = i-d; j >= 0 && v < nums[j]; j -= d) {//以增量d向前寻找插入位置
                    nums[j+d] = nums[j];
                }
                nums[j+d] = v;
            }
        }
        return nums;
    }
```

##

0x03 冒泡排序

给定一个整数数组 nums，将该数组升序排列。

通过相邻元素的比较和交换，使得每一趟循环都能找到未有序数组的最大值

空间：O(1)

最好：O(n)，初始即有序时 一趟冒泡即可

最坏：O(n^2)， 初始为逆序时

平均：O(n^2)

稳定性：是

```cpp
    vector<int> sortArray(vector<int>& nums) {
        int n = nums.size();
        bool sorted = false;//无序
        for(int i = n-1; i >=0 && !sorted; i--){//0~i为无序区
            sorted = true;//假定已有序
            for(int j = 0; j < i; j++){//相邻比较 大的沉底
                if(nums[j]>nums[j+1]) swap(nums[j],nums[j+1]),sorted = false;//发生交换 则无序
            }
        }
        return nums;
    }
```

##

0x04 快速排序

给定一个整数数组 nums，将该数组升序排列。

选择一个元素作为基数（通常是第一个元素），把比基数小的元素放到它左边，比基数大的元素放到它右边（相当于二分），再不断递归基数左右两边的序列。

空间：O(log n)，即递归栈深度

最好：O(nlog n)，其他的数均匀分布在基数的两边，此时的递归就是不断地二分左右序列。

最坏：O(n^2)，其他的数都分布在基数的一边，此时要划分n次了，每次O(n)

平均：O(nlog n)

稳定性：否

```
    void quick_sort(vector<int>& nums, int l , int r){//对下标为 l~r部分 排序
        if(l >= r ) return ;
        int x = nums[l], i = l-1, j = r+1;//x为划分中枢 i为左半起点 j为右半起点
        while(i < j){
            while(nums[++i] < x);//左边寻大于x的数
            while(nums[--j] > x);//右边寻小于x的数
            if(i < j) swap(nums[i],nums[j]);//每次把左边大于x的数和右边小于x的交换即可
        }
        quick_sort(nums,l,j),quick_sort(nums,j+1,r);
        //因为结束时i在j的右边 j是左半段的终点,i是右半段的起点
    }
    vector<int> sortArray(vector<int>& nums) {
        int n = nums.size();
        quick_sort(nums,0,n-1);
        return nums;
    }
```

##

0x05 快排应用-第k大数

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

输入: [3,2,1,5,6,4] 和 k = 2

输出: 5

```
    int quick_select(vector<int>& nums, int l , int r, int k){
                                    //在下标为l~r的数中求第k大的数
        if(l >= r) return nums[l];
        int x = nums[l], i = l-1, j = r+1;
        while(i < j){//以x为枢纽 一次快排划分 此处大的在左边 小的在右边
            while(nums[++i] > x);
            while(nums[--j] < x);
            if(i < j) swap(nums[i],nums[j]);
        }
        int sl = j-l+1;//左半区间的长度
        if(sl>=k) return quick_select(nums, l,j,k);
        //k<=左半区间长度,则第k大数必然在左半段 并且是左半段的第k大数
        else return quick_select(nums, j+1, r, k - sl);
        //第k大数必然在右半段 并且是右半段的第k-sl大数
    }
    int findKthLargest(vector<int>& nums, int k) {
        int n = nums.size();
```

```
        return quick_select(nums,0,n-1,k);
    }
```

##

0x06 选择排序

给定一个整数数组 nums，将该数组升序排列。

和冒泡排序相似，区别在于选择排序是直接挑出未排序元素的最大值放后面，其它元素不动。无论如何都要O(n^2)

最好：O(n^2)

最坏：O(n^2)

平均：O(n^2)

稳定性：否

```
    vector<int> sortArray(vector<int>& nums) {
        int n = nums.size();
        for(int i = n-1; i >=0; i--){//0~i为无序部分
            int minpos = -1;
            for(int j = 0; j <= i; j++){//寻找0~i区间内的最大值的位置
                if(minpos ==-1||nums[j]>nums[minpos]) minpos = j;
            }
            swap(nums[i],nums[minpos]);//把挑出最大值放到后面
        }
        return nums;
    }
```

##

0x07 堆排序

输入一个长度为n的整数数列，从小到大输出前m小的数。

输入格式 第一行包含整数n和m。

第二行包含n个整数，表示整数数列。

输出格式 共一行，包含m个整数，表示整数数列中前m小的数。

输入样例：

5 3

4 5 1 3 2

输出样例：

1 2 3

根据数组建立一个堆（类似完全二叉树），每个结点的值都大于左右结点（大根堆，通常用于升序），或小于左右结点（小根堆堆，通常用于降序）。

空间：O(1)

最好：O(nlog n)，log n是调整小根堆所花的时间

最坏：O(nlog n)

平均：O(nlog n)

稳定性：否

```
#include<bits/stdc++.h>
using namespace std;
const int N = 100010;
int h[N];//堆
int n,m;
void down(int x){//小根堆 下调
    int p = x*2;//左孩子下标
    while(p <= n){
        if(p + 1 <=n && h[p+1] < h[p]) p++;//找到子节点的最小值
        if(h[x]<=h[p]) return;//调整完毕
        swap(h[x],h[p]);
        x = p;
        p = x*2;
    }
}
int main(){
    scanf("%d%d",&n,&m);
    for(int i = 1; i <= n; i++) scanf("%d",&h[i]);
    for(int i = n/2; i >=1; i--) down(i);//建堆,从最后一个叶子的父节点开始调整即可
    while(m--){
        printf("%d ",h[1]);//堆顶即为最小值
        swap(h[1],h[n]),n--,down(1);//删除堆顶
    }
    return 0;
}
```

##

0x08 归并排序

给定你一个长度为n的整数数列。

请你使用归并排序对这个数列按照从小到大进行排序。

并将排好序的数列按顺序输出。

递归将数组分为两个有序序列，再合并这两个序列

空间: O(n) 辅助备份数组

最好：O(nlog n)

最坏：O(nlog n)

平均：O(nlog n)

稳定性：是

输入样例：

5

3 1 2 4 5

输出样例：

1 2 3 4 5

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5+10;
int a[N],t[N];//原数组 t为归并的辅助数组
void merge_sort(int l, int r){//将l~r从小到大排序
    if(l >= r) return;
    int mid = l + r >> 1;
    merge_sort(l,mid),merge_sort(mid+1,r);//左右部分别排好序
    //合并左右两部分 ， k为待填充位置 每次选最小的去填
    //i为左部分的起始下标 j为右部边的起始下标 mid为左部分的边界
    for(int k = l,i = l, j = mid+1; k <= r; k++){
        if(j > r||i <= mid&&a[i] <= a[j]) t[k] = a[i++];//选择左部分的值去填的条件
        else t[k] = a[j++];//否则只能选右半部分了
    }
    for(int i = l; i <= r; i++) a[i] = t[i];
}
int main(){
    int n;
    scanf("%d",&n);
    for(int i = 0; i < n; i++) scanf("%d",&a[i]);
    merge_sort(0,n-1);
    printf("%d",a[0]);
    for(int i = 1; i < n; i++) printf(" %d",a[i]);
    return 0;
}
```

##

0x09 归并排序的应用-逆序对的数量

给定一个长度为n的整数数列，请你计算数列中的逆序对的数量。

逆序对的定义如下：对于数列的第 i 个和第 j 个元素，如果满足 i < j 且 a[i] > a[j]，则其为一个逆序对；否则不是。

输入样例：

6

2 3 4 5 6 1

输出样例：

5

解释:逆序对分别为 (2,1)(3,1)(4,1)(5,1)(6,1)

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef long long LL;
const int N = 100010;
int a[N],t[N];
LL ans;//逆序对数量
void merge_sort(int l, int r){//l~r从小到大排 排序过程中统计逆序对数量
    if(l >= r) return;
    int mid = l + r >> 1;
    merge_sort(l,mid),merge_sort(mid+1,r);
    for(int k = l,i=l,j=mid+1; k <= r; k++){
        if(j > r|| i<= mid&&a[i]<=a[j]) t[k] = a[i++];
        else ans += mid-i+1LL,t[k] = a[j++];
        //此时arr[i]>arr[j] 逆序 统计逆序对数量  则arr[i~mid]的元素均大于arr[j]构成逆序
    }
    for(int i = l; i <= r; i++) a[i] = t[i];
}
int main(){
    int n;
    scanf("%d",&n);
    for(int i = 0; i < n; i++) scanf("%d",&a[i]);
    merge_sort(0,n-1);
    printf("%lld",ans);
    return 0;
}
```

##

0x0a 基数排序

使用十个桶0-9，把每个数从低位到高位根据位数放到相应的桶里，以此循环最大值的位数次。

只能排列正整数，因为遇到负号和小数点无法进行比较。

空间：O(r) r个队列

最好：O(d(n+r)) d趟分配收集 一趟分配O(n)收集O(r)

最坏：O(d(n+r))

平均：O(d(n+r))

稳定性：是

```cpp
void radixSort(int arr[]) {
    int _max = (*max_element(arr, arr+len));
        // 计算最大值的位数
        int maxDigits = 0;
        while(_max) {
        maxDigits++;
        _max /= 10;
    }
    // 标记每个桶中存放的元素个数
    int bucketSum[10];
    memset(bucketSum, 0, sizeof(bucketSum));
    int div = 1;
    // 第一维表示位数即0-9，第二维表示里面存放的值
```

```
    int res[10][1000];
    while(maxDigits--) {
        int digit;
        // 根据数组元素的位数将其放到相应的桶里，即分配
        for(int i=0; i<len; i++) {
            digit = arr[i] / div % 10;
            res[digit][bucketSum[digit]++] = arr[i];
        }
        // 把0-9桶里的数放回原数组后再进行下一个位数的计算，即收集
        int index = 0;
        for(int i=0; i<=9; i++) {
            for(int t=0; t<bucketSum[i]; t++) {
                arr[index++] = res[i][t];
            }
        }
        memset(bucketSum, 0, sizeof(bucketSum));
        div *= 10;
    }
}
```