A.自动收小麦机

预处理在每一格倒水时水能流过的最短距离，查询时输出前缀和即可。

标程

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

using namespace std;
typedef long long ll;
typedef pair<int, int> PII;
const int N = 1e5 + 5, M = 1e5 + 5, INF = 0x3f3f3f3f;
int n, q, k;
ll h[N], sum[N];    // 麦田的高度和麦田小麦的前缀和
int tj[N], st[N];   // 前一个台阶的距离和最远流到的位置

int main() {
    cin >> n >> q >> k;
    for (int i = 1; i <= n; i++) {
        int x;
        cin >> x;
        // 求前缀和
        sum[i] = sum[i - 1] + x;
    }
    h[0] = -1;
    tj[1] = 1;
    int idx = 0;    // 记录距前一个台阶的距离
    for (int i = 1; i <= n; i++) {
        cin >> h[i];
        if (h[i] != h[i - 1]) tj[i] = idx = 1;
        else tj[i] = idx;
        idx++;
        if (tj[i] < k) st[i] = st[i - 1];   // 能够流入下一个台阶
        else st[i] = i - k + 1;             // 无法流入下一个台阶
    }
    while (q--) {
        int x;
        cin >> x;
        cout << sum[x] - sum[st[x] - 1] << endl;
    }
    return 0;
}
```

B.异星突击

由题可知，我们需要一种数据结构，可以快速的查询所有数中异或某个数**x**后大于**h**的数的数量。

对于异或值的查询可以想到用**trie**来解决，记录一下每个**节点下的数字数量**。每次查询时 从高位向低位贪心查询，当**h**当前位为**1**时，我们只能选择和**x**异或值为**1**的位置继续向下搜索，当当前位**h**为**0**时所有与**x**异或值为**1**的数字都符和条件，然后继续搜索当前位不符合条件但以后可能符和条件的数，即可保证所有选择的数都大于**h**。

标程

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

using namespace std;
typedef long long ll;
typedef pair<int, int> PII;
const int N = 1e5 + 5, M = 1e5 * 32 + 5, INF = 0x3f3f3f3f;
int n, hp, idx = 1;
int cnt[M], tr[M][2]; // 计数每个节点的数字数量 和子节点编号
// 插入
void insert(int x, int num) {
    int p = 1; // 从根节点开始
    for (int i = 30; i >= 0; i--) {
        int d = (x >> i) & 1;
        if (!tr[p][d]) tr[p][d] = ++idx;
        p = tr[p][d];

        cnt[p] += num;  // 当前节点上的数字数量+1
    }
}

int find(int x, int h) {
    int p = 1, res = 0; // 从根节点开始搜索
    for (int i = 30; i >= 0; i--) {
        int d = (x >> i) & 1;   // x这一位的值
        int dh = (h >> i) & 1;  // h这一位的值
        if (!dh) {              // h当前位为0,则选择异或后值为1的所有数都能使结果大于h
            res += cnt[tr[p][d ^ 1]];   // 异或后值为1的所有数数量
            p = tr[p][d];       // 继续查找当前位异或后为0的数
        } else {                // h当前位为1,想要异或后大于h只能选择查找当前值异或后值为1的数
            p = tr[p][d ^ 1];
        }
    }
    return res; // 返回结果
}

int main ()
{
    // 输入输出
    cin >> n >> hp;
    for (int i = 0; i < n; i++) {
        int op, x, h;
        cin >> op;
        if (op == 0) {
            cin >> x;
            insert(x, 1);

        } else if (op == 1) {
```

```
            cin >> x;
            insert(x, -1);
        } else if (op == 2) {
            cin >> x >> h;
            int res = find(x, h);
            cout << res << endl;
            if (res == 0) hp--;
        }
    }
    cout << hp << endl;
    return 0;
}
```

C.悲伤的RT

动态规划递推方程：$dp_i = max(dp_{i-1}, dp_{i-c} + \min_{j=i-c+1}^{i} a_j)$

关键点是选择长度恰好为c的段来分割一定是最优的，如果选择段长小于c则没有任何贡献，如果选择段长为$c < x < 2 * c$则可以分解成一个长度为c的段和多个长度为1的段，且贡献值相同，如果选择长度为$2 * c$则可以分解为两个长度为 c 的段，很容易得出长度为$2 * c$的段的贡献是小于等于两个长度为c的段的。

使用单调队列实现滑动窗口或st表来获取段长为c的区间最小值。

标程

单调队列做法

```
#include <iostream>
#include <cstring>
#include <queue>

using namespace std;

const int N = 1e6 + 10;

long long f[N];
int a[N], n, c;
deque<pair<int, int> > dq;

int main()
{
    cin >> n >> c;
    for (int i = 1; i <= n; i ++ )  cin >> a[i];
    for (int i = 1; i <= n; i ++ )
    {
        f[i] = f[i - 1];
        while (dq.size() && i - dq.front().second >= c) dq.pop_front();
        while (dq.size() && dq.back().first >= a[i])

        {
```

```
            dq.pop_back();
        }
        dq.push_back({a[i], i});
        if (i >= c)
        {
            f[i] = max(f[i], f[i - c] + dq.front().first);
        }
    }
    cout << f[n];
}
```

st表做法

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

using namespace std;
typedef long long ll;
typedef pair<int, int> PII;
const int N = 1e6 + 5, M = 26 + 5, INF = 0x3f3f3f3f;
int n, c;
int st[N][M];
ll dp[N];
int get(int l, int r) {
    int k = __lg(r - l + 1);
    return min(st[l - 1][k], st[r - (1 << k)][k]);
}
int main ()
{
    cin >> n >> c;
    for (int i = 0; i < n; i++)
        cin >> st[i][0];
    for (int i = 1; 1 << i <= n; i++) {
        for (int j = 0; j + (1 << i) <= n; j++) {
            st[j][i] = min(st[j][i - 1], st[j + (1 << i - 1)][i - 1]);
        }
    }
    for (int i = 1; i <= n; i++) {
        if (i == 1 && c == 1) dp[i] = get(i, i);
        if (i > 0)
            dp[i] = max(dp[i], dp[i - 1]);
        if (i >= c)
            dp[i] = max(dp[i], dp[i - c] + get(i - c + 1, i));
    }
    cout << dp[n];
    return 0;
}
```

D.固执的RT

签到题，对n个树枝的攻击力求和并判断攻击力之和是否大于m，如果大于m输出YES，否则输出NO。攻击力之和可能超出int范围。

标程

```
#include <iostream>

using namespace std;

int main()
{
    int n, m;
    long long sum = 0;
    cin >> n >> m;
    while (n -- )
    {
        int x;
        cin >> x;
        sum += x;
    }
    puts(sum >= m ? "YES" : "NO");
    return 0;
}
```

E.释怀的RT

可以将第i个心岩的照亮当成对[i - x, i - 1], [i + 1, i + x]这两个区间的每一个格子加1，最后判断格子是否大于0，如果大于0，说明至少被1个心岩照亮，否则就没有被照亮。区间加可以用差分来做。

标程

```
#include <iostream>

using namespace std;

const int N = 1e6 + 10;

int n, x, pre[N];
void solve() {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> x;
        pre[max(1, i - x)]++, pre[i]--;
        pre[i + 1]++, pre[min(n + 1, i + x + 1)]--;
    }
    int cnt = 0, now = 0;
    for (int i = 1; i <= n; i++) {
        now += pre[i];
        cnt += now > 0;
```

```
    }
    cout << cnt << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    solve();
    return 0;
}
```

F.最短距离

由于 k 较小，我们可以先用一个简单的树形 DP 预处理出每个节点的子树中距离该节点距离为 $0 \sim k$ 的节点数量，记为 $dp[n][k]$。

然后再进行一次换根 DP，计算每个节点的非子树节点距离该节点的距离为 $0 \sim k$ 的节点数量。同时分别给子树中距离该节点的距离为 $0 \sim k$ 的 dp 数组做一个根节点到该位置路径上的前缀和。

对于每个查询 (x, y, d)：

1. 先计算 x 和 y 的最近公共祖先 LCA。
2. 在以 LCA 为根的子树中，到达 $x \sim y$ 简单路径距离为 d 的节点数量
   $= (dp[x][d] + dp[y][d] - dp[LCA][d] - dp[fa[LCA]][d]) - (dp[x][d-1] + dp[y][d-1] - 2 \times dp[LCA][d-1])$
   （通过前缀和计算，在以一个节点为根的子树中，到该节点距离为$d-1$的节点到其父节点的距离为d，但是这些节点并不符合题意，故减去）。
3. 再加上距离 LCA 为 d 的非子树节点数量。
4. 总的结果就是上面两项相加。

总的时间复杂度：

预处理树形 DP 和换根 DP 均为 $O(n \times k)$。

查询时，计算最近公共祖先为 $O(\log n)$，计算结果为 $O(1)$。

共有 q 次查询。

故总的时间复杂度为 $O(n \times k + q \times \log n)$。

标程

```
#include <bits/stdc++.h>
using namespace std;
vector<int>num[100010];
int deep[100010],st[20][100010];
int down[100010][110],up[100010][110];
void dfs(int x,int fa){
    deep[x]=deep[fa]+1;st[0][x]=fa;

    for(int i=1;i<=19;i++)
```

```
    st[i][x]=st[i-1][st[i-1][x]];
    down[x][0]=1;
    for(auto i:num[x])if(i!=fa){
        dfs(i,x);
        for(int j=0;j<100;j++)
        down[x][j+1]+=down[i][j];
    }
}
void dfs1(int x,int fa){
    for(int j=0;j<=100;j++)
    down[x][j]+=down[fa][j];
    for(auto i:num[x])if(i!=fa){
        up[i][1]=1;
        for(int j=1;j<100;j++)
        up[i][j+1]=(down[x][j]-down[fa][j])+up[x][j]-down[i][j-1];
        dfs1(i,x);
    }
}
int lca(int x,int y){
    if(deep[x]<deep[y])
    swap(x,y);
    int h=deep[x]-deep[y];
    for(int i=19;i>=0;i--)
    if(h>=(1<<i))h-=1<<i,x=st[i][x];
    if(x==y)return x;
    for(int i=19;i>=0;i--)
    if(st[i][x]!=st[i][y])
    x=st[i][x],y=st[i][y];
    return st[0][x];
}
int main(){
    int n,t;scanf("%d%d",&n,&t);
    for(int i=1;i<n;i++){
        int x,y;scanf("%d%d",&x,&y);
        num[x].push_back(y);
        num[y].push_back(x);
    }dfs(1,0);dfs1(1,0);
    while(t--){
        int x,y,k;
        scanf("%d%d%d",&x,&y,&k);
        int mid=lca(x,y);
        int ans=down[x][k]+down[y][k]-down[mid][k]-down[st[0][mid]][k];
        ans-=down[x][k-1]+down[y][k-1]-2*down[mid][k-1];
        ans+=up[mid][k];
        printf("%d\n",ans);
    }
}
```

G.爬山

考察知识点：二分

时间复杂度：nlog(n)

因为缆车越高攀爬高度越低，所以符合二分单调性，对缆车高度进行二分，计算过程中会爆int需要开long long。

标程

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5+10;
typedef long long LL;
int n, p,a[N];
bool func(int h)
{
    LL sum = 0;
    for (int i = 1; i <= n;i++)
        if(a[i]>h)sum += a[i] - h;
    return sum*2>=p;//包括上山高度+下山高度，即攀爬高度*2
}
int main()
{
    LL sum = 0;
    cin >> n >> p;
    for (int i = 1; i <= n;i++)
        cin >> a[i],sum+=a[i];
    if(sum*2<p)//即使缆车高度为0也不可能完成运动指标
    {
        cout << -1;
        return 0;
    }
    int l = 0, r = 0x3f3f3f3f;
    while(l<=r)
    {
        int mid = l + r >> 1;
        if(func(mid))l = mid + 1;
        else r = mid - 1;
    }
    cout <<l-1;
    return 0;
}
```

H.欢乐颂

考察知识点：最小生成树+树的直径

时间复杂度：mlog(m)+n

（1）建图：根据弦值连接不同的琴柱。

（2）建树：按照边权（能量）>较小编号琴柱>较大编号琴柱的优先级，使用Kruskal算法求建立小生成树。

（3）求树的直径：可以用树形dp或搜索两次最长路。

标程：

```
#include<bits/stdc++.h>
#define LL long long
#define PII pair<int,int>
#define x first
#define y second
using namespace std;
const int N=1e5+10,M=1e6+10;
typedef struct edges{int x,y,w;}edges;
vector<int> v1,v2[M];//v1记录所有弦值的种类，v2记录每种弦值所具有的琴柱编号
vector<PII> v[N];//储存最小生成树
edges edge[M];//储存所有可以连接的的边
bool st[M];//标记弦值避免重复添加
int num[N],a[N];//num统计每个琴柱所拥有的弦值的和，a是并查集数组
int n,ans;
LL res,dp[N];//dp记录当前节点的到叶子节点最长距离
bool cmp(edges xx,edges yy)//排序
{
    if(xx.w!=yy.w)return xx.w<yy.w;
    else if(xx.x!=yy.x)return xx.x<yy.x;
    return xx.y<yy.y;
}
int find1(int x)//并查集
{
    if(x!=a[x])a[x]=find1(a[x]);
    return a[x];
}
void DP(int u,int p)//树形dp查找树的直径,u是当前节点，p是u的父节点
{
    for(auto it:v[u])
    {
        if(it.x==p)continue;
        DP(it.x,u);
        res=max(res,dp[it.x]+dp[u]+it.y);
        dp[u]=max(dp[u],dp[it.x]+it.y);
    }
}
int main()
{
    cin>>n;
    for(int i=1;i<=n;i++)

    {
```

```
        int k,x;
        cin>>k;
        while(k--)
        {
            cin>>x;
            num[i]+=x;
            v2[x].push_back(i);
            if(!st[x])v1.push_back(x),st[x]=1;
        }
        a[i]=i;
    }
    for(int it:v1)//建图
    {
        sort(v2[it].begin(),v2[it].end());//保证edge中小编号琴柱为edge.x
        for(int i=0;i<v2[it].size();i++)
            for(int j=i+1;j<v2[it].size();j++)
            edge[ans++]={v2[it][i],v2[it][j],(num[v2[it][i]]^num[v2[it][j]])+1};
    }
    sort(edge,edge+ans,cmp);//根据题目信息排序
    int sum=0;
    for(int i=0;i<ans&&sum!=n-1;i++)//Kruskal求最小生成树
    {
        int x=edge[i].x,y=edge[i].y,w=edge[i].w;
        int x1=find1(x),y1=find1(y);
        if(x1==y1)continue;
        a[x1]=y1;
        v[x].push_back({y,w});//建树
        v[y].push_back({x,w});
        sum++;
    }
    if(sum!=n-1)cout<<0<<endl;//无法建立最小生成树
    else
    {
        DP(1,0);//以1为根开始
        cout<<res<<endl;
    }
    return 0;
}
```

I.奶牛的寿命

考察知识点：模拟

时间复杂度：log(n)

保留最高位的1，其余的1尽量移到低位。

标程

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    cin >> n;
    bitset<31> m(n);
    int num = m.count() - 1;//num为二进制一的个数-1
    bool flag = 0;//判断有没有出现第一个1
    for (int i = 30; i >= 0;i--)
    {
        if(i<num)m[i] = 1;
        else if(flag)m[i] = 0;
        if(m[i]&1) flag = 1;
    }
    cout <<n-m.to_ulong();
    return 0;
}
```

J.睡美人

考察知识点：组合数

时间复杂度：nlog(n)

由于1号魔药可以随机改变顺序所以题目与字符串s的顺序无关，但是需要统计两种字符的数量。由于2号魔药可以合并多个魔毯为一个魔毯，我们考虑先可以按区间排列魔毯。

（1）当m为奇数时，序列两端一定为相同种类的魔毯区间且这种魔毯的区间总数量为m/2+1，另外一种魔毯区间数量自然就为m/2。然后开始划分区间，假设魔毯单体数量为sum需要划分的区间数量为num，那么就相当于在sum-1个位置中选择并切num-1刀，由于有两种魔毯所以需要求两次。

（2）m为偶数时和奇数计算方式一样，只不过两端魔毯区间种类不同，计算一次乘2即可（两种序列翻转其中一个后和另一序列相同）

组合数公式：

$$C_n^m = \frac{A_n^m}{m!} = \frac{n!}{m!\,(n-m)!}$$

标程：

```
#include<bits/stdc++.h>
using namespace std;
typedef long long LL;
const int mod=1e9+7;

const int N=1e6+10;
```

```cpp
int n,m;
string s;
LL num1[N],num2[N];//num1储存范围内所有数字的阶乘，num2储存范围内所有数字的阶乘的逆元
LL qmi(LL x,LL y)//快速幂
{
    LL sum=1;
    while(y)
    {
        if(y&1)sum=(sum*x)%mod;
        y>>=1;
        x=x*x%mod;
    }
    return sum;
}
void init()//初始化num1，num2
{
    num1[0]=num2[0]=1;
    for(int i=1;i<=n;i++)
    {
        num1[i]=num1[i-1]*i%mod;
        num2[i]=qmi(num1[i],mod-2);//费马小定理求逆元
    }
}
LL C(int x,int y)//求组合数
{
    if(x-y<0)return 0;
    return num1[x]*num2[y]%mod*num2[x-y]%mod;
}
int main()
{
    cin>>n>>m>>s;
    init();
    int sum1=0,sum2=0;//sum1储存0的个数，sum2储存1的个数
    for(int i=0;i<n;i++)s[i]=='0'?++sum1:++sum2;
    if(sum1<sum2)swap(sum1,sum2);
    int sum=sum2*2+(sum1==sum2?0:1);//sum为在使用完2号魔药后最多可以保留的魔毯数
    LL res=C(n,sum2);//使用完一号药水后所有可能的组合情况
    if(sum<m||m==1)cout<<0<<" "<<res;//不存在移除的可能，题目保证0和1至少出现一次所以m至少为2
    else
    {
        LL ans=0;
        int x=m/2,y=m/2;
        if(m&1) ans=(C(sum1-1,x)*C(sum2-1,y-1)+(ans+C(sum1-1,y-1)*C(sum2-1,x)))%mod;//m为奇数，隐含了x--
        else ans=(C(sum1-1,x-1)*C(sum2-1,y-1)*2)%mod;//m为偶数
        cout<<ans<<" "<<(res-ans+mod)%mod;
    }
    return 0;
}
```

K.卡特兰数

一个大于1的整数可以拆成若干个质数相乘，而质数相乘且结果末尾有0的只有2*5。所以一个数末尾0的数量就等于这个数2的因子数量和5的因子数量最小值。

标程

```
#include <iostream>

using namespace std;

const int N = 5e6 + 10;

int cnt2[N], cnt5[N];

int func(int x, int t)
{
    int res = 0;
    while (x % t == 0)
    {
        res ++ ;
        x /= t;
    }
    return res;
}

int main()
{
    int n;
    cin >> n;
    for (int i = 1; i <= n; i ++ )
    {
        cnt2[i] = cnt2[i - 1], cnt5[i] = cnt5[i - 1];

        cnt2[i] += func(4 * i - 2, 2);
        cnt5[i] += func(4 * i - 2, 5);

        cnt2[i] -= func(i + 1, 2);
        cnt5[i] -= func(i + 1, 5);
    }
    for (int i = 1; i <= n; i ++ )
    {
        cnt2[i] += cnt2[i - 1];
        cnt5[i] += cnt5[i - 1];
    }
    cout << min(cnt2[n], cnt5[n]);
}
```

L.最长连续相同字符

可以考虑用线段树来维护每个区间的最长连续相同字符。

首先确定线段树节点维护的信息，题目求最长连续相同字符，所以节点要储存当前区间中最长连续相同字符的左右边界和字符类型，用maxn储存这些信息。

接着考虑如何用子节点的信息维护父节点的信息。父节点的最长连续相同字符有三种可能：1.全部在左区间；2.全部在右区间；3.跨越两个区间。

1，2需要的信息在左右子节点的maxn里储存，3需要知道左子节点的后缀最长相同连续字符rmax（以左子节点右端点为末尾）和右子节点的前缀最长相同连续字符lmax（以右子节点的左端点为起始）。（以字符串“aaacsdfedddd”，“aaa”为该字符串的前缀最长相同连续字符，“dddd”为该字符串的后缀最长相同字符）。

由于左右子节点独立，父节点u.maxn有以下三种取值：

1. 左子节点的最长连续相同字符l.maxn。
2. 右子节点的最长连续相同字符r.maxn。
3. 当左子节点后缀最长连续相同字符的字符类型和右子节点前缀最长连续相同字符的字符类型相同时，可以为左子节点后缀最长连续相同字符l.rmax + 右子节点前缀最长连续相同字符r.lmax。

   接着考虑父节点的lmax和rmax如何由子节点转移。

   父节点的lmax有两种取值可能：
4. 左子节点的前缀最长连续相同字符l.lmax
5. 当左子节点的所有字符为同一种字符类型且与右子节点前缀最长连续相同字符的字符类型相同时，为l.lmax + r.lmax。

   父节点的rmax同理
6. 右子节点的后缀最长连续相同字符r.rmax
7. 当右子节点的所有字符为同一种字符类型且与左子节点前缀最长连续相同字符的字符类型相同时，为l.rmax + r.rmax。

标程

```
#include <iostream>

using namespace std;

const int N = 1e5 + 10;

struct SEG{
    int l, r, len;
    char ch;
};

struct node{
    int l, r;
    SEG maxl, maxr, maxn;
}tr[N << 2];
```

```
int n, m;
string a;

SEG max(SEG& a, SEG& b)
{
    if (a.len == b.len) return a.l < b.l ? a : b;
    return a.len > b.len ? a : b;
}

void pushup(node& u, node& l, node& r)
{
    u.maxl = l.maxl;
    if (l.maxl.r == l.r && l.maxl.ch == r.maxl.ch)
    {
        u.maxl = {l.maxl.l, r.maxl.r, r.maxl.r - l.maxl.l + 1,l.maxl.ch};
    }
    u.maxr = r.maxr;
    if (r.maxr.l == r.l && r.maxr.ch == l.maxr.ch)
    {
        u.maxr = {l.maxr.l, r.maxr.r, r.maxr.r - l.maxr.l + 1,r.maxr.ch};
    }
    u.maxn = max(l.maxn, r.maxn);
    if (l.maxr.ch == r.maxl.ch)
    {
        SEG t = {l.maxr.l, r.maxl.r, r.maxl.r - l.maxr.l + 1, l.maxr.ch};
        u.maxn = max(u.maxn, t);
    }
}

void pushup(int u)
{
    pushup(tr[u], tr[u << 1], tr[u << 1 | 1]);
}

void build(int u, int l, int r)
{
    if (l == r)
    {
        tr[u] = {l, r, {l, r, 1, a[l]}, {l, r, 1, a[l]}, {l, r, 1, a[l]}};
        return;
    }
    tr[u] = {l, r};
    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    pushup(u);
}

void modify(int u, int x, char ch)
{
    if (tr[u].l == x && tr[u].r == x)
    {
        tr[u] = {x, x, {x, x, 1, ch}, {x, x, 1, ch}, {x, x, 1, ch}};
```

```
        return;
    }
    int mid = tr[u].l + tr[u].r >> 1;
    if (x <= mid)   modify(u << 1, x, ch);
    else    modify(u << 1 | 1, x, ch);
    pushup(u);
}

node query(int u, int l, int r)
{
    if (tr[u].l >= l && tr[u].r <= r)   return tr[u];
    int mid = tr[u].l + tr[u].r >> 1;
    if (r <= mid)   return query(u << 1, l, r);
    if (l > mid)    return query(u << 1 | 1, l ,r);
    node left = query(u << 1, l, r), right = query(u << 1 | 1, l, r), res;
    pushup(res, left, right);
    return res;
}

int main()
{
    cin >> n >> m >> a;
    a = " " + a;
    build(1, 1, n);
    while (m -- )
    {
        int op;
        cin >> op;
        if (op == 1)
        {
            int l, r;
            cin >> l >> r;
            node ans = query(1, l, r);
            cout << ans.maxn.l << " " << ans.maxn.r << endl;
        }
        else
        {
            int x;
            char ch;
            cin >> x >> ch;
            modify(1, x, ch);
        }
    }
    return 0;
}
```