魔力屏障

题意分析

有一排魔力屏障从左向右排列，第 i 个屏障的临界值为 a_i。对于一次魔力值为 x 的攻击，如果 x 大于当前魔力屏障的临界值，那么将击碎该屏障，魔力值减半为 $\lfloor x/2 \rfloor$，到达下一个屏障。否则，攻击会在当前屏障停止。两次攻击可以叠加，魔力值为两者叠加的和。

对于a_i的每个前缀，求出以任意顺序，任意大小释放攻击，破除所有屏障所需的魔力值之和的最小值。

$1 \leq n \leq 70$，$1 \leq a_i \leq 150$

知识点

区间DP

题解

思路1

从样例中不难看出，从前到后逐个击破不一定是最优的。例如 1000 1 500，这时如果 1000 溢出的攻击落到 1 上，那么将会浪费很多。所以可能需要先消除中间的一部分位置。

对于$n \leq 10$ 的部分，考虑对于每个前缀各自计算答案。不难想到枚举击破屏障的顺序，我们仍需要确定每一时刻所需要叠加的新的攻击的大小。不难发现，任意时刻都选择恰好能击破当前的屏障一定是最优的。因为当前时刻如果花费更多代价，那么未来这部分魔力值可能会减半，不如推迟到之后需要的时候直接补足。所以在枚举击破屏障的顺序之后，叠加攻击的大小是唯一确定的，按照题意模拟即可。

这样能得到一个$O(n! \times \mathrm{poly(n)})$的做法，这部分复杂度是跑不了满分的，可以通过子任务 1，预期得分 30 分。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 80;
int ans;
vector<int> a, b;        //a[i]：每道屏障的临界值，b[i]：当前已经积累的魔力值
void dfs(int c, int P)
{
    bool ok = 1;         //是否已经击碎所有屏障
    for (int i = 0; i <= P; ++i)
        if (b[i] < a[i]) ok = 0;
    if (ok == 1) {
        ans = min(ans, c);       //击碎所有屏障需要的魔力值
        return;
    }
    //枚举补足哪个位置的魔力值
    for (int i = 0; i <= P; ++i) {
        if (b[i] < a[i]) {
            c += a[i] - b[i];
            vector<int> bb(b);
            b[i] = a[i];
            //模拟向后击碎屏障和魔力叠加
            for (int j = i + 1, val = b[i] / 2; j <= P; ++j)
                if (b[j] >= a[j]) continue;
                else if (b[j] < a[j]) {
```

```
                    b[j] += val;
                    if (b[j] < a[j]) break;
                    val = b[j] / 2;
                }
            dfs(c, P);
            b = bb;
            c -= a[i] - b[i];
        }
    }
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n; cin >> n;
    a.resize(n); b.resize(n);
    for (int i = 0; i < n; ++i) cin >> a[i];
    for (int i = 0; i < n; ++i) {
        ans = INT_MAX;
        dfs(0, i);      //对于a[0]~a[i]进行搜索
        cout << ans << " ";
    }
    return 0;
}
```

思路2

考虑区间 DP，设 $dp_{l,r,k}$ 表示将 $[l,r]$ 内的屏障全部击碎，剩下的攻击魔力值为 k 时，所需花费魔力总和的最小值，那么对于前 i 个数字组成的序列，答案就是

$$\min_{j\geq a[i]/2}\{dp_{1,i,j}\}$$

初始化 $dp_{i,i,a[i]/2}=a[i]$。

下面考虑 DP 状态的转移。

首先是关于魔力值叠加的转移。枚举区间内的一个位置 mid，钦定 mid 位置的剩余魔力值会直接叠加到 r 剩余的部分，那么 $[l,mid]$ 与 $[mid+1,r]$ 的部分就可以独立处理。枚举最后的 k 点魔力值分别有多少是左右区间贡献的，于是有

$$dp_{l,r,k}=\min_{l\leq mid\leq r}\min_{p\leq k}(dp_{l,mid,p}+dp_{mid+1,r,k-p})$$

其次是叠加后击破下一个的转移。钦定 $[l,r-1]$ 的叠加后剩余的魔力值为 p，用于击破下一个位置，那么此时击破下一个位置只需额外花费 $\max\{0,a_r-p\}$，并且击破后剩余魔力值也可以计算出来。

考虑转移的复杂度，可以发现 k 一定 $\leq\sum a_i$，因为答案至多为 $\sum a_i$，所以多出的魔力值一定是无用的。这样可以通过子任务 3。

思路3

类似思路2，进一步可以发现在最优解中，任意时刻叠加的魔力值之和一定不会超过 $\max a_i$，否则多出的部分在遇到下一个屏障后必定毫无作用，反而还要减半，不如在需要的时候在实行攻击。也即 $k\leq\max a_i$ 即可。

记 $V=\max a_i$，复杂度为 $\mathcal{O}(n^3V^2)$，并且带有 $\frac{1}{8}$ 的小常数，可以通过子任务 $1,2,3,4$，预期得分 100 分。

易错点:

- 在 DP 转移中，叠加后击破下一个的转移需要单独讨论。
- DP 状态设计中 k 这一维的大小需要仔细分析题目性质。

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pii;
const int N = 80, M = 152;
int dp[N][N][M], n, a[N], mx;
int main()      //dp[i][j][k]表示将[i,j]内的屏障全部击碎，剩下的攻击魔力值为k的最小代价
{
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
        mx = max(mx, a[i]);
    }
    for (int i = 1; i <= n; ++i) {
        for (int k = 0; k <= mx; ++k)
            dp[i][i][k] = 0x3f3f3f3f;
        dp[i][i][a[i]/2] = a[i];        //初始时，击碎单个屏障后会剩余a[i]/2的魔力
    }
    for (int len = 2; len <= n; ++len) {
        for (int l = 1; l + len - 1 <= n; ++l) {
            int r = l + len - 1;
            for (int k = 0; k <= mx; ++k)
                dp[l][r][k] = 0x3f3f3f3f;
            for (int mid = l + 1; mid <= r; ++mid) {
                for (int p = a[r]/2; p <= mx; ++p)
                    for (int k = a[r]/2; k <= p; ++k)
                        //关于魔力值叠加的转移，枚举中间位置mid，分为两部分
                        dp[l][r][p] = min(dp[l][r][p], dp[l][mid - 1][p - k] +
dp[mid][r][k]);
            }
            for (int p = 0; p <= mx; ++p) {
                int nxt = max(p, a[r]) / 2;
                // 关于叠加后击破下一个的转移，需要补充 max(0, a[r] - p) 的魔力值
                dp[l][r][nxt] = min(dp[l][r][nxt], dp[l][r - 1][p] + max(0, a[r]
- p));
            }
        }
    }
    for (int i = 1; i <= n; ++i) {
        int ret = 0x3f3f3f3f;
        for (int k = a[i] / 2; k <= mx; ++k)
            ret = min(ret, dp[1][i][k]);
        printf("%d ", ret);
    }
    return 0;
}
```

诡秘之主

题解

题意解释

题目描述了一棵二叉搜索树。探索团队按顺序进入二叉搜索树，每名队员根据规则选择向左或向右的子节点探索。除了序列0以外其它序列的值都可由你自行决定。探索的成功度为所有队员最终停留位置与根节点的距离的最大值，探索的收益为能获得的探索成功度的最小值。现在给出一队成员的序列，需要回答若干次查询，求出每次查询区间内所有子区间权值之和。

这道题求值的关键在于要发现权值是根据队员序列0和其它序列的分布情况计算得出的，序列0是无法改变的，且它们一定会组成一条向左的链，而其它序列的分布是可以在不与序列0冲突的情况下任意决定的。从这一点出发，利用好二叉搜索树的性质，我们便可以有一些初步的想法。

知识点提炼

二叉搜索树、树状数组、前缀和

核心解题思路

子任务1、2

以下记 s_l 表示 01 串的第 l 位。

$l=1,r=n$ 的情况。

首先求一个区间的权值。考虑判断一个高度 h 是否能够容纳下这么多人，设该区间中 0 的个数为 a，1 的个数为 x，第一个 0 前有 b 个 1。因为所有 0 最终一定会形成一条向左的链，所以最优选择应当是取出 $\min(b,h-a+1)$ 个第一个 0 前的 1，组成一条向左的链，将 0 接在上面，其余的 1 可以在放在全树除第一个 0 的左子树外的位置。那么 h 能够容纳下所有人需要满足以下条件：

- $h\geq a-1$，当 $b\geq 1$ 时需要满足 $h\geq a$。
- 第一个 0 前的 b 个 1 可以在 0 子树外的位置放下，需要满足：$2^{h+1}-2^a\geq b$。
- 其余的 1 可以在第一个 0 的左子树外的位置放下，需要满足：$2^{h+1}-2^{h-\min(b,h-a+1)}-1\geq x$。

接下来如果预处理出所有子区间的权值，具体地，针对每个查询区间，我们需要枚举每个可能的树高，验证是否可行，从中选择最小的权值作为当前区间的权值。然后询问时对所有子区间的权值求和，即可通过子任务1，时间复杂度$O(n^3+qn^2)$。

通过二分等方式预处理前缀和，并用ST表预处理出询问的值，可将复杂度降至$O(n^2\log n)$，可用于子任务1、2。

期望得分10~25分。

```
#include<bits/stdc++.h>
using namespace std;
const int MOD = 998244353;
const int MAXN = 1005;
int dp[MAXN][MAXN];      //用于保存预处理的权值
int seq[MAXN];           //输入的01序列
```

```
int n, q;
inline int calc1(int height,int a){
    if(height>=22) return 0x3f3f3f3f;
    return (1<<height+1)-(1<<a);
}
inline int calc2(int height,int a,int b){
    if(height>=22) return 0x3f3f3f3f;
    return (1<<height+1)-(1<<(height-min(b,height-a+1)))-1;
}
//预处理所有子区间的权值
void preprocess() {
    // 动态规划计算权值
    for (int len=1;len<=n;++len){
        for (int l=1;l+len-1<=n;++l){
            int r=l+len-1;
            //计算区间内0和1的数量
            int zeroCount=0,oneCount=0,b=0;
            for (int i=l;i<=r;++i){
                if (seq[i]==0) ++zeroCount;
                else{
                    ++oneCount;
                    if (zeroCount==0) ++b;
                }
            }
            if(zeroCount==0){
                dp[l][r]=log2(len);
                continue;
            }
            //枚举树高并验证是否可行
            for (int h=0;h<=len;++h){
                int a=zeroCount;
                int x=oneCount;
                //计算树高能否容纳所有人的条件
                bool valid=(h>=a-1)&(b==0||h>=a);
                valid=valid & (calc1(h,a)>=b) & (calc2(h,a,b)>=x);
                if (valid){
                    dp[l][r]=h;     //更新权值为树高
                    break;
                }
            }
        }
    }
}
//处理查询并返回结果
int processQuery(int l,int r){
    int ans=0;
    for (int i=l;i<=r;++i) {
        for (int j=i;j<=r;++j) {
            ans=(ans+dp[i][j]) % MOD;       //对所有子区间的权值求和
        }
    }
    return ans;
}
int main(){
    string S;
    cin>>n>>q>>S;
```

```
    for (int i=1;i<=n;++i) seq[i]=s[i-1]-'0';
    preprocess();       //预处理所有子区间的权值
    //处理每个查询并输出结果
    for (int i=0;i<q;++i){
        int l,r;
        cin>>l>>r;
        cout<<processQuery(l,r) << endl;
    }
    return 0;
}
```

子任务3

若字符串中没有0，可以任意构造二叉树的形态，此时最优解为完全二叉树，即铺满一层再铺下一次，单个区间答案即为区间长度二进制的位数。

接下来便可以单次询问$O(\log n)$地枚举所有可能的答案并统计和。

时间复杂度$O(q\log n)$。期望得分5分。

以下为单个区间答案的求法。

```
ans[l][r] = log2(len);
```

子任务4、5、6

考虑固定左端点 l 计算所有右端点的答案，可以按照区间内的 0 的个数将右端点分为以下几个部分：

- $a=0$：若区间长度为 x，则答案即为 $f(x)=\min\{h|2^{h+1}-1\geq x\}$，可以 $\mathcal{O}(n)$ 双指针预处理 f 即其前缀和，然后即可 $\mathcal{O}(1)$ 回答。
- $a\leq\log n$：这一部分右端点对应的 h 一定 $\leq\mathcal{O}(\log n)$，可以按照 a 的大小分成 $\mathcal{O}(\log n)$ 段，每段暴力枚举高度，判断该高度最多能容纳多少个 1。注意到左端点固定时，随着右端点向右移动，区间权值单调不降，可以使用双指针同时移动右端点与高度，复杂度是 $\mathcal{O}(\log n)$ 的。
- $a>\log n$，那么只要满足第一条，后两条一定会被满足。因此根据 $s[l]$ 是否为 0，可以得到这一部分的右端点对应的答案就是 a 或者 $a-1$。通过预处理 $pre_i=\sum_{j=1}^{i}(\sum_{k\leq j}[s_k=0])$ 即可 $\mathcal{O}(1)$ 完成。

对于区间 $[l,r]$ 的询问，只需要在上面的过程中预处理每个后缀的答案，然后用后缀 l 的答案减去后缀 $r+1$ 的答案，再减去跨过 r 的区间的权值即可。对后者也可以按照 0 的个数是否 $>\log n$ 分类：

- $a>\log n$：用 0 将序列分为若干段，枚举右端点所在段，可以找到要满足 0 个数 $>\log n$ 对应的左端点所在的区间，计算一段区间的左端点与一段区间的右端点对应的权值之和通过预处理前缀和依然可以 $\mathcal{O}(1)$ 计算。当右端点与 r 中间间隔超过 $\log n$ 个 0 后，对应的左端点区间就是恒定的了，可以一起处理，这部分的复杂度就是 $\mathcal{O}(\log n)$。
- $a\leq\log n$：这部分区间的权值一定 $<\log n$。考虑暴力枚举区间的权值，注意到如果将左右端点视作两维，那么权值为 h 的区间对应的是 $\mathcal{O}(n)$ 个 $1\times k$ 的矩形加，询问就是查询给定矩形与这些矩形的交，可以用离线树状数组做到 $\mathcal{O}(n\log^2 n+q\log n)$。

最终复杂度是 $\mathcal{O}(n\log^2 n+q\log n)$。

若没有完全实现询问操作，可得到子任务4的30分。

之后依据实现询问操作的复杂度和常数，可得到子任务5、6的20~40分。

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

```cpp
//快读快写
namespace iobuff{
    const int LEN=10000000;
    char in[LEN+5],out[LEN+5];
    char *pin=in,*pout=out,*ed=in,*eout=out+LEN;
    inline char gc(void){
        #ifdef TQX
        return getchar();
        #endif
        return pin==ed&&(ed=(pin=in)+fread(in,1,LEN,stdin),ed==in)?EOF:*pin++;
    }
    inline void pc(char c){
        pout==eout&&(fwrite(out,1,LEN,stdout),pout=out);
        (*pout++)=c;
    }
    inline void flush(){fwrite(out,1,pout-out,stdout),pout=out;}
    template<typename T> inline void read(T &x){
        static int f;
        static char c;
        c=gc(),f=1,x=0;
        while(c<'0'||c>'9') f=(c=='-'?-1:1),c=gc();
        while(c>='0'&&c<='9') x=10*x+c-'0',c=gc();
        x*=f;
    }
    template<typename T> inline void putint(T x,char div='\n'){
        static char s[20];
        static int top;
        top=0;
        x<0?pc('-'),x=-x:0;
        while(x) s[top++]=x%10,x/=10;
        !top?pc('0'),0:0;
        while(top--) pc(s[top]+'0');
        pc(div);
    }
}
using namespace iobuff;
using namespace std;
typedef long long ll;
const int N = 2e5 + 10, mod = 998244353, B = 20;
char s[N];
int n, q, stk[N], top, pze[N], pre[N], mx, lim[N];
ll suf[N], ret[N], ppze[N], anq[N], ssuf[N];
int ql[24][N], qr[24][N];

// 计算树高为 height, 需容纳的位置数量
inline int calc(int bef, int height) {
    if (height >= 22) return 0x3f3f3f3f;
    return (1 << (height + 1)) - (1 << (height - bef)) - 1;
}

// 计算树高为 height, 需容纳的剩余位置数量（不包括 0）
inline int space(int bef, int height) {
    if (height >= 22) return 0x3f3f3f3f;
    return (1 << (height + 1)) - (1 << (height - bef + 1));
}
```

```cpp
// 预处理，初始化计算数组
inline void init() {
    // 输入 n 和 q
    scanf("%d%d", &n, &q);
    // 输入队员的序列01串
    scanf("%s", s + 1);

    // 初始化 pre 数组和 lim 数组
    pre[0] = 0;
    for (int i = 0; i <= 22; ++i) lim[i] = n;
    for (int i = 1, now = 0; i <= n; ++i) {
        while ((1 << (now + 1)) - 1 < i) ++now;
        pre[i] = pre[i - 1] + now;
        lim[now] = i;
    }

    // 计算 pze 数组和 ppze 数组
    pze[0] = ppze[0] = 0;
    for (int i = 1; i <= n; ++i) {
        pze[i] = pze[i - 1] + (s[i] == '0');
        ppze[i] = ppze[i - 1] + pze[i - 1];
    }

    // 初始化 suf 数组和 ssuf 数组
    suf[n + 1] = ssuf[n + 1] = 0;
    for (int i = n; i >= 1; --i) {
        suf[i] = suf[i + 1] + pze[i];
        ssuf[i] = ssuf[i + 1] + suf[i];
    }

    // 初始化 ans 为 0，stk 数组为栈顶元素为 n+1
    ll ans = 0;
    stk[top = 1] = n + 1;

    // 倒序遍历数组 s，从每个位置出发计算树的权值
    for (int i = n; i >= 1; --i) {
        // 如果当前位置是 0，则将其入栈
        if (s[i] == '0') stk[++top] = i;

        // 计算从当前位置出发的树的 bef
        int bef = stk[top] - i, ct = 0, now = bef;

        // 如果bef非 0，则将 ans 加上 pre[bef]
        if (bef) ans += pre[bef];

        // 初始化数组 ql 和 qr
        for (int j = 0; j <= 22; ++j) ql[j][i] = n + 1, qr[j][i] = 0;

        // 计算数组 ql 和 qr
        for (int j = 1; j <= 22; ++j) {
            int ll = lim[j - 1] + 1, rr = min(lim[j], bef);
            if (ll <= rr) ql[j][i] = min(ql[j][i], ll + i - 1), qr[j][i] =
max(qr[j][i], rr + i - 1);
            mx = max(mx, j);
            if (lim[j] >= bef) break;
        }
```

```
        // 遍历栈 stk
        for (int j = top; j > 1; --j) {
            int x = stk[j]; ++ct;

            // 当 ct > B 时，采用特殊处理，将计算结果累加到 ans 中并退出循环
            if (ct > B) {
                if (bef) ans += suf[x];
                else ans += suf[x] - (n - x + 1);
                ans -= 1ll * pze[i - 1] * (n - x + 1);
                break;
            }

            // 计算树高 be 和 height，并使得当前高度合法
            int be = min(bef, 1), height = ct + be - 1;
            while (space(be, height) < bef) ++be, ++height;

            // 计算区间 [l, r] 并更新 ans 和数组 ql 和 qr
            int l = now, las = now; now += stk[j - 1] - stk[j] - 1; int r = now;
            while (l <= r) {
                while (calc(be, height) < l) {
                    if (be < bef) ++be;
                    ++height;
                }
                int nr = min(calc(be, height), r);
                mx = max(mx, height);
                ql[height][i] = min(ql[height][i], (l - las) + stk[j]);
                qr[height][i] = max(qr[height][i], (nr - las) + stk[j]);
                ans += 1ll * height * (nr - l + 1);
                l = nr + 1;
            }
        }

        // 将当前位置的权值记录到 ret 数组中
        ret[i] = ans;
    }

    // 将栈 stk 反转，保证 stk[1] 是最后一个 0 的位置
    reverse(stk + 1, stk + top + 1);
}


// 计算查询区间 [l1, r1] 和 [l2, r2] 的重叠权值
inline ll calc(int l1, int r1, int l2, int r2) {
    if (l1 > r1 || l2 > r2) return 0;
    ll ans = 0;
    int x2 = pze[r1] - pze[l1 - 1], x1 = r1 - l1 + 1 - x2;
    ans += (suf[l2] - suf[r2 + 1]) * x1;
    ans += (suf[l2] - suf[r2 + 1] - (r2 - l2 + 1)) * x2;
    ans -= 1ll * (r2 - l2 + 1) * (ppze[r1] - ppze[l1 - 1]);
    return ans;
}

int L[N],R[N],tot,first[N],nxt[N<<1];
pair<int,int> val[N<<1];
inline void add(int x,pair<int,int> y){
```

```
    val[++tot]=y;
    nxt[tot]=first[x];first[x]=tot;
}

// 计算查询区间的权值之和
inline ll solve(int id, int l, int r) {
    L[id] = l; R[id] = r;
    ll ans = ret[l] - ret[r + 1];
    int pos = lower_bound(stk + 1, stk + top + 1, r + 1) - stk;
    if (pos > B + 1) ans -= calc(l, stk[pos - B - 1], r + 1, stk[pos] - 1);
    for (int i = pos; i < top; ++i) {
        if (i > B) {
            if (stk[i - B] >= r) {
                ans -= calc(l, r, stk[i], n);
                break;
            }
            ans -= calc(l, stk[i - B], stk[i], stk[i + 1] - 1);
        }
    }
    return ans;
}

namespace BIT {
    ll t1[N], t2[N], sum;

    // 单点更新 [l, r] 的值 v
    inline void update(int l, int r, ll v) {
        sum += (r - l + 1) * v;
        for (int x = l; x <= n; x += (x & -x)) t1[x] += v, t2[x] += v * l;
        for (int x = r + 1; x <= n; x += (x & -x)) t1[x] -= v, t2[x] -= v * (r +
1);
    }

    // 区间查询 [1, r] 的权值之和
    inline ll qry(int r) {
        ll res = 0;
        for (int i = r; i > 0; i -= (i & -i)) res += t1[i] * (r + 1) - t2[i];
        return res;
    }

    // 查询区间 [1, l] 的权值之和
    inline ll query(int l) { return sum - qry(l - 1); }
}

int main() {
    init();

    // 对每次查询进行处理
    for (int i = 1, l, r; i <= q; ++i) {
        read(l); read(r);
        anq[i] = solve(i, l, r);
    }

    // 构建逆序对列表
    for (int j = 1; j <= q; ++j) {
        int l = L[j], r = R[j];
```

```
        if (l > 1) add(l - 1, make_pair(j, r + 1));
        add(r, make_pair(-j, r + 1));
    }

    // 利用树状数组计算每次查询的权值之和
    for (int j = 1; j <= n; ++j) {
        for (int i = 1; i <= mx; ++i) {
            if (ql[i][j] <= qr[i][j]) BIT::update(ql[i][j], qr[i][j], i);
        }
        for (int v = first[j]; v; v = nxt[v]) {
            ll ans = BIT::query(val[v].second);
            if (val[v].first > 0) anq[val[v].first] += ans;
            else anq[-val[v].first] -= ans;
        }
    }

    // 输出结果
    for (int i = 1; i <= q; ++i) putint((anq[i] + mod) % mod, '\n');
    flush();
    return 0;
}
```

其它思路

本题的查询操作使用主席树也可以实现，即用主席树查询给定矩形与这些矩形的交，但实际上主席树常数较大可能无法通过。

本题易错点

要注意所选择的数据结构的常数。

另外，本题的判断条件涉及位运算左移，容易溢出，需要依据数据规模具体判断。

示例如下：

```
//子任务
inline int calc1(int height,int a){
    if(height>=22) return 0x3f3f3f3f;
    return (1<<height+1)-(1<<a);
}
inline int calc2(int height,int a,int b){
    if(height>=22) return 0x3f3f3f3f;
    return (1<<height+1)-(1<<(height-min(b,height-a+1)))-1;
}
//正解
inline int calc(int bef,int height){
    if(height>=22) return 0x3f3f3f3f;
    return (1<<height+1)-(1<<height-bef)-1;
}
inline int space(int bef,int height){
    if(height>=22) return 0x3f3f3f3f;
    return (1<<height+1)-(1<<height-bef+1);
}
```

博弈

题意解释

给定两个人的树上博弈规则。小 N 初始时在节点 S，小 Y 需要通过一些操作“强迫”小 N 最后到达节点 T。小 N 想要让小 Y 的操作尽量多，而小 Y 希望自己的操作尽量少。求解两人都采取最优决策时，小 Y 的操作次数。 每一轮中操作如下：

- 小 Y 可选的操作有：恢复一条小 N 删除的边；或者删除一条边；或者跳过这次操作（不计入答案）。
- 小 N 接下来执行：如果没有相邻的点，则停留在原地；否则选择一条出边走过去，并将走过的边删除。

关键在于，从特殊情况入手，分析两个人采取的最优决策。发现如果小 N 向下走，一定会走到某个叶子节点，这样的答案是容易计算的，进而通过树形DP解决。

知识点提炼

博弈，树形DP，二分答案

核心解题思路

子任务1

$n \leq 10$ 的部分，可以爆搜。每条边有三种状态：在树中，被小 N 删除，以及被小 Y 删除。每个博弈的局面可由边的状态和小 N 所处的位置来表示。按照题目要求进行模拟转移即可。

注意到每条边至多会被小 Y 恢复/删除一次，所以答案不会超过 $2n$。可以通过测试点1。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 11;
int n, s, t;
vector<int> e[N];
int id[N][N], a[N];     //a[]存放边的三种状态
//当前位于pos，剩余res步，轮到先手/后手，must_choose 防止两人都不动，死循环，规定后手不动时先手必须动
int dfs(int pos, int res, int o, int must_choose)
{
    if (res < 0) return 0;
    if (pos == t) return 1;
    //先手
    if (o == 0) {
        int ans = 0;
        //跳过本次操作
        if (!must_choose) ans |= dfs(pos, res, o ^ 1, 0);
        if (ans) return 1;
        //删除边
        for (int i = 0; i < n - 1; ++i)
            if (a[i] == 0) {
```

```
                a[i] = 2;
                ans |= dfs(pos, res - 1, o ^ 1, 0);
                a[i] = 0;
                if (ans) return 1;
            }
        //修复边
        for (int i = 0; i < n - 1; ++i)
            if (a[i] == 1) {
                a[i] = 0;
                ans |= dfs(pos, res - 1, o ^ 1, 0);
                a[i] = 1;
                if (ans) return 1;
            }
        //只要有一种可行，就是 Yes
        return ans;
    }
    else {
        int ans = 1;
        //记录有多少可走的儿子
        int cnt = 0;
        for (auto v : e[pos]) {
            int ID = id[v][pos];
            if (a[ID] == 0) {
                ++cnt;
                a[ID] = 1;
                ans &= dfs(v, res, o ^ 1, 0);
                a[ID] = 0;
                if (!ans) return 0;
            }
        }
        if (cnt == 0) ans &= dfs(pos, res, o ^ 1, 1);
        return ans;
    }
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cin >> n >> t >> s;
    for (int i = 0; i < n - 1; ++i) {
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
        e[v].push_back(u);
        id[u][v] = id[v][u] = i;
    }

    for (int i = 0; i <= n; ++i) {
        if (dfs(s, i, 0, 0)) {
            cout << i << endl;
            return 0;
        }
    }
    return 0;
}
```

子任务2

子任务2具有特殊性质 **存在边** (S,T)。不妨假设这是一个以 T 为根的树，那么小 N 现在的目标是使得到达根节点时，小 Y 的操作尽量多。

存在边 (S,T) 时，如果小 Y 不做任何操作的话，小 N 在第一轮中的最优决策显然是走向 S 的某个儿子（如果存在），并且在接下来每一轮中只能走向某个儿子，直至到达某个叶子节点，之后无路可走。

假设小 N 被困在了某个叶子节点 u 上（ u 到 fa_u 之间的边被小 N 删除），那么接下来小 Y 的最优决策一定是将 T 到 u 路径上的所有分叉边（除 $path(u,T)$ 外的边）断掉，然后将 u 到 T 沿路上被小 N 删除的边恢复，这样就可以强迫小 N 只能一直向上走到 T。

考虑小 N 从 u 向下走到某个儿子的过程，记录 f_v 表示在以 v 为根的子树中，初始时小 N 从 v 出发，最后被迫回到 v 的最小代价。 f_u 是容易 DP 求出的：在 u 的子树中，小 N 一定会选择走 f_v 最大的一个儿子，那么小 Y 就会选择堵住这个儿子，小 N 只能选择儿子中 f 的次大值，于是有：

$$f_u = \mathrm{2nd}\max_{v\in son_u} f_v + |son_u|$$

其中 $|son_u|$ 这部分代价，1 来自堵住最大的 f_v，$|son_u|-1$ 来自向上走时，堵住所有 u 连出去的分叉边，并恢复向上走的这一条路。注意一些特殊情况：当 $|son_u|=1$ 时，直接堵住这一个儿子即可，$f_u=1$。当 $|son_u|=0$ 时，无需操作 $f_u=0$。

那么存在边 (S,T) 时，问题的答案就是 f_S。复杂度 $O(n)$，可以通过测试点 2。

```
#include<bits/stdc++.h>
using namespace std;
namespace iobuff{
    const int LEN=1000000;
    char in[LEN+5],out[LEN+5];
    char *pin=in,*pout=out,*ed=in,*eout=out+LEN;
    inline char gc(void){
        return pin==ed&&(ed=(pin=in)+fread(in,1,LEN,stdin),ed==in)?EOF:*pin++;
    }
    template<typename T> inline void read(T &x){
        static int f;
        static char c;
        c=gc(),f=1,x=0;
        while(c<'0'||c>'9') f=(c=='-'?-1:1),c=gc();
        while(c>='0'&&c<='9') x= 10*x+c-'0',c=gc();
        x*=f;
    }
}
using namespace iobuff;
const int N=1e6+10;
struct node{
    int v,nxt;
}e[N<<1];
int n,rt,s,dp[N],dep[N],fa[N],first[N],cnt,de[N];
inline void add(int u,int v){
    e[++cnt].v=v; e[cnt].nxt=first[u]; first[u]=cnt;
}
//dp[u]表示以v为根的子树中，初始时小N从u出发，最后被迫回到u的最小代价。
//g[u]表示根节点到u的路径中所有分叉边数量
inline void dfs(int u,int f){
    dep[u]=dep[f]+1;
    fa[u]=f;
```

```
    int son=de[u]-(f!=0);
    int mx=0,se=0;
    for(int i=first[u];i;i=e[i].nxt){
        int v=e[i].v;
        if(v==f) continue;
        dfs(v,u);
        //mx记录子树中dp最大值，dp[v]记录次大值
        if(dp[v]>mx) se=mx, mx=dp[v];
        else if(dp[v]>se) se=dp[v];
    }
    dp[u]=se+son;
}
int main(){
    read(n); read(rt); read(s);
    for(int i=1,u,v;i<n;++i){
        read(u); read(v);
        add(u,v); add(v,u); de[u]++; de[v]++;
    }
    dfs(rt,0);
    printf("%d\n",dp[s]);
    return 0;
}
```

子任务3，4

一般情况，小 N 第一轮中不一定直接走向 S 的某个儿子，而有可能先向上走一段距离，在走到一个点后再向子树中走。在向上走的过程中，小 Y 的策略是难以确定的，可能会堵住小 N 向上走的路，也有可能选择堵住一个 f 值较大的儿子，放小 N 继续向上走。

考虑将最优化问题转为判定问题。固定小 Y 的操作次数 T，判断能否在 T 次操作内结束博弈。 用 g_u 表示根节点到 u 的路径中所有分叉路数量，可以通过一次 dfs 预处理得到 g。

第一轮中，如果存在 $v \in son_S$，使得 $f_v + g_S \leq T$，对于小 Y 而言这样的儿子必须全部被堵住，接下来小 N 唯一的策略就是沿着父亲向上走。注意到小 Y 是不能主动删除向父亲走的边的，因为按照题目条件这样的边无法恢复。在接下来的一轮中类似，如果小 Y 没有需要堵的儿子，那么就将自己的操作“预留”下来，等到某个祖先需要时再使用。

这样，模拟一次小 N 从 S 跳到根的过程，判断两个条件：其一是由于小 N 和小 Y 是轮流操作的，如果某一时刻需要堵的儿子总数超出了小 Y 当前可以操作的轮数，那么不可行；其二是如果操作总数超过 T，那么不可行。

注意到答案的一个上界是 n，因为总可以等待小 N 自己走到一个叶子节点后，再开始执行操作，每条边至多需要恢复/删除一次。这样暴力枚举是 $O(n^2)$ 的，可以通过子任务3。

一个明显的性质是答案具有单调性，所以二分答案并按照上述方法检查，复杂度为$O(n \log n)$，可以通过子任务4，预期得分 100 分。

```
#include<bits/stdc++.h>
using namespace std;
namespace iobuff{
    const int LEN=1000000;
    char in[LEN+5],out[LEN+5];
    char *pin=in,*pout=out,*ed=in,*eout=out+LEN;
    inline char gc(void){
        return pin==ed&&(ed=(pin=in)+fread(in,1,LEN,stdin),ed==in)?EOF:*pin++;
    }
```

```
    template<typename T> inline void read(T &x){
        static int f;
        static char c;
        c=gc(),f=1,x=0;
        while(c<'0'||c>'9') f=(c=='-'?-1:1),c=gc();
        while(c>='0'&&c<='9') x= 10*x+c-'0',c=gc();
        x*=f;
    }
}
using namespace iobuff;
const int N=1e6+10;
struct node{
    int v,nxt;
}e[N<<1];
int n,rt,s,dp[N],dep[N],g[N],fa[N],first[N],cnt,de[N];
inline void add(int u,int v){
    e[++cnt].v=v; e[cnt].nxt=first[u]; first[u]=cnt;
}
//dp[u]表示在以v为根的子树中，初始时小N从u出发，最后被迫回到u的最小代价。
//g[u]表示根节点到u的路径中所有分叉边数量
inline void dfs(int u,int f){
    dep[u]=dep[f]+1;
    fa[u]=f;
    int son=de[u]-(f!=0);
    if (f) g[u]+=son-1;
    int mx=0, se=0;
    for(int i=first[u];i;i=e[i].nxt){
        int v=e[i].v;
        if (v==f) continue;
        g[v]=g[u];
        dfs(v,u);
        //mx记录子树中dp最大值，dp[v]记录次大值
        if (dp[v]>mx) se=mx, mx=dp[v];
        else if (dp[v]>se) se=dp[v];
    }
    dp[u]=se+son;
}
//判断能够用x次操作结束
inline bool check(int x){
    int u=s,las=-1, pre=0, dis=0;
    while(u!=rt){
        int nw=pre;
        for(int i=first[u];i;i=e[i].nxt){
            int v=e[i].v;
            //计算每棵子树的代价，包括之前走过的步数
            //不能走来时的路，起点处要特殊处理
            if(v!=las && v!=fa[u] && dp[v]+g[u]+nw+1-(las!=-1)>x) pre++;
        }
        //如果需要堵住的子树个数pre超过当前轮数dis，或者超过总步数x，答案为No
        if (pre>dis+1||pre>x) return false;
        las=u; u=fa[u]; dis++;
    }
    //可以一路跳到根上，那么是Yes
    return true;
}
int main(){
```

```
    read(n); read(rt); read(s);
    for(int i=1,u,v;i<n;++i){
        read(u); read(v);
        add(u,v); add(v,u); de[u]++; de[v]++;
    }
    //dfs求出dp,g
    dfs(rt,0);
    int l=0, r=1e9, ans=0;
    while(l<=r){
        int mid=(l+r)>>1;
        if(check(mid)) ans=mid, r=mid-1;
        else l=mid+1;
    }
    printf("%d\n",ans);
    return 0;
}
```

本题易错点

本题的难点和易错点主要在于思考两人所采取的最优决策，比较容易忽略小 N 可以选择先向上走几步的决策。

另外，直接进行最优化问题的求解非常复杂，转化为判定性问题之后就会迎刃而解。

地雷

题意解释

一排地雷，每个地雷都有四个属性 p_i, q_i, r_i, s_i，拆除第 i 个地雷的代价为 $(p_{i-1}-q_i)^2+(p_i-r_{i+1})^2+(p_{i+1}-s_{i+2})^2$。拆除一个地雷后，该地雷左右两侧的地雷会连接到一起。在最坏的情况下，需要花多少代价才能拆除所有地雷。

关键在于要发现拆除一个地雷，实际上相等于把左右两个区间连接起来，代价来自于左右两边未被拆除的雷，然后一起对接下来的拆除造成影响，与区间DP的转移方式一致。

知识点提炼

区间DP

核心解题思路

子任务1

对于$n \leq 20$的情形，可以直接$O(n \times 2^n)$枚举所有可能的情形，求出最大代价。

期望得分20分。

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 80, mxn = 20;
int n, p[maxn], q[maxn], r[maxn], s[maxn];
```

```
long long f[10+(1<<mxn)];   //f[i]表示拆 i 需要的最大代价
int cnt, a[30], tag[10 + (1 << mxn)];   //在某状态下，没炸的地雷标号
void get(int x)     //统计在x状态下还没炸过的地雷
{
    cnt = 0;
    for (int i = 0; i < n; i++)
        if (x & (1 << i))       // i 号地雷没炸过
        {
            cnt++;
            a[cnt] = i + 1;     //标记这颗地雷
            tag[i + 1] = x;
        }
    a[cnt + 1] = n+1;   //若右侧没有地雷，等价于有一个各项属性均为0的地雷
    a[cnt + 2] = n+2;
}
int main()
{
    memset(tag, -1, sizeof(tag));
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> p[i];
    for (int i = 1; i <= n; i++) cin >> q[i];
    for (int i = 1; i <= n; i++) cin >> r[i];
    for (int i = 1; i <= n; i++) cin >> s[i];
    int m = 1 << n;
    f[0] = 0;
    for (int i = 0; i < m; i++)
    {
        get(i);
        int tot = 0;
        for (int j = 1; j <=n; j++)
            if (!((1 << j - 1) & i))    //没炸过
            {
                while (tot <= cnt && a[tot + 1] < j)
                    ++tot;
                long long sum = 0;
                //统计代价
                sum += (p[a[tot]] - q[j]) * (p[a[tot]] - q[j]);
                sum += (p[j] - r[a[tot + 1]]) * (p[j] - r[a[tot + 1]]);
                sum += (p[a[tot + 1]] - s[a[tot + 2]]) * (p[a[tot + 1]] - s[a[tot
+ 2]]);
                //更新答案
                f[i | (1 << j - 1)] = max(f[i | (1 << j - 1)], f[i] + sum);
            }
    }
    cout << f[m - 1];
    return 0;
}
```

子任务2

考虑区间 dp，每次枚举区间内最晚被删除的数。

设 $f_{i,j}$ 表示区间 $[i,j]$的最大删除代价。

转移枚举区间内最晚删除的数 k。有转移：

$$f_{i,j}=\max_{i\le k\le j}(f_{i,k-1}+f_{k+1,j}+val(i-1,k,j+1))$$

其中

$$val(a,b,c)=(p_a-q_b)^2+(p_b-r_c)^2+(p_c)^2$$

复杂度 $\mathcal{O}(n^3)$。

利用了$s_i=0$的特殊性质，期望得分20分。

```
#include<bits/stdc++.h>
using namespace std;
const int N=72;
int n,p[N],q[N],r[N],s[N],f[N][N];
inline int S(int x){return x*x;}
inline int solve(int a,int b,int c){    //计算排除第a个地雷的代价
    return S(p[a]-q[b])+S(p[b]-r[c])+S(p[c]);
}
int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;++i) scanf("%d",&p[i]);
    for(int i=1;i<=n;++i) scanf("%d",&q[i]);
    for(int i=1;i<=n;++i) scanf("%d",&r[i]);
    for(int i=1;i<=n;++i) scanf("%d",&s[i]);
    memset(f,-0x3f,sizeof(f));
    //初始化
    for(int i=1;i<=n+1;++i){
        f[i][i-1]=0;
        f[i][i]=solve(i-1,i,i+1);
    }
    //从长度为2开始依次计算长度为len的区间的f值
    for(int len=2;len<=n;++len){
        for(int l=1;l+len-1<=n;++l){
            int r=l+len-1;      //区间的右边界
            int &g=f[l][r];     //引用当前f[l][r]的值
            for(int mid=l;mid<=r;++mid){
                int tt=solve(l-1,mid,r+1);
                g=max(g,tt+f[l][mid-1]+f[mid+1][r]);
            }
        }
    }
    printf("%d\n",f[1][n]);
    return 0;
}
```

子任务3、4

依旧考虑区间 dp，每次枚举区间内最晚被删除的数。

设 $f_{i,j,t,u}$ 表示当前考虑区间 $[i,j]$，满足 $i-1,j+1$ 都比区间 $[i,j]$ 更晚删除，在 $j+1$ 后第一个比 $[i,j]$ 更晚删除的数编号是 t（也就是枚举了区间 $[i,j]$ 内最晚被删除的数 k 后，钦定删除 k 时 k 右侧第二个数是 t）。这样钦定会导致一些限制，于是 u 表示限制了区间中在 u 左侧的数都比 u 更早删除。具体为什么这样限制参见下方转移。
f 就是满足以上条件时区间 $[i,j]$ 的最大删除代价。

转移枚举区间内最晚删除的数 k 以前左侧区间对应的 t（设为 v），根据 u 的定义，k 必须 $\geq u$。而 v 显然不会超过 $j+1$。有转移：

$$f_{i,j,t,u} = \max_{u \leq k \leq j, k+1 \leq v \leq j+1} (f_{i,k-1,v,u} + f_{k+1,j,t,v} + val(i-1,k,j+1,t))$$

其中

$$val(a,b,c,d) = (p_a - q_b)^2 + (p_b - r_c)^2 + (p_c - s_d)^2$$

解释：
钦定了 v 作为左区间的 t，即要求删除 $[i,k-1]$ 最后一个数时，其右侧第二个数为 v。因此 $[k+1,v]$ 中的数一定都比 v 更早删去。（也就是引入限制 u 的原因）而只要满足了这一限制，就可以通过合理地合并两个子区间内部的删除顺序得到一个满足 $t=v$ 的条件的 $[i,j]$ 删除顺序。

对于向右区间的转移，直接将大区间的 t 继承到了右区间，实际上这里有可能发生变化，比如类似31452(数字小表示先删除)。但选择让 t 不变一定不劣，因为 t 只影响 $p_{j+1} - s_t$ 这一项。$j+1$ 没有变化，因此最优的 t 也不会发生变化。（如果这里选择改变为 t'，那么在大区间一定也可以改变为 t'，因此 t' 不优于 t）

复杂度 $\mathcal{O}(n^6)$，但常数非常小，可以通过。

若没有发现选择让 t 不变一定不劣，而选择再循坏一维，则无法通过子任务4。

期望得分100分。

```
#include<bits/stdc++.h>
using namespace std;
const int N=72;
int n,p[N],q[N],r[N],s[N],f[N][N][N][N];    //f[l][r][x][y]：某一时刻x成为最左边的数
inline int S(int x) { return x*x; }
inline int solve(int a, int b, int c, int d) {    //计算排除第a个地雷的代价
    return S(p[a] - q[b]) + S(p[b] - r[c]) + S(p[c] - s[d]);
}
int main() {
    scanf("%d", &n);
    for(int i=1; i<=n; ++i) scanf("%d", &p[i]);
    for(int i=1; i<=n; ++i) scanf("%d", &q[i]);
    for(int i=1; i<=n; ++i) scanf("%d", &r[i]);
    for(int i=1; i<=n; ++i) scanf("%d", &s[i]);
    memset(f, -0x3f, sizeof(f));
    //初始化
    for(int i=1; i<=n+1; ++i){
        f[i][i-1][i-1][i] = 0;
        for(int j=i+1; j<=n+1; ++j){
            f[i][i][i][j] = f[i][i][i-1][j] = solve(i-1, i, i+1, j);
            f[i][i-1][i-1][j] = 0;
        }
    }
    //从长度为2开始依次计算长度为len的区间的f值
    for(int len=2; len<=n; ++len){
        for(int l=1; l+len-1<=n; ++l){
            int r = l + len - 1;    //区间的右边界
            for(int x=l-1; x<=r; ++x){
                for(int y=r+1; y<=n+1; ++y){
                    int &g = f[l][r][x][y];     //引用当前f[l][r][x][y]的值
                    for(int mid=max(x,l); mid<=r; ++mid){
```

```
                        int tmp = (mid==x) ? l-1 : x;        //选择合理的限制
                        int tt = solve(l-1, mid, r+1, y);    //计算排除第mid个地雷的
代价
                        //计算当前情况下f[l][r][x][y]的最大值
                        g = max(g, tt + f[l][mid-1][tmp][r+1] + f[mid+1][r][mid]
[y]);
                        for(int t=mid+1; t<=r; ++t)
                            g = max(g, tt + f[l][mid-1][tmp][t] + f[mid+1][r][t]
[y]);
                    }
                }
            }
        }
    }
    printf("%d\n", f[1][n][0][n+1]);
    return 0;
}
```

本题易错点

若没有发现选择让 t 不变一定不劣，而选择再循坏一维，会导致效率降低。

若没有发现当s_i不为0时，会对其它区间的删除顺序造成影响，会导致无法正确计算答案。