算经

——《编程题集》第 II 版

weiyong1024@gmail.com

感谢我的好朋友：吕甘霖、徐英浩、姜予名
硕士生导师杨晨阳教授
Megvii 的同学们
永远的融融
以及修行路上所有帮助过我的人

第一章：基础篇

——常用经典算法、数论函数和数据结构的 Cpp 实现

1.1 Sieve of Eratosthenes / 埃拉托色尼筛选法
1.2 Binary search / 二分搜索
1.3 Ternary search / 三分搜索
1.4 Greatest cammon divisor / 最大公约数
1.5 Quick power/快速幂
1.6 Combination / 模 1000000007 意义下的组合数
1.7 Inverse % MOD / 模 1000000007 意义下前 N 个正整数的乘法逆元
1.8 Binary index tree / Fenwick tree / 二进制索引树
1.9 Segment tree / 线段树
    1.9.1 区间和线段树
    1.9.2 区间最大值线段树
    1.9.3 区间最小值线段树
    1.9.4 线段树的 C++类模版
1.10 Shortest path / 最短路径
    1.10.1 Dijkstra/单源最短路径
    1.10.2 Floyd/所有点对间最短路径
1.11 Minimum spanning tree (MST)/最小生成树
1.12 Local sorting/本地排序
    1.12.1 Quick sort/快速排序
    1.12.2 Merge sort/归并排序
1.13 Pattern matching/模式匹配
    1.13.1 Knuth-Morris-Pratt /KMP 算法
    1.13.2 Boyer-Moore/BM 算法
1.14 LRUcache/最常访问缓存
1.15 Union-Find/并查集
1.16 Tire/字典树
1.17 Brian-Kernighan/统计正整数二进制表示中 1 的数量
1.18 Minimum cover circle / 最小圆覆盖
1.19 Colouring / 图的顶点着色问题

1.1 Sieve of Eratosthenes / 埃拉托色尼筛选法

寻找n以内所有素数。一句话描述：建立n长空表，p从2到n递增，若当前元素未被标记，则记录该元素为素数，并标记该元素在n以内的所有倍数为合数。据 Codeforces 水友所述该算法的时间复杂度为$T\sim O(n\log\log n)$

核心代码:

```
const int N = 10000005;
bool vis[N];
std::vector<int> Eratosthenes_sieve() {
    std::vector<int> primes;
    for (int i = 2; i < N; i++) {
        if (!vis[i]) primes.push_back(i);
        for (int j = i << 1; j < N; j += i)
            if (!vis[j]) vis[j] = true;
    }
    return primes;
}
```

上述代码在 C02TR2TEHTD7 上运行，N为10^7时运行时间约1~2s，N为10^8时运行时间约17s。

应用 1：生成N以内所有整数的最小素因数。

核心代码:

```
const int N = 10000005;
int min_fac[N];
void Eratosthenes_sieve() {
    for (int i = 2; i < N; i++) {
        if (min_fac[i] == 0) min_fac[i] = i;
        for (int j = i << 1; j < N; j += i)
            if (min_fac[j] == 0) min_fac[j] = i;
    }
}
```

1.2 Binary search/二分搜索

· 用于寻找阈值的二分搜索

若问题具有大于等于某一阈值“可行”，小于该阈值“不可行”的二段性，或者属于寻找满足条件的最小 XX 数量问题，则用寻找阈值的二分搜索求解。

核心代码（check()函数用于判定当前值是否“可行”）：

连续型：

```
double bi_search() {
    double l = 0, r = 1e9;
    for (int i = 0; i < 300; i++) [
        double m = (l + r) / 2l;
        if (check(m)) r = m;
        else l = m;
    }
    return l;
}
```

离散型（$[l,r]$维护的是剩余搜索域）：

```
int bi_search() {
    int ans = −1;
    int l = 0, r = n − 1;
    while (l <= r) {
        int m = (l + r) >> 1;
        if (check(m)) ans = m, r = m − 1;
        else l = m + 1;
    }
    return ans;
}
```

· 用于判存在性的二分搜索

以“判断数组$a[0 \ldots n-1]$中是否存在值为tar的项”为例，$[l,r]$维护的是剩余搜索域。

```
bool bi_search(int tar) {
    int l = 0, r = n − 1;
    while (l <= r) {      // Search range [l, r].
        int mid = (l + r) >> 1;
        if (a[mid] == tar) return true;
        else if (a[mid] < tar) l = mid + 1;
        else r = mid − 1;
    }
    return false;
}
```

1.3 Ternary search/三分搜索

· 标准三分搜索:

当需要求解**单峰连续函数**$f(x)$在区间$[L,R]$上的极/最值时，可使用标准三分搜索算法。算法每次迭代将原始区间向目标点缩进$\frac{2}{3}$，理论上最终收敛到极值点或最值线段左端（实际中由于浮点误差会出现判等错误，故当原函数存在最值线段时，最终定位往往为线段内的随机点，但返回的最值仍有效）。

在$[-1000,1000]$上搜索单峰连续函数$f(x)$极小值的代码:

```
double ternary_search() {
    double lo = -1000, hi = 1000;
    for (int i = 0; i < 300; i++) {
        double m1 = (2 * lo + hi) / 3;
        double m2 = (lo + 2 * hi) / 3;
        if (f(m1) >= f(m2)) lo = m1;
        else hi = m2;
    }
    return f(lo);
}
```

注: 迭代 100 次后区间长度为原来的2.4597×10^{-18}倍，而迭代 300 次后区间长度为原来的1.4881×10^{-53} 倍。

· 二维三分搜索/平面三分搜索:

当需要求解单封连续二元函数$f_{x,y}(x,y)$在矩形区域$x \in [L_x,R_x], y \in [L_y,R_y]$上的级/最值时，可使用平面三分搜索算法。具体地，只需分两层调用标准三分搜索算法即可找到极值。从几何意义来看，平面三分搜索的过程是利用三分搜索求解固定x时函数在y方向上的极值，它是一个x的函数$f_x(x)$，然后利用三分搜索求解$f_x(x)$在x方向上的极值。

以在$[-1000,1000] \times [-1000,1000]$上搜索单峰函数$f(x)$极小值为例，代码如下:

```
double t_search_y(double x) {
    double lo = -1000, hi = 1000;
    for (int i = 0; i < 300; i++) {
        double m1 = (2 * lo + hi) / 3;
        double m2 = (lo + 2 * hi) / 3;
        if (f(x, m1) >= f(x, m2)) lo = m1;
        else hi = m2;
    }
    return f(x, lo);
}
double t_search_x() {
    double lo = -1000, hi = 1000;
    for (int i = 0; i < 300; i++) {
        double m1 = (2 * lo + hi) / 3;
        double m2 = (lo + 2 * hi) / 3;
        if (t_search_y(m1) >= t_search_y(m2)) lo = m1;
        else hi = m2;
    }
    return t_search_y(lo);
}
```

常见**数论函数**:

1.4 gcd/最大公约数

算法：对于每轮迭代,
大数，小数 = 小数，大数对小数的余数

· std 命名空间标准库中<algorithm>库中的函数 std::__gcd()
（2017.9.24 更新）：这个并不是 C++标准库中的函数，在 Xcode 更新到 9.0 后该函数无法运行。

· 基于 Euclid 算法的 C 语言实现:
迭代版

```
int gcd ( int a, int b ) {
   int c;
   while (a != 0)
        c = a, a = b % a, b = c;
   return b;
}
```

递归版

```
int gcdr ( int a, int b ) {
      if (a == 0) return b;
      return gcdr (b % a, a);
}
```

欧几里德算法的时间复杂度为$T\sim O(\max(a,b))$。

1.5 Quick-power/快速幂

在模MOD意义下求幂值a^b，$T\sim O(\log b)$。

算法：对于每轮迭代，若b是偶数，则将指数减半，否则分离出一个底数到累乘器，再将指数减半。

核心代码：

```
const int MOD = 1000000007;

int qpw(int a, int b) {    // Quick power for a^b.
    int ans = 1;
    while (b) {
        if (b & 1) ans = 1ll * ans * a % MOD;
        a = 1ll * a * a % MOD;
        b >>= 1;
    }
    return ans;
}
```

1.6 模1000000007意义下的组合数C_n^m

利用 1.5 节快速幂算法，求$maxn = 200000$以内的组合数。

在主程序执行前运行 preProcess()函数以获得阶乘及其逆元的备忘录。

核心代码:

```
const int maxn = 200000;
const int MOD = 1000000007;

int qpw(int a, int b) {     // Quick power for a^b.
    int ans = 1;
    while (b) {
        if (b & 1) ans = 1ll * ans * a % MOD;
        a = 1ll * a * a % MOD;
        b >>= 1;
    }
    return ans;
}

int fac[maxn + 1], facinv[maxn + 1];
void pre_process() {
    fac[0] = 1;
    for (int i = 1; i <= maxn; i++)
        fac[i] = 1ll * fac[i - 1] * i % MOD;
    facinv[maxn] = qpw(fac[maxn], MOD - 2);
    for (int i = maxn - 1; i >= 0; i--)
        facinv[i] = 1ll * facinv[i + 1] * (i + 1) % MOD;
}

int C(int x, int y) {     // Combination in MOD: C_x^y.
    if(x < y) return 0;
    return 1ll * fac[x] * facinv[y] % MOD * facinv[x - y] % MOD;
}
```

注：以上组合数函数的输入满足$0 \le y \le x \le 200000$时会输出有意义的结果。当$x < y$时输出为 0（也是有意义的不是吗？）。

注 2：在模1000000007意义下组合数的返回值在$[0, 10000000006]$之间，故返回一个 int32 型变量即可。

根据费马小定理：$a^{p-1} \equiv 1 (\mathrm{mod}\, p)$

上式可拆成：$a \times a^{p-2} \equiv 1 (\mathrm{mod} p)$，可见$a^{p-2}$是$a$在$\mathrm{mod} p$意义下的逆元。

1.7 Inverse % MOD / 模 1000000007 意义下前 maxn 个数的乘法逆元

对于任意正整数x，模 1000000007 意义下的乘法逆元基于费马小定理可以用快速幂在$T\sim O(\log MOD)$的时间内求出——即$x^{-1}=qpw(x,MOD-2)$。

利用**动态规划**可以在$O\sim T(maxn)$时间内求出前 maxn 个正整数的乘法逆元，存储在 inv[maxn + 1]数组中。

假如我们要求解x在 mod p意义下的逆元，则将p分成x的余数和倍数两部分:

$$p=(p\ \%\ i)+\lfloor p/i\rfloor=a+b*x$$

于是有:

$$a+b*x\equiv 0\mathrm{mod}(p)$$

将a移到右边:

$$b*x\equiv -a\mathrm{mod}(p)$$

进而得到了最优子结构:

$$x^{-1}=(-b)\times a^{-1}\mathrm{mod}(p)=(p-\left\lfloor\frac{p}{i}\right\rfloor)\times(p\%i)^{-1}\mathrm{mod}(p)$$

由于$(p\%i)<x$，故在求解x^{-1}时，$(p\%i)^{-1}$是已经求解过的重叠子问题，可以直接使用。

完整代码:

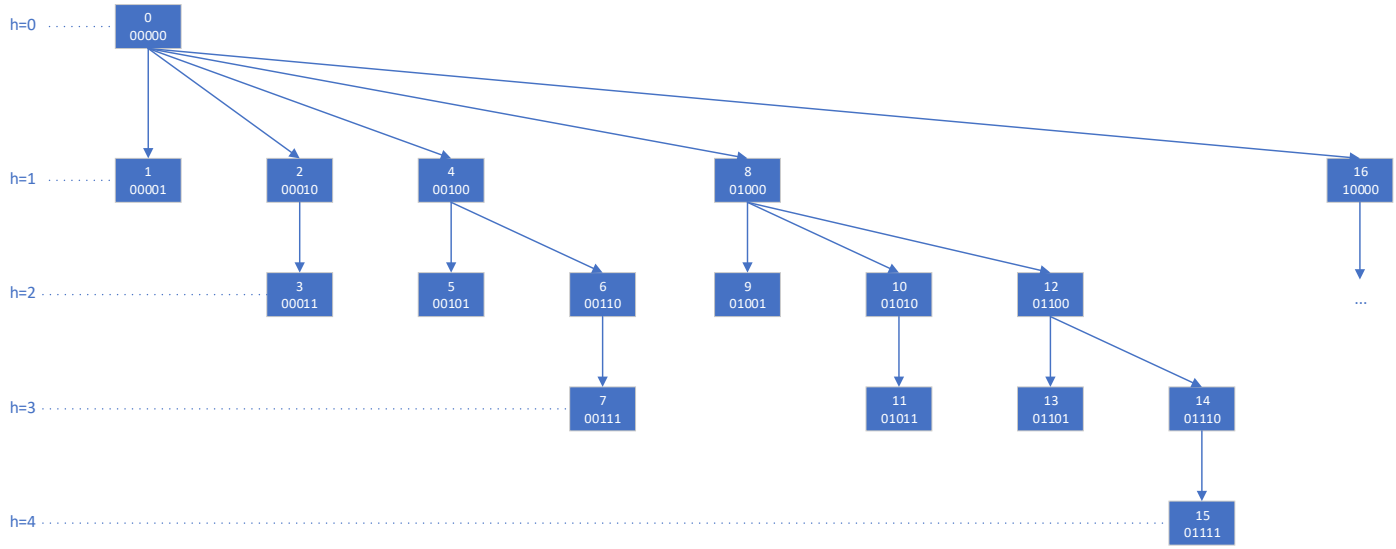
```
#include <bits/stdc++.h>

using namespace std;
using ll = long long;

const int maxn = 1000005;
const ll MOD = 1000000007;

ll qpw(ll a, ll b) {
    ll ans = 1;
    while (b) {
        if (b & 1) ans = ans * a % MOD, b--;
        a = a * a % MOD;
        b >>= 1;
    }
    return ans;
}

ll inv[maxn + 1];
void pre_process() {
    inv[1] = 1;
    for (int i = 2; i <= maxn; i++)
        inv[i] = (MOD - MOD / i) * inv[MOD % i] % MOD;
}
```

1.8 Binary index tree / Fenwick / 二进制索引树

BIT 支持在$O(\log n)$时间内更新改变数组中指定元素的值和计算数组前缀和的操作。

变量声明:
arr[N]——原始数组
BIT[N + 1]——树状数组

原始数组下标使用范围: $[0, n-1]$
树状数组下标使用范围: $[1, n]$



算法:

树中节点编号 i 代表 arr[]中前 i 个元素。每个节点中存储的是相比父节点多出来的那一部分元素的和。故对于任意节点，其回溯到根节点的路径上的元素和就是原数组的前缀。

注: 若x为正，则$x\&-x$为x低位1所代表的数。

1.8.1 BIT 与前缀和

支持操作:

· 改变原数组指定元素的值 arr[u] += v

add(u, v);

· 计算原数组前 u+1 个元素的和，即 arr[]下标在[0, u]的元素和

psum(u)

· 建立 BIT（根据 **arr[0...n − 1]** 的内容向 **BIT[1...n]** 添值）

build()

核心代码（基本功能）:

```
const int maxn = 100;

int arr[maxn];
int n;

int bit[maxn + 1];

void add(int u, int v) {      // Point upd: arr[u] += v.
    for (u++; u <= n; u += u & -u)
        bit[u] += v;
}

int psum(int u) {      // Sum of arr[0...u].
    int ans = 0;
    for (u++; u; u -= u & -u)
        ans += bit[u];
    return ans;
}

void build() {      // Build tree.
    for (int i = 0; i < n; i++)
        add(i, arr[i]);
}
```

扩展功能:

· 计算区间和（利用前缀和做差）:

```
int range_sum(int l, int r) {   // Sum of [arr[l], arr[r]).
    return prefix_sum(r - 1) - prefix_sum(l - 1);
}
```

与线段树的区别:

对于 BIT 和线段树的区别，从树状数组索引更新的方式可见端倪:

BIT 的索引更新方式为减去或加上最低位1，线段树的索引更新方式为左移或右移1位。这导致 BIT 的**父子节点所表征的区间没有交集**，一枝中的所有节点一起构成一个索引所决定前缀的内容；而线段树的父子节点之间是有重合的，甚至父节点p存储的值就是由两个儿子节点（$p << 1$和$p << 1 | 1$）直接决定的。这也解释了为什么 BIT 不支持**维护**区间最值，而线段树却支持，因为线段树为此**额外**付出了$S\sim O(n)$的空间复杂度。

注：BIT 的空间复杂度$S\sim O(n)$，线段树的空间复杂度$S\sim O(2n)$。

1.8.2 BIT 的类模版:

完整代码:

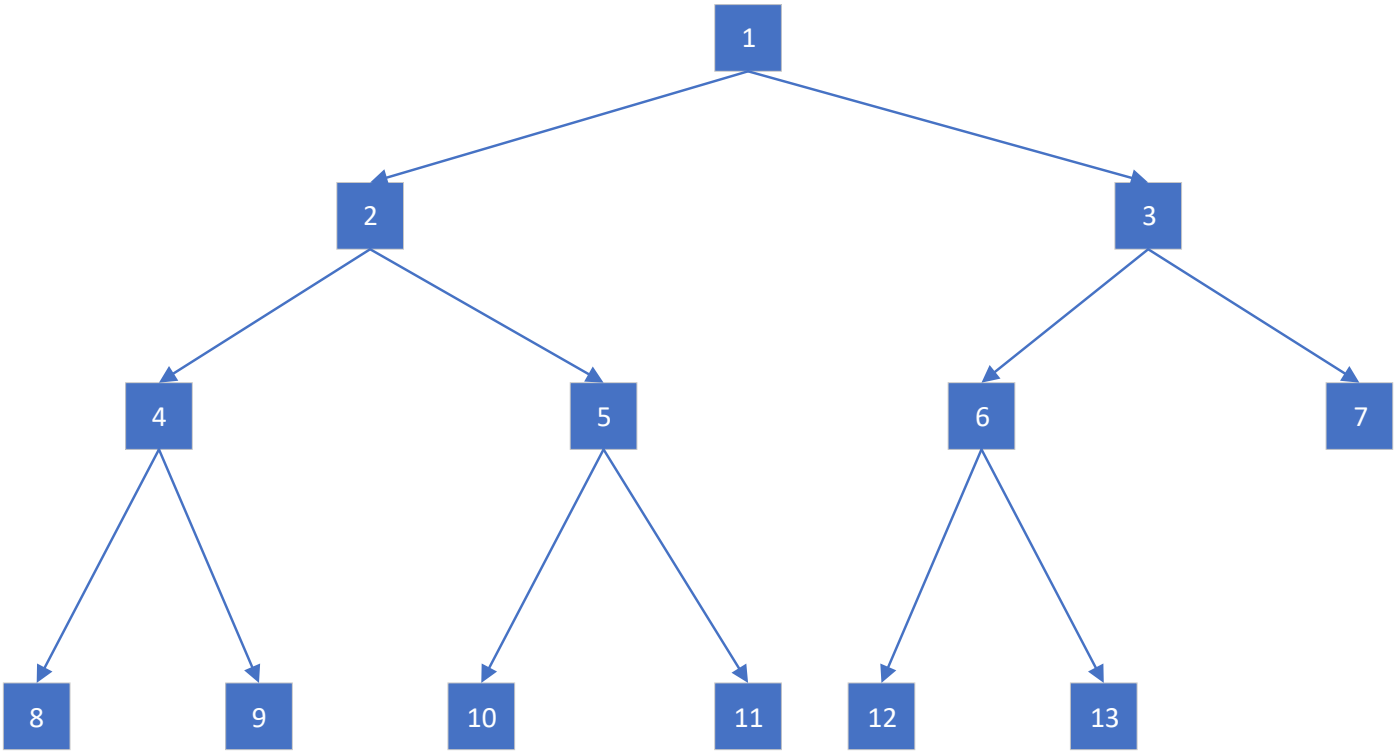
```
template<typename T> class BIT {
public:
    BIT(int _sz) : sz(_sz) {
        bit = vector<T>(sz + 1, 0);
    }
    void add(int x, T val) {
        for (x++; x <= sz; x += x & -x)
            bit[x] += val;
    }
    T query(int x) {      // Sum [0, x]
        T ans = 0;
        for (x++; x; x -= x & -x)
            ans += bit[x];
        return ans;
    }
    T query(int l, int r) { // Sum [l, r)
        return query(r - 1) - query(l - 1);
    }
private:
    int sz;
    vector<T> bit;
};
```

1.9 Segment tree / 线段树

相较 BIT，线段树除了支持在$O(\log n)$时间内更新制定元素和计算区间和以外，还支持在$O(\log n)$时间内计算区间最值，然而却为此多付出了的$O(n)$空间开销。

变量声明:
arr[N]——原始数组
SGMT[N << 1]——树状数组

说明:
原始数组下标使用范围：$[0, n-1]$
树状数组下标使用范围：$[1, 2n-1]$



算法:

线段树是一棵有奇数个节点的完全二叉树。

由于线段树的后半段（叶节点）存储了原数组的内容，故线段树的操作无论建立、求和或者更新值都是先定位到叶节点然后自底向上进行。

具体地，区间和操作是先将表征左右端点指针定位在也节点相应位置，然后一步步上移紧缩，过程每当端点出现奇数则需要“接漏”——其中左端是闭区间所以先加再缩，右端是开区间所以先缩再加。

1.9.1 区间和线段树

支持操作:

· 改变原数组指定元素的值 arr[u] = v

upd(u, v);

· 计算原数组 arr 下标为$[u,v)$区间的元素和

range_sum(u, v);

· 建立线段树（根据 **arr[0...n − 1]** 的内容向 **SGMT[1...2 * n − 1]** 添值）

build();

核心代码:

```
const int maxn = 105;

int arr[maxn];
int n;

int SGMT[maxn << 1]; // Segment tree for arr[N].

void build() {   // Build segment tree SGMT[maxn << 1] for arr[maxn].
    for (int i = 0; i < n; i++) // Set leaf nodes. - Copy arr[0...n - 1] to the second half SGMT[n...2 * n - 1].
        SGMT[n + i] = arr[i];
    for (int i = n - 1; i; i--) // Generate non-leaf nodes. – Fill SGMT[1...n - 1] according to leaf nodes SGMT[n...2 * n - 1].
        SGMT[i] = SGMT[i << 1] + SGMT[i << 1 | 1];
}

int range_sum(int l, int r) { // Sum query: [ arr[l], arr[r] ).
    int ans = 0;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) { // 'l' tracks on the left side of unvisited range, 'r - 1' tracks on the right side.
        if (l & 1) ans += SGMT[l++]; // Update result according to left visited range.
        if (r & 1) ans += SGMT[--r]; // Update result according to right visited range.
    }
    return ans;
}

void upd(int p, int val) {   // Point upd: arr[p] = val.
    for (SGMT[p += n] = val; p > 1; p >>= 1)   // Set leaf node new value, then back track to root.
        SGMT[p >> 1] = SGMT[p] + SGMT[p ^ 1];
}
```

注：对于增量式的点更新操作，只需将更新函数给叶节点赋值改为给叶节点增量即可（需要改动的部分在代码中标红）。

1.9.2 区间最大值线段树

支持操作:

· 改变原数组指定元素的值 arr[u] = v

upd(u, v);

· 返回原数组 arr 下标为$[u,v)$区间的元素的最大值

query_max(u, v);

· 建立线段树（根据 **arr[0...n - 1]** 的内容向 **SGMT[1...2 * n - 1]** 添值）

build();

核心代码:

```
const int N = 105;
const int INF = 1000000;

int arr[N];
int n;

int SGMT[N << 1]; // Segment tree for arr[N].

void build() {   // Build segment tree SGMT[N << 1] for ret[N].
    for (int i = 0; i < n; i++) // Set leaf nodes. - Copy ret[N] to second half of SGMT[N << 1].
        SGMT[n + i] = arr[i];
    for (int i = n - 1; i; i--) // Generate non-leaf nodes.
        SGMT[i] = std::max(SGMT[i << 1], SGMT[i << 1 | 1]);
}

int query_max(int l, int r) { // Ran max query: [ arr[l], arr[r] ).
    int ans = -INF;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) { // 'l' tracks on the left side of unvisited range, 'r - 1' tracks on the right side.
        if (l & 1) ans = std::max(ans, SGMT[l++]); // Update result according to left visited range.
        if (r & 1) ans = std::max(ans, SGMT[--r]); // Update result according to right visited range.
    }
    return ans;
}

void upd(int p, int val) {   // Point upd: arr[p] <- val.
    for (SGMT[p += n] = val; p > 1; p >>= 1)   // Set leaf node new value, then back track to root.
        SGMT[p >> 1] = std::max(SGMT[p], SGMT[p ^ 1]);
}
```

注: 点更新若从赋值式改为增量式, 修改方式同前（标蓝语句的赋值运算符改为+=）。

1.9.3 查询区间最小值的线段树

在查询最大值版本的基础上，只需改变相应**函数名**、所有**取最大值**函数变成**取最小值**、**最值记录变量**的初值改变即可（需要改动的部分在代码中标红）。

核心代码:

```
const int N = 105;
const int INF = 1000000;

int arr[N];
int n;

int SGMT[N << 1]; // Segment tree for arr[N].

void build() {  // Build segment tree SGMT[N << 1] for ret[N].
    for (int i = 0; i < n; i++) // Set leaf nodes. - Copy ret[N] to second half of SGMT[N << 1].
        SGMT[n + i] = arr[i];
    for (int i = n - 1; i; i--) // Generate non-leaf nodes.
        SGMT[i] = std::min(SGMT[i << 1], SGMT[i << 1 | 1]);
}

int RMinQ(int l, int r) { // Ran max query: [ arr[l], arr[r] ).
    int ans = INF;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) { // 'l' tracks on the left side of unvisited range, 'r - 1' tracks on the right side.
        if (l & 1) ans = std::min(ans, SGMT[l++]); // Update result according to left visited range.
        if (r & 1) ans = std::min(ans, SGMT[--r]); // Update result according to right visited range.
    }
    return ans;
}

void upd(int p, int val) {  // Point upd: arr[p] <- val.
    for (SGMT[p += n] = val; p > 1; p >>= 1)  // Set leaf node new value, then back track to root.
        SGMT[p >> 1] = std::min(SGMT[p], SGMT[p ^ 1]);
}
```

以上所有原始代码（点赋值更新的求区间和的线段树、点赋值更新的求区间最大值饿线段树）、修改代码（点增量更新、求区间最小值）均测试可用。

关于为什么数组表示的二叉树一般不使用下标0:

从上至下，从左至右对一棵满二叉树编号，如跟节点编号为1，则节点p的左右儿子和父亲分别为$p << 1, p << 1|1, p >> 1$，节点之间的转移可以只通过位运算完成。

然而若根节点从0开始编号，则节点 p 的左右儿子分别为$2p + 1, 2p + 2$，父亲的索引与p的奇偶性有关，索引转移因而复杂得多。

1.9.4 实现线段树的 C++类模版

以区间和线段树为例:

```
template<typename T> class SGT {
public:
	SGT() {}
	SGT(int _sz) : sz(_sz) {
		sgt = vector<T>(sz << 1, 0);
	}
	T query(int l, int r) {
		T ans = 0;
		for (l += sz, r += sz; l < r; l >>= 1, r >>= 1) {
			if (l & 1) ans += sgt[l++];
			if (r & 1) ans += sgt[--r];
		}
		return ans;
	}
	void add(int x, T val) {
		for (sgt[x += sz] += val; x > 1; x >>= 1)
			sgt[x >> 1] = sgt[x] + sgt[x ^ 1];
	}
private:
	int sz;
	vector<T> sgt;
};
```

1.10 Shortest path／最短路径

1.10.1 单源最短路／Dijkstra 算法

图结构用邻接矩阵 g[maxn][maxn]表示，运行一次 Dijkstra(int s)函数得到从 s 出发到所有点的最短路径长度，存储在数组 dis[maxn]中。路径的前驱关系存储在数组 prev[maxn]中，可用于生成具体路径。$T/S\sim O(|V|^2)$

算法中已访问顶点集合用 vis[true]表示，已有最短路径存储在 dis[]中，前驱顶点存储在 prev[]中。

核心代码:

```
const int INF = 1e9;

const int maxn = 10;

int n;
int g[maxn][maxn];

bool vis[maxn];
int dis[maxn];
int prev[maxn];

void Dijkstra(int s) {
    for (int i = 0; i < n; i++) {    // Init.
        dis[i] = g[s][i];
        vis[i] = i == s ? true : false;
        prev[i] = dis[i] == INF ? -1 : s;
    }

    for (int i = 0; i < n - 1; i++) {    // n - 1 steps.
        int mn = INF;
        int u = s;
        for (int j = 0; j < n; j++)      // Find the closest vertex u can be reached from s, put it in U set.
            if (!vis[j] && dis[j] < mn)
                u = j, mn = dis[j];
        vis[u] = true;
        for (int j = 0; j < n; j++)      // Update the distance of vertexes in V set according to the u which is the point put ino V lastly.
            if (!vis[j] && g[u][j] < INF)
                if (dis[u] + g[u][j] < dis[j])
                    dis[j] = dis[u] + g[u][j], prev[j] = u;
    }
}
```

算法上述实现的时间复杂度为$T\sim O(|V|^2)$，适合稠密图。用基于堆结构的优先队列维护已访问点集V可以进一步优化之。算法共有$|V|$步，每一步的复杂度为**寻找最近点**和**更新距离**复杂度之和。根据 CLRS，二叉堆实现时$T\sim O((V+E)logV)$，斐波那契堆实现时$T\sim O(VlogV+E)$。

Dijkstra 算法总是选择剩余点集中最近的顶点加入到已访问点集，属于**贪心法**，欲证明其正确性即是证明贪心过程每一步的局部最优在该问题中就是全局最优解。

1.10.2 所有点对两两之间的最短路径 / Floyd 算法

图结构用邻接矩阵 g[maxn][maxn]表示，运行一次 Floyd()函数得到所有点对两两之间的最短路径长度，存储在矩阵 p[maxn][maxn]中。顶点之间的中继关系存储在矩阵 path[maxn][maxn]中，可用于生成具体路径。

$T\sim O(V^3)/S\sim O(V^2)$

核心代码:

```
const int INF = 1e9;

const int maxn = 10;

int n;
int g[maxn][maxn];

int p[maxn][maxn];
int path[maxn][maxn];

void Floyd() {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            p[i][j] = g[i][j], path[i][j] = -1;

    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (p[i][k] + p[k][j] < p[i][j])
                    p[i][j] = p[i][k] + p[k][j], path[i][j] = k;
}
```

Floyd 算法的**重叠子问题**是“可以使用前k个顶点作为中转点的最短路径”，属于**动态规划**算法。

1.11 Minimum spanning tree (MST)/最小生成树

1.11.1 Prim 算法:

维护一个 MST 点集，初始只有任一点（一般取索引为0的点），每次取所有一端在 MST 点集内部另一端在 MST 点集外部的边中权值最小的一条，将该边的另一端加入 MST 点集并记录该边，则经过$N-1$步得到最小生成树。

Prim 算法属于贪心法，其正确性基于 MST 性质:

MST 性质： *对于一颗**正在构造中**的最小生成树，设U,V分别为树的顶点集及其补集。若边(u,v)为一端在当前点集中$u \in U$，另一端不在当前点集中$v \in V$，且权重最小的边，则必然存在一颗包含该边(u,v)的最小生成树。*

核心代码:

```
int G[maxn][maxn];

int next_vertex(int low_cost[], bool mst_set[]) {
    int mn = INF, mn_idx = -1;
    for (int i = 0; i < n; i++)
        if (!mst_set[i] && low_cost[i] < mn)
            mn = low_cost[i], mn_idx = i;
    return mn_idx;
}

ll prim_MST() {
    int par[n];   // Store which inner vertex to connect for outer vertexes.
    int low_cost[n]; // lowest cost for outer vertexes to connect to a inner vertex.
    bool mst_set[n]; // Mark inner vertices.

    // Initially put 1st vertex into mst_set.
    mst_set[0] = true, par[0] = -1;
    for (int i = 1; i < n; i++)
        low_cost[i] = G[0][i], mst_set[i] = false, par[i] = 0;

    for (int i = 0; i < n - 1; i++) {     // There are n - 1 steps constructing MST.
        int u = next_vertex(low_cost, mst_set);
        mst_set[u] = true;
        for (int v = 0; v < n; v++)
            if (!mst_set[v] && G[u][v] < low_cost[v])
                par[v] = u, low_cost[v] = G[u][v];
    }

    ll ans = 0;
    for (int i = 1; i < n; i++)
        ans += G[i][par[i]];
    return ans;
}
```

最终，par 数组中存储了 MST 的信息。

1.11.2 Kruskal 算法:

Kruskal 算法的思路很清晰，需要使用 union-find 这种数据结构，该算法的步骤如下:

1. 新建图 G，图 G 中有与原图相同的顶点，但没有边;
2. 将原图中所有的边按权值从小到大排序;
3. 从权值最小的边开始，如果这两个边连接的两个顶点在图 G 内不在一个连通分量中，则添加这条边到图 G 中;
4. 重复 3，直到图 G 只剩一个连通分量。

（注：该图只关注动态连通性，用并查集实现就好。）

1.12 local sorting/本地排序

对长度为n的序列进行个本地排序的时间复杂度可以达到$T\sim O(n\log n)$。常用的实现方法有快速排序、归并排序和堆排序。

1.12.1 quick sort/快速排序

快排的主要流程是**“分边”操作**，也是$T\sim O(n\log n)$中的那个$T\sim O(n)$，无需开辟额外空间。

核心代码（经典数组实现）：

```
void quick(int *a, int l, int r) {   // Sort a[l...r].
    if (l >= r) return;
    int i = l + 1, j = r;
    while (1) {    //
        while (!(a[l] < a[i] || i == r))
            i++;
        while (!(a[l] >= a[j] || j == l))
            j--;
        if (i < j) {
            int tmp = a[i];
            a[i] = a[j], a[j] = tmp;
        } else break;
    }
    int tmp = a[l];
    a[l] = a[j], a[j] = tmp;
    quick(a, l, j - 1);
    quick(a, j + 1, r);
}
```

注：以上“分边”操作出自唐发根老师的《数据结构》教材，用两个指针分别从序列两端夹逼将整个序列按相对第一个元素的大小分类。

核心代码（单链表实现）：

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode() {};
    ListNode(int _val) : val(_val), next(nullptr) {};
};

void quick(ListNode *l, int n) {      // Sort linkedlist l with n node.
    if (n <= 1) return;

    ListNode *p, *q, *tail;
    int n1 = 1;

    tail = l;
    for (int i = 0; i < n - 1; i++)
        tail = tail->next;
```

```
    p = l, n1 = 0;
    while (p != tail && p->val <= tail->val)
        p = p->next, n1++;

    q = p;
    while (1) {
        while (!(q->val <= tail->val || q == tail))
            q = q->next;
        if (q != tail) {
            int tmp = p->val;
            p->val = q->val;
            q->val = tmp;
            p = p->next, n1++;
            q = q->next;
        } else break;
    }
    int tmp = p->val;
    p->val = q->val;
    q->val = tmp;

    quick(l, n1);
    quick(p->next, n - n1 - 1);
}

void quick_sort(ListNode *list) {    // Quick sort linkedlist pointed by list.
    int n = 1;
    ListNode *p = list;
    while (p->next != nullptr)
        p = p->next, n++;
    quick(list, n);
}
```

注：与数组不同的是，单链表的扫描方向只能从左向右，以上分边操作出自《算法导论》中快拍的描述。

注 2：如果希望单链表快排只对指针操作（不交换节点内容），那么需要更多的临时指针。对于每层排序以输入链表表头节点为基准，将链表重组为分别包含小于和大于等于表头节点的两个新链表，对这两个新链表递归排序后与单独的节点拼接并返回新表头。

1.12.2 merge sort/归并排序

归并排序主要流程是**“归并”操作**，也是$T\sim O(n\log n)$中的那个$T\sim O(n)$，需要开辟额外空间。

核心代码:

```
void merge(int *a, int l, int m, int r) {     // Merge two sub-arraies a[l...m] & a[m + 1...r] into a[l...r].
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = a[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = a[m + 1 + j];

    int i = 0, j = 0, k = l;      // Initial index of left array, right array, merged array.
    while (i < n1 && j < n2)
        if (L[i] <= R[j]) a[k++] = L[i++];
        else a[k++] = R[j++];
    while (i < n1)
        a[k++] = L[i++];
    while (j < n2)
        a[k++] = R[j++];
}

void merge_sort(int *a, int l, int r) {     // Sort array a[l...r].
    if (l >= r) return;
    int m = (l + r) >> 1;
    merge_sort(a, l, m);
    merge_sort(a, m + 1, r);
    merge(a, l, m, r);
}
```

1.13 Pattern matching/模式匹配算法

1.14 LRUcache/最常访问缓存

实现一个数据结构，支持 get 和 set 操作:

· get(key): 如果缓存中存在键 key，则返回其对应的值，否则返回-1;

· set(key, value): 设置缓存中键 key 对应的值或向缓存中加入键值映射(key, value)。当缓存达到其容量，加入新映射前去掉最不常访问的那对映射。

分析:

二级映射，键到键值对的地址，键值对的地址到值。中间级键值对用双向链表组织。

常规地，映射关系为键值对的动态集合。用平衡二叉查找树 map 或 Hash 结构 unordered_map 存储每个节点的地址可以保证在 T~O(log(n))或 O(1)的时间内查找节点。为刻画缓存中元素的访问频次顺序，使将某个元素移动至队头的操作足够高效，用双向链表实现缓存内的键值对动态集合。

综上，用映射关系用键值对刻画，键值对的动态集合用双向链表存储，最后用一个 unordered_map 存储每个节点（键值对）的地址。

完整代码:

```
class LRUcache {
private:
    struct cache_node {
        int k, v;
        cache_node() {}
        cache_node(int _k, int _v) : k(_k), v(_v) {}
    };
public:
    LRUcache(int capacity) {
        this->capacity = capacity;
    }
    int get(int key) {
        if (cache_map.find(key) == cache_map.end()) return -1;
        cache_list.splice(cache_list.begin(), cache_list, cache_map[key]);
        cache_map[key] = cache_list.begin();
        return cache_map[key]->v;
    }
    void set(int key, int value) {
        if (cache_map.find(key) == cache_map.end()) {
            if (cache_list.size() == capacity) {
                cache_map.erase(cache_list.back().k);
                cache_list.pop_back();
            }
            cache_list.push_front(cache_node(key, value));
            cache_map[key] = cache_list.begin();
        } else {
            cache_map[key]->v = value;
            cache_list.splice(cache_list.begin(), cache_list, cache_map[key]);
            cache_map[key] = cache_list.begin();
        }
```

```
    }
private:
    int capacity;
    list<cache_node> cache_list;
    unordered_map<int, list<cache_node>::iterator> cache_map;
};
```

1.15 Union-Find/并查集

用于维护动态连通性问题的数据结构。并查集由一个整型数组和两个函数构成。输入 pre[]记录了每个节点的前导节点，find()是查找，union()是合并。

并查集每次 union 操作随机将一个门派的掌门作为另一个门派掌门的直接上级；而每次 find 操作在逐层向上找到掌门之后又将路径中所有过程节点都改成掌门的直接下级。所以说，当 union 操作的增多会导致树状数据结构趋于不平衡，而 find 操作很多的时候，将最终导致所有节点都以掌门为直接上级（汇报距离变成 1）。

完整代码:

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

using namespace std;

class union_find {
public:
    union_find(int _sz) : sz(_sz) {
        pre = vector<int>(sz);
        for (int i = 0; i < sz; i++)
            pre[i] = i;
    }
    int find(int x) {
        // Find root.
        int r = x;
        while (r != pre[r])
            r = pre[r];
        // Path compress.
        while (x != r) {
            int tmp = pre[x];
            pre[x] = r;
            x = tmp;
        }
        return r;
    }
    void join(int x, int y) {
        int rx = find(x), ry = find(y);
        if (rx != ry)
            pre[rx] = ry;
    }
    int size() {
        return sz;
    }
private:
    int sz;
    vector<int> pre;
};
```

1.16 Tire/字典树

作为维护单词集合的一种数据结构，插入时间和查找时间都是单词长度的线性函数$T\sim O(l)$。相比以字符串作为节点内容，基于字符串比较排序的 RB-Tree 实现的单词集合（字典）单步操作$T\sim O(llogn)$的复杂度，字典树的操作的时间复杂度更低。然而 Tire 是一颗度数等于字符表大小的多叉树，具有较高的空间复杂度。

在实现的过程中需要注意的点:

· 类中用链式结构在内存堆空间内维护了一棵树，析构函数中要递归删除。

下面的 Tire 类，支持由 26 个小写英文字母构成单词的插入、查找和删除操作。

完整代码:

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

using namespace std;

class Tire {
public:
    Tire() {}
    ~Tire() { tire_del(head); }
    void insert(string word) {
        tire_node *cur = head;
        for (int i = 0; i < word.size(); i++) {
            if (cur->next[word[i] - 'a'] != nullptr) cur = cur->next[word[i] - 'a'];
            else {
                tire_node *tmp = new tire_node(false);
                cur->next[word[i] - 'a'] = tmp;
                cur = tmp;
            }
            if (i == word.size() - 1) cur->isword = true;
        }
    }
    bool search(string word) {
        tire_node *cur = head;
        for (int i = 0; i < word.size(); i++) {
            if (cur->next[word[i] - 'a'] == nullptr) return false;
            cur = cur->next[word[i] - 'a'];
        }
        return cur->isword;
    }
    void erase(string word) {
        tire_node *cur = head;
        for (int i = 0; i < word.size(); i++) {
            if (cur->next[word[i] - 'a'] == nullptr) {
                cout << "\"" << word << "\" " << "was not in Tire." << endl;
                return;
            }
            cur = cur->next[word[i] - 'a'];
        }
```

```
        cur->isword = false;
    }

private:
    static const int alphabat_sz = 26;
    struct tire_node {
        bool isword;
        tire_node *next[alphabat_sz];
        tire_node() {}
        tire_node(bool _isword) : isword(_isword) {
            for (int i = 0; i < alphabat_sz; i++)
                next[i] = nullptr;
        }
    };
    tire_node *head = new tire_node(false);
    void tire_del(tire_node *cur) {
        for (int i = 0; i < alphabat_sz; i++)
            if (cur->next[i] != nullptr) tire_del(cur->next[i]);
        delete cur;
    }
};
```

注（关于测试）:

对于一个新完成的 class，测试样例的设计原则如下：首先所设计测试样例应能够触发类中所有函数的所有分支；其次设计样例考虑边界情形和极端情形。

1.17 Brian Kernighan/统计正整数二进制表示中 1 的数量

很显然，对于正整数n，朴素的算法可以遍历其二进制表示中的所有位，统计其中1的数量。但对于二进制位中1特别少的情况，用如下技巧可以使过程大大加速，同时保持最坏为$T\sim O(\log n)$

注意 n&(n−1)可以去掉n的最低位1，利用这个操作，我们实际上只需n的二进制表示中1的数量的迭代次数。

核心代码:

```
int brian_kernighan(int n) {
    int ans = 0;
    while (n)
        n = n & (n − 1), ans++;
    return ans;
}
```

实际使用:

int __builtin_popcount(unsigned int)是 GCC 的一个内建函数（不是 C++函数），这个函数通过使用 CPU 的某些特殊指令在$T\sim O(1)$时间内完成上述操作。

```
__builtin_popcount(int)
__builtin_popcountl(long int)
__builtin_popcountll(long long)
```

1.18 Minimum covering circle／最小圆覆盖

给定平面上n个点$P_1, P_2, \ldots, P_n$，找到覆盖所有点的具有最小半径的圆。

贪心算法:

1. 在n个点中任取三点A, B, C。
2. 找到包含三点的最小圆。
3. 在n个点中选出与当前圆心距离最小的点D。

判断：若D在园内，则当前圆为所求，算法结束;

否则：找到包含A, B, C, D的最小圆，该圆必由A, B, C, D中的三个唯一决定，以这三个点为新的A, B, C进入第 2 步。

核心代码:

```
struct point {
    double x, y;
    point() {};
    point(double _x, double _y) : x(_x), y(_y) {};
};

bool operator == (point a, point b) {
    return (a.y == b.x) && (a.y == b.y);
}

bool operator != (point a, point b) {
    return !(a == b);
}

double dist(point a, point b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

struct circle {
    point o;
    double r;
    circle() {};
    circle(point _o, double _r) : o(_o), r(_r) {};
};

bool operator == (circle a, circle b) {
    return (a.o == b.o) && (a.r == b.r);
}

bool operator != (circle a, circle b) {
    return !(a == b);
}

circle get_circle_from_2p(point a, point b) {
    circle ans;
    ans.o.x = (a.x + b.x) / 2;
```

```
    ans.o.y = (a.y + b.y) / 2;
    ans.r = dist(a, b) / 2;
    return ans;
}

circle get_circle_from_3p(point a, point b, point c) {
    if (std::abs((b.x * c.y - c.x * b.y) - (a.x * c.y - c.x * a.y) + (a.x * b.y - b.x * a.y)) < eps) return circle(point(), -1);

    if (std::abs(a.y - b.y) < std::abs(a.y - c.y) && std::abs(a.y - b.y) < std::abs(b.y - c.y)) std::swap(a, c);
    if (std::abs(a.y - c.y) < std::abs(a.y - b.y) && std::abs(a.y - c.y) < std::abs(b.y - c.y)) std::swap(a, b);
    double k1 = - (a.x - b.x) / (a.y - b.y);
    double k2 = - (a.x - c.x) / (a.y - c.y);
    double d1x = (a.x + b.x) / 2, d1y = (a.y + b.y) / 2;
    double d2x = (a.x + c.x) / 2, d2y = (a.y + c.y) / 2;
    circle ans;
    ans.o.x = (k1 * d1x - d1y - k2 * d2x + d2y) / (k1 - k2);
    ans.o.y = k1 * (ans.o.x - d1x) + d1y;
    ans.r = dist(a, ans.o);
    return ans;
}

circle cover(point a, point b, point c) {    // Return the min circle which cover a, b and c.
    std::vector<circle> anss;

    circle tmp;
    tmp = get_circle_from_2p(a, b);
    if (tmp.r > 0 && dist(tmp.o, c) < tmp.r + eps)
        anss.emplace_back(tmp);
    tmp = get_circle_from_2p(a, c);
    if (tmp.r > 0 && dist(tmp.o, b) < tmp.r + eps)
        anss.emplace_back(tmp);
    tmp = get_circle_from_2p(b, c);
    if (tmp.r > 0 && dist(tmp.o, a) < tmp.r + eps)
        anss.emplace_back(tmp);
    tmp = get_circle_from_3p(a, b, c);
    if (tmp.r > 0)
        anss.emplace_back(tmp);

    circle ans(point(), INF);
    for (circle x : anss)
        if (ans.r > x.r) ans = x;
    return ans;

}
```

```
std::tuple<point, point, point> cover(point a, point b, point c, point d) { // Find 3 points among a, b, c, d generating the smallest circle covering 4 points.
    std::vector<std::tuple<circle, point, point, point>> anss;
    std::tuple<point, point, point> ans;
    circle mn(point(), INF);
    circle tmp;
    // Generate from 2 points.
    tmp = get_circle_from_2p(a, b);
    if (tmp.r > 0 && dist(tmp.o, c) < tmp.r + eps && dist(tmp.o, d) < tmp.r + eps)
        if (mn.r > tmp.r) mn = tmp, ans = std::tie(a, b, c);
    tmp = get_circle_from_2p(a, c);
    if (tmp.r > 0 && dist(tmp.o, b) < tmp.r + eps && dist(tmp.o, d) < tmp.r + eps)
        if (mn.r > tmp.r) mn = tmp, ans = std::tie(a, b, c);
    tmp = get_circle_from_2p(a, d);
    if (tmp.r > 0 && dist(tmp.o, b) < tmp.r + eps && dist(tmp.o, c) < tmp.r + eps)
        if (mn.r > tmp.r) mn = tmp, ans = std::tie(a, b, d);
    tmp = get_circle_from_2p(b, c);
    if (tmp.r > 0 && dist(tmp.o, a) < tmp.r + eps && dist(tmp.o, d) < tmp.r + eps)
        if (mn.r > tmp.r) mn = tmp, ans = std::tie(a, b, c);
    tmp = get_circle_from_2p(b, d);
    if (tmp.r > 0 && dist(tmp.o, a) < tmp.r + eps && dist(tmp.o, c) < tmp.r + eps)
        if (mn.r > tmp.r) mn = tmp, ans = std::tie(a, b, d);
    tmp = get_circle_from_2p(c, d);
    if (tmp.r > 0 && dist(tmp.o, a) < tmp.r + eps && dist(tmp.o, b) < tmp.r + eps)
        if (mn.r > tmp.r) mn = tmp, ans = std::tie(a, c, d);
    // Generate from 3 point
    tmp = get_circle_from_3p(a, b, c);
    if (tmp.r > 0 && dist(tmp.o, d) < tmp.r + eps)
        if (mn.r > tmp.r) mn = tmp, ans = std::tie(a, b, c);
    tmp = get_circle_from_3p(a, b, d);
    if (tmp.r > 0 && dist(tmp.o, c) < tmp.r + eps)
        if (mn.r > tmp.r) mn = tmp, ans = std::tie(a, b, d);
    tmp = get_circle_from_3p(a, c, d);
    if (tmp.r > 0 && dist(tmp.o, b) < tmp.r + eps)
        if (mn.r > tmp.r) mn = tmp, ans = std::tie(a, c, d);
    tmp = get_circle_from_3p(b, c, d);
    if (tmp.r > 0 && dist(tmp.o, a) < tmp.r + eps)
        if (mn.r > tmp.r) mn = tmp, ans = std::tie(b, c, d);

    return ans;
}

circle smallest_covering_circle(std::vector<point> &points) {
    if (points.size() == 1) return circle(points[0], 0);
    if (points.size() == 2) return get_circle_from_2p(points[0], points[1]);
```

```
    point a = points[0], b = points[1], c = points[2];
    while (1) {
        circle tmp = cover(a, b, c);
        double mx = -INF;
        point d;
        for (point x : points)
            if (x != a && x != b && x != c)
                if (mx < dist(x, tmp.o)) mx = dist(x, tmp.o), d = x;
        if (dist(d, tmp.o) < tmp.r + eps) return tmp;
        else {
            std::tie(a, b, c) = cover(a, b, c, d);
        }
    }
}
```

一些待整理的内容:

二分图的最大匹配
MKP 模式匹配算法

第二章：分类篇

——算法编程常见考点例题汇总

2.1 动态规划
2.2 数据结构
2.3 预处理
2.4 通过数学规律找到常数解
2.5 二分搜索
2.6 图论问题
2.7 数论问题
2.8 计数问题
2.9 计算几何
2.10 算法内功

2.1.1 Codeforces 796E (Round #408)

Exam Cheating (动态规划)

题述：

Zane 爱上的女孩儿遇到了麻烦。一次考试中有n道题编号从1到n，$1 \le n \le 1000$，女孩儿的左右坐着两个大神，大神会作答部分题。在监考老师走神的时间内女孩儿一共可以偷看p次旁边大神的卷子，$1 \le p \le 1000$，每次可以看到一个大神作答的任意连续k题，$1 \le k \le \min(n, 50)$。请通过计算帮助女孩儿确定通过作弊最多可以获得多少道题的正确答案。

输入：

第一行为三个整数$n, p, k(1 \le n, p \le 1000, 1 \le k \le \min(n, 50))$——分别为题目数量、最大偷看次数和一次偷看可以看到的最大连续题目数。

第二行开始是一个整数$r(0 \le r \le n)$，表示大神 A 作答的题目数量。接下来的r个整数$a_1, a_2, \dots, a_r(1 \le a_i \le n)$，表示大神 A 作答的题目序号，升序排列（$a_i < a_{i+1}$）。

第三行开始是一个整数$s(0 \le s \le n)$，表示大神 B 作答的题目数量。接下来的s个整数$b_1, b_2, \dots, b_s(1 \le b_i \le n)$，表示大神 B 作答的题目序号，升序排列（$b_i < b_{i+1}$）。

输出：

输出一个整数——女孩儿通过作弊获得的最多答案数量。

测试样例：

Input	Output
6 2 3 3 1 3 6 4 1 2 5 6	4
8 3 3 4 1 3 5 6 5 2 4 6 7 8	7

注释：

令(x, l, r)表示偷看大神 A/B$(x = 0/1)$卷子上从l到r的所有问题。

测试样例 1 中，女孩儿通过操作$(1, 1, 3), (2, 5, 6)$可以获得4道题的正确答案。

测试样例 2 中，女孩儿通过操作$(1, 3, 5), (2, 2, 4), (2, 6, 8)$可以获得7个问题的正确答案。

Tutorial 的动态规划解:

前缀上的动态规划。首先规定每次看旁边人的答案，**都看接下来的连续k道新题**。备忘录（状态／重叠子问题）设计：dp$[i][j][a][b]$记录通过j次偷看可以在前i个问题中获得的最大正解数量，同时大神 A 的第$i+1, \dots, i+a$题答案、大神 B 的第$i+1, \dots, i+b$题答案也顺带获得（它们因对后续状态的影响不同而作为状态属性），当然这二者的取值区间为$0 \le a, b \le k-1$。从0到n对i的每个取值模拟j次偷看的过程。最终答案为dp$[n][..][..][..]$的最大值。

如上算法的复杂度是$T/S\sim O(npk^2)$。对于加速，我们发现最多只需$p=2\times\left\lceil\frac{n}{k}\right\rceil$次偷看就能获得大神 A、B 所答的所有题目，于是将偷看次数p缩小到$p=min(p,2*(n/k+1))$将使时间复杂度上界优化到$T\sim O(n^2k)$，满足要求。对于空间开销，注意到$\mathrm{dp}[i][..][..][..]$仅利用了$\mathrm{dp}[i-1][..][..][..]$的信息，故可以只存储 4D 备忘录的前两列。

完整代码:

```
#include <algorithm>
#include <iostream>

const int INF = 1e9;

const int maxn = 1005, maxp = 1005, maxk = 55;

int A[maxn], B[maxn];
int n, p, k;
void read() {
    scanf("%d %d %d", &n, &p, &k);
    if (p > 2 * (n + k - 1) / k) p = 2 * (n + k - 1) / k;
    int r;
    scanf("%d", &r);
    for (int i = 0; i < r; i++) {
        int tmp;
        scanf("%d", &tmp);
        A[tmp] = 1;
    }
    int s;
    scanf("%d", &s);
    for (int i = 0; i < s; i++) {
        int tmp;
        scanf("%d", &tmp);
        B[tmp] = 1;
    }
}

int dp[2][maxp][maxk][maxk];
void solve() {
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < maxp; j++)
            for (int a = 0; a < maxk; a++)
                for (int b = 0; b < maxk; b++)
                    dp[i][j][a][b] = -INF;

    dp[0][0][0][0] = 0;
    for (int i = 1; i <= n; i++) {
        int cur = i & 1, prev = !cur;
```

```
            for (int j = 1; j <= p; j++)
                dp[cur][j][k - 1][0] = std::max(dp[cur][j][k - 1][0], dp[prev][j - 1][0][0] + A[i]);
            for (int j = 1; j <= p; j++)
                for (int a = 0; a < k - 1; a++)
                    dp[cur][j][a][0] = std::max(dp[cur][j][a][0], dp[prev][j][a + 1][0] + A[i]);

            for (int j = 1; j <= p; j++)
                dp[cur][j][0][k - 1] = std::max(dp[cur][j][0][k - 1], dp[prev][j - 1][0][0] + B[i]);
            for (int j = 1; j <= p; j++)
                for (int b = 0; b <= k - 1; b++)
                    dp[cur][j][0][b] = std::max(dp[cur][j][0][b], dp[prev][j][0][b + 1] + B[i]);

            for (int j = 1; j <= p; j++)
                for (int a = 0; a < k - 1; a++)
                    dp[cur][j][a][k - 1] = std::max(dp[cur][j][a][k - 1], dp[prev][j - 1][a + 1][0] + (A[i] | B[i]));
            for (int j = 1; j <= p; j++)
                for (int b = 0; b < k - 1; b++)
                    dp[cur][j][k - 1][b] = std::max(dp[cur][j][k - 1][b], dp[prev][j - 1][0][b + 1] + (A[i] | B[i]));
            for (int j = 1; j <= p; j++)
                for (int a = 0; a < k - 1; a++)
                    for (int b = 0; b < k - 1; b++)
                        dp[cur][j][a][b] = std::max(dp[cur][j][a][b], dp[prev][j][a + 1][b + 1] + (A[i] | B[i]));

            for (int j = 0; j <= p; j++)
                dp[cur][j][0][0] = std::max(dp[cur][j][0][0], dp[prev][j][0][0]);

            for (int j = 0; j <= p; j++)
                for (int a = 0; a < k; a++)
                    for (int b = 0; b < k; b++)
                        dp[prev][j][a][b] = -INF;
        }

        int ans = -INF;
        for (int j = 0; j <= p; j++)
            for (int a = 0; a < k; a++)
                for (int b = 0; b < k; b++)
                    ans = std::max(ans, dp[n & 1][j][a][b]);
        printf("%d\n", ans);
}

int main() {
        read();
        solve();
}
```

Q: 备忘录更新的过程中大量的 for(int i = 1; i <= p; i++)是怎么分类的？

A: 按状态转移方程不同分类（如是否又偷看一次），发展到状态 dp[i][j][a][b]的前态可能有若干种，每种的状态转移方程不同，在它们的结果中取最大值。

Q: 怎样保证状态更新方式的完备性？

A: 解释分类：看一眼 A 的以及其带来的好处，看一眼 B 的以及其带来的好处，注意以上状态转移方程按有没有对面余量分为两类。和不看。

关于本题 **4D-dp** 的**算法设计思路**：首先设计**备忘录**，即定义**状态**。然后设计**生成备忘录的过程**，即寻找**状态转移方程**。该过程分为再看一眼 A 的以及带来的好处并更新相应部分备忘录和再看一眼 B 的以及带来的好处并更新相应备忘录两条主线。在实际操作过程中发现在是否存在另一方好处的情况下状态转移方程不同，故更新过程的实现再具体细分为四部分，最后不看也会更新一部分，故总共有 5 个对 j 的循环模拟整个作弊过程。从正常认知的角度分析该 4D-dp 复杂在于其为对于i和j二者的**分层动规**，首先内层**模拟作弊过程**需要**动规**进行，然后是对**目标**（看几道题）迭代更新的又要一层**动规**。

（2017.7.9）本题状态转移的写法：

首先对于**前缀上的动态规划**，最外层循环为描述前缀长度的递增变量。由于永远是用少一位前缀所对应的状态$dp[i-1][..][..][..]$**更新**当前前缀对应的状态$dp[i][..][..][..]$，所以主循环内的每个循环体顺序实质上是任意的。

对于主循环体的写法，可以遵循如下思路：首先，我们将所有状态按照不用新信息、用$A[i]$作为新信息、用$B[i]$作为新信息、用$A[i]|B[i]$作为新信息分为 4 大类。其中，使用新信息更新状态还分为用一次偷看更新和无需偷看（用储备信息）更新。具体的：

1. 用$A[i]$更新的状态——无 B 的储备信息

· 偷看 A: $dp[i][1 \dots p][k-1][0]$

· 利用 A 的储备信息: $dp[i][1 \dots p][0 \dots k-2][0]$

2. 用$B[i]$更新的状态——无 A 的储备信息

· 偷看 B: $dp[i][1 \dots p][0][k-1]$

· 利用 B 的储备信息: $dp[i][1 \dots p][0][0 \dots k-2]$

3. 用$A[i]|B[i]$更新的状态——两边要么有储备信息，要么偷看一眼

· 偷看 A 并利用 B 的储备信息: $dp[i][1 \dots p][k-1][0 \dots k-2]$

· 偷看 B 并利用 A 的储备信息: $dp[i][1 \dots p][0 \dots k-2][k-1]$

· 利用 A、B 的储备信息: $dp[i][2 \dots p][0 \dots k-2][0 \dots k-2]$

4. 不用新信息更新的状态——两边都没有储备信息

· $dp[i][0 \dots p][0][0]$

2.1.2 Codeforces 819B (Round 421, Div1)

Mister B and PR Shifts（动态规划，频次表）

题述：

前些日子 Mister B 截获了一段来自外太空的信号，欲破译之。

经过系列的变换信号最终可由一个排列p或其循环移位表征。为了进一步研究 Mister B 需要一些理论基础，为此他需要得到该排列的所有循环移位中偏移量最小的一个。

定义排列p的偏移量为$\sum_{i=1}^{n}|p_i| - i$。

请找出排列p的所有循环移位中具有的最小偏移量的那一个并给出相应的偏移量。如果有多个解，给出任意一个即可。

定义排列循环移位的序号$k(0 \le k < n)$为从原序列变为该排列的循环右移次数，例如：

- $k = 0$ —— $p_1, p_2, \dots, p_n$
- $k = 1$ —— $p_n, p_1, \dots, p_{n-1}$
-
- $k = n - 1$ —— $p_2, p_3, \dots, p_1$

输入：

第一行为一个整数$n(2 \le n \le 10^6)$——排列长度。

第二行为n个由空格分隔的整数$p_1, p_2, \dots, p_n (1 \le p_i \le n)$——排列的元素，输入保证所有元素互不相同。

输出：

输出两个整数：排列p所有循环移位的最小偏移量和该循环移位的序号，如果有多个解，输出任意一个即可。

测试样例：

Input	Output
3 1 2 3	0 0
3 2 3 1	0 1
3 3 2 1	2 1

注释：

测试样例一中的原排列为“顺序排列”，其本身具有最小偏移量0。

测试样例二中原排列的偏移量为4，1 号循环移位$(1, 2, 3)$的偏移量为0，2 号循环移位$(3, 1, 2)$的偏移量为4，故 1 号循环移位具有最小偏移量。

测试样例三中原排列的偏移量为4，1 号循环移位$(1, 3, 2)$的偏移量为2，2 号循环移位$(2, 1, 3)$的偏移量也是2，故 1 号、2 号循环移位都是最优解。

参考 zoomswk / 姜予名的动态规划解:

首先，纯模拟的复杂度为本地循环移位和计算排列偏移量二者复杂度的乘积，即$T \sim O(n) \times O(n) = O(n^2)$，超时。

既然不可避免地要计算出并比较原排列所有n个循环移位的偏移量找到其中最小的一个，那么这个过程的每一步需要在常数时间内完成，于是问题转换为怎样在$T \sim O(1)$内由第i个循环移位的偏移量得到第$i+1$个循环移位的偏移量。

对任意排列，将其中元素分成三类——大于标准、等于标准和小于标准，**维护三类元素的各自数量**。排列循环右移1位之后，最右端元素x的比较标准从n变成1，引起的偏移量变化为$2x - n - 1$，其它$n-1$个元素的比较标准增加1，于是新的偏移量可以这样计算：新偏移量=原偏移量−原来前$n-1$个元素中在标准之上的元素数+原来前$n-1$个元素中在标准之下的元素数+原来前$n-1$个元素中等于标准的元素数+原来末元移到首位引起的变化。具体地，用d_i表示i号循环移位的偏移量，$more, less, equal$分别表示当前排列中的所有元素在标准之上、标准之下、等

于标准的元素数量。按照最后一个元素变化前后所属的类型分三类讨论，由d_i表示d_{i+1}的方法具体表示如下:

若当前最后一个元素为$p_{n-i}=1$，则其移位前属于 less，移位后属于 equal。则$d_{i+1}=d_i-more+(less-1)+equal+2p_{n-i}-n-1$，而三个统计量的变化方式为:

· $less=(less-1)+equal$

· $equal=h_{i+1}$

· $more=n-less-equal$

其中，$h[i+1]$为记录初始排列中经过$i+1$步循环右移后等于标准的元素数（频次哈希表），可在读入初始排列时一趟生成。

类似地，若当前最后一个元素$p_{n-i}=n$，则其移位前属于 equal，移位后属于 more。则$d_{i+1}=d_i-more+less+(equal-1)+2p_{n-i}-n-1$，三个统计量的更新方式:

· $less=less+(equal-1)$

· $equal=h_{i+1}$

· $more=n-less-equal$

最后，若当前元素为其他值$p_{n-i}=2\dots n-1$，则其移位前属于 less，移位后属于 more。则$d_{i+1}=d_i-more+(less-1)+equal+2p_{n-i}-n-1$，三个统计量的更新:

· $less=(less-1)+equal$

· $equal=h_{i+1}$

· $more=n-less-equal$

注意上述三种情况的逻辑完全相同，可以合并成一种情况。

以上算法$T\sim O(n)$。

细节描述（生成右移距离频次哈希表h_i）: 遍历原排列，若$p_i\geq i$则h_{p_i-i}加1，否则h_{n-i+p_i}加1。

完整代码:

```
#include <iostream>

using ll = long long;

const int maxn = 1000005;

int p[maxn], h[maxn];
int n;
void read() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &p[i]);
        if (p[i] >= i) h[p[i] - i]++;
        else h[n - (i - p[i])]++;
    }
}

void solve() {
    int more = 0, equal = 0, less = 0;
    ll d_cur = 0;
    for (int i = 1; i <= n; i++) {
        if (p[i] > i) more++;
        else if (p[i] == i) equal++;
```

```
            else less++;
            d_cur += std::abs(p[i] - i);
        }

        ll ans = d_cur;
        int idx = 0;
        for (int i = 1; i < n; i++) {
            d_cur = d_cur - more + equal + less + 2 * p[n - i + 1] - n - 2;
            if (d_cur < ans) ans = d_cur, idx = i;
            int less_cur = equal + less - 1;
            int equal_cur = h[i];
            int more_cur = n - less_cur - equal_cur;
            less = less_cur, equal = equal_cur, more = more_cur;
        }

        printf("%lld %d\n", ans, idx);
}

int main() {
        read();
        solve();
}
```

2.1.3 Codeforces 815C (Round 419)

Karen and Supermarket (Tree, DFS DP)

题述：

回家的路上，Karen 想去超市买些东西。

她想买的东西太多了，由于她还只是一个学生所以只有b美元的预算。

超市里有n种商品，每种商品的售价为c_i，且每种商品 Karen 只需要一件。

最近正值超市搞活动，Karen 作为老顾客得到了n张代金券，第i张代金券能在购买第i种商品时抵掉d_i美元。然而超市的促销也是有套路的——代金券的使用要遵循如下规则：对于$i \geq 2$，欲使用第i种商品的代金券，需要在已使用第x_i种商品的代金券的条件下。

Karen 想知道在不超过预算的前提下自己最多可以购买多少件商品。

输入：

第一行为两个整数n和$b(1 \leq n \leq 5000, 1 \leq b \leq 10^9)$——分别为超市内的商品数和 Karen 的预算。

接下来的n行每行描述一个商品。

- 其中第i行开始为两个整数c_i和$d_i(1 \leq d_i < c_i \leq 10^9)$——分别为第$i$种商品的售价和代金券的抵值额度。
- 如果$i \geq 2$，则再后跟一个整数$x_i(1 \leq x_i < i)$——表示欲使用第i种商品的代金券需要先使用第x_i种商品的代金券。

输出：

输出一行为一个整数——Karen 可购买商品的最大数量。

测试样例：

Input	Output
6 16 10 9 10 5 1 12 2 1	4

20 18 3 10 2 3 2 1 5	
5 10 3 1 3 1 1 3 1 2 3 1 3 3 1 4	5

注释：

测试样例一中，Karen 可以购买如下4件商品：

- 用代金券购买第1种商品消费$10-9=1$美元。
- 用代金券购买第3种商品消费$12-2=10$美元。
- 用代金券购买第4种商品消费$20-18=2$美元。
- 直接购买第6种商品消费2美元。

上述购买方案总共消费15美元在预算之内。请注意 Karen 无法使用第6种商品的代金券，因为她没有使用第5种商品的代金券。

测试样例二中，Karen 的预算足够她使用代金券购买全部商品。

参考 Tian.Xie 的 dfs_dp 解（自顶向下 dp 解）:

超市的促销规则中，所有n张代金券的使用顺序构成一棵**根树**结构（将所有商品按照到商品 1 的 BFS 距离分层），在这棵树上使用动态规划。建立重叠子问题，对于每个节点，我们求解在该节点使用／不使用代金券的情况下从该节点的子树中购买各种数量商品的最小花销。为此开辟备忘录$dp[N][2][N]$，其中$dp[i][0/1][j]$表示在第i件商品使用／不使用代金券的情况下从节点i的子树中购买j件商品的最小开销。

完整代码:

```
#include <algorithm>
#include <vector>
#include <cstdio>

const int maxn = 5005;
const int INF = 1e9;

int n, b;
int c[maxn], d[maxn], par[maxn];
std::vector<int> chi[maxn];
void read() {
	scanf("%d %d", &n, &b);
	scanf("%d %d", &c[0], &d[0]);
	for (int i = 1; i < n; i++) {
		scanf("%d %d %d", &c[i], &d[i], &par[i]);
		par[i]--;
		chi[par[i]].emplace_back(i);
	}
}
```

```
int dp[maxn][2][maxn], dp_tmp[2][maxn];
int sz[maxn];
void dfs_dp(int u) {
	for (int v : chi[u])
		dfs_dp(v);

	sz[u] = 1;
	dp[u][0][0] = 0;
	dp[u][0][1] = c[u];
	dp[u][1][0] = 0;
	dp[u][1][1] = c[u] - d[u];

	for (int v : chi[u]) {
		for (int j = 0; j <= sz[u] + sz[v]; j++)
			if (j <= sz[u]) dp_tmp[0][j] = dp[u][0][j], dp_tmp[1][j] = dp[u][1][j];
			else dp_tmp[0][j] = dp_tmp[1][j] = INF;
		for (int i = 0; i <= sz[u]; i++)
			for (int j = 0; j <= sz[v]; j++) {
				dp_tmp[0][i + j] = std::min(dp_tmp[0][i + j], dp[u][0][i] + dp[v][0][j]);
				if (i > 0) dp_tmp[1][i + j] = std::min(dp_tmp[1][i + j], dp[u][1][i] + std::min(dp[v][0][j], dp[v][1][j]));
			}
		for (int j = 0; j <= sz[u] + sz[v]; j++) {
			dp[u][0][j] = dp_tmp[0][j];
			dp[u][1][j] = dp_tmp[1][j];
		}
		sz[u] += sz[v];
	}
}

void solve() {
	for (int i = 0; i < maxn; i++)
		for (int j = 0; j < maxn; j++)
			dp[i][0][j] = dp[i][1][j] = INF;

	dfs_dp(0);

	int ans = 0;
	for (int j = 1; j <= n; j++)
		if (dp[0][0][j] <= b || dp[0][1][j] <= b) ans = j;
		else break;
	printf("%d\n", ans);
}

int main() {
	read();
```

```
    solve();
}
```

dfs_dp 总结（自顶向下动态规划问题）：**Codeforces 815C** 和 **Codeforces 814D** 两道题有如下共同点——两道题都可以建模为**根树结构**，**根树**结构的问题在**动态规划**时可直接利用问题特征将树的拓扑结构作为**最优子结构**，并在每个节点上建立考虑以该节点为根的**子树**中所有节点的**重叠子问题**，则单步**状态转移**就在层间进行。这种方法保证了$O(n)$的子问题处理规模（每个节点最多被处理两次，一次作为父节点、一次作为子节点），即备忘录的首维是以一次线性方式处理的。

2.1.4 Codeforces 814D (Round 418)

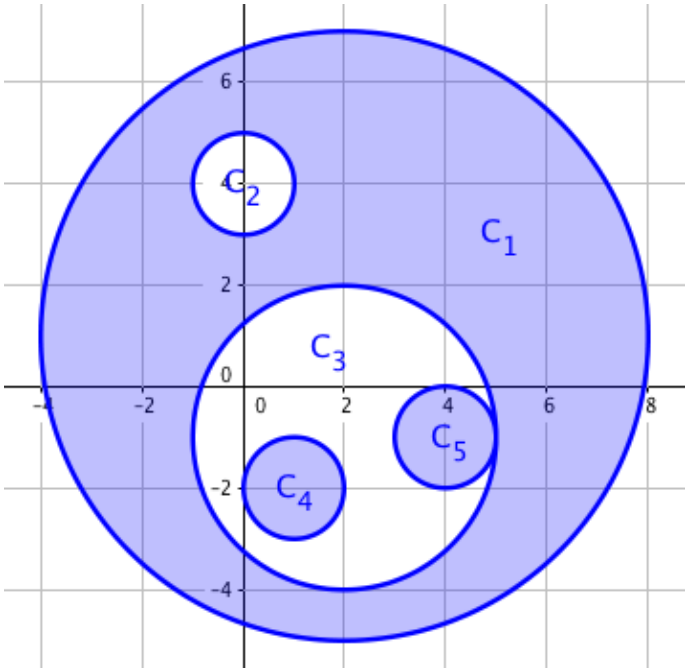
An overnight dance in discotheque

题述：

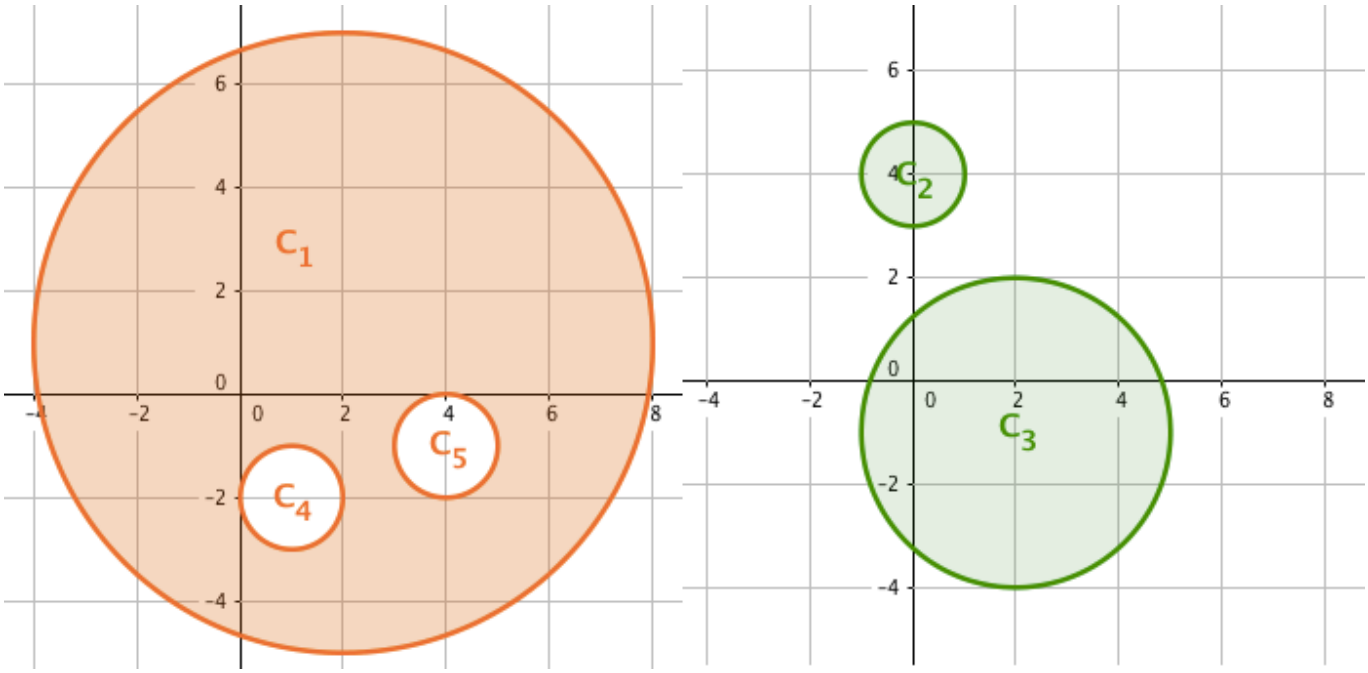
迪斯科舞厅里稍微拥挤一点并不会影响朋友们娱乐，但尽可能宽敞一些不是更好吗？

迪斯科舞厅被视作一个无限大xy −平面，总共有n个舞者。当某个人开始跳舞以后，她就只在自己的活动范围内移动，活动范围为一个由圆心(x_i, y_i)和半径r_i表征的园形区域C_i。任意两个区域边界的公共点不超过2个，这意味着对于任意数对$(i, j)(1 \le i < j \le n)$，$C_i$和$C_j$要么不相交，要么一个是另一个的子集。请注意两个区域可能有一个公共点，但不会有完全重合的两个区域。

Tsukihi 作为众多舞者中的一员，定义了舞厅的宽敞度为有奇数个正在跳舞的舞者可移动区域交集的面积。如下图所示，当所有舞者都在舞动时阴影部分的面积就是舞厅的宽敞度。



不过毕竟没有能能整晚连续地跳舞，所以整个夜晚的时间被分成午夜前和午夜后两个半场。每个舞者仅在一个半场的时间内跳舞，另一个半场则和朋友们一起坐下休息。整场舞厅的宽敞度为两个半场的宽敞度的和。如下图所示为使上例宽敞度最大的半场划分方法：



对所有舞者不同的上下半场分组将导致不同的整场宽敞度。你的任务是找到最大可达的宽敞度。

输入：

输入第一行为一个正整数$n(1 \le n \le 1000)$——舞者数量。

接下来的n行每行描述一个舞者的活动范围，第i行包含3个由空格分隔的整数$x_i, y_i, r_i(-10^6 \le x_i, y_i \le 10^6, 1 \le r_i \le 10^6)$，表征一个圆心位于$(x_i, y_i)$，半径为$r_i$的圆形活动范围。

输出：

输出一个十进制数字——通过把所有舞者分组到上下半场可达的最大整场宽敞度。

当绝对或相对误差之一不超过10^{-9}时答案会被接受。具体地，假设你提交的答案为a，陪审团的参考值为b，则答案被接受当且仅当$\frac{|a-b|}{\max(|b|,1)} \le 10^{-9}$。

测试样例：

Input	Output
5 2 1 6 0 4 1 2 -1 3 1 -2 1 4 -1 1	138.23007676
8 0 0 1 0 0 2 0 0 3 0 0 4 0 0 5 0 0 6 0 0 7 0 0 8	289.02652413

注释：

第一个测试样例就是题述中的例子。

注：此题用到根树的通用存储方式——用一个数组纪录父亲节点+用一个向量数组记录孩子节点。

注 2：此题中用于判断两圆相互包含的方法：圆心距≤半径差。

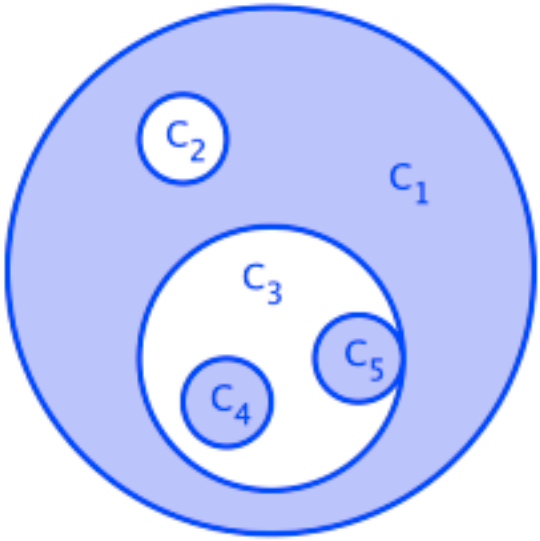
注 3：本题程序主体在计算圆面积时只计算半径平方，只在最终结果乘上圆周率π。这样的操作使得程序主体计算过程从浮点计算变为 64 位整型计算，大大提高了机器计算效率。

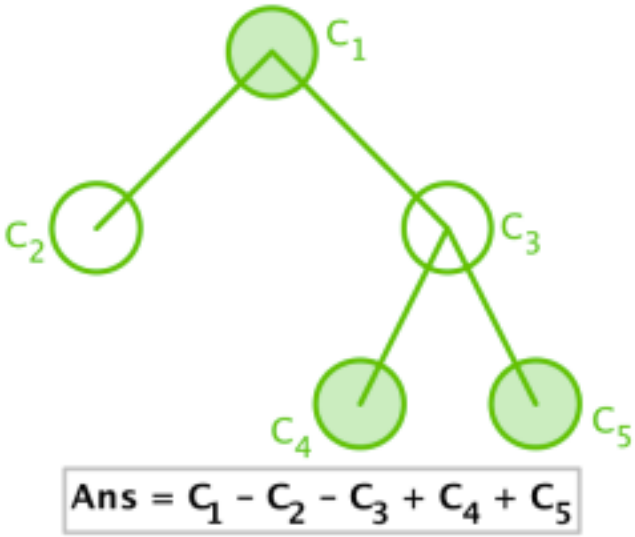
Tutorial 分析:

圆周不相交，也就是说，每个圆要么完全被别的圆包含，要么就是最外层的那些圆之一。所以你有没有从中看到一个树／森林结构呢？

对每个圆C_i构造一个节点，其权重为圆的面积πr_i^2。其父节点是那个直接包含它的圆，也就是所有包含C_i的圆中半径最小的那个。如果一个圆本身处于最外层，那么它对应于一个根。整个树结构可以在$T\sim O(n^2)$的时间内构建起来。

考虑只有一棵树的情况：宽敞度就等于所有深度为偶数的节点的权重和 减去 所有深度为奇数的节点的权重和。





现在，我们把原始树／森林分隔成两个不相交集。这启发我们构思一个 dp 解——对于一个节点u，考虑它在第一个集合和第二个集合中祖先节点个数的奇偶性。在这样的状态定义下，令$f[u][0/1][0/1]$表示当u在第一和第二个集合中分别会有偶／奇数个祖先节点时，（原树中）以其为根的子树最大可达的宽敞度（注意u在原树中的所有后代不一定与u在相同分组）。

这个递归可以自底向上地在$T\sim O(1)$内完成，最终答案为所有根节点u的$f[u][0][0]$的和。时间复杂度，建树$T\sim O(n^2)$，动规$T\sim O(n)$。完整递归过程见如下代码：

完整代码：

```
#include <algorithm>
#include <cmath>
#include <vector>
#include <cstdio>

const int N = 1005;
const double PI = acos(-1);
int x[N], y[N], r[N];
int par[N]; // Father node.
std::vector<int> chi[N];      // Children nodes.
int n;

void read() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%d %d %d", &x[i], &y[i], &r[i]);
}

bool contain(int u, int v) {
    return (long long) (x[u] - x[v]) * (x[u] - x[v]) + (long long) (y[u] - y[v]) * (y[u] - y[v]) <= (long long) (r[u] - r[v]) * (r[u] - r[v]);
}

void build_tree() {
```

```
    for (int i = 0; i < n; i++) {
        par[i] = -1;
        for (int j = 0; j < n; j++)
            if (r[j] > r[i] && contain(i, j))
                if (par[i] == -1 || r[par[i]] > r[j]) par[i] = j;
        chi[par[i]].push_back(i);
    }
}

long long f[N][2][2];
void dfs_dp(int u) {
    long long g[2][2] = {{0}};
    for (int v : chi[u]) {
        dfs_dp(v);
        for (int i = 0; i <= 1; i++)
            for (int j = 0; j <= 1; j++)
                g[i][j] += f[v][i][j];
    }
    for (int i = 0; i <= 1; i++)
        for (int j = 0; j <= 1; j++)
            f[u][i][j] = std::max(
                // u is assigned to the first group.
                g[i^1][j] + (long long) r[u] * r[u] * (i == 0 ? 1 : -1),
                // u is assigned to the second group.
                g[i][j^1] + (long long) r[u] * r[u] * (j == 0 ? 1 : -1)
                );
}

void solve() {
    long long ans = 0;
    for (int i = 0; i < n; i++)
        if (par[i] == -1) {
            dfs_dp(i);
            ans += f[i][0][0];
        }
    printf("%.8lf\n", (double)ans * PI);
}

int main() {
    read();
    build_tree();
    solve();
}
```

2.1.5 Codeforces 814C (Round 418)

An impassioned circulation of affection (动态规划，答案表)

题述：

Nadeko 的生日就要到了！家里的房间在她的装饰下，墙上最显眼的位置挂着一串彩色剪纸构成的花环。她想哥哥 Koyomi 一定会喜欢的！

不过现在的花环似乎还不能让 Nadeko 完全满意，她决定再好好修整一番。花环由n片剪纸构成，从左到右编号$1 \ldots n$，第i片的颜色s_i由一个小写英文字母表示。Nadeko 将对最多m片剪纸重新上色（仍由一个小写英文子母表示）。在这之后，她将在这一串剪纸中找到仅有颜色为c（哥哥 Koyomi 最喜欢的颜色）的剪纸组成的子串，将其长度定义为花环的 Koyomity。

例如，假设花环可以写为“kooomo”，哥哥 Koyomi 最喜欢的颜色是‘o’。在所有只含有‘o’的子串中，“ooo”的长度为3是其中最长的一个，故这个花环的 Koyomity 为3。

但是随之而来的问题是 Nadeko 并不确定哥哥最喜欢的颜色是什么，对于如何修整花环她也摇摆不定。她有q种方案，每种方案可由一个整数m_i和一个小写字母‘c_i’表示，意义同上。你的任务是计算每种计划下最大可达的 Koyomity。

输入：

输入第一行为一个整数$n(1 \le n \le 1500)$——花环的长度。

第二行为由n个小写英文字母$s_1 s_2 \ldots s_n$构成的字符串——花环的初始颜色。

第三行为一个整数$q(1 \le q \le 200{,}000)$——Nadeko 的修整方案数量。

接下来的q行分别描述q种方案，其中第i行为一个整数m_i（最大重绘颜色数）后跟一个空格然后是一个小写英文字母‘c_i’（哥哥最喜欢的颜色）。

输出：

输出 q 行。对于每个方案，输出一行包含一个整数——可达的最大 Koyomity。

测试样例：

Input	Output
6 koyomi 3 1 o 4 o 4 m	3 6 5
15 yamatonadeshiko 10 1 a 2 a 3 a 4 a 5 a 1 b 2 b 3 b 4 b 5 b	3 4 5 7 8 1 2 3 4 5
10 aaaaaaaaaa	10 10

2 10 b 10 z	

注释：

样例一中，共有三种方案：

- 方案一中，最多只能重绘1片剪纸。将‘y’重绘成‘o’使得花环变为“kooomi”，此时 Koyomity 为可达最大值3。
- 方案二中，最多可以重绘4片剪纸。“oooooo”的 Koyomity 为最大值6。
- 方案三中，最多可以重绘4片剪纸。“mmmmmi”和“kmmmmm”都具有最大的 Koyomity 为6。

WeiYong 的 2D-dp 解：

该题要求连续回答最多200,000个问题，于是每一个问题的回答一定得在$T\sim O(1)$完成。对于本题中的输入形式，数字m_i的取值空间$1 \le m_i \le n \le 1500$，字符c_i的取值空间为长度26的字母表，于是问题空间大小为$26 * 1500$，故用二维数组预先记录所有问题的答案。于是原问题转化为以下问题：给定目标字符tar，在$T\sim O(n^2)$内求解m从1取到n的答案。该问题可动态规划求解：

dp[i][j]记录从在允许替换最多 i 个字符的情况下，以第 j 为结尾的全 tar 子串最大长度，则[1]转移方程：$dp[i][j] = \begin{cases} dp[i][j-1]+1 & s[j]=tar \\ dp[i-1][j-1]+1 & s[j]\neq tar \end{cases}$[2]边界条件：$dp[0][j] = (s[j] == tar), dp[i][0] = (s[0] == tar)$，[3]所求：$memo[tar][m] = \max_j dp[m][j]$。基于以上分析，可写出如下核心代码：

状态转移：

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j < n; j++)
        if (s[j] == tar) dp[i][j] = dp[i][j - 1] + 1;
        else dp[i][j] = dp[i - 1][j - 1] + 1;
```

平凡解：

```
for (int j = 0, counter = 0; j < n; j++) {
    if (s[j] == tar) counter++;
    else counter = 0;
    dp[0][j] = counter;
}
for (int i = 0; i <= n; i++)
    dp[i][0] = (s[0] == tar);
```

所求：

```
memo[tar][m] = INT_MIN;
for (int j = 0; j < n; j++)
    memo[tar][m] = max(memo[tar][m], dp[m][j]);
```

完整代码：

```
#include <algorithm>
#include <vector>
#include <map>
#include <cstdio>

const int N = 1505;
int dp[2][N];
```

```
char s[N];
std::map<char, std::vector<int>> memo;
int n, q;

void read() {
	scanf("%d", &n);
	scanf("%s", s);
}

void calculate(std::vector<int> &memo_tmp, char tar) {
	int cur = 0, prev = 1;
	memo_tmp[0] = INT_MIN;
	for (int j = 0, counter = 0; j < n; j++) {
		if (s[j] == tar) counter++;
		else counter = 0;
		dp[cur][j] = counter;
		memo_tmp[0] = std::max(memo_tmp[0], dp[cur][j]);
	}
	for (int j = 0; j < n; j++)
		dp[prev][j] = j == 0 ? 1 : 0;
	for (int i = 1; i <= n; i++) {
		cur = !cur, prev = !prev;
		int mx = dp[cur][0];
		for (int j = 1; j < n; j++) {
			dp[cur][j] = s[j] == tar ? dp[cur][j - 1] + 1 : dp[prev][j - 1] + 1;
			mx = std::max(mx, dp[cur][j]);
			memo_tmp[i] = mx;
		}
		for (int j = 0; j < n; j++)
			dp[prev][j] = j == 0 ? 1 : 0;
	}
}

void pre_cal() {
	for (int i = 0; i < n; i++) {
		char c = s[i];
		if (memo.find(c) != memo.end()) continue;
		memo[c].resize(n + 5);
		calculate(memo[c], c);
	}
}

void solve() {
	scanf("%d", &q);
	for (int i = 0; i < q; i++) {
```

```
            int m;
            char tar;
            scanf("%d %c", &m, &tar);
            if (memo.find(tar) != memo.end())
                  printf("%d\n", memo[tar][m]);
            else printf("%d\n", m);
      }
}

int main() {
      read();
      pre_cal();
      solve();
}
```

2.1.6 Codeforces 811C (Round 416, Div2)

Vladik and Memorable Trip (动态规划)

题述：

Vladik 曾经有过这样一次有趣的旅行。

Vladik 在车站等车时，共有n个乘客（包括 Vladik）在排队等车。他们按照顺序站成一排，其中第i个人的目的地是城市$a[i]$。

列车长在乘客队伍中划定了若干不相交的区间段（区间段的并**不用**覆盖整个序列），位于同一区段的游客将被分配到相同的车厢。区间段的划分遵循以下规则：若区段中有一个乘客的目的地是城市x，那么所有以城市x为目的地的游客都得在该区间段内，进而都进入同一车厢。这意味着，对于任意游客，他要么位于包含起目的城市的车厢，要么不上车。

一个区段（一个车厢）的舒适度定义为该区间段内所有目的城市编号的按位异或值，整个列车的舒适度定义为所有车厢的舒适度之和。

请帮助 Vladik 计算列车的最大可达舒适度。

输入：

第一行为一个整数$n(1 \le n \le 5000)$——排队人数。

第二行为n个由空格分隔的整数$a_1, a_2, \dots, a_n (0 \le a_i \le 5000)$——$a_i$为第$i$个人的目的城市。

输出：

输出一个整数——列车的最大可能舒适度。

测试样例：

Input	Output
6 4 4 2 5 2 3	14
9 5 1 3 1 5 2 4 2 5	9

注释：

测试样例一中，一种最优的划分方式如下：$[4,4]\ [2,5,2]\ [3]$，相应的舒适度为$4 + (2 \oplus 5) + 3 = 4 + 7 + 3 = 14$

测试样例二中，一种最优的划分方式如下：5 1 [3] 1 5 [2, 4, 2] 5，相应的舒适度为$3 + (2 \oplus 4) = 3 + 6 = 9$

参考 unused 的 dfs_dp 解（惊艳代码）:

前缀上的动态规划，dps_dp(i)返回子序列$a_1 \dots a_i$的结果，顶层函数 dfs_dp(1)返回原问题的解，递归过程中生成并调用备忘录$dp[i]$。整个过程在$T\sim O(n)$时间内完成。

代码**惊艳**之处:

· 用于记录当前值在数组中最早／最晚出现位置的数组$L[a[i]]$和$R[a[i]]$，在原数组的输入过程中一趟生成。

· 利用了按位异或的性质$a \oplus b \le a + b$将整个过程从 Tutorial 的$T\sim O(n^2)$复杂度降至$T\sim(n)$复杂度。

· 当用$a[1 \dots n]$存储原始数据更方便操作时就不要从$a[0]$开始存数据。

完整代码:

```
#include <algorithm>
#include <cstdio>

const int N = 1000005;

int a[N], L[N], R[N];
int n;
void read() {
```

```
	scanf("%d", &n);
	for (int i = 1; i <= n; i++) {
		scanf("%d", &a[i]);
		if (L[a[i]] == 0) L[a[i]] = i;
		R[a[i]] = i;
	}
}

int dp[N];
bool vis[N];
int dfs_dp(int u) {
	if (u == 0) return 0;
	if (!vis[u]) {
		vis[u] = true;
		dp[u] = dfs_dp(u - 1);
		int rl = L[a[u]];
		bool over = false;
		int cur_xor = 0;
		for (int v = u; v >= rl; v--) {
			if (R[a[v]] > u) {
				over = true;
				break;
			}
			rl = std::min(rl, L[a[v]]);
			if (v == L[a[v]]) cur_xor ^= a[v];
		}
		if (!over) dp[u] = std::max(dp[u], cur_xor + dfs_dp(rl - 1));
	}
	return dp[u];
}

void solve() {
	printf("%d\n", dfs_dp(n));
}

int main() {
	read();
	solve();
}
```

2.1.7 Codeforces 768D (Round #399)

Jon and orbs (动态规划，滚动数组，答案表)

题述：

为了抵御百行者的入侵，Jon Snow 要收集一种生物——orbs。一共有k种不同的 obrs，他每种至少需要一个。墙北堰木树下有一只 orbs 皇后，每天产一个小 orbs，这只小 orbs 为每个种类的可能性完全相同。由于墙北地形险要危机四伏，Jon 想确定一个最少的天数，使得当他排出一个巡逻队员去墙北收集 orbs 时小 orbs 种类齐全的概率不少于$\frac{p_i-\epsilon}{2000}$，其中$\epsilon < 10^{-7}$ 。

为了有备无患，他想得到q个不同p_i值所对应的答案。由于 Jon 正忙于和 Sam 探讨战略，他向你寻求帮助。

输入：

第一行为两个由空格分隔的整数$k, q(1 \le k, q \le 1000)$——orbs 的种类数和问题数。

接下来的q行每行为一个整数$p_i(1 \le p_i \le 1000)$——第i个问题。

输出：

输出q行，其中第i行为第i个问题的答案。

测试样例：

Input	Output
1 1 1	1
2 2 1 2	2 2

参考 Tutorial 的动态规划算法:

模拟种群发展过程，备忘录为一个记录第n天 obrs 的**种类数量离散分布表**——$dp[n][0 \dots k]$记录第n天有$0 \dots k$种不同 orbs 的概率。根据**下一天是否产生新的物种**，有状态转移方程：$dp[n][x] = dp[n-1][x]\frac{x}{k} + dp[n-1][x-1]\frac{k-x+1}{k}$，问题的解是满足$dp[n][k] > \frac{p_i-\epsilon}{2000}$的最小$n$。程序中这个条件可以写为$2000 * dp[n][k] > p_i - \epsilon$。由于 dp[n][..]只和 dp[n - 1][..]有关，故备忘录在时间维度上交替迭代即可。

完整代码:

```
#include <cstdio>

const int K = 1005;
const int P = 1005;
const double eps = 1e-7;

int k, q;
void read() {
    scanf("%d %d", &k, &q);
}

double dp[2][K];
int ans_tb[P];
void solve() {
    int p = 1;
```

```
    dp[0][0] = 1;
    for (int i = 1; p <= 1000; i++) {
        int cur = i & 1, prev = !cur;
        dp[cur][0] = 0;
        for (int j = 1; j <= k; j++)
            dp[cur][j] = (dp[prev][j] * j + dp[prev][j - 1] * (k - j + 1)) / k;
        while (p - eps <= 2000 * dp[cur][k] && p <= 1000)
            ans_tb[p++] = i;
    }

    for (int i = 0; i < q; i++) {
        int tmp;
        scanf("%d", &tmp);
        printf("%d\n", ans_tb[tmp]);
    }
}

int main() {
    read();
    solve();
}
```

注：备忘录的生成方式如下

图例：默认值，设定值 / 平凡解，计算值

$dp[i][j]$	0	1	2	3	4	5	6	...	k
0	1	0	0	0	0	0	0		0
1	0	1	0	0	0	0	0		0
2	0	$\frac{1}{k}$	$\frac{k-1}{k}$	0	0	0	0		0
3	0	$\frac{1}{k^2}$	$\frac{3(k-1)}{k^2}$	$\frac{(k-1)(k-2)}{k^2}$	0	0	0		0
...	...	...	...	...	...	...	...		...

· 第一行为设定值（利用全局变量的默认零值），第1个值要人工设定为1。

· 从第二行开始每行第1个元素为设定值。

· 从第二行开始每行的第2到第$k+1$个元素由**上方**和**左上方**的元素计算得到。

2.1.8 Codeforces 777E (Round #401)

Hanoi Factory (动态规划，BIT)

题述：

你一定听说过经典的汉诺塔问题，但你知不知道有一家工厂专门生产找个游戏用的铁环？很久以前，又一个埃及的统治者汉诺工厂的工人们修建一座尽可能高的塔。对于这个突如其来的命令工人们并没有实现的准备，于是他们着手使用已经生产过的铁环来搭建这座塔。

工厂库存n个铁环。对于其中第i个铁环，其内径为a_i，外径为b_i，高为h_i。目标是选择这些铁环中的一部分，并按照以下限制条件堆叠成塔：

1. 外径非递减序，即若铁环 j 在铁环 i 上必有：$b_j \le b_i$。
2. 环之间不嵌套，即若铁环 j 在铁环 i 上必有：$b_j > a_i$。
3. 总高度最大。

输入：

第一行为一个整数$n(1 \le n \le 10^5)$——工厂库存的铁环数量。

接下来的n行中，每行为三个整数$a_i, b_i, h_i (1 \le a_i, b_i, h_i \le 10^9, b_i > a_i)$——分别为第$i$个铁环的内径、外经和高度。

输出：

输出一个整数——塔能搭建的最大高度。

测试样例：

Input	Output
3 1 5 1 2 6 2 3 7 3	6
4 1 2 1 1 3 3 4 6 2 5 7 1	4

注释：

测试样例 1 中，最优方案是使用全部铁环并按照自底向上 3, 2, 1 的顺序堆叠。

测试样例 2 中，将铁环 3 放到铁环 4 上高度为3，将铁环 1 放到铁环 2 上得到高度为4。

参考 XLightDog，基于 BIT 加速备忘录的动态规划：

首先将所有铁环按外经从大到小排序，对于外径相同的铁环按内径从大到小排序（这就是重叠子问题的**拓扑顺序**）。**动态规划模拟从下往上摞铁环的过程**，**重叠子问题**为：前i个铁环能够达到的最大高度。**依次将当前铁盘摞在顶层内径小于其外径的最高已有组合上**，将顶层内径对应的最大高度记录在备忘录中。这里状态转移需要备忘录中索引小于当前铁环外径的前缀最大值，用 BIT 存储备忘录将总时间复杂度从$T\sim O(n^2)$进一步优化到$T\sim O(n\log n)$。

备忘录：memo$[a_i]$记录以当前出现过的所有内径a_i的圆盘为顶的最大堆叠高度。则状态转移方程$memo[a_i] = h_i + max_{a_j<b_i}(memo[a_j])$。进一步，用有序取值空间中的坐标表征$a_i, b_i$，将存储备忘录的空间复杂度从$S\sim O(\max\{a_i, b_i\})$优化到$S\sim O(n)$。

注 1：对于外径相同的铁环，应先考虑内径大的后考虑内径小的。或者把外经相同的铁环视作一个整体，其内径为最小内径。

注 2：BIT 可以用于 prefix-max 问题的前提是 UPD 操作只**进行非负点更新**。

完整代码：

```
#include <algorithm>
#include <iostream>

const int N = 100005;
const long long INF = INT64_MAX;

struct ring {
	int a, b, h;
};

int n;
ring rings[N];
void read() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++)
		scanf("%d %d %d", &rings[i].a, &rings[i].b, &rings[i].h);
}

bool cmp(ring u, ring v) {
	return u.b > v.b || (u.b == v.b && u.a > v.a);
}

int sp[N << 1];

long long BIT[(N << 1) + 1];

long long prefix_max(int p) {
	long long ans = -INF;
	for (p++; p; p -= p & -p)
		ans = std::max(ans, BIT[p]);
	return ans;
}

void upd(int p, long long val) {
	for (p++; p <= N << 1; p += p & -p)
		BIT[p] = std::max(BIT[p], val);
}

void solve() {
	for (int i = 0; i < n; i++) {
		sp[i << 1] = rings[i].a;
		sp[(i << 1) + 1] = rings[i].b;
	}
	std::sort(sp, sp + 2 * n);
	int m = (int)(std::unique(sp, sp + 2 * n) - sp);
```

```
    std::sort(rings, rings + n, cmp);
    for (int i = 0; i < n; i++) {
        rings[i].a = (int)(std::lower_bound(sp, sp + m, rings[i].a) - sp);
        rings[i].b = (int)(std::lower_bound(sp, sp + m, rings[i].b) - sp);
    }

    for (int i = 0; i < n; i++) {
        long long tmp_h = prefix_max(rings[i].b - 1) + rings[i].h;
        upd(rings[i].a, tmp_h);
    }

    printf("%lld\n", prefix_max((N << 1) - 1));
}

int main() {
    read();
    solve();
}
```

Nero 同样基于 BIT 的动态规划代码(利用 STL 的简洁实现):

```
#include <vector>
#include <algorithm>
#include <tuple>
#include <stdio.h>

int main()
{
    int n;
    scanf("%d", &n);
    std::vector<std::tuple<int, int, int>> vec(n);
    std::vector<int> ss(n);
    for (int i = 0; i < n; ++ i) {
        int a, b, h;
        scanf("%d%d%d", &a, &b, &h);
        vec[i] = std::make_tuple(b, a, h);
        ss[i] = a;
    }
    std::sort(ss.begin(), ss.end());
    ss.erase(std::unique(ss.begin(), ss.end()), ss.end());
    std::sort(vec.begin(), vec.end());
    std::vector<long long> bit(ss.size(), 0);
    long long result = 0;
    for (int i = n - 1; i >= 0; -- i) {
        int a, b, h;
        std::tie(b, a, h) = vec[i];
```

```
            long long tmp = 0;
            for (int d = (int)(std::upper_bound(ss.begin(), ss.end(), b - 1) - ss.begin()) - 1; d >= 0; d -= ~d & d + 1)
                tmp = std::max(tmp, bit[d]);
            tmp += h;
            result = std::max(tmp, result);
            for (int d = std::lower_bound(ss.begin(), ss.end(), a) - ss.begin(); d < ss.size(); d += ~d & d + 1)
                bit[d] = std::max(bit[d], tmp);
        }
        printf("%lld\n", result);
}
```

0 base 树状数组——Nero 说:

“这不是循环减 1，这是 0-base 的树状数组。

~d&d+1 算的是 d+1 的二进制表示中最低位 1 代表的值。”

2.1.9 Codeforces 788C (Codeforces Round #407)

Function Again (动态规划)

题述：给定一个序列$a_1, a_2, \ldots, a_n$，定义函数$f(l,r) = \sum_{i=l}^{r-1}(-1)^{i-l}|a_i - a_{i+1}|$，求函数$f(l,r)$的最大值。
参数范围：$2 \le n \le 1e5$，$-1e9 \le a_i \le 1e9$，$1 \le l < r \le n$

WeiYong 的 1D-dp 解（个人首个独立动归解，纯纪念）:

存储绝对差分$d[i] = |a[i] - a[i+1]|$，备忘录：s[N], t[N]，其中$s[i] = \max f(i,r), t[i] = \min f(i,r)$，最终解为$\max s[i]$。对于每个$i$，$s[i]$的取值有两种可能，$d[i]$或$d[i] - t[i+1]$。

完整代码:

```
const int N = 100000 + 10;
int a[N];
long long d[N], s[N], t[N];
int n;

int main() {
    ios::sync_with_stdio(false);
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> a[i];
    for (int i = 0; i < n - 1; i++) {
        d[i] = 1ll * a[i] - a[i + 1];
        if (d[i] < 0) d[i] = -d[i];
    }

    s[n - 2] = t[n - 2] = d[n - 2];
    for (int i = n - 3; i >= 0; i--) {
        s[i] = max(d[i] - t[i + 1], d[i]);
        t[i] = min(d[i] - s[i + 1], d[i]);
    }
    long long mx = INT64_MIN;
    for (int i = 0; i < n - 1; i++)
        mx = max(mx, s[i]);

    cout << mx << endl;

    return 0;
}
```

2.1.10 Codeforces 792C (Educational Codeforces Round #18)

Divided by Three (动态规划，数论)

题述：

黑板上写着一个正整数n，长度不超过10^5个十进制位，你希望通过去掉尽量少的数位使之变成一个美丽数。

满足以下条件的数字被称为美丽数：至少一位，无前缀零，3的倍数。例如：$0, 99, 10110$是美丽数，$00, 03, 122$不是美丽数。

编写程序对于给定的正整数n，通过去掉尽量少的数位使之转化为一个美丽数。你可以去掉任意一组数位。例如，这些数位不必在n中相邻。

如果无法通过去掉某些位使n变成美丽数，输出-1。如果有多组解，输出任意一个即可。

输入：

输入一行为一个没有前缀零的正整数$n(1 \le n < 10^{100000})$。

输出：

输出一个数字——在去掉最少的数位后的到的美丽数。若无解输出-1。

测试样例：

Input	Output
1033	33
10	0
11	-1

注释：

测试样例 1 中，去掉首位后得到三的倍数，此时仍存在前缀零，进一步将其去掉后得到美丽数。故最少去掉的位数为2。

Square1001 的 1D-dp 解:

$dp[i][j]$表示原十进制数字的前$i-1$位的满足所有位和模3余j的子串的最大长度。

完整代码:

```
#include <algorithm>
#include <string>
#include <iostream>

const int N = 100005;
int a[N], dp[N][3]; // dp[i][j] represents length of a[0...i-1]'s longest subsrting whose sum % 3 == j.
std::string s;

int main() {
    std::ios::sync_with_stdio(false);
    std::cin >> s;
    for (int i = 0; i < s.size(); i++)
        a[i] = s[i] - '0';

    dp[0][0] = dp[0][1] = dp[0][2] = INT_MIN;
    for (int i = 0; i < s.size(); i++)
        for (int j = 0; j < 3; j++) {
            dp[i + 1][j] = std::max(dp[i][j], dp[i][(j - a[i] + 9) % 3] + 1);
            if (a[i] != 0 && j % 3 == a[i] % 3)
                dp[i + 1][j] = std::max(dp[i + 1][j], 1);
```

```
        }

    if (dp[s.size()][0] < 0) {
        if (s.find('0') != std::string::npos)
            std::cout << 0 << std::endl;
        else std::cout << -1 << std::endl;
    } else {
        int m = 0;
        std::string ret;

        int i = (int)s.size() - 1;
        while(1) {
            if (dp[i + 1][m] == 1 && a[i] != 0 && a[i] % 3 == m % 3) {
                ret += s[i];
                break;
            } else if (dp[i][(m - a[i] + 9) % 3] + 1 == dp[i + 1][m]) {
                ret += s[i];
                m = (m - a[i] + 9) % 3;
            }
            i--;
        }
        reverse(ret.begin(), ret.end());

        std::cout << ret << std::endl;
    }

    return 0;
}
```

该解应注意以下几点：前缀零的消除（生成备忘录的过程、生成解的过程），生成解过程中的结束判定。

2.1.11 Astar 2017 资格赛 1004

度度熊的午饭时光(动态规划)

复杂度限制：

- 时间限制：2000/1000 ms (Java/Others)
- 内存限制：32768K/32768K (Java/Others)

题述：

度度熊最期待每天的午饭时光，因为早饭菜品清淡，晚饭减肥不敢吃太多（胖纸的忧伤 T.T）。

百度食堂的午餐超级丰富，祖国各大菜系应有尽有，度度熊在每个窗口都有爱吃的菜品，而且他还为喜欢的菜品打了分，吃货的情怀啊(>.<)。

但是，好吃的饭菜总是很贵，每天的午饭预算有限，请帮度度熊算一算，怎样打饭才能买到最好吃的饭菜？（不超过预算、不重样、午餐得分最高的情况下，选择菜品序号加和最小，加和相等时字典序最小的组合）。

输入：

第一行一个整数T，表示T组数据。每组测试数据将以如下形式从标准输入读入：

B

N

$score\_1\ cost\_1$

$score\_2\ cost\_2$

⋮

$score\_N\ cost\_N$

第一行，正整数$B(0 \le B \le 1000)$，代表午餐的预算。

第二行，正整数$N(0 \le N \le 100)$，代表午餐可选的菜品数量。

从第三行到第$N+2$行，每行两个正整数，以空格分隔，$score_i$表示菜品的得分，$cost_i$表示菜品的价格$(0 \le score_i, cost_i \le 100)$。

输出：

对于每组数据，输出两行：第一行输出 “Case #i:”。代表第i组测试数据。第二行输出菜品的总得分和总花费，以空格分隔。第三行输出所选菜品的序号，菜品序号从1开始，由空格分隔。

测试样例：

Input	Output
2 29 6 9 10 3 4 6 5 7 20 10 9 15 11 0 2 2 23 10 12	Case #1: 34 29 2 3 5 6 Case #2: 0 0

WeiYong 的动态规划解:

预算范围$[0, 1000]$，菜品数量范围$[0, 100]$。作前缀上的动态规划，求解子问题：前i种菜品以j的预算可以获得的最高得分。以备忘录$dp[i][j]$记录之。按照当前菜品是否购买写出如下状态转移方程:

$$dp[i][j] = \begin{cases} \max\{dp[i-1][j], dp[i-1][j-c[i]] + s[i]\} & j \geq c[i] \\ dp[i-1][j] & j < c[i] \end{cases}$$

题目中要求菜品序号和最小，就是要求作前缀上的动态规划，如作后缀上的动态规划，则菜品序号和最大。另一方面，输出要求字典需最小，即要求升序输出菜品序号。

完整代码:

```
#include <algorithm>
#include <vector>
#include <cstring>
#include <cstdio>

const int maxn = 105;
const int maxb = 1005;

int n, b;
int s[maxn], c[maxn];

void read() {
    scanf("%d %d", &b, &n);
    for (int i = 1; i <= n; i++)
        scanf("%d %d", &s[i], &c[i]);
}

int dp[maxn][maxb];
std::vector<int> ans;

void generate_ans() {
    ans.clear();
    if (dp[n][b] == 0) return;

    for (int idx = n, cost = b; idx >= 1; idx--) {
        if (dp[idx][cost] == dp[idx - 1][cost]) continue;
        else {
            ans.push_back(idx);
            cost -= c[idx];
        }
    }
}

void solve(int t) {
    memset(dp, 0, sizeof(dp));
    for (int i = 1; i <= n; i++)
        for (int j = 0; j <= b; j++)
            if (j >= c[i]) dp[i][j] = std::max(dp[i - 1][j], dp[i - 1][j - c[i]] + s[i]);
            else dp[i][j] = dp[i - 1][j];
```

```
	generate_ans();

	int cost = 0;
	for (int i = 0; i < ans.size(); i++)
		cost += c[ans[i]];
	printf("Case #%d:\n%d %d\n", t, dp[n][b], cost);
	for (int i = (int)ans.size() - 1; i >= 0; i--)
		printf("%d%c", ans[i], " \n"[i == 0]);
}

int main() {
	int T;
	scanf("%d", &T);
	for (int t = 1; t <= T; t++) {
		read();
		solve(t);
	}
}
```

2.2.1 Codeforces 817E (Educational Codeforces Round 23)

Choosing the Commander (Tire，数据结构)

题述：

你也许还记得上轮比赛中，Vova 正在玩一款名为《暴怒帝国》的战略游戏。

Vova 想建立一个大规模的军队，却忘了其中最重要的一个人——指挥官。于是他决定招募一个指挥官，人选为最受士兵尊敬的那个人。

每个士兵用一个整数p_i表征其个性，每个指挥官由两个整数p_j, l_j分别表示其个性和领导力。士兵i尊重指挥官j当且仅当$p_i \oplus p_j < l_j$（$\oplus$是按位异或运算符）。

初始 Vova 军队为空，军队会发生如下三类变动：

- 1 p_i——一个个性为p_i的士兵进入军队。
- 2 p_i——一个个性为p_i的士兵离开军队。
- 3 p_i l_i——Vova 招募了一个个性p_i，领导力为l_i的指挥官。

对于所有第三类变动，Vova 想知道被招募的指挥官受当前军队内多少士兵尊重。

输入：

第一行为一个整数$q(1 \le q \le 100000)$——变动数量。

接下来的q行每行描述一次变动，格式如上。

输出：

每次发生第三类变动时，输出一个整数——尊重当前指挥官的士兵数量。

测试样例：

Input	Output
5 1 3 1 4 3 6 3 2 4 3 6 3	1 0

注释：

测试样例中，经过前两次变动后军队中有两个士兵个性分别为3和4。然后 Vova 招募了一个个性为6领导力为3的指挥官，只有一个士兵尊重他（$4 \oplus 6 = 2 < 3, 3 \oplus 6 = 5 > 3$）。随着个性为4的士兵离开，新招募的指挥官不受任何士兵尊敬。

2.2.2 Codeforces 785E (Round #404)

Anton and Permutation (数据结构，Sqrt Decomposition)

题述：

Anton 喜欢排列，尤其喜欢玩转元素的顺序。注意n个元素的排列为一个序列$\{a_1, a_2, \dots, a_n\}$，其中1到n之间的每个整数出现一次。

这天，Anton 拿到了一个新排列并开始玩转它。他进行以下操作q次：交换排列中某两个位置的元素。每次交换之后他问他的朋友 Vanya，新排列中有多少逆序对。逆序对的数量为满足$1 \le i < j \le n$且$a_i < a_j$的有序数对(i, j)的数量。

Vanya 已经厌倦于回答 Anton 的 SB 问题。他希望你能写一个程序代替他回答这些问题。

初始时 Anton 的排列为$\{1, 2, \dots, n\}$，即对于所有$1 \le i \le n$有$a_i = i$。

输入：

第一行为两个整数n和$q(1 \le n \le 200000, 1 \le q \le 50000)$——分别为排列长度和操作次数。

接下来的q行每行包含两个整数$l_i, r_i(1 \le l_i, r_i \le n)$——Anton 第$i$次操作所交换的两个元素的序号。注意 Anton 第i此操作的两个元素的序号可能相同。排列中元素序号从1开始。

输出：

输出q行。其中第i行为第i次操作后排列的逆序数。

测试样例：

Input	Output
5 4 4 5 2 4 2 5 2 2	1 4 3 3
2 1 2 1	1
6 7 1 4 3 5 2 3 3 3 3 6 2 1 5 1	5 6 7 7 10 11 8

注释：

考虑第一个测试样例。

第一次操作后排列为$\{1, 2, 3, 5, 4\}$。只有一个逆序对：$(5, 4)$。

第二次操作后排列为$\{1, 5, 3, 2, 4\}$。有四个逆序对：$(2, 3), (2, 4), (2, 5), (3, 4)$。

第三次操作后排列为$\{1, 4, 3, 2, 5\}$。有三个逆序对：$(2, 3), (2, 4), (3, 4)$。

第四次操作后排列没有变化，故逆序对的数量仍为三个。

mr.banana 基于 sorted-segment 的解法：

首先考虑蛮力法统计逆序数，遍历排列中每一组数对，累计反序数对数量，这一步的时间复杂度为$T \sim O(n^2)$，总复杂度为$T \sim O(qn^2) \approx 2 \times 10^{15}$。对于现代计算机需要数年完成。

我们可以在$T \sim O(n)$的时间内统计一次交换两个数据引起逆序数的变化量——为二者组成的数对+位置和值同时在二者之间的元素数×2，此时总复杂度优化成$T \sim O(qn) \approx 10^{10}$。

更进一步，如建立 Sorted Segment，将原数组按分段排序存储。则建立 Sorted Segment 的时间为 $T\sim O(n\log n)$；而对于每一步操作，更新某一段的时间为$T\sim O(\sqrt{n}\log\sqrt{n})$，统计变化量的时间为$T\sim O(\sqrt{n}\log\sqrt{n})$，于是此时总复杂度为$T\sim O((q\sqrt{n}+n)\log\sqrt{n})\approx 1.97\times 10^8$，现代计算机可以在秒级时间内完成。

完整代码:

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

const int maxn = 200005;
const int SQ = 450;

int a[maxn];
std::vector<int> seg[500];
int n, q;
void read() {
	scanf("%d %d", &n, &q);
	std::iota(a, a + n, 1);
	for (int i = 0; i < n; i++)
		seg[i / SQ].push_back(a[i]);
}

void upd(int u) { // update the segment with idx u.
	seg[u].clear();
	for (int i = u * SQ; i < (u + 1) * SQ && i < n; i++)
		seg[u].push_back(a[i]);
	std::sort(seg[u].begin(), seg[u].end());
}

int count(int l, int r, int small, int large) {
	if (small > large) std::swap(small, large);
	int ans = 0;
	while (l % SQ && l < r) {
		if (small < a[l] && a[l] < large) ans++;
		l++;
	}
	while (r % SQ && l < r) {
		if (small < a[r - 1] && a[r - 1] < large) ans++;
		r--;
	}
	for (int i = l / SQ; i < r / SQ; i++)
		ans += lower_bound(seg[i].begin(), seg[i].end(), large) - upper_bound(seg[i].begin(), seg[i].end(), small);
	return ans;
}
```

```
int operate(int l, int r) {
	int ans = 1;
	std::swap(a[l], a[r]);
	upd(l / SQ);
	upd(r / SQ);
	ans += count(l, r + 1, a[l], a[r]) << 1;
	return a[l] > a[r] ? ans : -ans;
}

void solve() {
	long long ans = 0;
	for (int i = 0; i < q; i++) {
		int l, r;
		scanf("%d %d", &l, &r);
		if (l != r) {
			l--, r--;
			if (l > r) std::swap(l, r);
			ans += operate(l, r);
		}
		printf("%lld\n", ans);
	}
}

int main() {
	read();
	solve();
}
```

可见，算法的关键在于找到一种更新和查询都快捷的数据结构。适用于此题的数据结构就是 **Sqrt Decomposition**。

总结：什么样的问题适合用 sorted-segment 组织数据。

2.2.3 Codeforces 798E (Round #410)

Mike and code of a permutation (构造性算法，线段树，图论，拓扑排序)

题述：

Mike 发现了一种新的对排列编码的方式。对于排列$P=[p_1,p_2,\dots,p_n]$，编码方式如下：

令n长序列$A=[a_1,a_2,\dots,a_n]$为编码结果。对于从1到n的每一个i，将第一个未被标记的满足$p_i<p_j$的$j(1\le j\le n)$赋给a_i（即令$a_i=j$）并标记j。若这样的j不存在，就给a_i赋-1（即令$a_i=-1$）。

Mike 忘记了原始排列，但它记录了编码之后的结果。你的任务很明确：找到任意一个编码后与原始排列结果相同的排列。

保证至少存在一个可行的原始排列。

输入：

第一行为一个整数$n(1\le n\le 500000)$——排列的长度。

第二行为n个整数$a_1,a_2,\dots,a_n\left(1\le a_i\le n或a_i=-1\right)$——Mike 的排列的编码结果。

保证 A 中的所有非负值互不相同。

输出：

输出一行n个整数$p_1,p_2,\dots,p_n(1\le p_i\le n)$——编码结果和输入相同的一个排列。注意排列中的数字应互不相同。

测试样例：

Input	Output
6 2 -1 1 5 -1 4	2 6 1 4 5 3
8 2 -1 4 -1 6 -1 8 -1	1 8 2 7 3 6 4 5

注释：

对于测试样例一的输出排列

$i=1$，最小的j是2，因为$p_2=6>2=p_1$

$i=2$，没有满足条件的j，因为6是排列中最大的数

$i=3$，最小的j是1，因为$p_1=2>1=p_3$

$i=4$，最小的j是5，因为$p_5=5>3=p_4$

$i=5$，没有满足条件的j，因为2已经被标记过了

$i=6$，最小的j是4，因为$p_4=4>3=p_6$

Tutorial（作者）的拓扑排序算法:

关于 Mike 的编码方式的分析：首先，编码结果$a_1,a_2,\dots,a_n$反映了原排列下标的一种顺序信息。那么自然想到利用拓扑排序恢复序列。

构造有向图，以全排列$p_1,p_2,\dots,p_n$的索引为顶点，以利用编码结果可以确定满足$p_a>p_b$的有序点对(a,b)为有向边，则通过对该图进行拓扑排序即可得到一个可行的全排列。具体地，设S为索引集拓扑排序的逆序结果，则可以通过给p_{S_i}赋i得到一组可行解。

实现拓扑排序前用$a_i=n+1$代替$a_i=-1$。

怎么找到这些有向边呢？定义由a的值到索引的映射序列$b_1,b_2,\dots,b_n$，$b_i=\begin{cases}j & \exists a_j=i\\ n+1 & \text{others}\end{cases}$。首先，所有满足$b_i\ne n+1$的有序点对$(i,b_i)$都构成一条有向边。另一方面，原排列还需要满足在编码第i位时，遍历所有$1\le j<a_i$的索引j的过程中，对于其中的未标记的索引j应有$p_i>p_j$。怎么知道索引是否已经被标记了呢？只需看j被谁标记了，若b_j（标记j的点）在i后面（即$b_j>i$），则编码第i位时，j尚未被标记。综上，存在有向边(i,j)，当且仅当（$j=b_i$）或（$1\le j<a_i$且$i\ne j$且$b_j>i$）。

怎样快速找到由顶点i出发的所有边呢？假设我们已经访问过一些顶点，对于访问过的顶点j我们给b_j赋 0（已访问标记），现在给定顶点i，我们希望快速找到由其直接指向的一个顶点j或返回空，如果这个操作能在小于集合规模平方的复杂度内完成则问题解决。如前所述，第一类条件$j = b_i$且$1 \le b_i \le n$，容易检查。第二类条件$1 \le j < a_i$且$b_j > i$，考虑索引值在$1 \le j < a_i$中b_j最大的那个j。若$b_j > i$则我们找到一个有向边(i, j)，否则返回空。可以用**线段树**来查找这样的j，每访问一个顶点更新一次**线段树**（作该顶点的已访问标记）。整体上我们访问n个顶点最多查询**线段树**$2 \times (n-1)$次（其中$(n-1)$次访问新顶点，$(n-1)$次返回空）。

$T \sim O(N \log N) / S \sim O(N)$

注：此题关于**线段树**的实现可作为$S \sim O(4n)$线段树模版。

跟进：此题中的**线段树**采用自顶向下递归实现的**树状数组**方法，$S \sim O(4n)$。进一步**线段树**有更通用的自底向上迭代实现的**树状数组**表示，见 796F，$S \sim O(2n)$。

完整代码：

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

const int maxn = 500005;

int a[maxn], b[maxn];

std::pair<int, int> t[maxn << 2];

void build(int u, int v, int node) {
    if (u == v) t[node] = {b[u], u};
    else {
        int m = (u + v) >> 1;
        build(u, m, node << 1);
        build(m + 1, v, node << 1 | 1);
        t[node] = std::max(t[node << 1], t[node << 1 | 1]);
    }
}

void del(int p, int u, int v, int node) {
    if (u == v) t[node] = {0, p};
    else {
        int m = (u + v) >> 1;
        if (p <= m) del(p, u, m, node << 1);
        else del(p, m + 1, v, node << 1 | 1);
        t[node] = std::max(t[node << 1], t[node << 1 | 1]);
    }
}

std::pair<int, int> query(int l, int r, int u, int v, int node) {
    if (l > r) return {0, 0};
    if (l <= u && v <= r) return t[node];
    std::pair<int, int> ans = {0, 0};
```

```
    int m = (u + v) >> 1;
    if (l <= m) ans = std::max(ans, query(l, r, u, m, node << 1));
    if (m + 1 <= r) ans = std::max(ans, query(l, r, m + 1, v, node << 1 | 1));
    return ans;
}

int n;
void read() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        b[i] = n + 1;
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        if (a[i] != -1) b[a[i]] = i;
        else a[i] = n + 1;
    }
}


bool vis[maxn];
std::vector<int> S;
void dfs(int idx) {
    vis[idx] = true;
    del(idx, 1, n, 1);
    if (b[idx] != n + 1 && !vis[b[idx]]) dfs(b[idx]);
    while (1) {
        auto tmp = query(1, a[idx] - 1, 1, n, 1);
        if (tmp.first > idx) dfs(tmp.second);
        else break;
    }
    S.push_back(idx);
}

int ans[maxn];
void solve() {
    build(1, n, 1);
    for (int i = 1; i <= n; i++)
        if (!vis[i]) dfs(i);
    for (int i = 0; i < n; i++)
        ans[S[i]] = i + 1;

    for (int i = 1; i <= n; i++)
        printf("%d%c", ans[i], " \n"[i == n]);
}
```

```
int main() {
    read();
    solve();
}
```

关于此题实现中的若干注意事项:

· 为什么对于不存在后位较大值的索引i，$a[i]$的值要从-1改为$n+1$？

——因为对于该顶点出发的第二类有向边需要在$[1,n]$的范围内找到最大值。代码中用**红色加粗标出**的位置需要$a[i]-1$的值为n。

· 为什么对于没有被作为后位较大值标记其他前位的索引i，$b[i]$的值要设为$n+1$？

——因为它作为比较过的点（不够标记）时，应该被判定作为一条有向边。代码中用**黄色加粗标出**的位置需要$tmp.first$的值大于n。

2.2.4 Codeforces 796F (Round #408)

Sequence Recovery (Segment Tree, Bitmask)

题述：

Zane 曾经有一个长度为n的良序列$a_1, a_2, \ldots, a_n$——但他不小心把这个序列给丢了。

良序列定义为每个元素a_i是整数且满足$0 \le a_i \le 10^9$。

然而，Zane 记得在游戏过程中自己对这个序列所做过的m次操作。

操作一共有两类：

1. 给定l和r，找到下标i满足$l \le i \le r$的所有整数的最大值。
2. 给定k和d，将d作为值赋给下标为k的整数。

游戏结束后，他将序列恢复到所有操作进行之前的状态。也就是说，序列a恢复到未受所有两类操作影响的状态。接着他就在某时某刻丢掉了这个序列。

幸运的是，Zane 记得自己做过的操作以及它们的顺序，而且他还记得所有第一类操作的结果。更进一步，在所有以相同顺序进行相同操作会返回相同结果的良序列中，他知道他的序列有最大的 cuteness 值。

序列的 cuteness 定义为序列内所有数字的按位异或的值，例如，Zane 的序列的 cuteness 为$a_1 \oplus a_2 \oplus \ldots \oplus a_n$。

Zane 知道基于自己可以提供的关于那个序列的信息可能不足以完全将其复原，所以他将会很高兴找到一个长度为n的良序列$b_1, b_2, \ldots, b_n$满足：

1. 当以相同的顺序进行相同操作时，序列b能和序列a产生相同的结果。
2. 和原始序列a有相同的 cuteness 值。

如果这样的序列存在，请找到它。否则，就意味着 Zane 可能记错了某些信息，这也不是不可能。

输入；

第一行为两个整数n和$m(1 \le n, m \le 3 \times 10^5)$——分别为 Zane 的序列$a$的长度和游戏过程中的操作数量。

接下来的m行，其中第i行开始为一个整数$t_i(t_i \in \{1, 2\})$——为第i次操作的种类。

如果操作种类为 1（$t_i = 1$），则后跟三个整数l_i, r_i, x_i——分别为区间的左、右端点，和下表位于区间$[l_i, r_i]$的整数的最大值。

如果操作种类为 2（$t_i = 2$），则后跟两个整数k_i和$d_i(1 \le k_i \le n, 0 \le d_i \le 10^9)$——表示下表为$k_i$的整数再次次操作后被赋值为$d_i$。

保证对所有(i, j)有$x_i \ne x_j$，其中$1 \le i < j \le m$且$t_i = t_j = 1$。

操作的输入顺序与游戏中的执行顺序相同。也就是说，先输入的操作先执行，后输入的操作后执行，以此类推。

输出：

如果合法的良序列不存在，输出“NO”（不带引号）。

否则，在第一行输出“YES”（不带引号），然后再第二行输出 n 个由空格分隔的整数$b_1, b_2, \ldots, b_n(1 \le b_i \le 10^9)$。

如果有多个满足条件的解，输出任意一个即可。

测试样例：

Input	Output
5 4 1 1 5 19 1 2 5 1 2 5 100 1 1 5 100	YES 19 0 0 0 1
5 2 1 1 5 0 1 1 5 100	NO

注释：

测试样例一中，不难验证该良序列符合题意。特别地，其 cuteness 值为$19 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 19$

测试样例二中，两次操作的记录显然矛盾，故不存在合法良序列。

Tutorial 的分析与基于线段树的算法：

首先，分析每个整数的取值上界。注意到当一个整数被 UDP 操作赋值后，其后出现的 RMQ 操作将不能再提供关于其初始值的任何信息。那么总可以用 naive 的方法在$T\sim O(nm)$内获得所有整数的取值上界——被某次 UPD 赋值前所有包含该位置 RMQ 结果的最小值，但这样的算法显然太慢。

用**线段树**优化。相比于常规的点更新和区间查询，这里采用点查询和区间更新。无需 lazy propagation，这个过程可以在$T\sim O((n+m)\log n)$内完成。

至此，每个整数的当前值为满足记录的取值上界。对该序列过一遍所有操作，用另一个线段树检查该序列是否满足所有 RMQ 记录的结果，如果不满足，则可以确信无解。为什么呢？因为如果该序列 RMQ 的结果比记录小，我们也无法使其更大，这是由于这个较小的结果要么是被其他 RMQ 限制了上界要么是被 UPD 强制赋值；而如果该序列 RMQ 的结果比记录大，则只有一种可能——两类操作的记录有冲突。

最后还剩一步——使$(b_1 \oplus b_2 \oplus \ldots \oplus b_n)$最大，考虑如下两种情况：

Case #1. 自由数的数量大于 1（自由数是没有 RMQ 限制其最大取值的数）。这时将其中一个自由数置为$2^{29}-1$（低 29 位置 1），其他所有数置为$[2^{29}, 10^9]$内的任意值即可（高位置 1）。

Case #2. 否则，对于序列中出现过两次以上的取值上界，将其中一个最高位 1 置零并将所有低位置 1，如此得到一个按位或最大值 orMax。若还有一个自由数，则从高位到低位给 orMax 补 1，只要保证该自由数不超过10^9。

注：此题中线段树 t2[N << 1]的迭代实现，用作基础篇线段树（迭代）实现的通用模版。

完整代码：

```
#include <algorithm>
#include <map>
#include <iostream>

const int maxn = 300005, maxm = 300005;
const int INF = 2e9;

int ret[maxn];
int n;

int t[maxn << 1];

void upd(int l, int r, int x) {
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if (l & 1) t[l] = std::min(t[l], x), l++;
```

```
            if (r & 1) --r, t[r] = std::min(t[r], x);
        }
}

int find_max(int p) {
        int ans = INF;
        for (p += n; p > 0; p >>= 1)
            ans = std::min(ans, t[p]);
        return ans;
}

int t2[maxn << 1];

void build() {
        for (int i = 0; i < n; i++)
            t2[i + n] = ret[i];
        for (int i = n - 1; i > 0; i--)
            t2[i] = std::max(t2[i << 1], t2[i << 1 | 1]);
}

void upd(int p, int val) {
        for (t2[p += n] = val; p > 1; p >>= 1)
            t2[p >> 1] = std::max(t2[p], t2[p ^ 1]);
}

int find_max(int l, int r) {
        int ans = -INF;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (l & 1) ans = std::max(ans, t2[l++]);
            if (r & 1) ans = std::max(ans, t2[--r]);
        }
        return ans;
}

int type[maxm], a[maxm], b[maxm], c[maxm];
int m;
void read() {
        scanf("%d %d", &n, &m);
        for (int i = 0; i < n << 1; i++)
            t[i] = INF;
        for (int i = 0; i < n; i++)
            ret[i] = INF + 1;
        for (int i = 0; i < m; i++) {
            scanf("%d", &type[i]);
            if (type[i] == 1) {
```

```
            scanf("%d %d %d", &a[i], &b[i], &c[i]);
            a[i]--, b[i]--;
            upd(a[i], b[i] + 1, c[i]);
        } else {
            scanf("%d %d", &a[i], &b[i]);
            a[i]--;
            if (ret[a[i]] == INF + 1) ret[a[i]] = find_max(a[i]);
        }
    }
    for (int i = 0; i < n; i++)
        if (ret[i] == INF + 1) ret[i] = find_max(i);
}

std::map<int, int> mp;
void solve() {
    // Check validity of ret[0 ... n - 1].
    build();
    for (int i = 0; i < m; i++) {
        if (type[i] == 1) {
            if (find_max(a[i], b[i] + 1) != c[i]) {
                printf("NO\n");
                return;
            }
        } else upd(a[i], b[i]);
    }

    // Answer exist.
    printf("YES\n");
    for (int i = 0; i < n; i++)
        mp[ret[i]]++;

    // If amount of free number is more than 1.
    if(mp[INF] > 1) {
        for (int i = 0; i < n; i++)
            if (ret[i] == INF) {
                ret[i] = 1 << 29, mp[INF]--;
                if (mp[INF] == 0) ret[i] = (1 << 29) - 1;
            }
        for (int i = 0; i < n; i++)
            printf("%d%c", ret[i], " \n"[i == n - 1]);
        return;
    }

    // Otherwise, find the best value for the only free number(if any).
    int or_ret = 0;
```

```
    for (int i = 0; i < n; i++) {
        if (ret[i] == 0 || ret[i] == INF) continue;
        mp[ret[i]]--;
        if (mp[ret[i]]) {
            int tmp = ret[i], pow = 0;
            while (tmp) pow++, tmp >>= 1;
            ret[i] = (1 << (pow - 1)) - 1;
        }
        or_ret |= ret[i];
    }
    int tobe = 0;
    for (int cur = 29; cur >= 0; cur--) {
        if (or_ret & (1 << cur)) continue;
        if (tobe + (1 << cur) <= 1e9) tobe += (1 << cur);
    }
    for (int i = 0; i < n; i++)
        printf("%d%c", ret[i] == INF ? tobe : ret[i], " \n"[i == n - 1]);
}

int main() {
    read();
    solve();
}
```

关于使用$S\sim(2N)$和$S\sim(4N)$两种**树状数组**实现**线段树**的总结：

线段树提出的初衷是利用折半（二分）的思想将区间不断二分到每个元素，通过建立关于这$N-1$个区间操作的备忘录，如区间查询（RMQ）、点更新（UPD），从而在对数时间内得到任意区间的操作结果。

首先，**$4N$树状数组**的实现遵循了上文提出**线段树**的初衷，其创建、RMQ 以及 UPD 操作都是自顶向下采用“二分区间”的思想，递归实现。时间复杂度方面，建树用时$T\sim O(2N-1)$——即**线段树**的总节点数，RMQ 和 UPD 用时都是$T\sim O(\log N)$——即**线段树**的深度；空间复杂度方面，**树状数组**本身开销$S\sim O(4N)$虽然当$N=2^k$时$S\sim O(2N)$，另外递归栈的开销为$S\sim O(\log N)$——即**线段树**的深度。

当$N=2^k$不成立时，为了保持**树状数组**索引和**线段树**节点的对应关系，**$4N$树状数组**中的一部分空间是一直不被使用的，最坏情况下未利用空间达到$2N$。注意到**线段树**的叶节点数总是N，而非叶子结点的数量总是$N-1$，为了设计一种充分利用空间的存储结构，于是有了**$2N$树状数组**实现的**线段树**，我们将**线段树**的叶子节点（即原数组元素）存储在**树状数组**的后半部分，然后根据**树状数组**索引和**线段树**节点对应关系自底向上迭代生成前$N-1$个数组元素（即非叶子节点）。注意这个自底向上生成的**线段树**已经具有和开始设计时不同的逻辑结构，其特点之一是非叶子节点的区间长度都是偶数（自底向上建树的直接结果）。由于可以直接定位叶子节点在**树状数组**中的位置，故其 RMQ 和 UPD 也都是自底向上进行。时间复杂度方面，建树用时$T\sim O(2N-1)$——即**线段树**的总节点数，RMQ 和 UPD 用时都是$T\sim O(\log N)$——即**线段树**的深度；空间复杂度方面，**树状数组**本身开销$S\sim O(2N)$，无其他开销。

总的来说，**$4N$树状数组**实现的**线段树**是由逻辑结构出发设计存储结构，而**$2N$树状数组**实现的**线段树**则是在前者基础上由优化的存储结构设计的逻辑结构。前者构建和使用都是依据二分区间的原则自顶向下进行，而后者的构建和使用都从定位叶结点开始依据**树状数组**索引和**线段树**节点的对应关系自底向上进行。最后，二者都是用$tree[1]$存储整个区间的信息，首地址$tree[0]$不被使用。

2.2.5 Codeforces 817F (Educational Codeforces Round 23)

MEX Queries

题述：

给定一组整数的集合，初始为空，进行n次 Query：

Query 的种类有以下三种：

- 1 l r ——增加区间$[l,r]$内所有缺失的整数。
- 2 l r ——移除所有$[l,r]$之间的整数。
- 3 l r ——反转区间$[l,r]$——增加$[l,r]$内所有缺失的，并移除$[l,r]$内所有存在的整数。

每次 Query 之后输出当前集合的 MEX 值——当前集合未出现的最小正整数。

输入：

第一行为一个整数$n(1 \le n \le 10^5)$。

接下来的n行每行为3个整数$t,l,r(1 \le t \le 3, 1 \le l \le r \le 10^{18})$——Query 类型、左右边界。

输出：

每次 Query 后输出当前集合的 MEX 值。

测试样例：

Input	Output
3 1 3 4 3 1 6 2 1 3	1 3 1
4 1 1 3 3 5 6 2 4 4 3 1 6	4 4 4 1

注释：

对于测试样例一，每次 Query 后集合如下：

1. $\{3,4\}$——增加$[3,4]$
2. $\{1,2,5,6\}$——$\{3,4\}$被移除，$[1,6]$之间的其他数字被加入
3. $\{5,6\}$——$\{1,2\}$被移除

2.3.1 Codeforces 822D (Round 422, Div 2)

My pretty girl Noora (预处理)

题述：

Noora 就读的 Pavlopolis 大学正准备举办一场“Pavlopolis 小姐”的选美比赛，下面说明一下比赛规则。

比赛由若干轮次构成。假设初始有n个女孩儿参加选美，参赛选手被分为人数相同的若干组，用x表示单组选手的数量。x可以取任意值（即每轮的x值可以不同）。组内采取两两比较的单循环方式，例如如果某组内有x个参赛选手，那么将进行$\frac{x(x-1)}{2}$次两两比较，然后每一组的第一名入围下一轮比赛。举例来说，如果n个参赛选手被分为x人一组，则$\frac{n}{x}$人会入围下一轮次。直到剩下一个选手，获得“Paclopolis 小姐”的殊荣。

但对于委员会来说，两两比较的操作太枯燥了。他们希望恰当地选取每轮的分组数量以使得全部过程的“两两比较”次数达到最小。如果初始参赛人数为n，则用$f(n)$表示这一最少的比较次数。

举办方脑子瓦特了，他们给了 Noora 三个整数t,l,r，然后叫这个可怜的女孩儿计算如下表达式的值：$t^0f(l)+t^1f(l+1)+\cdots+t^{r-l}f(r)$。由于结果可能较大，只需在模 1000000007 的意义下计算即可。举办方承诺如果 Noora 能给出上式的结果他们就让 Noora 当选美比赛的志愿者。但对于可怜的小 Noora 来说数学并不是强项，她向 Leha 求助，Leha 转而寻求你的帮助。

输入：

输入一行包含三个整数$t,l,r(1\le t<10^9+7,2\le l\le r\le 5\times 10^6)$.

输出：

输出一个整数——模 1000000007 意义下上式的结果。

测试样例：

Input	Output
2 2 4	19

注释：

上例中，表达式为$(2^0f(2)+2^1f(3)+2^2f(4))\bmod 1000000007$。

$f(2)=1$，2 个女孩儿为一组，一次比较即可决出优胜者。

$f(3)=3$，3 个女孩儿为一组，两两比较共3次可以决出优胜者。

$f(4)=3$，4 个女孩儿分为两组，每组2个选手，第一轮共进行2次比较，第二轮一次比较，共 2+1=3 次比较。如果第一轮将4个参赛选手视作一组，那么需要$C_4^2=6$次两两比较才能决出优胜者。

将上述讨论结果带入表达式中，得到$(2^0\times 1+2^1\times 3+2^2\times 3)\bmod 1000000007=19$。

WeiYong 的动态规划解

输入规模达到5×10^6，我们需要在线性时间内算出体重表达式的值，那么每一项都得在 T~O(1)时间内得到。于是我们不需要预先生成$t^0,t^1,\ldots,t^{r-l}$和$f(1),f(2),\ldots,f(n)$的值。模 1000000007 意义下t的指数很容易在$T\sim O(n)$时间内生成。对于$f(n)$，显然它和$f(n/k)$有关。我们首先在草纸上写出$f(1)\ldots f(18)$的值:

$f(1)=0$
$f(2)=1$
$f(3)=\frac{3\times 2}{2}=3$
$f(4)=\min\left\{\begin{matrix}\frac{4\times 3}{2}=6\\ 2\times f(2)+f(2)=2\times 1+1=3\end{matrix}\right\}=3$
$f(5)=\frac{5\times 4}{2}=10$

$$f(6)=\min\begin{Bmatrix}\dfrac{6\times5}{2}=15\\2\times f(3)+f(2)=2\times3+1=7\\3\times f(2)+f(3)=3\times1+3=6\end{Bmatrix}=6$$

$$f(7)=\frac{7\times6}{2}=21$$

$$f(8)=\min\begin{Bmatrix}\dfrac{8\times7}{2}=28\\2\times f(4)+f(2)=2\times3+1=7\\4\times f(2)+f(4)=4\times1+3=7\end{Bmatrix}=7$$

$$f(9)=\min\begin{Bmatrix}\dfrac{9\times8}{2}=36\\3\times f(3)+f(3)=3\times3+3=12\end{Bmatrix}=12$$

$$f(10)=\min\begin{Bmatrix}\dfrac{10\times9}{2}=45\\2\times f(5)+f(2)=2\times10+1=21\\5\times f(2)+f(5)=5\times1+10=15\end{Bmatrix}=15$$

$$f(11)=\frac{11\times10}{2}=55$$

$$f(12)=\min\begin{Bmatrix}\dfrac{12\times11}{2}=66\\2\times f(6)+f(2)=2\times6+1=13\\3\times f(4)+f(3)=3\times3+3=12\\4\times f(3)+f(4)=4\times3+3=15\\6\times f(2)+f(6)=6\times1+6=12\end{Bmatrix}=12$$

$$f(13)=\frac{13\times12}{2}=78$$

$$f(14)=\min\begin{Bmatrix}\dfrac{14\times13}{2}=91\\2\times f(7)+f(2)=2\times21+1=43\\7\times f(2)+f(7)=7\times1+21=28\end{Bmatrix}=28$$

$$f(15)=\min\begin{Bmatrix}\dfrac{15\times14}{2}=105\\3\times f(5)+f(3)=3\times10+3=33\\5\times f(3)+f(5)=5\times3+10=25\end{Bmatrix}=25$$

$$f(16)=\min\begin{Bmatrix}\dfrac{16\times15}{2}=120\\2\times f(8)+f(2)=2\times7+1=15\\4\times f(4)+f(4)=4\times3+3=15\\8\times f(2)+f(8)=8\times1+7=15\end{Bmatrix}=15$$

$$f(17)=\frac{17\times16}{2}=136$$

$$f(18)=\min\begin{Bmatrix}\dfrac{18\times17}{2}=153\\2\times f(9)+f(2)=2\times12+1=25\\3\times f(6)+f(3)=3\times6+3=21\\6\times f(3)+f(6)=6\times3+6=24\\9\times f(2)+f(9)=9\times1+12=21\end{Bmatrix}=21$$

...

通过观察很容易总结出若n为素数，则$f(n)=\frac{n(n-1)}{2}$，否则$f(n)=(n/a)f(a)+f(n/a)$，其中a是n最小的素因数（证明见后）。最后的问题是预先生成2到5×10^6范围内所有整数的最小素因数，可利用埃拉托色尼筛选法（逆向乘法）在$T\sim O(n\log\log n)$（近线性）时间内得到。

完整代码:

```
#include <cstdio>

const int N = 5000005;
const int MOD = 1000000007;

int t, l, r;
void read() {
    scanf("%d %d %d", &t, &l, &r);
}

int t_pow[N], f[N], min_fac[N];
void pre_process() {
    t_pow[0] = 1;
    for (int i = 1; i < N; i++)
        t_pow[i] = 1ll * t_pow[i - 1] * t % MOD;

    for (int i = 0; i < N; i++)
        min_fac[i] = i;
    for (int i = 2; i < N; i++) {
        if (min_fac[i] == i)
            for (int j = i << 1; j < N; j += i)
                if (min_fac[j] > i) min_fac[j] = i;
    }

    f[1] = 0;
    for (int i = 2; i < N; i++) {
        if (min_fac[i] < i) f[i] = (1ll * (i / min_fac[i]) * f[min_fac[i]] + f[i / min_fac[i]]) % MOD;
        else f[i] = 1ll * i * (i - 1) / 2 % MOD;
    }
}

void solve() {
    pre_process();

    long long ans = 0;
    for (int i = l; i <= r; i++) {
        ans += 1ll * t_pow[i - l] * f[i] % MOD;
        ans %= MOD;
    }
```

```
	printf("%d\n", (int)ans);
}

int main() {
	read();
	solve();
}
```

2.3.2 Codeforces 817D (Educational Codeforces Round 23)

Imbalanced Array (预处理，栈，线性时间算法)

题述：

给定n长序列a，定义区间的 imbalance 为区间极差，序列的 imbalance 为所有子区间的极差和。举例来说，序列$[1,4,1]$的 imbalance 为9，这是因为这个序列的6个子区间$[1],[1,4],[1,4,1],[4],[4,1],[1]$的极差分别为$0,3,3,0,3,0$。

给定序列求其 imbalance。

输入：

第一行为一个整型$n(1 \le n \le 10^6)$——序列长度。

第二行为由空格分隔的n个整数$a_1,a_2,\dots,a_n(1 \le a_i \le 10^6)$——序列元素

输出：

输出一行为一个整数——序列a的 imbalance。

测试样例：

input	output
3 1 4 1	9

Tutorial 分析:

首先，计算所有区间最大值的和s_1 与 所有区间最小值的和s_2，则序列 imbalance 就是s_1-s_2的值。以所有区间最大值的求法为例。首先对于0到$n-1$间的每个索引值i，用$l[i]$记录从$a[i]$开始向左最后一个满足$a[j]<a[i]$的索引值j，用$r[i]$记录从$a[i]$开始向右最后一个满足$a[j]\le a[i]$的索引值j，则以$a[i]$为**最左侧最大值**的区间数量为$(i-l[i]+1)*(r[i]-i+1)$，故最大值和可在$T\sim O(n)$为由$\sum_{i=0}^{n-1} a[i]*(i-l[i]+1)*(r[i]-i+1)$求得。

那么$l[i]$和$r[i]$怎样在线性时间内求解呢？以求解子区间最大值和过程中生成的数组$l[0\dots n-1]$的过程为例，从左到右当要求解$l[cur]$时，**栈内当前元素为所有可能成为$l[cur]-1$的索引**，具体地，从栈顶到栈底是数组$a[]$中从$a[cur-1]$开始向左的所有非降序元素的索引（相应地，生成$r[N]$的过程中为升序）。

由于计算完$l[i]$时索引i必位于栈顶，故生成$l[0\dots n-1]$的过程中每个元素进／出栈恰好为一次，故整个过程的$T\sim O(n)$。具体代码如下:

注：本体中求所有子区间最大值和的方法可推广到其他程序的相同子问题。

完整代码:

```
#include <stack>
#include <cstdio>

#define ll long long

const int N = 1000005;

int n;
int a[N];
void read() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++)
		scanf("%d", &a[i]);
}

int l[N], r[N];
```

```
std::stack<int> stk;

ll min_sum() {
    // Generate l[], r[].
    while (!stk.empty()) stk.pop();
    for (int i = 0; i < n; i++) {
        while (!stk.empty() && a[stk.top()] > a[i])
            stk.pop();
        if (stk.empty()) l[i] = 0;
        else l[i] = stk.top() + 1;
        stk.push(i);
    }
    while (!stk.empty()) stk.pop();
    for (int i = n - 1; i >= 0; i--) {
        while (!stk.empty() && a[stk.top()] >= a[i])
            stk.pop();
        if (stk.empty()) r[i] = n - 1;
        else r[i] = stk.top() - 1;
        stk.push(i);
    }
    // Calculate max sum.
    ll ans = 0;
    for (int i = 0; i < n; i++)
        ans += (i - l[i] + 1) * (r[i] - i + 1) * a[i];
    return ans;
}

ll max_sum() {
    // Generate l[].
    while (!stk.empty()) stk.pop();
    for (int i = 0; i < n; i++) {
        while (!stk.empty() && a[stk.top()] < a[i])
            stk.pop();
        if (stk.empty()) l[i] = 0;
        else l[i] = stk.top() + 1;
        stk.push(i);
    }
    // Generate r[].
    while (!stk.empty()) stk.pop();
    for (int i = n - 1; i >= 0; i--) {
        while (!stk.empty() && a[stk.top()] <= a[i])
            stk.pop();
        if (stk.empty()) r[i] = n - 1;
        else r[i] = stk.top() - 1;
        stk.push(i);
```

```
    }
    // Calculate min sum.
    ll ans = 0;
    for (int i = 0; i < n; i++)
        ans += (i - l[i] + 1) * (r[i] - i + 1) * a[i];
    return ans;
}

void solve() {
    ll ans = 0;
    ans += max_sum();
    ans -= min_sum();
    printf("%lld\n", ans);
}

int main() {
    read();
    solve();
}
```

注：实现过程中注意的问题，根据题述我们存储的是从目标开始向一个方向最后一个满足条件的索引值。为了达到相同效果当然也可以通过存储第一个不满足条件的索引值，然而在这种情况下需要注意的是，如果一直到边界都满足条件，则不满足条件的索引值应该相应为-1/n。

2.4.1 Codeforces 815B (Round 419, Div 1)

Karen and Test (纯数学，模式识别，组合数)

题述：

Karen 到学校了，今天她有一场数学考试。

考试是关于加减法的。很不巧，老师们都在忙着给 Codeforces 出题，没有足够的时间出一套完整的卷子。于是正常考试只有一道题，用于评分。

n个整数排成一行，Karen 的任务是交替在相邻两个数之间作加法和减法，并将结果写在它们的正下方一行。对如此得到的下一行进行相同的操作，知道最后一行只剩一个数字，规定第一个运算符为加。

注意，如果一行的最后一组运算是加法，则下一行第一组运算为减法，反之亦然（and vice versa）。作为评阅，老师只会检查最后一行的那个数字，如果正确就给满分，否则零分。

Karen 为考试已经做了充足的准备，但她担心自己可能在计算过程中某处出错而导致最终结果错误。你能给出按照如上过程得到的最后一行的数字吗？

由于最终结果可能很大，只需输出其模$10^9 + 7$的非负余数即可。

输入：

第一行为一个整数n，第一行给出的整数数量。

接下来的一行包含n个整数$a_i(1 \le a_i \le 10^9)$，为初始一行的所有整数。

输出：

输出一行为一个整数——最后一行的结果。（模 1000000007 意义下）

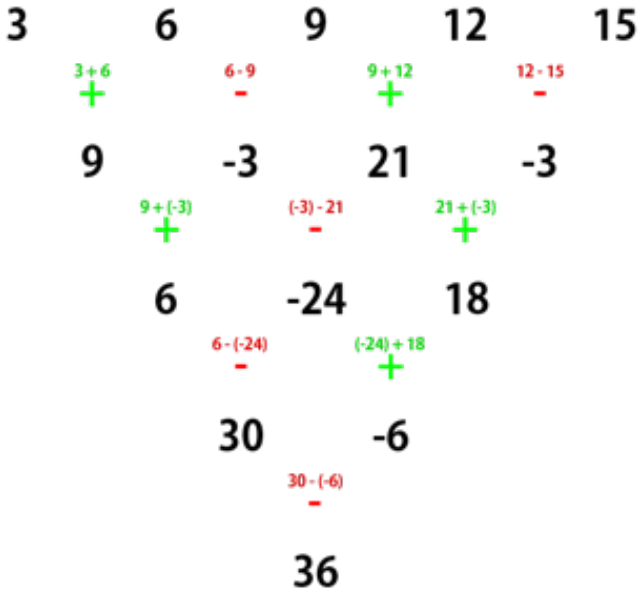
测试样例：

Input	Output
5 3 6 9 12 15	36
4 3 7 5 2	1000000006

注释：

样例 1 中，写在第一行的数字为$3, 6, 9, 12, 15$。

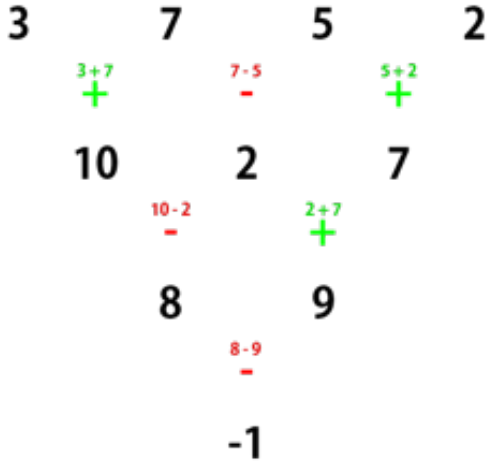
Karen 的操作如下：



最后一行的数字（模1000000007）为36。

样例 2 中，写在第一行的数字为$3, 7, 5, 2$。

Karen 的操作如下：



最后一行的数字为-1，模1000000007结果为1000000006.

参考 Tutorial 的线性解:

首先，输入规模上限2×10^5，如使用纯模拟，算法会以$T\sim O(n\text{^}2)$的复杂度超时。不难发现，这道题的关键在于找到最后一行由第一行n个数字线性组合的组合系数，它们由输入规模n唯一确定。如果能找到第一行第i个元素的组合系数对输入规模n的闭式解，那么就能在$T\sim O(n)$内得到原问题的解。

由于预感到最终系数会以组合数以类似二项式系数的形式表达，将原始数据用代数符号$a_0, a_1, \ldots, a_{n-1}$表示，我们在纸上写下输入规模$n = 1 \ldots 9$时的计算过程，试图从中发现一些规律。

我们发现：$n = 8, 6, 4, 2$时倒数第二行的两个多项式的组合系数为二项式系数、$n = 9, 5, 1$时最后一行的组合系数为二项式系数、$n = 7, 3$时倒数第三行的组合系数是二项式系数。于是我们猜测：组合系数随n的增长以4为周期成规律分布。具体地，当$n \equiv 0 (\mathrm{mod} 4)$时，倒数第二行两个多项式分别由原始数据中的全部**奇数项**和全部**偶数项**通

过二项式系数组合得到，二者通过**减法**得到最后一个多项式，用类似的方式总结出$n \equiv 1(\mathrm{mod}4)$，$n \equiv 2(\mathrm{mod}4)$，$n \equiv 3(\mathrm{mod}4)$时的情况可得到如下代码:

完整代码:

```
#include <cstdio>

const int N = 200005;
const int MOD = 1000000007;

int n;
int a[N];
void read() {
	scanf("%d", &n);
	for(int i = 0; i < n; i++)
		scanf("%d", &a[i]);
}

int fac[N << 1], facinv[N << 1];
int pw(int a, int b) {	// Quick power for a^b.
	int ans = 1;
	while (b) {
		if (b & 1) ans = 1ll * ans * a % MOD;
		a = 1ll * a * a % MOD;
		b >>= 1;
	}
	return ans;
}

void preProcess() { // Inverse elements in MOD.
	fac[0] = 1;
	for (int i = 1; i <= 200000 << 1; i++)
		fac[i] = 1ll * fac[i - 1] * i % MOD;
	facinv[400000] = pw(fac[400000], MOD - 2);
	for (int i = 399999; i >= 0; i--)
		facinv[i] = 1ll * facinv[i + 1] * (i + 1) % MOD;
}

int C(int x, int y) {	// Combination in MOD: C_x^y.
	if(x < y) return 0;
	return 1ll * fac[x] * facinv[y] % MOD * facinv[x - y] % MOD;
}

void solve() {
	preProcess();
	long long ans = 0;
	if (n % 4 == 0) {
```

```
        long long c1= 0, c2 = 0;
        for (int i = 0; i <= n / 2 - 1; i++)
            c1 += 1ll * C(n / 2 - 1, i) * a[2 * i] % MOD, c1 %= MOD;
        for (int i = 0; i <= n / 2 - 1; i++)
            c2 += 1ll * C(n / 2 - 1, i) * a[2 * i + 1] % MOD, c2 %= MOD;
        ans = (c1 - c2 + 2 * MOD) % MOD;
    } else if (n % 4 == 1) {
        for (int i = 0; i <= n / 2; i++)
            ans += 1ll * C(n / 2, i) * a[2 * i] % MOD, ans %= MOD;
    } else if (n % 4 == 2) {
        long long c1 = 0, c2 = 0;
        for (int i = 0; i <= n / 2 - 1; i++)
            c1 += 1ll * C(n / 2 - 1, i) * a[2 * i] % MOD, c1 %= MOD;
        for (int i = 0; i <= n / 2 - 1; i++)
            c2 += 1ll * C(n / 2 - 1, i) * a[2 * i + 1] % MOD, c2 %= MOD;
        ans = (c1 + c2 + 2 * MOD) % MOD;
    } else {      // n % 4 == 3.
        long long c1 = 0, c2 = 0, c3 = 0;
        for (int i = 0; i <= n / 2 - 1; i++)
            c1 += 1ll * C(n / 2 - 1, i) * a[2 * i] % MOD, c1 %= MOD;
        for (int i = 0; i <= n / 2 - 1; i++)
            c2 += 1ll * C(n / 2 - 1, i) * a[2 * i + 1] % MOD, c2 %= MOD;
        for (int i = 0; i <= n / 2 - 1; i++)
            c3 += 1ll * C(n / 2 - 1, i) * a[2 * i + 2] % MOD, c3 %= MOD;
        ans = (c1 + 2 * c2 - c3 + 3ll * MOD) % MOD;
    }
    printf("%lld\n", ans);
}

int main() {
    read();
    solve();
}
```

注：第一次提交时代码中红色部分没有取摸导致 64 位整型溢出错误。

2.4.2 京东 2017 秋招笔试

求幂 (纯数学，计数问题)

题述：

东东在学习数学的过程中发现了一种有趣的等式：例如$27^2 = 9^3$。东东好奇在整数 n 以内满足这种形式的等式数量。请帮助东东找到满足$1 \le 1, b, c, d \le n$的所有等式$a^b = c^d$的数量。由于最终结果可能较大只需给出其模1000000007的余数。

输入：

输入一个整数n，如上所述。

输出：

满足$1 \le 1, b, c, d \le n$的所有等式$a^b = c^d$的数量，结果对1000000007取模。

参数限制：

$1 \le n \le 100000$

测试样例：

Input	Output
2	6

测试样例中，$n = 6$，合法的等式有：

$1^1 = 1^1$

$1^1 = 1^2$

$1^2 = 1^1$

$1^2 = 1^2$

$2^1 = 2^1$

$2^2 = 2^2$

共6组。

官方解析:

完整代码:

```
#include <algorithm>
#include <set>
#include <iostream>

using ll = long long;

const int MOD = 1000000007;

int n;

std::set<int> st;
void solve() {
    ll ans = 1ll * n * n % MOD;
    for (int i = 2; i * i <= n; i++) {
        if (st.find(i) != st.end()) continue;
        int tmp = i;
        int cnt = 0;
        while (tmp <= n) {
            st.insert(tmp);
```

```
                cnt++;
                tmp *= i;
            }
            for (int x = 1; x <= cnt; x++)
                for (int y = 1; y <= cnt; y++) {
                    int g = std::__gcd(x, y);
                    int xx = x / g, yy = y / g;
                    ans += n / std::max(xx, yy), ans %= MOD;
                }
        }
        ans += 1ll * (n - st.size() - 1) * n, ans %= MOD;
        std::cout << ans << std::endl;
}

int main() {
        std::cin >> n;
        solve();
}
```

2.5.1 Codeforces 780B (Round 403)

The Meeting Place Cannot Be Changed (二分搜索)

题述：

Bytecity 的主干路是一条南北向的直线。作为地标，从道路最南端的建筑物开始向南有以米丈量的坐标。

沿公路的某些位置上分布着n个朋友，其中第 i 个位于坐标为x_i的位置上并可以以不超过v_i的速度朝南 / 北任意方向移动。

你的任务是计算使所有n个朋友汇聚到道路上某点所需的最短时间。注意这里汇合点的坐标不一定是整数。

输入：

第一行为一个整数n——朋友的数量。

第二行为n个由空格分隔的整数$x_1, x_2, \dots, x_n (1 \le x_i \le 10^9)$——朋友们所处的位置，单位为米。

第三行为n个由空格分隔的整数$v_1, v_2, \dots, v_n (1 \le v_i \le 10^9)$——每个朋友的移动速度上限，单位为米每秒。

输出：

输出可使所有n个朋友汇聚到一点的最短时间。

当且仅当绝对或相对误差不超过10^{-6}时答案被视为正确的。具体地，假设所提交的答案为a，参考答案为b，则满足$\frac{|a-b|}{\max(1,b)} \le 10^{-6}$时答案被接受。

测试样例：

Input	Output
3 7 1 3 1 2 1	2.000000000000
4 5 10 3 2 2 3 2 4	1.400000000000

注释：

测试样例 1 中，所有朋友可以在2秒内汇聚到坐标为5的点。为此，第一个人要向南全速移动，第二个和第三个人向北全速移动。

201530800126_10 的二分搜索算法:

本问题的解具有大于等于某一阈值可行，否则不可行的二值性。用二分搜索求解。具体的，判定给定时间t所有人能否汇合的算法：首先让所有人在t时间内以最大速度往右走，记录位于最左端的人的坐标L；然后让所有人在t时间内以最大速度向左走，记录位于最右端的人的坐标R，若$L <= R$，则可以汇合，否组不可以。

完整代码:

```
#include <algorithm>
#include <cstdio>
using namespace std;

const int N = 60005;
int x[N], v[N];
int n;

bool check(double t) {
    double L = 1e9, R = 0;
    for (int i = 0; i < n; i++)
        L = min(L, x[i] + v[i] * t), R = max(R, x[i] - v[i] * t);
```

```
	return R <= L;
}

int main() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++)
		scanf("%d", &x[i]);
	for (int i = 0; i < n; i++)
		scanf("%d", &v[i]);

	double l = 0, r = 1e9;
	for (int i = 0; i < 100; i++) {
		double m = (l + r) / 2;
		if (check(m)) r = m;
		else l = m;
	}

	printf("%.12lf\n", l);
}
```

2.6.1 Codeforces 780E (Round #403)

Undergroung Lab (DFS 相关，图)

题述：给定连通图$|E| = n, |V| = m$。允许自环和连接相同点的多边。现有k人，每人限最多经过$\left\lceil \frac{2n}{k} \right\rceil$个顶点。设计所有$k$人的路线使得这些路线加在一起可以覆盖所有点。

参数范围：$1 \le n \le 2e5, 1 \le m \le 2e5, 1 \le k \le n$

unused 的一次深搜解:

所有人经过的总顶点数为$2n$。注意到一次 dfs 所经过的边构成一棵树，共$2n-2$跳，故路径经过顶点总数为$2n-1$，于是可将 dfs 路径拆分成 k 段作为答案。

完整代码:

```
const int N = 200005;
vector<int> gra[N];
bool vis[N];
vector<int> dfsPath;      // dfs path consisted of 2 * n - 1 vertices.
int n, m, k;

void dfs(int cur) {
    vis[cur] = true;
    dfsPath.push_back(cur);
    for(int nxt : gra[cur]) {
        if(vis[nxt]) continue;
        dfs(nxt);
        dfsPath.push_back(cur);
    }
}

int main() {
    scanf("%d %d %d", &n, &m, &k);
    dfsPath.reserve(2 * n - 1);
    for(int i = 0; i < m; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        if(u == v) continue;
        gra[u].push_back(v);
        gra[v].push_back(u);
    }

    dfs(1);
    int base = int(dfsPath.size() / k);
    for(int i = 0, cur = 0; i < k; i++) {
        int len = base + (dfsPath.size() % k > i);
        printf("%d ", len);
        while(len--) {
            printf("%d ", dfsPath[cur++]);
        }
```

```
            printf("\n");
        }

        return 0;
}
```

2.6.2 Codeforces 780F (Round #403)

Axel and Marston in Bitland (bitmask，图论，矩阵)

题述：Axel 和 Marston 去 Bitland 旅行。共有n个城区，由m条单向公路连接，这些公路有步行道和骑行路两类，两个城区之间可能有多条公路，可能有自环，但不存在某两条公路拥有同样的起止点和类型。两人从城区 1 出发规划旅行路径，Axel 喜欢步行，Marston 喜欢骑行，为了让两人都能玩得开心他们决定按照如下原则规划路径：

1. 从步行公路开始。
2. 假设已有路径以 01 串s表示（0 表示步行，1 表示骑行），则接下来的路径为$\bar{s}$。

前面若干条路径的类型如下：0110100110010110…，按照这个类型表规划路径，当在下一步没有表中指定类型的路径时返程。找到两人按照上述规则能有的最大路径长度，若长度超过10^{18}返回-1。

参数范围：$1 \le n \le 500, 0 \le m \le 2n^2$

unused 利用数字二进制表示及 vector<array<bitset>>记录多步可达性矩阵的定时间解法：观察路径类型表的特征，为递归反对称结构。最终结果在10^{18}之内，故用$\lceil 18\log_2 10 \rceil = 60$个二进位可以表示。

完整代码:

```
const int N = 500;

using matrix = array<bitset<N>, N>;
matrix operator * (const matrix &m1, const matrix &m2) {       // Reload '*' for array<bitset<N>, N>.
    matrix ans;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for(int k = 0; k < N; k++)
                ans[i][j] = ans[i][j] | (m1[i][k] & m2[k][j]);
    return ans;
}

vector<matrix> m1, m2;   // m1 is set of pedestrian reachability matrices, m2 is the biking one.
int n, m;    // Vertex number and edge number.

int main() {
    scanf("%d %d", &n, &m);
    m1.resize(60), m2.resize(60);
    for(int i = 0; i < m; i++) {       // Set up initial pedestrain and biking neighbourhood matrices.
        int a, b, c;
        scanf("%d %d %d", &a, &b, &c);
        if(c == 0) m1[0][a-1][b-1] = true;
        else m2[0][a-1][b-1] = true;
    }
    for(int i = 1; i <= 59; i++) {   // Set up 2-power steps reachability matrices.
        m1[i] = m1[i - 1] * m2[i - 1];
        m2[i] = m2[i - 1] * m1[i - 1];
    }

    matrix mm;
    for(int i = 0; i < N; i++)   // Search ans according to it's binary representation.
```

```
        mm[i][i] = true;
    bool flag = true;
    long long ans = 0;
    for(int i = 59; i >= 0; i--)
        if((mm * (flag ? m1[i] : m2[i]))[0].any()) {
            ans += 1ll << i;
            mm = mm * (flag ? m1[i] : m2[i]);
            flag = ~flag;
        }

    if(ans > 1e18) printf("-1\n");
    else printf("%lld\n", ans);

    return 0;
}
```

2.6.3 Codeforces 796C (Round #408)

Bank Hacking (树，计数)

题述：

尽管小狗 Inzane 找到了它最喜欢的骨头，然而他的主人 Zane 快回来了。为了找到主人小狗 Zane 需要一些钱，然而它现在两手空空。为解燃眉之需，Zane 决定入侵银行。

n家银行编号1到n，$1 \le n \le 3e5$，由$n-1$条网线连接。初始全部在线，每家银行都有各自的入侵难度a_i。

定义一些术语：银行i和j邻接当且仅当二者间有一条网线直接相连，银行i和j半邻接当且仅当二者和同一个在线银行邻接。

当某银行被入侵之后其被迫下线，所有与其邻接或半邻接的银行的入侵难度相应增加 1。

Inzane 从某家银行开始入侵，当然第一家银行的入侵难度不能超过 Inzane 的能力。然后它开始依次入侵其他的银行直至所有银行瘫痪，但它入侵的银行x必须遵从以下规则：

- 银行x在线，即银行x还未被入侵。
- 银行x与至少一个离线银行邻接。
- 银行x的入侵难度小于等于 Inzane 的黑客技能。

请帮助 Inzane 确定使所有银行瘫痪至少需要的黑客技能。（保证存在使所有银行瘫痪的入侵顺序）

输入：

第一行为一个整数$n(1 \le n \le 3 \times 10^5)$——银行数量。

第二行为n个整数$a_1, a_2, \ldots, a_n(-10^9 \le a_i \le 10^9)$——银行的入侵难度。

接下来的$n-1$行每行包含两个整数$u_i, v_i(1 \le u_i, v_i \le n, u_i \ne v_i)$——表示银行$u_i$和$v_i$由一条网线直接连接。

银行之间的连接方式保证若 Inzane 的黑客技能足够高则一定能入侵所有银行。

输出：

输出一个整数——Inzane 入侵所有银行所需最少黑客技能。

测试样例：

Input	Output
5 1 2 3 4 5 1 2 2 3 3 4 4 5	5
7 38 -29 87 93 39 28 -55 1 2 2 5 3 2 2 4	93

1 7 7 6	
5 1 2 7 6 7 1 5 5 3 3 4 2 4	8

测试样例 1 中，Inzane 只需5的黑客技能就能入侵所有银行，步骤如下：

- 初始所有一行入侵难度为$[1, 2, 3, 4, 5]$
- 入侵银行 5，剩下银行的入侵难度变为$[1, 2, 4, 5, -]$
- 入侵银行 4，剩下银行的入侵难度变为$[1, 3, 5, -, -]$
- 入侵银行 3，剩下的银行入侵难度变为$[2, 4, -, -, -]$
- 入侵银行 2，剩下的银行入侵难度变为$[3, -, -, -, -]$
- 入侵银行 1，完活儿

测试样例 2 中，Inzane 按照$4, 2, 3, 1, 5, 7, 6$的顺序入侵，按此顺序需要黑客技能93。

Toturial 分析:

首先分析并描述整个 hack 的过程，**第一个被 hack 的银行i需要a_i的难度，与之邻接的所有银行j需要$a_j + 1$的难度，其他所有银行k需要$a_k + 2$的难度，与入侵顺序无关。**于是最终答案只可能是a_m，$a_m + 1$，$a_m + 2$中的一个（其中a_m为a_i中的最大值）。用两个变量x, y分别记录以$a_m + 2$和$a_m + 1$被访问的节点数量（频次）。

以上算法$T \sim O(n)$

完整代码:

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

const int N = 300005;
const int INF = INT32_MAX;

int a[N];
std::vector<int> nei[N];
int n;
void read() {
	scanf("%d", &n);
	for (int i = 1; i <= n; i++)
		scanf("%d", &a[i]);
	for (int i = 0; i < n - 1; i++) {
		int u, v;
		scanf("%d %d", &u, &v);
		nei[u].push_back(v);
		nei[v].push_back(u);
	}
}
```

```
void solve() {
    int mx = -INF;
    for (int i = 1; i <= n; i++)
        mx = std::max(mx, a[i]);

    int x = 0, y = 0;    // x, y are the number of vertexes that visited with strength (mx + 2) and (mx + 1) respectively.
    for (int i = 1; i <= n; i++)
        if (a[i] == mx) x++;
        else if (a[i] == mx - 1) y++;

    int ans = mx + 2;
    for (int i = 1; i <= n; i++) {   // Start hacking from vertex i.
        // Adjust x and y for current loop.
        if (a[i] == mx) x--;
        else if (a[i] == mx - 1) y--;
        for (int j = 0; j < nei[i].size(); j++)
            if (a[nei[i][j]] == mx) x--, y++;
            else if (a[nei[i][j]] == mx - 1) y--;
        // Generate ans according to x and y.
        if (x == 0) {
            ans = mx + 1;
            if (y == 0) {
                ans = mx;
                break;
            }
        }
        // Recover x and y for next loop.
        if (a[i] == mx) x++;
        else if (a[i] == mx - 1) y++;
        for (int j = 0; j < nei[i].size(); j++)
            if (a[nei[i][j]] == mx) x++, y--;
            else if (a[nei[i][j]] == mx - 1) y++;
    }

    printf("%d\n", ans);
}

int main() {
    read();
    solve();
}
```

注：本题利用了树的一个性质——**与同一顶点邻接的不同顶点之间互不邻接**，保证了“第一个被 hack 的银行i需要a_i的难度，与之邻接的所有银行j需要a_j+1的难度，其他所有银行k需要a_k+2的难度，与入侵顺序无关。”成立。故此题中的树结构是必要的。

2.6.4 Codeforces 796D (Round #408)

Police Stations (构造性算法，最短路，树，BFS)

题述：

Inzane 最终找到了 Zane 同时还剩下了一笔不菲的资产，他们决定建立一个自己的国度。

然而治理一个国家可不简单，总有小偷和恐怖分子等恶势力团伙企图扰乱社会秩序。作为还击，Zane 和 Inzane 颁布了一条法令：每个城市距离最近警局的公路路程不能超过$d$km。

整个国家由n个城市构成，编号$1,\dots,n$。这些城市由$n-1$条公路连接，每条公路的长度都是1km。初始这些公路使得城市之间两两可达。有k个警察局分布在某些城市里，特别地，城市结构满足上述条件，另外一个城市可以有多个警察局。

然而，Zane 觉得没有必要设立多达 n-1 条公路。国家出现了参政问题，所以 Zane 希望通过关闭以下公路以最小化用于公路维护的开支。

请帮助 Zane 确定可以关闭的最大公路数并具体指出这些公路。

输入：

第一行为3个整数$n,k,d(2\le n\le 3\times 10^5,1\le k\le 3\times 10^5,0\le d\le n-1)$——分别为城市数、警局数、每个城市到最近警局的距离上限。

第二行为k个整数$p_1,p_2,\dots,p_k(1\le p_i\le n)$——表示每个警局所位于的城市。

接下来的$n-1$行每行包含两个整数$u_i,v_i(1\le u_i,v_i\le n,u_i\ne v_i)$——第$i$条公路连接的两个城市。

输入保证城市之间通过公路两两连通。同时，保证每个城市到最近警局的距离不超过$d$km。

输出：

第一行打印一个整数s——可以关闭的最大公路数量。

第二行打印s个互不相同的整数——被关闭的公路序号，顺序任意。

如果有多组解，输出任意一个即可。

测试样例：

Input	Output
6 2 4 1 6 1 2 2 3 3 4 4 5 5 6	1 5
6 3 2 1 5 6 1 2 1 3 1 4	2 4

1 5 5 6	

测试样例 1 中，若关闭 5 号公路，所有城市仍可以在$k = 4$km 的距离内到达警局。
测试样例 2 中，最大可关闭公路集合唯一$\{4,5\}$，但第二行的输出顺序可以为$[4,5]$也可以为$[5,4]$。

Tutorial 分析:

考虑以所有带警察局的顶点为根进行 BFS，删去所有通向已访问顶点的边，用一个一维数组标记。这样的结果是每个本身没有警察局的顶点连接到一个与其最邻近的警察局，满足要求。用k'表示带警察局的顶点数，则整个过程将删去$k'-1$条边（注意到原图为树），从而**将图分解成k'个各自有一个带警察局的节点的连通树分量**，再少一条边将会产生不与警察局连通的节点，故得到的是满足要求且边数最少的子图。

完整代码:

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

const int N = 300005;

std::queue<std::pair<int, int>> q;      // BFS queue.
std::vector<std::pair<int, int>> way[N];   // Neighbourhood table.
int n, k, d;
void read() {
    scanf("%d %d %d", &n, &k, &d);
    for (int i = 0; i < k; i++) {
        int p;
        scanf("%d", &p);
        q.push({p, 0});
    }
    for (int i = 1; i <= n - 1; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        way[u].push_back({v, i});
        way[v].push_back({u, i});
    }
}

bool vis[N];      // Visited memo.
bool clo[N];      // Mark whether road i should be closed.
void solve() {
    while (!q.empty()) {      // BFS
        int pos = q.front().first;
        int from = q.front().second;
        q.pop();
        if (vis[pos]) continue;
        vis[pos] = true;
        for (int i = 0; i < way[pos].size(); i++)     // Search all children.
```

```
                if (way[pos][i].first != from) {
                    if (vis[way[pos][i].first])
                        clo[way[pos][i].second] = true;
                    else q.push({way[pos][i].first, pos});
                }
        }

        int cnt = 0;
        for (int i = 1; i <= n - 1; i++)
            if (clo[i]) cnt++;
        printf("%d\n", cnt);
        for (int i = 1; i <= n - 1; i++)
            if (clo[i]) printf("%d\%c", i, " \n"[i == n - 1]);
}

int main() {
        read();
        solve();
}
```

注：若原图不是树，该算法仍然适用。作者 zoomswk 给出的解释是如果给定的是任意图，那么解法就太显然了。

“It will still work. The solution would be more obvious could the input be any graph, wouldn't it? XD”

关于 BFS 队列和 DFS 栈:

BFS 队列和 **DFS 栈**分别为二者**迭代实现**所依赖的数据结构，需使用 BFS/DFS 算法时应开辟相应物理存储。而基于 DFS 算法本身的**递归**特点，深搜操作可以用 C++递归实现，此时的 **DFS 栈**为编程语言的**函数调用栈**。

2.6.5 Codeforces 936B (Round #467)

Sleep Game (有限状态机，DFS)

题述：

Petya 和 Vasya 进行了一场游戏。游戏规则如下：游戏基于一张n个顶点m条边的有向图，其中一个顶点上有一枚棋子。起初棋子位于顶点s，玩家轮流沿有向边移动棋子。Petya 先手，直到一方无路可走则该玩家失败。如果游戏轮次超过10^6则宣判平局。

由于 Vasya 昨夜消耗了大量的体力，游戏刚一开始他就睡着了，Petya 可得好好利用这个机会，他决定代替 Vasya 移动棋子。

你的任务是帮助 Petya 判断其能否获胜，或至少达到平局。

输入：

输入第一行为两个整数n和m，分别为图的顶点数和边数$(2 \le n \le 10^5, 0 \le m \le 2 \cdot 10^5)$。

接下来的n行为图中边的信息。其中，第i行开始一个整数c_i表示该顶点的出度，后跟c_i个整数$a_{i,j}(1 \le a_{i,j} \le n, a_{i,j} \ne i)$表示所有由顶点$i$一步可达的顶点。

最后一行一个整数s为棋子初始所在的顶点。

输出：

如果 Petya 能赢在第一行输出“Win”，在下一行输出 Petya 赢得游戏的顶点移动序列$v_1, v_2, \dots, v_k (1 \le k \le 10^6)$，其中$v_1 = s$，而$v_k$是一个出度为零的顶点。为使 Petya 获胜整个序列的长度应为偶数。

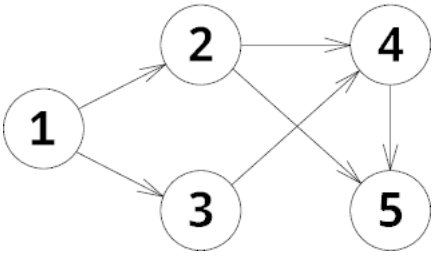
如果 Petya 不能赢的游戏但能达到平局，输出“Draw”，否则输出“Lose”。

测试样例：

Input	Output
5 6 2 2 3 2 4 5 1 4 1 5 0 1	Win 1 2 4 5
3 2 1 3 1 1 0 2	Lose
2 2 1 2 1 1 1	Draw

样例解释：

对于测试样例 1，有向图如下：



开始时棋子位于顶点1，Petya 先将棋子移动到顶点2，然后代替 Vasya 将棋子移到顶点4，然后再将棋子移动到顶点5，此时 Vasya 无法继续移动棋子，Petya 获胜。

测试样例 2 中，有向图如下：

起初棋子位于顶点2，Petya 只能将棋子移动到顶点1，而后又只能代替 Vasya 将棋子移动到顶点3，此时 Petya 无法继续移动棋子，故输掉游戏。

测试样例 3 中，有向图如下：

Petya 无法获胜，但它可以抑制绕环移动，故最终为平局。

作者基于状态图的线性算法:

将题目翻译如下：Petya 想要判断自己能否赢得比赛，若果不能，则判断能否构成平局。

先做分析：显然，如果能赢，则要从起点开始移动奇数步到达一个汇点并输出这个路径（由偶数个顶点构成，第一个点为起点s，最后一个点为一个汇点）。如果这种情况不存在，但同时从s点出发能进入一个环内，那么只要沿着环移动10^6步就能达到平局；另一方面，是否存在其他的平局呢？——答案是否定的，注意到图中顶点数最大为10^5，于是不可能移动10^6步而不出现环。那么如果这两个条件都不满足，自然输掉游戏。

对上面的分析进行总结，根据游戏规则，在逻辑层面上:

· 赢：从起始点s开始经过奇数步到达一个汇点;

· 平：在不能赢的情况下，从起始点s移动能够到达一个环内;

· 输：剩下的情况。

进一步，怎样判断从s点开始能否通过奇数步到达一个汇点呢？按照从s点开始到达每个点步骤数的奇偶性，可以建立如下状态图，共有$2n$个顶点，原图中第i个顶点分别对应状态图中的$(i|0),(i|1)$两个顶点，而对原图中的每条有向边(u,v)，分别对应状态图中的有向边$((u|0),(v|1))$和$((u|1),(v|0))$，于是状态图中共有$2m$条有向边。基于此，游戏胜利条件转化为在状态图从$(s|0)$可达的点中是否存在一个顶点$(x|1)$满足x在原图中为一个汇点。

进一步总结，游戏规则在实现层面上:

· 赢：状态图从$(s|0)$可达的顶点中存在一个顶点$(x|1)$满足x在原图中为一个汇点;

· 平：在不能赢的情况下，原图中从起始点s移动能够到达一个环内;

· 输：剩下的情况。

对于赢和平的判断分别基于原图和状态如图的 DFS，时间复杂度$T\sim O(V+E)$。

完整代码:

```
#include <vector>
#include <cstdio>

const int maxn = 100005;

std::vector<int> g[maxn];    // Original graph.
int n, m;
int s;
void read() {
    scanf("%d %d", &n, &m);
    for (int i = 0; i < n; i++) {
        int num;
        scanf("%d", &num);
```

```
            for (int j = 0; j < num; j++) {
                int tmp;
                scanf("%d", &tmp);
                g[i].emplace_back(tmp - 1);
            }
        }
        scanf("%d", &s);
        s--;
}

std::vector<int> sg[maxn << 1]; // State graph.

bool vis1[maxn << 1];
std::vector<int> cur_path;   // Record current path for output.
bool dfs1(int p) {   // Return whether from node p can reach a winning-node in state-graph.
        if (vis1[p]) return false;

        vis1[p] = true;
        cur_path.emplace_back(p);
        if ((p & 1) && g[p >> 1].empty()) return true;
        else {
            for (int i = 0; i < sg[p].size(); i++)
                if (dfs1(sg[p][i])) return true;
            cur_path.pop_back();
            return false;
        }
}

bool vis2[maxn];
bool dfs2(int p) {   // Return whether from node p can reach a circle in original-graph.
        if (vis2[p]) return true;     // Circle found.

        vis2[p] = true;
        for (int i = 0; i < g[p].size(); i++)
            if (dfs2(g[p][i])) return true;
        vis2[p] = false;
        return false;
}

void solve() {
        // Build state-graph.
        for (int i = 0; i < n; i++)
            for (int j = 0; j < g[i].size(); j++) {
                sg[(i << 1) + 0].emplace_back((g[i][j] << 1) + 1);
                sg[(i << 1) + 1].emplace_back((g[i][j] << 1) + 0);
```

```
        }

    // Search state-graph for winning.
    if (dfs1((s << 1) + 0)) {   // DFS on state-grath from (s|0) to find a winning point.
        printf("Win\n");
        for (int i = 0; i < cur_path.size(); i++)
            printf("%d%c", (cur_path[i] >> 1) + 1, " \n"[i == cur_path.size() - 1]);
        return;
    }

    // Search original-graph for draw.
    if (dfs2(s)) printf("Draw\n");
    else printf("Lose\n");
}

int main() {
    read();
    solve();
}
```

2.7.1 Codeforces 937B (Round #467)

Vile Grasshoppers (数论——素数间隔)

题述：

天气真好，今天很适合爬上门口的松树眺望远处景色。

松树的树干上有很多横向的树枝，从下向上依次编号$2, \dots, y$，其中2号到p号的树枝被你的敌人——邪恶蚂蚱占据着。这些蚂蚱以其灵活的跳跃能力著称：在x号树枝上的蚂蚱可以跳跃至编号为$2x, 3x, \dots, \left\lfloor \frac{y}{x} \right\rfloor x$的树枝上。

基于以上信息，你决定找一个不会被邪恶蚂蚱打扰的树枝欣赏风景，同时为了看得更远你还希望所选的树枝尽量高。

换句话说，你的目标是找到那个不会有蚂蚱到来的最高的树枝或者告知这样的树枝不存在。

输入：

唯一一行包含两个整数p和$y(2 \le p \le y \le 10^9)$。

输出：

输出符合条件的最高的树枝编号，如不存在输出-1。

测试样例：

Input	Output
3 6	5
3 4	-1

样例解释：

测试样例 1 中，位于2号树枝的蚂蚱可以到达编号为$2, 4, 6$的树枝，而编号为3的树枝上本来就有一只蚂蚱，故最终答案为5。

测试样例 2 中，显然没有符合条件的树枝。

基于 Tutorial 分析的算法:

本题要找到y以内不被$[2, p]$中数整除的最大的数。显然是从y开始向下一个一个搜索，找到第一个最小素因子大于p的数。

这个算法为什么可行呢？首先，找到 n 的最小素因子的复杂度为$T \sim O(\sqrt{n})$，由于10^9以内连续素数的最大间隔不超过300，故最多搜索300个数就能得到最终结论。故总复杂度为：$T \sim O(300\sqrt{10^9})$

完整代码:

```
#include <cmath>
#include <cstdio>

int p, y;
void read() {
    scanf("%d %d", &p, &y);
}

int min_fac(int n) {
    int rn = sqrt(n);
    for (int i = 2; i <= rn; i++)
        if (n % i == 0) return i;
    return n;
}

void solve() {
```

```
    for (int i = y; i > p; i--) // At most 300 step as prime gap in billion is less than 300.
        if (min_fac(i) > p) {
            printf("%d\n", i);
            return;
        }
    printf("-1\n");
}

int main() {
    read();
    solve();
}
```

2.8.1 Codeforces 785D (Round #404)

Anton and School – 2 (计数问题，组合数，乘法逆元，快速幂)

题述：满足以下条件的括号串被称为 RBRS 串：非空，偶数个字符，前一半字符全部为 '(' 后一半为 ')'。求给定字符串（长度不超过 200000）中的 RBRS 子串数，结果对(1e9 + 7)取模。

参数范围：$|s| \leq 2e5$

9 大 topcoder 的思路：首先长度为$x + y$的字符串 "((...())...)" 中 RBRS 子串的数量为C_{x+y-1}^{x}。于是原问题解可用如下方式求解，从左向右，每遇到 '(' 则计算以之为末位 '(' （保证计算不重复）的 RBRS 子串数，累加所有结果即为所求。

关于预处理，首先记录原数组中每个位置上左边 '(' 和右边 ')' 的数量。在计算组合数前记录取模意义下阶乘的值，同时记录阶乘的**乘法逆元**。阶乘逆元　　需要从大到小递推，利用费马小定理根据$(400000!)^{-1} = (400000!)^{MOD-2}$，使用**快速幂**算法在对数时间内完成。

注：本题中利用乘法逆元在模 1000000007 求组合数的函数及其前期准备（紫色部分）可以作为通用模块，其中标蓝部分视具体问题的规模而更改。

完整代码:

```
const int N = 200005;
const int MOD = 1000000007;
int lef[N], rig[N];
int fac[N << 1], facinv[N << 1];
char s[N];

int pw(int a, int b) {     // Quick power for a^b.
    int ans = 1;
    while (b) {
        if (b & 1) ans = 1ll * ans * a % MOD;
        a = 1ll * a * a % MOD;
        b >>= 1;
    }
    return ans;
}

void preProcess() { // Inverse elements in MOD.
    fac[0] = 1;
    for (int i = 1; i <= 200000 << 1; i++)
        fac[i] = 1ll * fac[i - 1] * i % MOD;
    facinv[400000] = pw(fac[400000], MOD - 2);
    for (int i = 399999; i >= 0; i--)
        facinv[i] = 1ll * facinv[i + 1] * (i + 1) % MOD;
}

int C(int x, int y) {     // Combination in MOD: C_x^y.
    if(x < y) return 0;
    return 1ll * fac[x] * facinv[y] % MOD * facinv[x - y] % MOD;
}

int main() {
```

```
    preProcess();
    scanf("%s", s);

    int len = (int)strlen(s);
    lef[0] = s[0] == '(';
    for (int i = 0; i < len; i++)
        if(s[i] == '(') lef[i] = lef[i - 1] + 1;
        else lef[i] = lef[i - 1];
    rig[len - 1] = s[len - 1] == ')';
    for (int i = len - 2; i >= 0; i--)
        if (s[i] == ')') rig[i] = rig[i + 1] + 1;
        else rig[i] = rig[i + 1];

    int ans = 0;
    for (int i = 0; i < len; i++)
        if (s[i] == '(') ans = (ans + C(lef[i] + rig[i+1] - 1, lef[i])) % MOD;

    printf("%d\n", ans);

    return 0;
}
```

2.8.2 Sougou 2017 秋招笔试

钝角三角形个数 (计数方法，计算几何，双指针，线性时间算法，细节实现)

题述：

给定圆上n个点，用其所对应的圆心角坐标表示，请问从这n个点中取出三个点能够构成多少个不同的钝角三角形。

输入：

输入第一行一个整数n为点的数量。接下来的n行中的第i行一个双精度浮点数表示第i个点所对应的圆心角。输入按照圆心角升序排列。

输出：

输出一个整数表示由这些点可以构成的不同钝角三角形的数量。

参数限制：

$1 \le n \le 300000$

测试样例：

Input	Output
6 0.000000 1.000000 2.000000 169.000000 358.000000 359.000000	20

WeiYong 的线性双指针算法:

以圆周上的三点构成钝角三角形当且仅当三个点两两之间距离小于直径（三点位于直径同一侧，或圆周上包含三个点的最短弧所对的圆心角小于180°）。我们的算法是计数以每个点为起点逆时针方向可以构成钝角三角形的数量，具体地，**用两个指针i和p分别记录起点和从起点开始逆时针方向不超过180°所能达到的最远点。**则从点i开始以之为长变端点在逆时针方向上可以构成的钝角三角形数量为$1+2+\cdots+m-1=\frac{m(m-1)}{2}$，其中$m$为逆时针方向从$i$到$p$之间除起点外的点数。

如果从i开始逆时针180°范围内不存在其他点，那么点i自然成为点$i+1$的p点。

整个过程i走n步，p走最多$\frac{3}{2}n$步，时间复杂度$T\sim O(n)$。

完整代码:

```
#include <iostream>

using ll = long long;

const int maxn = 300005;

double a[maxn];
int n;
void read() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%lf", &a[i]);
```

```
}

bool check(int i, int j) {
    if (j >= i) return a[j] - a[i] < 180;
    else return a[j] + 360 - a[i] < 180;
}

ll count(int i, int j) {
    if (j >= i) {
        int m = j - i;
        return   1ll * m * (m - 1) / 2;
    } else {
        int m = j + n - i;
        return 1ll * m * (m - 1) / 2;
    }
}

int next(int i) {
    if (i == n - 1) return 0;
    else return i + 1;
}

void solve() {
    if (n == 1 || n == 2) {
        printf("0\n");
        return;
    }

    ll ans = 0;
    int p = 0;
    for (int i = 0; i < n; i++) {
        while (check(i, next(p)) && next(p) != i)
            p = next(p);
        ans += count(i, p);
    }
    printf("%lld\n", ans);
}

int main () {
    read();
    solve();
}
```

算法编程心得：细节实现题，明确每个变量的所代表的意义和每一步应处于的位置（或取值），在此基础上结合问题本身性质处理边界情形。

2.8.3 TCO 2018 Beijing Regional 热身题目

田忌赛马（二分答案区间，最小值的最大值，顺序统计量）

题述：

齐国的大将田忌很喜欢赛马，有一回他被齐威王请来赛k天马。经过精心的准备，现在田忌有n匹速度分别为$a_1,a_2,\ldots,a_n$的马，齐威王则有n匹速度分别为$b_1,b_2,\ldots,b_n$的马。每天要进行n场比赛，每场比赛田忌和齐威王分别会派出一匹马，每匹马每天只能上场一次，每场比赛的精彩程度是上场的两匹马的速度之和。为了让比赛更精彩，任意不同的两天里进行的比赛不能完全相同，也就是说，对于任意不同的两天，存在至少一匹马在这两天比赛的对手不同。你需要计算出这k天总计$k\times n$场比赛中精彩程度最小值的最大可能值。

输入：

第一行包含两个整数n和k $(1\le n\le 50, 1\le k\le \min(n!,10^9))$，分别表示田忌和齐威王各自拥有的马匹数以及赛马的天数。

第二行包含n个整数 $a_1,a_2,\ldots,a_n$ $(1\le a_i\le 10^9)$，表示田忌拥有的n匹马的速度。

第三行包含n个整数 $b_1,b_2,\ldots,b_n$ $(1\le b_i\le 10^9)$，表示齐威王拥有的n匹马的速度。

输出：

输出一个整数，表示这k天总计 $k\times n$ 场比赛中精彩程度最小值的最大可能值。

测试样例：

Input	Output
2 1 1 2 1 3	3
2 2 1 2 1 3	2

算法：

将该问题作如下转化，固定序列a中的元素顺序，则序列b的每一个排列与a组合都可以构成一天的比赛，所以共有$n!$种不同的比赛组合可供观赏。对于b的每种排列，计算a,b对应位置元素和的最小值。问题要求这$n!$个最小值中第k大的那一个。

显然求出所有$n!$个最小值再去其中第k大的数是不现实的。采用二分答案区间的思路，元素和的取值范围是$[0,2e9]$，对于每个值m，我们需要判断出元素和最小值大于等于m的b的排列的数量是否大于等于k，这个函数可以在序列a有序的基础上，按a中数字非降序的处理顺序在$T\sim O(N^2)$的时间内完成。利用这个判断函数，可以使用二分搜索确定最终答案。

整个过程的时间复杂度$T\sim O(N\text{^}2\log(range))$，其中$range=2e9$。

完整代码：

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

using namespace std;
using ll = long long;

const int maxn = 55;
```

```
int a[maxn], b[maxn];
int n, k;
void read() {
	cin >> n >> k;
	for (int i = 0; i < n; i++)
		cin >> a[i];
	for (int i = 0; i < n; i++)
		cin >> b[i];
}

int c[maxn];
bool check(ll m) {
	memset(c, 0, sizeof c);
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
			if (a[i] + b[j] >= m) c[i]++;

	for (int i = 0; i < n; i++)
		if (c[i] - i < 1) return 0;
	ll ans = 1;
	for (int i = 0; i < n; i++) {
		ans *= c[i] - i;
		if (ans >= k) return true;
	}
	return false;
}

void solve() {
	sort(a, a + n);
	ll l = 0, r = 2e9 + 1;
	while (l < r - 1) {
		ll m = (l + r) >> 1;
		if (check(m)) l = m;
		else r = m;
	}
	cout << l << endl;
}

int main() {
	read();
	solve();
}
```

2.9.1 TCO 2017 Round A1 div1

1000-points problem（计算几何，Simpson 公式）

题述：

给定 XY 平面上的凸多边形，其顶点顺时针顺序为(x[0], y[0]), (x[1], y[1]), …，用两个向量 vector<int> x 和 y 表示。为了实现上的简便该多边形及其表示方式满足某些限制条件，详见限制条件部分。将该多边形沿 y 轴旋转将得到一个 3D 实体，请计算该实体的体积。

定义：

Class: PolyonRotation

Method: getVolume

Parameters: vector<int>, vector<int>

Returns: double

Method signature: double getVolume(vector<int> x, vector<int> y)

(Be sure your method is public)

资源限制：

时间：2s

内存：256MB

栈限制：256MB

注意：

返回值的最大绝对或相对误差为 1e-9。

限制条件：

· x 中的元素数在 3 到 50 之间（包括边界）；

· y 中的元素数和 x 相同；

· x 和 y 中元素的值在-100 到 100 之间；

· 多边形是凸的；

· 多边形有且仅有两个顶点位于 y 轴上，用(0, Ymin)和(0, Ymax)表示，一种 Ymin<Ymax；

· 所有顶点从(0, Ymax)开始以顺时针列出；

· 所有顶点的 y 坐标都在 Ymin 和 Ymax 之间；

· 任意三点不共线。

测试样例：

0)

{0, 1, 1, 0}

{1, 1, 0, 0}

Returns: 3.141592653589793

1)

{0, 1, 0, -1}

{2, 1, 0, 1}

Returns: 2.0493951023931953

2)

{0, 3, 0, -2, -2}

{2, 0, -3, -1, 1}

Returns: 49.91641660703782

3)

{0, 3, 3, 0, -1, -1}

{2, 2, -2, -2, -1, 1}

Returns: 113.09733552923255

wxh010910 的纯数值算法:

根据题目的限制条件，由于位于 y 轴上的两个顶点分别为该凸多边形的最高点和最低点，故经过旋转后改图多边形的任意一个水平截面都是一个圆心落在 y 轴上的圆，故旋转体的体积可以通过水平截面的面积沿 y 轴方向积分得到。

显然积分区间就是[Ymin, Ymax]。由于多边形的顶点坐标都是整数，故旋转体轮廓的“拐点”必然出现在所有的整数点上，也就是说相邻整数之间的部分必然构成圆台（圆锥）。于是我们可以将积分区间[Ymin, Ymax]分成若干个长度为 1 的小区间。这样一来，旋转体在每个区间上的体积就可以用 Simpson 公式得到。

接下来最后的一个任务就是任意给定高度 y，返回旋转体相应截面的面积。注意到截面由原凸多边形位于该高度且具有最大绝对 x 坐标的点决定，那么只需搜索多边形所有的边即可找到这个半径的长度。

完整代码:

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

using namespace std;
using ll = long long;
using ldb = long double;

class PolygonRotation {
public:
    double getVolume(vector<int> x, vector<int> y) {
        this->x = x, this->y = y;
        n = (int)x.size();

        ldb ans = 0;
        y_mn = upb, y_mx = lob;
        for (int i = 0; i < n; i++)
            y_mn = min(y_mn, y[i]), y_mx = max(y_mx, y[i]);
        for (int i = y_mn; i < y_mx; i++)
            ans += simpson(i, i + 1, eps, cal(i, i + 1));
        return double(pi * ans);
    }

private:
    const ldb pi = acos(-1);
    const ldb eps = 1e-15;
    const int upb = 100, lob = -100;

    vector<int> x, y;
    int y_mn, y_mx;
    int n;

    inline ldb f(ldb p) {    // The area of section with y = p.
        ldb s = 0;
```

```
        for (int i = 0; i < n; i++) {
            int x1 = x[i], x2 = x[(i + 1) % n];
            int y1 = y[i], y2 = y[(i + 1) % n];
            if (y1 > y2) swap(x1, x2), swap(y1, y2);
            if (y1 - eps < p && p < y2 + eps) {
                if (y1 == y2) s = max(s, (ldb)max(x1, x2) * max(x1, x2));
                else {
                    ldb x = x1 + (ldb)(x2 - x1) / (y2 - y1) * (p - y1);
                    s = max(s, x * x);
                }
            }
        }
        return s;
    }

    inline ldb cal(ldb l, ldb r) {
        return (f(l) + f(r) + f((l + r) / 2) * 4) * (r - l) / 6;
    }

    inline ldb simpson(ldb l, ldb r, ldb eps, ldb last) {
        ldb mid = (l + r) / 2;
        ldb L = cal(l, mid), R = cal(mid, r);
        if (fabs(L + R - last) < eps) return last;
        return simpson(l, mid, eps / 2, L) + simpson(mid, r, eps / 2, R);
    }
};
```

2.9.1 Codeforces 878B (Round #443)

Teams Formation (算法设计，细节实现，数据处理)

题述：

Berland 团队信息学奥林匹克开赛在即，本届赛事举办地为一个偏远的小城，只能通过班车往来。班车内部有n个座位，其中第i个座位只能提供给来自第i个城市的选手。

这一天共有m趟班车，每趟班车都满载n个乘客。乘客下车后按照他们到达的先后顺序排成一行，其中来自同一趟班车的乘客按照他们的座位顺序排列（i.e. 如果我们写下每个乘客的出发地，将得到序列$a_1, a_2, \dots, a_n$重复m次）。

之后开始组队，每支队伍由队列中相邻且来自相同城市的k个选手组成。一个队伍形成后就留开队列，这个过程持续到队列中不再存在任何来自相同城市的连续k个选手。

请帮助主办方计算组队过程结束后队列中剩下的选手的数量。可以证明这个结果与组队顺序无关。

输入：

第一行为三个整数$n, k, m (1 \le n \le 10^5, 2 \le k \le 10^9, 1 \le m \le 10^9)$。

第二行为n个整数$a_1, a_2, \dots, a_n (1 \le a_i \le 10^5)$，其中$a_i$是城市编号，只有来自城市$a_i$的选手可以坐在班车的第$i$个座位上。

输出：

输出最终队列的人数。

测试样例：

Input	Output
4 2 5 1 2 3 1	12
1 9 10 1	1
3 2 10 1 2 1	0

注释：

测试样例 2 中，队列由来自同一城市的十名选手组成，其中九人组队，最终剩下一人。

Tutorial 的分析与算法

首先，先看一趟班车内队列的行为，用一个**二元组**（城市，来自这个城市的人数）栈模拟。当来自同一城市的人数达到k，就去掉这个二元组。

假设已经得到了这样一个栈，长度为r，其中的二元组为(c_i, d_i)。现在考虑两个这样的栈之间的相互作用，若位于首尾的二元组满足$c_1 = c_r$且$d_1 + d_r \ge k$，就可以有新的组队产生。如果不等式取等，那么第二个和倒数第二个（penultimate）二元组就可以继续尝试组队。我们来确定满足对于任意$i = 1, \dots, p$有$c_i = c_{r+1-i}$且$d_i + d_{r+1-i} = k$的最大p，由于i关于$i \mapsto \mathrm{r} + \mathrm{i} - 1$具有对称性，故如果$p \ge \left\lceil \frac{r}{2} \right\rceil$，则$p = r$。

单独考虑$p = r$的情形，这意味着两趟车中的乘客全部组队。如果 m 是偶数则最终答案为零，否则最终答案为$\sum_{i=1}^{r} d_i$。

类似地，考虑r为奇数且$p = \left\lfloor \frac{r}{2} \right\rfloor$的情形（若$r$为偶数且$p = \left\lfloor \frac{r}{2} \right\rfloor$，则为$p = r$的情形）。在所有“边界组队”完成之后，队列成如下形态：第一车的左边界——md_{p+1}个来自城市c_{p+1}的人——第 m 车的右边界。如果$k = 0 \bmod (md_{p+1})$则中间部分完全组队，两侧对消，最终答案为零。否则，中间部分组队，过程结束，最终答案为$\sum_{i=1}^{p} d_i + md_{p+1} \% k + \sum_{i=r+1-p}^{r} d_i$。

最后，对于更小的p，“边界组队”结束后，过程结束，最终答案为$\sum_{i=1}^{p} d_i + (m-1)(\sum_{i=p+1}^{r-p} d_i - (d_{p+1} + d_{r-p} > k) * k) + \sum_{i=r+1-p}^{r} d_i$。

完整代码:

```
#include <vector>
#include <cstdio>

using ll = long long;

const int maxn = 100005;

int n, k, m;
int a[maxn];
void read() {
	scanf("%d %d %d", &n, &k, &m);
	for (int i = 0; i < n; i++)
		scanf("%d", &a[i]);
}

struct pan {
	int city;
	int num;
	pan() {};
	pan(int _city, int _num) : city(_city), num(_num) {};
};

std::vector<pan> seats;
void solve() {
	seats.emplace_back(pan(a[0], 1));
	for (int i = 1; i < n; i++)
		if (seats[seats.size() - 1].city == a[i]) {
			if (seats[seats.size() - 1].num + 1 == k) seats.pop_back();
			else seats[seats.size() - 1].num++;
		} else seats.emplace_back(pan(a[i], 1));

	int l = (int)seats.size();
	int p = -1;
	while ((p + 1) < l && (seats[(p + 1)].city == seats[l - 1 - (p + 1)].city) && (seats[p + 1].num + seats[l - 1 - (p + 1)].num == k)) p++;

	ll ans = 0;
	if (p == l - 1) {
		if (m & 1)
			for (int i = 0; i < l; i++)
				ans += seats[i].num;
	} else if ((l & 1) && (p == l / 2 - 1)) {
		int mid = 1ll * seats[p + 1].num * m % k;
		if (mid != 0) {
```

```
                for (int i = 0; i <= p; i++)
                    ans += seats[i].num;
                ans += mid;
                for (int i = p + 2; i < l; i++)
                    ans += seats[i].num;
            }
        } else {
            ll ans1 = 0, ans2 = 0;
            for (int i = p + 1; i < l - (p + 1); i++)
                ans1 += seats[i].num;
            ans1 = seats[p + 1].city == seats[l - 1 - (p + 1)].city && seats[p + 1].num + seats[l - 1 - (p + 1)].num > k ?
ans1 - k : ans1;
            for (int i = 0; i < l; i++)
                ans2 += seats[i].num;
            ans = ans1 * (m - 1) + ans2;
        }

        printf("%lld\n", ans);
}

int main() {
        read();
        solve();
}
```

关于本题的整体思考:

从完整队列形态开始，进行如下四种操作:

首先对原始数据进行清理:

· *m*段（原始序列）——>**内部组队**——>*m*段（一趟车内不可组队）

确定数据的特征*p*值，并根据*p*的取值进行如下处理步骤:

· *m*段（一趟车内不可组队）——>**边界组队**——>3段（首左+所有中+尾右）

· 3段——>**中部组队**——>3段（中部完全组队）或2段（中部不完全组队）

· 若剩下2段：2段——>**首尾对消**——>无

本题数据的**关键特征**在于找到单车p值。

注（一句话证明最终结果于组队顺序无关）:

Codeforces 822C (Round 422, Div 2)

Hacker, pack your bags! (哈希表)

题述：

众所周知，转移注意力最好的方法就是去做最喜欢的事，对于 Leha 来说这件事就是工作。

于是这黑客全力工作以转移注意力，他开始入侵全世界的电脑。老板给了他长达x天假期，正如大多数人一样，Leha 来到旅行社。旅行社有n种旅程，第i种旅程由三个整数$l_i, r_i, cost_i$表证——分别为启程日期、返程日期和费用，旅程长度为$r_i - l_i + 1$。

Leha 希望将假期分成两段，同时希望花费最小的费用。于是 Leha 希望选择不重叠的两个旅程i和$j(i \neq j)$，长度和刚好为x，总花销越少越好。两个旅程i, j不重叠当且仅当满足：$r_i < l_j$或$r_j < l_i$。

帮助 Leha 选择旅程吧！

输入：

第一行为两个整数n和$x(2 \leq n, x \leq 2 \times 10^5)$——可供选择的旅程数量和 Leha 的假期长度。接下来的n行每行为三个整数$l_i, r_i, cost_i(1 \leq l_i \leq r_i \leq 2 \times 10^5, 1 \leq cost_i \leq 10^9)$表征第$i$个旅程。

输出：

输出一个整数——Leha 的最小花费，若不存在长度和为x的不重叠旅程则输出-1。

测试样例：

Input	Output
4 5 1 3 4 1 2 5 5 6 1 1 2 4	5
3 2 4 6 3 2 4 1 3 5 4	-1

注释：

测试样例 1 中 Leha 选择第1和第3个旅程。总长度为$(3 - 1 + 1) + (6 - 5 + 1) = 5$，总花销为$4 + 1 = 5$。

测试样例 2 中所有旅程的长度都是3，故无法选出两个长度和为2的旅程。

（超时）WeiYong 的排序夹逼算法:

典型的**在一组不同的数中找到和为定值的两个数（存在线性时间解）**的变体。先将所有 voucher 按照长度排序，然后左右夹逼寻找所有满足长度和条件的旅程对，找到其中不重叠的那些里票价最低的。故平均复杂度$T \sim O(n\log n)$最坏情况$T \sim O(n^2)$，最坏情况为**所有元素的值相同且任意两个元素都满足长度和条件**，此时算法退化为蛮力搜索。该算法在第 36 个测试样例上超时。

（超时）完整代码:

```
#include <algorithm>
#include <cstdio>
#include <iostream>

const int N = 200005;
const int INF = INT_MAX;

struct voucher {
    int l, r, cost;
```

```
};

bool cmp(voucher a, voucher b) {
	return (a.r - a.l) < (b.r - b.l);
}

bool seperate(voucher a, voucher b) {
	return a.r < b.l || b.r < a.l;
}

int sum(voucher a, voucher b) {
	return (a.r - a.l + 1) + (b.r - b.l + 1);
}

int dur(voucher a) {
	return a.r - a.l + 1;
}

int n, x;
voucher v[N];
void read() {
	scanf("%d %d", &n, &x);
	for (int i = 0; i < n; i++)
		scanf("%d %d %d", &v[i].l, &v[i].r, &v[i].cost);
}

void solve() {
	int mn_cost = INF;
	std::sort(v, v + n, cmp);

	int i = 0, j = n - 1;
	while (i < j) {
		if (sum(v[i], v[j]) > x){
			j--;
			continue;
		}
		if (sum(v[i], v[j]) < x) {
			i++;
			continue;
		}

		int l1 = dur(v[i]), l2 = dur(v[j]), j0 = j;
		while (i < j0 && dur(v[i]) == l1) {
			j = j0;
			while (i < j && dur(v[j]) == l2) {
```

```
                    if (seperate(v[i], v[j])) mn_cost = std::min(mn_cost, v[i].cost + v[j].cost);
                    j--;
                }
                i++;
            }
        }
        printf("%d\n", mn_cost == INF ? -1 : mn_cost);
}

int main() {
        read();
        solve();
}
```

kriii 基于 hash 表的遍历算法:

从 WeiYong 算法的最坏情况可以看出，超时的原因在于便利了太多相互**重叠**的元素组合。Kriii 给出的算法通过恰当地组织数据巧妙地解决了这个问题。下面对这个算法作一总结:

首先，我们希望只遍历那些可行的旅程组，从中找到价钱最小的组合。如何唯一遍历所有可行旅程对呢？对于任意一个旅程，如果能找到并一次遍历所有它作为后半段的可行旅程对，就实现了不重复地遍历所有可行旅程对。对于任意旅程，可以作为前半段从而构成可行旅程对的旅程有两个要求——[1] 长度和为x、[2] 不重叠。那么只需构建以时长为索引、以所有具有该时长的旅程的集合为值的哈希表，则可以直接定位所有具有特定时长的旅程，如果这些旅程又是按照开始时间有序排列，那么又可以做到只遍历其中不重叠的部分。$T\sim O(n)$

上述算法的实现如下:

完整代码（STL 二元组实现版本）:

```
#include <algorithm>
#include <vector>
#include <cstdio>

const int N = 200005;
const int INF = 2000000005;

std::vector<std::pair<int, int>> ht[N];

int n, x;
void read() {
        scanf("%d %d", &n, &x);
        for (int i = 0; i < n; i++) {
                int l, r, cost;
                scanf("%d %d %d", &l, &r, &cost);
                ht[r - l + 1].push_back({l, cost});
        }
}

void solve() {
```

```
    for (int i = 1; i <= x - 1; i++)
        std::sort(ht[i].begin(), ht[i].end());

    int ans = INF;
    for (int d = 1; d <= x - 1; d++) {   // For all length for the second part of vacation.
        auto &u = ht[d], &v = ht[x - d];
        int mn_v = INF; // Min of v[0] ... v[j].
        for (int i = 0, j = 0; i < u.size(); i++) {
            while (j < v.size() && v[j].first + (x - d) - 1 < u[i].first)
                mn_v = std::min(mn_v, v[j].second), j++;
            if (mn_v != INF) ans = std::min(ans, u[i].second + mn_v);
        }
    }

    printf("%d\n", ans == INF ? -1 : ans);
}

int main() {
    read();
    solve();
}
```

注：标红语句若不加判断将导致 32 位 int 型溢出。

注 2：注意 mn_v 的复用，使得 u[]和 v[]在一趟遍历完。若对每个 u[i]，v[j]都从头开始遍历，造成的重复计算仍会使第 36 个测试点超时。

完整代码 2（结构体实现的版本）：

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

const int X = 200005;
const int INF = 2000000005;

struct voucher {
    int l, r, cost;
};

std::vector<voucher> ht[X];
int n, x;
void read() {
    scanf("%d %d", &n, &x);
    for (int i = 0; i < n; i++) {
        voucher tmp;
        scanf("%d %d %d", &tmp.l, &tmp.r, &tmp.cost);
```

```
            ht[tmp.r - tmp.l + 1].push_back(tmp);
        }
}

bool cmp(voucher a, voucher b) {
        return a.l < b.l;
}

void solve() {
        for (int i = 1; i <= x - 1; i++)
                sort(ht[i].begin(), ht[i].end(), cmp);

        int ans = INF;
        for (int d = 1; d <= x - 1; d++) {
                auto &u = ht[d], &v = ht[x - d];
                int mn = INF;
                for (int i = 0, j = 0; i < u.size(); i++) {
                        while (j < v.size() && v[j].r < u[i].l) {
                                mn = std::min(mn, v[j].cost);
                                j++;
                        }
                        if (mn != INF) ans = std::min(ans, u[i].cost + mn);
                }
        }

        printf("%d\n", ans == INF ? -1 : ans);
}

int main() {
        read();
        solve();
}
```

Codeforces 821C (Round 420, Div 2)

Okabe and Boxes (贪心法，单调栈)

题述：

Okabe 和超级大黑客 Darus 正在整理箱子，n个箱子编号$1 \ldots n$，初始堆栈内没有箱子。

Okabe 个人控制欲极强，他给 Darus 下了$2n$条指令，其中n条为将某个箱子放入堆栈，另外n条为将栈顶箱子弹出并丢弃。Okabe 希望 Darus 丢弃箱子的顺序按编号从小到大排列。当然，这就意味着当需要被丢弃的箱子不在栈顶时，丢弃操作无法进行。

Darus 可以在两条指令之间的时间内将栈内的箱子调整成任意顺序。

请帮助 Darus 计算为了实现按照编号顺序丢弃箱子，最少需要做多少次栈内箱子顺序的调整。

输入：

第一行为一个整数$n(1 \le n \le 3 \times 10^5)$——箱子的数量。

接下来的$2n$行每行开始为一个取值或为“add”或为“remove”的字符串。如果为“add”那么还会在后跟一个整数$x(1 \le x \le n)$——表示 Darus 需要将x压栈。

输入保证共有n次 add 指令，所有 add 指令的操作数不同，其余n次为 remove 指令。另外，保证每个箱子在需要出栈之前已经入栈。

输出：

输出 Darus 需要对栈内箱子排序的最少次数。

测试样例：

Input	Output
3 add 1 remove add 2 add 3 remove remove	1
7 add 3 add 2 add 1 remove add 4 remove remove remove add 6 add 7 add 5 remove remove remove	2

注释：

测试样例一中，Darus 需要在 3 号箱子入栈后对栈内排序。

测试样例二中，Darus 需要在 4 号箱子和 7 号箱子入栈后对栈内排序。

WeiYong 的贪心算法:

首先，Darus 只需要在必要的时候对栈内排序（即每当需要弹出的箱子不在栈顶时）。证明很简单：对多的箱子排序总比对少的箱子排序要好，因为这样总能使更多的箱子按序排放。于是得到贪心解：模拟所有步骤，每当应该弹出的箱子不在栈顶时就对栈内元素从顶到底升序排序。

Tutorial 优化:

以上算法$T\sim O(n^2logn)$。注意到每当我们重排栈内元素，总可以使的当前栈内的所有元素处于其最优的位置（最优的意义为不该它在栈顶的时候它就不在栈顶，即**这些元素不会再成为导致重排的原因**——满足该条件的栈内序为栈底到栈顶的降序排列），于是可以不再考虑它们，把栈清空即可。该算法运行过程中每只箱子只进、出栈各一次，摊还复杂度$T\sim O(n)$。

完整代码:

```
#include <stack>
#include <cstdio>

int n;
void read() {
    scanf("%d", &n);
}

char ins[10];
std::stack<int> stk;
void solve() {
    int ans = 0;
    int cur = 1;
    for (int i = 0; i < n * 2; i++) {
        scanf("%s", ins);
        if (ins[0] == 'a') {
            int tmp;
            scanf("%d", &tmp);
            stk.push(tmp);
        } else {
            if (!stk.empty()) {
                if (stk.top() == cur) {
                    stk.pop();
                } else {
                    ans++;
                    while (!stk.empty()) stk.pop();
                }
            }
            cur++;
        }
    }
    printf("%d\n", ans);
}

int main() {
```

```
    read();
    solve();
}
```

Codeforces 816B (Round 419, Div 2)

Karen and Coffee (BIT，区间覆盖数组)

题述：

为了保持一天的好精神，Karen 需要喝点儿咖啡。

作为一个咖啡狂热爱好者，Karen 想知道煮咖啡的最佳温度。的确，她读了大量的有关咖啡制作的书，包括广受好评的《咖啡烹制艺术》。

她有n个咖啡烹制秘方，每个秘方推荐了一个煮咖啡的最佳温度区间$[l_i, r_i]$。

Karen 认为，当且仅当一个温度在至少k个秘方的推荐范围内时，可以以此作为自己的烹调温度。

Karen 性格易变，所以她提出了q个问题，在每个问题中她给出一个温度区间$[a, b]$，你的任务是告诉她该区间内可行的整数温度值的数量。

输入：

第一行为三个整数$n, k, q(1 \le k \le n \le 200000, 1 \le q \le 200000)$——秘方数、可行温度的最少推荐秘方数、提问数。

接下来的n行每行描述一个秘方。具体地，第i行包含两个整数$l_i, r_i(1 \le l_i \le r_i \le 200000)$——第$i$个秘方推荐温度区间的左右端点。

接下来的q行每行描述一次提问。具体地，第i行包含两个整数$a_i, b_i(1 \le a_i \le b_i \le 200000)$——第$i$次提问温度区间的左右端点。

输出：

对于每次提问，打印一行一个整数——问题区间内的可行整温度数。

测试样例：

Input	Output
3 2 4 91 94 92 97	3 3

97 99 92 94 93 97 95 96 90 100	0 4
2 1 1 1 1 200000 200000 90 100	0

注释：

测试样例一中，Karen 知道三个咖啡烹制秘方，所推荐得温度分别为$[91,94],[92,97],[97,99]$，当某温度被至少两个秘方推荐过则被视为可行。

Karen 问了四个问题：在第一个问题中她想知道$[92,94]$内的可行整温度数，结果为3个：$92,93,94$；在第二个问题中欲求区间为$[93,97]$，结果为3个：$93,94,97$；在第三个问题中欲求区间为$[95,96]$，该区间内没有可行温度；在第四个问题中欲求区间为$[90,100]$，结果为4个：$92,93,94,97$。

测试样例二中，Karen 知道两个秘方——分别只推荐了单温度点$1,200000$，可行温度的最少被推荐次数为1。

在 Karen 唯一的提问中，欲求区间内没有被推荐过的温度点。

参考 kipa00 的 BIT 解：

题目涉及区间操作，直观想到用**线段树**或**二进索引树**组织数据。基于此思路，以算法的 BIT 实现为例，我们首先根据输入构建记录整个温度区间$[1,200000]$上每个温度点是否可行的标记数组$mark[200001]$，然后对于每个问题区间查询标记数组$mark[]$的区间和即可。

怎样计算每个温度点每推荐的次数呢？用一个数组$cov[200001]$的**前缀和**表示相应温度点被推荐的次数（将其称为**区间包含数组**）。初始置零，对于每个推荐温度区间$[l_i,r_i]$，$a[l_i]$++，$a[r_i+1]$—即可达到预期目的。标记数组的区间和$rangeSum(l,r)$可由两端点前缀和的差$prefixSum(r)-prefixSum(l-1)$得到。实现过程中，可将同一树状数组复用两遍，分别以$cov[]$和$mark[]$为原数组，注意中间需要让 BIT 归零。

完整代码：

```
#include <cstring>
#include <cstdio>

const int N = 200005;
const int Q = 200005;
const int UPB = 200005;

int n, k, q;
int l[N], r[N];
int a[Q], b[Q];
void read() {
	scanf("%d %d %d", &n, &k, &q);
	for (int i = 0; i < n; i++)
		scanf("%d %d", &l[i], &r[i]);
	for (int i = 0; i < q; i++)
		scanf("%d %d", &a[i], &b[i]);
}
```

```
int BIT[UPB + 1];

void add(int u, int v) {       // arr[u] <- v.
    for (u++ ; u < UPB; u += u & -u)
        BIT[u] += v;
}

int prefix_sum(int u) { // Sum of arr[0]...arr[u].
    int ans = 0;
    for (u++ ; u; u -= u & -u)
        ans += BIT[u];
    return ans;
}

int mark[UPB];
void solve() {
    for (int i = 0; i < n; i++)
        add(l[i], 1), add(r[i] + 1, -1);
    for (int i = 0; i < UPB; i++)
        if (prefix_sum(i) >= k) mark[i] = 1;
    memset(BIT, 0, sizeof(BIT));
    for (int i = 0; i < UPB; i++)
        if (mark[i] == 1) add(i, 1);
    for (int i = 0; i < q; i++)
        printf("%d\n", prefix_sum(b[i]) - prefix_sum(a[i] - 1));
}

int main() {
    read();
    solve();
}
```

参考 Tian.xie 的$T\sim O(n+q)$解:

虽然此题涉及标记数组 mark[]区间和、区间包含数组cov[]的前缀和操作，然而此二者在建立之后，都只有区间查询操作而没点更新。故直接构建二者的前缀和数组即可在常数时间完成查询任务。

完整代码:

```
#include <cstring>
#include <cstdio>

const int N = 200005;
const int Q = 200005;
const int UPB = 200005;

int n, k, q;
int l[N], r[N];
```

```
int a[Q], b[Q];
void read() {
	scanf("%d %d %d", &n, &k, &q);
	for (int i = 0; i < n; i++)
		scanf("%d %d", &l[i], &r[i]);
	for (int i = 0; i < q; i++)
		scanf("%d %d", &a[i], &b[i]);
}

int cov[UPB], mark[UPB];
void solve() {
	for (int i = 0; i < n; i++)		// Generate cov[UPB].
		cov[l[i]]++, cov[r[i] + 1]--;
	for (int i = 1; i < UPB; i++)	// Generate prefix-sum of cov[UPB].
		cov[i] += cov[i - 1];
	for (int i = 0; i < UPB; i++)	// Generate mark[UPB].
		if (cov[i] >= k) mark[i] = 1;
	for (int i = 1; i < UPB; i++)	// Generate prefix-sum of mark[UPB].
		mark[i] += mark[i - 1];
	for (int i = 0; i < q; i++)
		printf("%d\n", mark[b[i]] - mark[a[i] - 1]);
}

int main() {
	read();
	solve();
}
```

Codeforces 798D (Round #410)

Mike and Distribution (排序，构造性算法)

题述：

Mike 一直关注着社会的公平性问题并为之深深困扰，以至于做编程题的时候也在想着。现在，Mike 有两个长度为n的正整数序列$A=[a_1,a_2,\dots,a_n]$和$B=[b_1,b_2,\dots,b_n]$，基于这两个序列 Mike 提出了一个关于公平性的问题。

为了测试你对“不公平”现象的敏感程度，Mike 要你找到原始序列的“不公平”子集。具体地，他要你选出k个互不相同的数字$P=[p_1,p_2,\dots,p_k]$，对于其中任意p_i有$1\le p_i\le n$，作为从A,B两个序列中筛选元素的下标集。子集P被定义为“不公平”当且仅当：$2(a_{p_1}+\cdots+a_{p_k})$大于$A$中所有元素和，且$2(b_{p_1}+\cdots+b_{p_k})$大于$B$中所有元素和。若$k$过大则上述子集很容易找到，故限制$k$不能超过$\left\lfloor\frac{n}{2}\right\rfloor+1$。

Mike 保证满足上述条件的解存在，那么，请满足他的好奇心吧！

输入：

第一行为一个整数$n(1\le n\le 10^5)$——序列长度。

第二行为n个由空格分隔的整数$a_1,a_2,\dots,a_n(1\le a_i\le 10^9)$——序列 A 中的元素。

第二行为n个由空格分隔的整数$b_1,b_2,\dots,b_n(1\le b_i\le 10^9)$——序列 B 中的元素。

输出：

第一行为一个整数k——所选子集的元素数，不超过$\left\lfloor\frac{n}{2}\right\rfloor+1$。

第二行为k个整数$p_1,p_2,\dots,p_k(1\le p_i\le n)$——下标序列$P$中的元素，顺序任意，但要互不相同。

测试样例：

Input	Output
5 8 7 4 8 3 4 2 5 3 7	3 1 4 5

Tutorial 算法:

将$a_1,a_2,\dots,a_n$非升序排列得到$a_{c_1},a_{c_2},\dots,a_{c_n}$，满足$a_{c_i}\ge a_{c_{i+1}}$。首先考虑$n$为奇数的情况，令$p_1=c_1$，从$c_2$开始两个两个考虑，对于$(c_{2k},c_{2k+1})$，将$b_{c_{2k}}$和$b_{c_{2k+1}}$中较大者的下标赋给$p_{k+1}$，不难证明这样得到的下标集合符合题意。

草纸上进行如下模拟 / 演算（演算技巧：奇数一般取 5/7，偶数一般取 4/6）:

n为奇数	$\begin{cases}a_{c_1}\ge a_{c_2}\ge a_{c_3}\ge a_{c_4}\ge a_{c_5}\\ b_{c_1},(b_{c_2}\le b_{c_3}),(b_{c_4}\ge b_{c_5})\end{cases}$
n为偶数	$\begin{cases}a_{c_1}\ge a_{c_2}\ge a_{c_3}\ge a_{c_4}\ge a_{c_5}\ge a_{c_6}\\ b_{c_1},(b_{c_2}\le b_{c_3}),(b_{c_4}\ge b_{c_5}),b_{c_6}\end{cases}$

对于n为偶数的情况，将序列分成两部分考虑——除最后一项外的前奇数项$a_{c_1},a_{c_2},\dots,a_{c_{n-1}}$可完全划归到$n$为奇数的情况。再加入最后一项$p_{\frac{n}{2}+1}=c_n$即可。$T\sim O(n\log n)$

注：此题解存在为算法本身保证，与输入无关。

完整代码:

```
const int N = 100005;
pair<int, int> a[N];
int b[N], p[N];
int n;
```

```
int main() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++)
		scanf("%d", &a[i].first), a[i].second = i;
	for (int i = 0; i < n; i++)
		scanf("%d", &b[i]);

	sort(a, a+n), reverse(a, a+n);
	if (n & 1) {
		p[0] = a[0].second;
		for (int i = 1; i <= n / 2; i++)
			p[i] = b[a[2 * i - 1].second] > b[a[2 * i].second] ? a[2 * i - 1].second : a[2 * i].second;
		printf("%d\n", n / 2 + 1);
		for (int i = 0; i <= n / 2; i++)
			printf("%d ", p[i] + 1);
		printf("\n");
	} else {
		p[0] = a[0].second;
		for (int i = 1; i < n / 2; i++)
			p[i] = b[a[2 * i - 1].second] > b[a[2 * i].second] ? a[2 * i - 1].second : a[2 * i].second;
		p[n / 2] = a[n - 1].second;
		printf("%d\n", n / 2 + 1);
		for (int i = 0; i <= n / 2; i++)
			printf("%d ", p[i] + 1);
		printf("\n");
	}

	return 0;
}
```

第三章：竞赛篇

——Google 校招赛及 Google/Facebook 杯赛题目

3.1 Kickstart 2017 Round A
3.2 Kickstart 2017 Round B
3.3 Kickstart 2017 Round C
3.4 Kickstart 2017 Round D
3.5 Kickstart 2017 Round E
3.6 Kickstart 2017 Round F
3.7 Kickstart 2017 Round G
3.8 Kickstart 2018 Round A
3.9 Kickstart 2018 Round B
3.10 Kickstart 2018 Round C
3.8 CodeJam 2017 Round 2
3.9 CodeJam 2017 Round 3
3.10 CodeJam 2017 World Final
3.11 HackerCup 2017 Round 3
3.12 HackerCup 2017 World Final

3.1 Kickstart 2017 Round A

Problem A. Square Counting (计数问题，平方级数和，立方级数和，乘法逆元)

题述：

Panda 先生最近爱上了一个叫做 Square off 的新游戏。在这个游戏中，玩家需要在一个均匀分布的点阵中找到尽量多不同的正方形。玩家通过指出其四个顶点来标记一个正方形。正方形的四条边需等长，当然，具体长度可取任意值，而且正方形的边也无需平行于点阵的轴。每找到一个正方形玩家得一分。两个正方形被视作不同当且仅当它们的顶点不完全相同。

Panda 先生现在拿到了一个R行C列的点阵。那么他能在点阵内找到多少不同的正方形呢？由于结果可能很大，输出其模1000000007的余数即可。

输入：

第一行为一个整数T——测试样例的数量。接下来的T行，每行为两个整数R和C——分别为点阵每行、每列的点数。

输出：

对于么个测试样例，输出一行包含$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是点阵中能找到的最多的正方形数量。

参数限制：

$1 \le T \le 100$

小数据集：

$2 \le R \le 1000$

$2 \le C \le 1000$

大数据集：

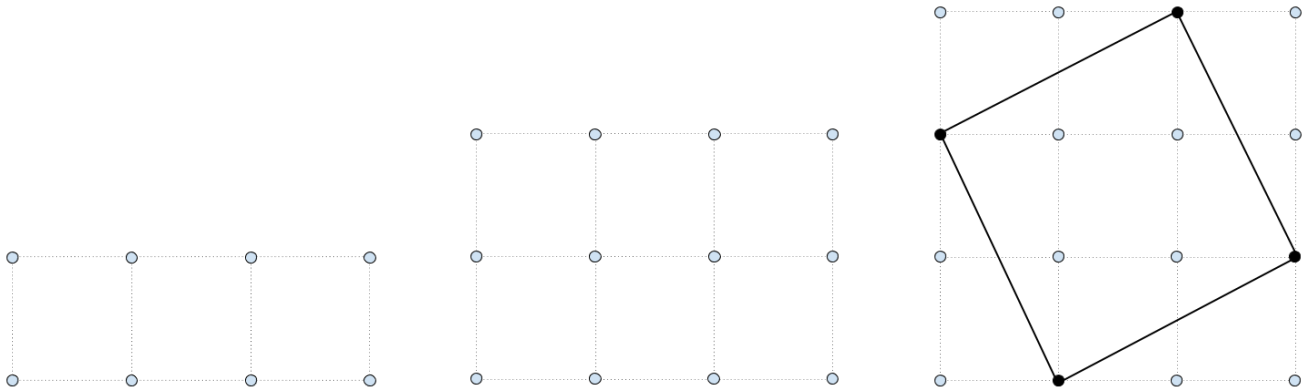
$2 \le R \le 10^9$

$2 \le C \le 10^9$

测试样例：

Input	Output
4 2 4 3 4 4 4 1000 500	Case #1: 3 Case #2: 10 Case #3: 20 Case #4: 624937395

下图给出前三个测试样例的点阵，及第三个测试样例中一个合法的正方形。



WeiYong 的直接公式解:

不妨设$R \le C$（否则转置点阵）。首先，边长为k的正方形的内切正方形数量为k，不难得到下表:

正方形边长	在点阵中的数量	其中内切正方形数量
1	$(R-1)*(C-1)$	1

2	$(R-2)*(C-2)$	2
…	…	…
i	$(R-i)*(C-i)$	i
…	…	…
$R-1$	$1*(C-R+1)$	$R-1$

最终求得所有正方形的总数$n=\frac{R^2(R-1)C}{2}-\frac{(R+C)(R-1)R(2R-1)}{6}+\frac{R^2(R-1)^2}{4}$。编程时变量用 64 位整型存储，先计算加项再计算减项，除2,4,6用其在**模 MOD 乘法群**上的逆元表示，常数逆元用费马小定理推论$a^{-1}=a^{MOD-2}$+快速幂预先求得。

完整代码:

```
const int MOD = 1000000007;
int T, inv_6, inv_4, inv_2;
long long R, C;

int pw(int a, int b) {
	int ans = 1;
	while (b) {
		if (b & 1) ans = 1ll * ans * a % MOD;
		a = 1ll * a * a % MOD;
		b >>= 1;
	}
	return ans;
}

void read() {
	scanf("%lld %lld", &R, &C);
}

int solve() {
	long long ans = 0;
	if (R > C) swap(R, C);
	ans += R * R % MOD * C % MOD * (R - 1) % MOD * inv_2 % MOD;
	ans += R * R % MOD * (R - 1) % MOD * (R - 1) % MOD * inv_4 % MOD;
	ans -= (R + C) * (R - 1) % MOD * R % MOD * (2 * R - 1) % MOD * inv_6 % MOD;
	ans += MOD;
	ans %= MOD;
	return (int)ans;
}

int main() {
	freopen("in.txt", "r", stdin);
	freopen("out.txt", "w", stdout);
	scanf("%d", &T);
	inv_2 = pw(2, MOD - 2), inv_4 = pw(4, MOD - 2), inv_6 = pw(6, MOD - 2);
```

```
    for (int t = 0; t < T; t++) {
        read();
        int ans = solve();
        printf("Case #%d: %d\n", t + 1, ans);
    }

    return 0;
}
```

数学知识回顾:

群公理: 闭、结合、幺、逆; n的所有余数构成整数**模n乘法群**。

费马小定理: $a^{p-1} \equiv 1 (\mathrm{mod}\, p)$

3.1 Kickstart 2017 Round A

Problem B. Pattern Overlap (动态规划，带通配符的模式匹配)

题述：

Alice 喜爱阅读，她买了很多的书。她把她的书放在两个箱子里，每个箱子上有一个标签，其上的模式匹配该箱子内所有书籍的书名。一个模式由大 / 小写的英文字母和通配符*构成。通配符*可以替代零至四个字。例如，书名分别为 GoneGirl 和 GoneTomorrow 的书都可以被放在带有模式 Gone**作为标签的箱子内，而书名为 TheGoneGirl 或 GoneWithTheWind 的书则不可以。

Alice 好奇是否存在一本书同时可以被放在两个箱子中，也就是说，是否存在一个书名同时和两个箱子上的模式匹配。

输入：

第一行为一个整数T——测试样例数量。接着是T个测试样例，每个测试样例包含两行：每行为一个由大 / 小写英文字母或*构成的字符串。

输出：

对于每个测试样例，输出一行包含$Case\ \#x: y$，其中x是从1开始的测试样例序号，若存在同时匹配两个模式的字符串则y为 True，否则y为 False。

参数限制：

$1 \le T \le 50$

小数据集：

每个模式的长度在1到200之间

每个模式内的通配符数量不超过5个

大数据集：

每个模式的长度在1到2000之间

测试样例：

Input	Output
3 **** It Shakes*e S*speare Shakes*e *peare	Case #1: TRUE Case #2: TRUE Case #3: FALSE

测试样例 1 中，题目为 It 的书同时和两种模式匹配。注意*可以匹配 0 个字符。

测试样例 2 中，题目为 Shakespare 的书同时和两种模式匹配。

测试样例 3 中，没有任何书名能同时和两个模式匹配。特别地，Shakespare 因为开头缺少5个字符故不能和第二个模式匹配。

WeiYong1024 的深搜解:

$dfs(int\ i, int\ j)\{\}$ 返回$s1[i \dots l_1 - 1]$和$s2[j \dots l_2 - 1]$是否匹配，顶层调用函数$dfs(0,0)$。具体地，探底条件为$i = l_1$或$j = l_2$；如未探底则分$s1[i] = '*'$和$s1[i] \neq '*'$两类条件讨论。

注：如下实现无法通过小数据集，不知问题出现在何处。（已解决）

完整代码:

```
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;
```

```
string s1, s2;
int T;

bool dfs(int i, int j) {        // Return whether s1[i...l1-1] and s2[j...l2-1] match each other.
    if (i == s1.size()) {     // Current s1[i] reaches its end.
        for (int p = j; p < s2.size(); p++)
            if (s2[p] != '*') return false;
        return true;
    } else if (s1[i] == '*') {    // Current s1[i] is '*'.
        int p = 1;
        while (s1[i + p] == '*' && i + p < s1.size())
            p++;
        for (int q = 0; q <= 4 * p && j + q < s2.size(); q++)
            if (s2[j + q] == s1[i + p] || s2[j + q] == '*')
                if (dfs(i + p, j + q)) return true;
        if (s2.size() - j <= 4 * p) return dfs(i + p, (int)s2.size());
        return false;
    } else {        // Current s1[i] is not '*'
        if (s2[j] == s1[i])
            return dfs(i + 1, j + 1);     // Current s2[j] == si[i].
        else if (s2[j] == '*') { // Current s2[j] is '*'
            int p = 1;
            while (s2[j + p] == '*' && j + p < s2.size())
                p++;
            for (int q = 0; q <= 4 * p && i + q < s1.size(); q++)
                if (s1[i + q] == s2[j + p] || s1[i + q] == '*')
                    if (dfs(i + q, j + p)) return true;
            if (s1.size() - i <= 4 * p) return dfs((int)s1.size(), j + p);
            return false;
        }
        else return false;
    }
}

int main() {
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    ios::sync_with_stdio(false);
    cin >> T;

    for (int t = 0; t < T; t++) {
        cin >> s1 >> s2;
        bool ans = dfs(0, 0);
        cout << "Case #" << t + 1 << ": " << (ans ? "TRUE" : "FALSE") << endl;
```

```
    }

    return 0;
}
```

此版本无法通过小数据集。以 Tian.Xie 提交的代码为准，WeiYong1024 在第 11,12,13,14,18,21,26,27,32 共 9 个测试点处给出的结果不同，且都是把 TRUE 错给成 FALSE。通过考察第 32 个测试点的运行情况，**找到问题——当一侧为通配符扫描另一侧字符计数时不应该考虑通配符**。

修改代码后如下：

（小数据集通过大数据集超时）完整代码：

```
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;

string s1, s2;
int T;

bool dfs(int i, int j) {       // Return whether s1[i...l1-1] and s2[j...l2-1] match each other.
    if (i == s1.size()) {      // Current s1[i] reaches its end.
        for (int p = j; p < s2.size(); p++)
            if (s2[p] != '*') return false;
        return true;
    } else if (s1[i] == '*') {    // Current s1[i] is '*'.
        int p = 1;
        while (s1[i + p] == '*' && i + p < s1.size())
            p++;
        int counter = 0;       // Num of non-'*' characters after j in s2[].
        for (int q = 0; counter <= 4 * p; q++) {
            if (j + q == s2.size())
                return dfs(i + p, j + q);
            if (s2[j + q] == '*') {
                if (dfs(i + p, j + q)) return true;
                continue;
            }
            if (s2[j + q] == s1[i + p])
                if (dfs(i + p, j + q)) return true;
            counter++;
        }
        return false;
    } else {       // Current s1[i] is not '*'
        if (s2[j] == s1[i])
            return dfs(i + 1, j + 1);     // Current s2[j] == si[i].
        else if (s2[j] == '*') { // Current s2[j] is '*'
            int p = 1;
```

```
            while (s2[j + p] == '*' && j + p < s2.size())
                p++;
            int counter = 0;
            for (int q = 0; counter <= 4 * p; q++) {
                if (i + q == s1.size())
                    return dfs(i + q, j + p);
                if (s1[i + q] == '*') {
                    if (dfs(i + q, j + p)) return true;
                    continue;
                }
                if (s1[i + q] == s2[j + p])
                    if (dfs(i + q, j + p)) return true;
                counter++;
            }
            return false;
        }
        else return false;
    }
}

int main() {
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    ios::sync_with_stdio(false);
    cin >> T;

    for (int t = 0; t < T; t++) {
        cin >> s1 >> s2;
        bool ans = dfs(0, 0);
        cout << "Case #" << t + 1 << ": " << (ans ? "True" : "False") << endl;
    }

    return 0;
}
```

此版通过小数据集，但大数据集用 8 分钟跑出 22/50 个测试点，显然超时。

（2017.7.15）注：这里通过 22/50 个测试点，并不是直观意义上的跑完了一半，而是前 22 个测试点规模较小，可以在 3s 内跑完，而第 23 个测试点在剩下的 7 分 50 多秒内都跑不完。

考虑增加备忘录的动规加速。开辟 4MB 连续内存存储备忘录，秒过大数据集，实现如下：

（最终版）完整代码：

```
#include <algorithm>
#include <string>
#include <iostream>

const int N = 2005;
```

```
std::string s1, s2;

void read() {
    std::cin >> s1 >> s2;
}

short dp[N][N];
bool dfs(int i, int j) {      // Return whether s1[i...l1-1] and s2[j...l2-1] match each other.
    if (dp[i][j] != -1) return dp[i][j];

    if (i == s1.size()) {     // Current s1[i] reaches its end.
        for (int p = j; p < s2.size(); p++)
            if (s2[p] != '*') return dp[i][j] = 0;
        return dp[i][j] = 1;
    } else if (s1[i] == '*') {   // Current s1[i] is '*'.
        int p = 1;
        while (s1[i + p] == '*' && i + p < s1.size())
            p++;
        int counter = 0;      // Num of non-'*' characters after j in s2[].
        for (int q = 0; counter <= 4 * p; q++) {
            if (j + q == s2.size())
                return dp[i][j] = dfs(i + p, j + q);
            if (s2[j + q] == '*') {
                if (dfs(i + p, j + q)) return dp[i][j] = 1;
                continue;
            }
            if (s2[j + q] == s1[i + p])
                if (dfs(i + p, j + q)) return dp[i][j] = 1;
            counter++;
        }
        return dp[i][j] = 0;
    } else {      // Current s1[i] is not '*'
        if (s2[j] == s1[i])
            return dp[i][j] = dfs(i + 1, j + 1);     // Current s2[j] == si[i].
        else if (s2[j] == '*') { // Current s2[j] is '*'
            int p = 1;
            while (s2[j + p] == '*' && j + p < s2.size())
                p++;
            int counter = 0;
            for (int q = 0; counter <= 4 * p; q++) {
                if (i + q == s1.size())
                    return dp[i][j] = dfs(i + q, j + p);
                if (s1[i + q] == '*') {
                    if (dfs(i + q, j + p)) return dp[i][j] = 1;
```

```
                continue;
            }
            if (s1[i + q] == s2[j + p])
                if (dfs(i + q, j + p)) return dp[i][j] = 1;
            counter++;
        }
        return dp[i][j] = 0;
    }
    else return dp[i][j] = 0;
  }
}

bool solve() {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      dp[i][j] = -1;
  return dfs(0, 0);
}

int main() {
  freopen("bin.txt", "r", stdin);
  freopen("bout.txt", "w", stdout);
  std::ios::sync_with_stdio(false);
  int T;
  std::cin >> T;

  for (int t = 0; t < T; t++) {
    read();
    bool ans = solve();
    std::cout << "Case #" << t + 1 << ": " << (ans ? "True" : "False") << std::endl;
  }

  return 0;
}
```

关于**测试**：测试的较高境界在于用简单、小规模的数据暴露出程序中的问题（让程序输出错误结果）。这个命题基于一个事实——即一般来说对于一个尚存错误的程序，在通过阅读代码无法定位错误的情况下，其错误往往只有一到两个，这个错误可能暴露在小规模数据下，也可能暴露在大规模数据下。

3.1 Kickstart 2017 Round A

Problem C. Space Cubes (二分搜索，立体几何)

题述：

“Look at the stars, look how they shine for you.”——玩酷乐队，“Yellow”

在遥远的银河，那里有很多的星星，每个星星都是一个（在三维空间中的）位置和半径确定的球体，星星之间可能相互有重叠。

星星们太美了，你想永远保留它们！你希望用两个边长相同的立方体放在空间的某处，使得对于每颗星星，至少有一个立方体将其完全包含（仅仅被两个立方体的并完全包含是不够的）。当星星上的任何一点都在立方体内部时我们说行星被立方体完全包含，恰好位于立方体表面的点也被视作在立方体内部。

立方体可以被放在空间中的任意位置，但它们的边必须平行于坐标轴。立方体与星星、立方体与立方体之间可以有重叠。

为了达到目标，立方体边长的最小整数取值是多少？

输入：

第一行为一个整数T——测试样例的数量，接着是T个测试样例。

每个测试样例开始一行为一个整数N——表示星星的数量。

接下来的N行每行为四个由空格分隔的整数X_i, Y_i, Z_i, R_i——分别为第i颗星星的空间坐标和半径。

输出：

对于每个测试样例，输出一行$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是解决上述问题的最短整数边长取值。

参数限制：

$1 \leq T \leq 100$

$-10^8 \leq X_i \leq 10^8$, for all i.

$-10^8 \leq Y_i \leq 10^8$, for all i.

$-10^8 \leq Z_i \leq 10^8$, for all i.

$-10^8 \leq R_i \leq 10^8$, for all i.

小数据集：

$1 \leq N \leq 16$

大数据集：

$1 \leq N \leq 2000$

测试样例：

Input	Output
3 3 1 1 1 1 2 2 2 1 4 4 4 1 3 1 1 1 2 2 3 4 1 5 6 7 1 3 1 1 1 1 1 1 1 1 9 9 9 1	Case #1: 3 Case #2: 5 Case #3: 2

测试样例 1 中，一种方案是将两个边长为3的立方体的最小坐标顶点分别放置在$(0, 0, 0)$和$(3, 3, 3)$的位置。

测试样例 2 中，一种解决方案是将两个边长为5的立方体的最小坐标顶点分别放置在$(-1,-1,-1)$和$(1,2,3)$的位置。

（错误算法）WeiYong1024 三个维度分别分析的解法:

首先，包含球体的问题等价于包含它的外接正方体。将原问题分解到三个维度上分别考虑，以 X 轴为例，问题等价于找到 X 轴上的两条等长线段，使得所有球体在 X 轴上的投影线段都至少包含在一个线段之内，令l_x表示符合上述条件线段的最小长度，则最终答案为$\max(l_x, l_y, l_z)$。

注：上述算法的如下实现无法通过 Small case，不知问题所在。

——上述算法是错误的：首先考虑采用上述算法后将三个维度重新组合起来得到的空间区域是什么，是两个正方体吗？并不是！虽然得到的区域投影到每个轴上是位于两端的两条等长线段，然而这两个线段在三个维度的组合有八种，去掉两个正方体交换位置的重复情况有四种，就是位于大长方体四个对角线两段的情况。而我的算法得到的区域是一个大长方体正中间扣去一个小长方体后剩下的几何体，由于该空间体积大于位于八个顶点的正方体的和，故得到的结果也应小于等于正确结果。

（上述错误算法的）完整代码:

```
struct seg {
    int idx;
    int l, r;
};

const int N = 20005;
seg segs[N];
seg l_order[N];
seg r_order[N];
int x[N], y[N], z[N], r[N];
int T, n;
unordered_map<int, int> mp;

void read() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%d %d %d %d", &x[i], &y[i], &z[i], &r[i]);
}

bool cmp1(seg a, seg b) {
    return a.l < b.l;
}
bool cmp2(seg a, seg b) {
    return a.r < b.r;
}

int cover() {
    for (int i = 0; i < n; i++)
        l_order[i] = segs[i], r_order[i] = segs[i];
    sort(l_order, l_order + n, cmp1);
    sort(r_order, r_order + n, cmp2);
```

```
    int a = -1, b = n;
    int L = r_order[0].l, R = l_order[n - 1].r;
    mp.clear();
    while(mp.size() < n) {
        if (r_order[a + 1].r - L < R - l_order[b - 1].l)
            mp[r_order[++a].idx]++;
        else if(r_order[a + 1].r - L > R - l_order[b - 1].l)
            mp[l_order[--b].idx]++;
        else mp[r_order[++a].idx]++, mp[l_order[--b].idx]++;
    }
    return max(r_order[a].r - L, R - l_order[b].l);
}

int solve() {
    int ans = INT_MIN;
    for (int i = 0; i < n; i++)
        segs[i].idx = i, segs[i].l = x[i] - r[i], segs[i].r = x[i] + r[i];
    ans = max(ans, cover());
    for (int i = 0; i < n; i++)
        segs[i].idx = i, segs[i].l = y[i] - r[i], segs[i].r = y[i] + r[i];
    ans = max(ans, cover());
    for (int i = 0; i < n; i++)
        segs[i].idx = i, segs[i].l = z[i] - r[i], segs[i].r = z[i] + r[i];
    ans = max(ans, cover());
    return ans;
}

int main() {
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    scanf("%d", &T);

    for (int t = 0; t < T; t++) {
        read();
        int ans = solve();
        printf("Case #%d: %d\n", t + 1, ans);
    }
    return 0;
}
```

Tian.Xie 的算法:

该问题的解有**大于等于某一阈值可行，小于则不可行的二段性**，所求就是参数（长度）的这个阈值。类似 Codeforces 780B 的思路，用**二分搜索**寻找**阈值**。具体地，给定一个线段长度判断以之为边长的两个正方体是否可以包含所有球体的方法：首先通过一次扫描所有球体确定它们公共的外接长方体位置，然后判断将两个小正方体分别放置在四组对角是否满足每个球体至少被一个正方体完全涵盖即可。$T\sim O(4N\log_2 2e9)$

为什么若四组对角不满足则其他位置也不满足？因为四组对角是唯一能让两个小正方体把长方体撑起来的位置，故其他位置必然不满足。

注：下述实现中的二分搜索可作为二分搜索阈值的模版（原则：$[l,r]$为所有未知元素）

完整代码：

```
#include <algorithm>
#include <iostream>

const int N = 2005;
int x[N], y[N], z[N], r[N];
int x_min, y_min, z_min, x_max, y_max, z_max;
int T, n;

void read() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%d %d %d %d", &x[i], &y[i], &z[i], &r[i]);
}

bool cover(int x1_min, int y1_min, int z1_min, int x1_max, int y1_max, int z1_max,
           int x2_min, int y2_min, int z2_min, int x2_max, int y2_max, int z2_max) {    // Whether the n bolls are covered by either one of the two cubes.
    for (int i = 0; i < n; i++) {
        int invalid = 0;
        if (!(x[i] - r[i] >= x1_min && y[i] - r[i] >= y1_min && z[i] - r[i] >= z1_min &&
              x[i] + r[i] <= x1_max && y[i] + r[i] <= y1_max && z[i] + r[i] <= z1_max))
            invalid++;
        if (!(x[i] - r[i] >= x2_min && y[i] - r[i] >= y2_min && z[i] - r[i] >= z2_min &&
              x[i] + r[i] <= x2_max && y[i] + r[i] <= y2_max && z[i] + r[i] <= z2_max))
            invalid++;
        if (invalid == 2) return false;
    }
    return true;
}

bool check(int len) {    // Whether len is enough for the two cubes to cover all balls.
    if (cover(x_min, y_min, z_min, x_min + len, y_min + len, z_min + len,
              x_max - len, y_max - len, z_max - len, x_max, y_max, z_max)) return true;
    if (cover(x_min, y_max - len, z_min, x_min + len, y_max, z_min + len,
              x_max - len, y_min, z_max - len, x_max, y_min + len, z_max)) return true;
    if (cover(x_min, y_min, z_max - len, x_min + len, y_min + len, z_max,
              x_max - len, y_max - len, z_min, x_max, y_max, z_min + len)) return true;
    if (cover(x_min, y_max - len, z_max - len, x_min + len, y_max, z_max,
              x_max - len, y_min, z_min, x_max, y_min + len, z_min + len)) return true;
    return false;
}
```

```
int solve() {
    x_min = y_min = z_min = INT_MAX;
    x_max = y_max = z_max = INT_MIN;
    for (int i = 0; i < n; i++) {     // Get the two representative vertexes of the circumscribed cuboid of all balls.
        x_min = std::min(x_min, x[i] - r[i]);
        y_min = std::min(y_min, y[i] - r[i]);
        z_min = std::min(z_min, z[i] - r[i]);
        x_max = std::max(x_max, x[i] + r[i]);
        y_max = std::max(y_max, y[i] + r[i]);
        z_max = std::max(z_max, z[i] + r[i]);
    }

    int l = 0, r = 4e8;
    int ans = r;
    while(l <= r) {
        int m = (l + r) >> 1;
        if (check(m)) ans = m, r = m - 1;
        else l = m + 1;
    }
    return ans;
}

int main() {
    freopen("cin.txt", "r", stdin);
    freopen("cout.txt", "w", stdout);
    scanf("%d", &T);

    for (int t = 0; t < T; t++) {
        read();
        int ans = solve();
        printf("Case #%d: %d\n", t + 1, ans);
    }
}
```

3.2 Kickstart 2017 Round B

Problem A. Math Encoder (计数问题，预处理)

题述：

Math 教授正在着手一项涉密项目，目前需要解决一个问题——找到将一组数字高效编码为一个数字的方法。基于大量研究基础，Mat 教授提出了以下三步高效编码数字的方法：

1. 找到这一组数字的所有非空子集，对于每个非空子集计算其极差（最大元减最小元）。注意对于只有一个成员的子集，该成员既是最大元也是最小元。全集也算作一个子集。
2. 将所有极差相加得到编码结果。
3. 最终结果对 1000000007 取余。

教授给出了如下例子并做以说明。那么给定一组数字，你能帮教授实现一个高效的函数来计算最终的编码结果吗？

输入：

第一行一个数字T为测试样例的数量，接下来为T个测试样例，每个测试样例由两行构成。

1. 第一行为一个正整数N——组内数字数量
2. 第二行为N个正整数K_i，按照非降序排列。

输出：

对于每一个测试样例，打印一行$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是编码结果。

由于结果可能很大只需输出结果对素数 100000007 取余的结果。

参数限制：

$1 \le T \le 25$.

$1 \le K_i \le 10000$, for all i.

$K_i \le K_{i+1}$, for all $i < N - 1$.

小数据集：

$1 \le N \le 10$

大数据集：

$1 \le N \le 10000$

测试样例：

Input	Output
1 4 3 6 7 9	Case #1: 44

样例解释：

1. 找到所有非空子集，计算极差：

$[3], max-min = 3-3 = 0$

$[6], max-min = 6-6 = 0$

$[7], max-min = 7-7 = 0$

$[9], max-min = 9-9 = 0$

$[3,6], max-min = 6-3 = 3$

$[3,7], max-min = 7-3 = 4$

$[3,9], max-min = 9-3 = 6$

$[6,7], max-min = 7-6 = 1$

$[6,9], max-min = 9-6 = 3$

$[7,9], max-min = 9-7 = 2$

$[3,6,7], max-min = 7-3 = 4$

$[3,6,9], max-min = 9-3 = 6$

$[3,7,9], max-min = 9-3 = 6$

$[6,7,9], max-min = 9-6 = 3$

$[3,6,7,9], max-min = 9-3 = 6$

2. 计算上一步得到的所有极差的和：

$3+4+6+1+3+2+4+6+6+3+6 = 44$

3. 将上一步得到的结果对 1000000007 取模：

$44\%1000000007 = 44$

WeiYong 比赛时“按子集”的计数算法：

对于非降序排列的一组数字$a_1 \le a_2 \le \cdots \le a_n$。共有$2^n - 1$个非空子集，每个非空子集的极差为子集中下标最大的元素 减去 子集中下标最小的元素。那么对于一对下标为i和$j(i<j)$的元素$(a_j - a_i)$，是多少个子集的极差呢？由下标 k 满足$i<k<j$的元素在自己中是否出现共有2^{j-i-1}个以$(a_j - a_i)$为极差的子集。故所有子集的极差为$\sum_{1\le i<j\le n}(a_j - a_i) \times 2^{j-i-1}$。

在线性时间预先生成以2为底，指数在0到N的幂值。

以上算法的时间复杂度为$T\sim O(TN^2)$。

完整代码：

```
#include <cstdio>

const int MOD = 1000000007;
const int N = 10005;

int power[N];
void setPower() {
	power[0] = 1;
	for (int i = 1; i <= N; i++)
		power[i] = (power[i - 1] << 1) % MOD;
}

int nums[N];
int n;
void read() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++)
		scanf("%d", &nums[i]);
}

long long solve() {
	long long ans = 0;
	for (int i = 0; i < n; i++)
		for (int j = i + 1; j < n; j++)
			ans += 1ll * (nums[j] - nums[i]) * power[j - i - 1], ans %= MOD;
	return ans;
}

int main() {
```

```
    freopen("ain.txt", "r", stdin);
    freopen("aout.txt", "w", stdout);
    setPower();
    int T;

    scanf("%d", &T);
    for (int t = 0; t < T; t++) {
        read();
        long long ans = solve();
        printf("Case #%d: %d\n", t + 1, (int)ans);
    }
}
```

注：上述代码在比赛中未考虑标红处两个大 int 型变量相乘溢出的问题，小数据集通过，大数据集计算结果全部溢出。

WeiYong1024 赛后“按集合成员”的计数算法:

上述算法的复杂度为$T\sim O(TN^2)$，10^9量级的乘加运算在 C02TR2TEHTD7 大数据集通过需要 1-2s。此题一定有更快的算法。

对于每一个测试样例，以上算法的复杂度为$T\sim O(N^2)$。具体分析，蛮力的时间复杂度正比于非空子集数量2^N-1，上述算法将区间最值成对考虑，以每一对数字作为最值的所有子集在常数时间内得到，故时间复杂度正比于“最大／最小对”的数量$\frac{(n-1)(n-2)}{2}$。

进一步，若我们能在常熟时间内把每个元素对编码的贡献直接得到，那么每个测试样例的时间复杂度将降低至$T\sim O(N)$。怎样计算每个元素对最终结果的贡献呢？注意到，对于元素$a_i(0\le i\le n-1)$，以其作为最小值的非空子集（包括$\{a_i\}$）数量为2^{N-i-1}（指数为i后面的元素数）。同样地，以a_i为最大值的非空子集数量为2^i（指数为i前面的元素数）。于是a_i对最终结果的贡献为(2^i-2^{N-i-1})。

上述算法总复杂度$T\sim O(TN)$。

完整代码:

```
#include <cstdio>

const int MOD = 1000000007;
const int N = 10005;

int power[N];
void setPower() {
    power[0] = 1;
    for (int i = 1; i <= N; i++)
        power[i] = (power[i - 1] << 1) % MOD;
}

int nums[N];
int n;
void read() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
```

```
        scanf("%d", &nums[i]);
}

long long solve() {
    long long ans = 0;
    for (int i = 0; i < n; i++) {
        ans += 1ll * nums[i] * power[i];
        ans -= 1ll * nums[i] * power[n - i - 1];
        ans %= MOD;
    }
    return ans;
}

int main() {
    freopen("ain.txt", "r", stdin);
    freopen("aout.txt", "w", stdout);
    setPower();
    int T;

    scanf("%d", &T);
    for (int t = 0; t < T; t++) {
        read();
        long long ans = solve();
        printf("Case #%d: %d\n", t + 1, (int)ans);
    }
}
```

3.2 Kickstart 2017 Round B

Problem B. Center (三分搜索，平面几何)

题述：

平面上有N个带权点，第i个点的坐标为(X_i, Y_i)，权值为W_i。在平面上找到一个中心点(X, Y)，使该中心到所有点的带权距离和最小。中心到第i个点的带权距离定义为$d_i = \max(|X - X_i|, |Y - Y_i|) * W_i$

输入：

第一行包含一个整数 T 为测试样例的数量，接着是 T 个测试样例。

每个测试样例开始一行为一个数字N。接下来的N行每行包含三个由空格分隔的实数X_i, Y_i, W_i。所有数字精确到十进制小数点后两位。

输出：

对于每一个测试样例，输出一行包含$Case\ \#x: y$，其中x是从 1 开始的样例序号，y是所有点(X, Y)到中心的带权距离和$\sum_{i=1}^{N} \max(|X - X_i|, |Y - Y_i|) * W_i$。

当y与标准答案的绝对或相对误差不超过10^{-6}时被视作正确。

参数限制：

$1 \le T \le 10$

$-1000.00 \le X_i \le 1000.00$

$-1000.00 \le Y_i \le 1000.00$

小数据集：

$1 \le N \le 100$.

$W_i = 1, 0$, for all i.

大数据集：

$1 \le N \le 10000$.

$1 \le W_i \le 1000.00$, for all i.

测试样例：

Input	Output
3 2 0.00 0.00 1.00 1.00 0.00 1.00 4 1.00 1.00 1.00 1.00 -1.00 1.00 -1.00 1.00 1.00 -1.00 -1.00 1.00 2 0.00 0.00 1.00 1.00 0.00 2.00	Case #1: 1.0 Case #2: 4.0 Case #3: 1.0

pps789 的二维三分搜索算法:

原问题中的函数为凸函数，可以用**二维三分搜索**找到函数在区域内的最值点。

完整代码:

```
#include <algorithm>
#include <cmath>
#include <cstdio>
```

```
const int N = 10005;

double X[N], Y[N], W[N];
int n;
void read() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++)
		scanf("%lf %lf %lf", &X[i], &Y[i], &W[i]);
}

double go(double x, double y){
	double ret = 0;
	for (int i = 0; i < n; i++)
		ret += std::max(std::abs(X[i] - x), std::abs(Y[i] - y)) * W[i];
	return ret;
}

double ternary_search(double x){
	double lo = -1000, hi = 1000;
	for (int i = 0; i < 100; i++){
		double m1 = (2 * lo + hi) / 3;
		double m2 = (lo + 2 * hi) / 3;
		double L = go(x, m1), R = go(x, m2);
		if (L <= R) hi = m2;
		else lo = m1;
	}
	return go(x, lo);
}

double solve(){
	double lo = -1000, hi = 1000;
	for (int i = 0; i < 100; i++){
		double m1 = (2 * lo + hi) / 3;
		double m2 = (lo + 2 * hi) / 3;
		double L = ternary_search(m1), R = ternary_search(m2);
		if (L <= R) hi = m2;
		else lo = m1;
	}
	return ternary_search(lo);
}

int main(){
	freopen("bin.txt", "r", stdin);
	freopen("bout.txt", "w", stdout);
```

```
    int T; scanf("%d", &T);
    for (int tc = 1; tc <= T; tc++){
        read();
        printf("Case #%d: %.10f\n", tc, solve());
    }
}
```

注：本体代码中的**三分搜索**可作为三分搜索模版。

关于**三分搜索**和**二分搜索**：

三分搜索解决的是单峰函数（上／下凸函数）的极值定位（或求解）问题；**二分搜索**解决的是指定元素在有序数组中的搜索（或定位）问题。

关于本题中的**二维三分搜索**：

本题求解二元函数$f_{x,y}(x,y)$在区域内的最大值，采用两个维度上的三分搜索。首先求出给定x坐标时函数在y方向上的最大值$f_x(x_0)=\max_y f_{x,y}(x_0,y)$，然后求出$f_x(x)$的最大值即为二元函数的区域最值。能这样求解的前提是$f_{x,y}(x,y)$是y的单峰函数（或称准凸函数，包括凸函数），同时$f_x(x)$是x的单峰函数。对于只有一个极值(x_0,y_0)的二元函数，这个条件成立。姜予名证法：用一个平行于 z 轴的平面截曲面，得到一个一维凸函数，存在一个极大值，移动这个平面，随该平面经过二维函数最大值点平面上极大值的变化趋势为先增大后减小，当平面经过(x_0,y_0)时达到全局最大值。

关于原函数是凸函数的证明（韩圣千老师空时信号处理课）：

首先，无穷阶范数是凸函数，乘一个常数保凸，多个凸函数的正权和仍保凸。另外关于凸函数的定义：符合 Jensen’ s ineuqlity----$f\big(\theta x+(1-\theta y)\big)\le\theta f(x)+(1-\theta f(y))$。

3.2 Kickstart 2017 Round B

Problem C. Christmas tree (动态规划)

题述：

给定N行M列的矩形网格，每个网格被涂上白色或绿色中的一种颜色。你的任务是找到最大的“圣诞树形状”中的绿色元宝数目。

为定义“圣诞树形状”，首先定义“良三角”如下：

我们称顶点位于第R行第C列，高度为h的指向上的只包含绿色元胞的等腰三角形为“良三角”。严格来说，这意味着位于(R, C)的元胞是绿色的，而且对于任意的$i \in [0, h-1]$第$R+i$行的第$C-i$到$C+i$列的元胞也是绿色的。

例如：

```
..#..
.####
#####
```

是一个高度为3的良三角。‘#’网格代表绿色，‘.’网格代表白色。注意这里有一个绿色元胞不属于该良三角，尽管它和该良三角临接。

```
..#..
.###.
####.
```

不是一个高度为3的良三角，因为第三行第五列的元胞是白色的。然而网格中存在多个高度为2的良三角。

```
...#.
.###.
#####.
```

不是一个高度为3的良三角，然而存在多个高度为2的良三角。

K-圣诞树的定义如下：

• 垂直方向上包含K个良三角

• 第$i+1$个良三角的顶端元胞必须和第i个良三角的底边任意一个元胞共享相同的边。这意味着，如果第i个良三角的底边在第r行，第c_1到c_2列，那么第$i+1$个良三角的顶点必须位于第$r+1$行，第c_1到c_2之间的任意一列。

例如，对于$K=2$：

```
...#...
..###..
.#####.
#######
..#....
.###...
#####..
```

是一个合法的 2-圣诞树。注意两个良三角的高度可以不同。

```
..#..
.###.
.#...
```

也是一个合法的 2-圣诞树。注意其中一个良三角只有一层一个元胞。

```
...#...
..###..
.#####.
.......
..#....
```

.###...
#####..
这两个高度为3的良三角不能构成 2-圣诞树，因为第二个良三角的顶点必须在第四行。
...#.
..###
.#...
###..
这两个高度为2的良三角不能构成 2-圣诞树，因为第二个良三角的顶点列号必须在3到5之间。

你需要找到包含绿色元胞最多的 K-圣诞树。

输入：

第一行为一个整数T为测试样例数量。接下来是T个测试样例。每个测试样例包含若干行：

- 第一行为3个由空格分隔的整数N, M, K——分别为网格的行数、列数和目标圣诞树的层数。
- 接下来的N行每行包含M个字符。每个字符或为'#'或为'.'——分别代表绿色和白色的元胞。

输出：

对于每个测试样例，输出一行包含$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是最大 K-圣诞树里包含的绿色元胞的数量。如果不存在 K-圣诞树，输出0。

参数限制：

$1 \leq T \leq 100$.

$1 \leq M \leq 100$.

$1 \leq N \leq 100$.

网格中的每个元胞比为'#'和'.'之一。

小数据集：

$K = 1$

大数据集：

$1 \leq K \leq 100$

测试样例：

Input	Output
4 3 5 1 ..#.. .###. ##### 3 5 1 .... 4 5 1 ##### ##### ##### ##### 4 5 2 ##### ##### #####	Case #1: 9 Case #2: 0 Case #3: 9 Case #4: 10

#####	

测试样例 1 中，最大的 1-圣诞树有 9 个绿色元胞：

..#..

.###.

#####

测试样例 2 中，不存在 1-圣诞树。

测试样例 3 中，最大的 1-圣诞树之一有 9 个绿色元胞：

#####

#####

#####

#####

测试样例 4 中，最大的 2-圣诞树之一有 10 个绿色元胞：

#####

#####

#####

#####

WeiYong 的动态规划解:

状态$dp[k][x][y]$存储以(x,y)为顶点的最大 k-圣诞树的面积，状态转移：$dp[k][x][y]$的值为所有以(x,y)为顶点的良三角下面一行的$dp[k-1][x][y]$的最大值 加上 当前良三角的面积。注意到$dp[k][..][..]$的值只与$dp[k-1][..][..]$有关，只需存储前两列。$S\sim O(MN)$

完整代码:

```
#include <algorithm>
#include <cstring>
#include <cstdio>

const int N = 105, M = 105;

char grids[N][M];
int n, m, k;
void read() {
	scanf("%d %d %d", &n, &m, &k);
	memset(grids, 0, sizeof(grids));
	for (int i = 0; i < n; i++)
		scanf("%s", grids[i]);
}

bool check(int x, int y1, int y2) { // Check if grid[x][y1...y2] are all '#'.
	for (int y = y1; y <= y2; y++)
		if (grids[x][y] != '#') return false;
	return true;
}

int hight_of_GT(int x, int y) {	// Return the height of largest triangle with top at (x, y).
	if (grids[x][y] != '#') return 0;
```

```
    int h = 1;
    while (x + h < n && y - h >= 0 && y + h < m && check(x + h, y - h, y + h))
        h++;
    return h;
}

int dp[2][N][M];
int height[N][M];
int solve() {    // Return area of the largest k-christmas tree.
    memset(height, 0, sizeof(height));
    for (int x = 0; x < n; x++)
        for (int y = 0; y < m; y++)
            height[x][y] = hight_of_GT(x, y);
    memset(dp, 0, sizeof(dp));
    for (int x = 0; x < n; x++)
        for (int y = 0; y < m; y++)
            dp[1][x][y] = height[x][y] * height[x][y];

    for (int kk = 2; kk <= k; kk++) {
        int cur = kk & 1, prev = !cur;
        for (int i = 0; i < n; i++) // update value of area of kk-christmas tree with top (i, j).
            for (int j = 0; j < m; j++)
                if (height[i][j] > 0) {
                    for (int ii = i + 1; ii - i <= height[i][j] && ii < n; ii++)      // (ii, jj) is the top of good triangle
below.
                        for (int jj = j - (ii - i - 1); jj <= j + (ii - i - 1); jj++)
                            if(dp[prev][ii][jj] > 0) dp[cur][i][j] = std::max(dp[cur][i][j], (ii - i) * (ii - i) +
dp[prev][ii][jj]);
                }

        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                dp[prev][i][j] = 0;
    }

    int ans = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            ans = std::max(ans, dp[k & 1][i][j]);

    return ans;
}

int main() {
    freopen("cin.txt", "r", stdin);
```

```
    freopen("cout.txt", "w", stdout);

    int T;
    scanf("%d", &T);
    for (int i = 0; i < T; i++) {
        read();
        int ans = solve();
        printf("Case #%d: %d\n", i + 1, ans);
    }
}
```

上述代码运行大数据集时间：62.176473s

2017.7.13 内容更新时第一次提交时漏写代码中**标红**的判断。

3.3 Kickstart 2017 Round C

Problem A. Ambiguous Cipher (纯数学)

题述：

Susie 和 Calvin 同班，Calvin 想给 Susie 传消息并让老师和同学们不知道其内容，以防小纸条传到不该传到的地方。Calvin 为此设计了一套加密系统。

Calvin 每次给 Susie 传递一个由大写字母构成的单词（他太渴望和 Susie 说话了）。每个单词按如下方式加密：

- 每个字母由其在字母表中的位置表示，例如 A=0, B=1, …, Z=25。
- 每一个字母的被加密成其相邻字母对应代码的和模 26 所对应的字母。
- 原文中的每个字母都用这种方式加密，且仅使用原文中相邻字母的和加密。

以 Calvin 传给 Susie 的一个纸条为例，Calvin 总是很饿，他想让 Susie 知道他又想吃饭了。于是他将 SOUP 加密的过程如下：

- S=18, O=14, U=20, P=15。
- 每个字符根据其左右的字符加密：

  第一个字符：14%26=14

  第二个字符：(18+20)%26=12

  第三个字符：(14+15)%26=3

  第四个字符：20%26=20
- 所得数字在字母表中对应的单词为 OMDU，即（写在纸条上的）密文。

首先保证密闻必可译码（例如 APE 就不可译码）。

然而，Calvin 的加密系统并不完善，其传递给 Susie 的某些密文存在多种译码方式，从而产生歧义！例如 BCB 可以被译码成 ABC、CBA 等

Susie 今晚要刷夜做课程项目，她没时间译码了，你帮帮她好吗？

输入：

第一行为测试样例数量T。接着是T个测试样例。每个测试样例为一个单行由大写字母构成的字符串 W。

输出：

对于每个测试样例，输出一行$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是译码结果或为 AMBIGUOUS（当存在译码歧义）。

参数限制：

$1 \leq T \leq 100$

W 只包含大写字母

W 存在译码结果（可能有歧义）

W 本身不会被译码成 AMBIGUOUS（仅当存在歧义时输出该词）

小数据集：

$2 \leq |W| \leq 4$

大数据集：

$2 \leq |W| \leq 50$

测试样例：

Input	Output
3 OMDU BCB AOAAAN	Case #1: SOUP Case #2: AMBIGUOUS Case #3: BANANA

注：最后一个测试样例不会出现在小数据集中。

Case #1 和 Case #2 如题干中所描述。

Case #3 中，BANANA 是唯一可加密成 AOAAAN 的单词。

WeiYong1024 的算法:

首先考虑n为偶数的情况，设$a[],b[]$分别为密文和原文字符对应字母表的位置，原文的偶数位$b[1,3,5,\dots,n-1]$可以从左向右唯一确定，奇数位从右向左唯一确定，具体地（在草纸上模拟$n=6$的情况）:

```
b[1] = a[0]                                  b[n - 2] = a[n - 1]
b[3] = (a[2] - b[1] + 26) % 26               b[n - 4] = (a[n - 3] - b[n - 2] + 26) % 26
...                                          ...
b[n - 1] = (a[n - 2] - b[n - 3] + 26) % 26   b[0] = (a[1] - b[2] + 26) % 26
```

然后是 n 为奇数的情况，在草纸上模拟$n=5$的情况不难得到，此时上述算法只能得到奇数位上的原文（在密文保证可译的情况下），而偶数位上的原文无法确认，故n为奇数时必存在歧义。

完整代码:

```
#include <string>
#include <iostream>

const int N = 50;

std::string str;
void read() {
    std::cin >> str;
}

int a[N], b[N];
std::string solve() {
    int n = (int)str.size();
    if (n & 1) return "AMBIGUOUS";

    for (int i = 0; i < n; i++)
        a[i] = str[i] - 'A';
    b[1] = a[0];
    for (int i = 3; i <= n - 1; i += 2)
        b[i] = (a[i - 1] - b[i - 2] + 26) % 26;
    b[n - 2] = a[n - 1];
    for (int i = n - 4; i >= 0; i -= 2)
        b[i] = (a[i + 1] - b[i + 2] + 26) % 26;

    std::string ans;
    for (int i = 0; i < n; i++)
        ans += b[i] + 'A';
    return ans;
}

int main() {
    freopen("ain.txt", "r", stdin);
    freopen("aout.txt", "w", stdout);
    std::ios::sync_with_stdio(false);
```

```
    int t;
    std::cin >> t;
    for (int i = 0; i < t; i++) {
        read();
        std::cout << "Case #" << i + 1 << ": " << solve() << std::endl;
    }
}
```

3.3 Kickstart 2017 Round C

Problem B. X Squared

题述：

“X Square”是今年最流行的玩具，类似于一款平面魔方。它形如一个$N \times N$的棋盘网格，其中$2N-1$个单元的贴纸标记“X”，其余单元的贴纸空白（题目中用“.”表示）。每次操作可以交换任意两行或两列的位置，目标是将标记“X”的单元位于网格的两个主对角线上，从而构成一个更大的“X”，下图给出了$N=5$的情形：

```
X...X
.X.X.
..X..
.X.X.
X...X
```

你的任务是解决你手中处于任意态的“X Square”，不过你淘气的妹妹可能移动了棋盘上的某些贴纸从而使玩具永远到不了目标态。那么如果给定初始态，能不能判定目标态的可达性呢？

输入：

第一行为测试样例数量T。接着是T个测试样例，每一个开始为一个整数N——棋盘宽度，接下来的N行每行为一个长度为N的字符串。第i行的第j格字符表示棋盘上第i行第j列的字符，其值为“X”或“.”。

输出：

对于每个测试样例，输出一行$Case\ \#x: y$，其中x是从1开始的样例序号，y为“IMPOSSIBLE”或 “POSSIBLE”取决于当前状态是否可达目标态。

参数限制：

$1 \le T \le 100$

$N \equiv 1 (\mathrm{mod} 2)$

棋盘上的网格恰好$2N-1$个为“X”，N^2-2N+1个为“.”

初始态不是目标态

小数据集：

$3 \le N \le 5$

大数据集：

$3 \le N \le 55$

测试样例：

Input	Output
2 3 ..X XX. XX. 3 ... XXX XX.	Case #1: POSSIBLE Case #2: IMPOSSIBLE

Case #1 中，可以通过以下两步达到目标态：

1. 交换 1、2 行。
2. 交换 2、3 列。

```
..X      XX.      X.X
XX.  ->  .X.  ->  .X.
XX.      XX.      X.X
```

Case #2 中，目标态不可达。

WeiYong1024 的分析:

交换行列位置使得棋盘变化的规律——每行／列中的成员组成不会发生变化。具体地，交换某两行使得每行中某两个元素的相对位置发生改变，但该行仍由原来的N个成员组成；每列的位置可能变化，但其成员组成和顺序都不发生改变。交换某两列的情况类似。

由于“X Square”的操作是可逆的，故能通过有限次操作到达目标态的状态也可以通过目标态经过有限次操作得到。由于目标态中有$N-1$个行／列有且仅有两个“X”，1个行／列有且仅有一个“X”，故目标态可以达到的状态也该有这个特点。那么只需检查题目中给出的初始态是否满足上述特点，

· 赛时 REJECTED 的算法：检查有$N-1$个行／列有且仅有两个“X”，1个行／列有且仅有一个“X”

· 赛后 CORRECT 的算法：检查每行只能有1或2个 X，若数量为2，则其对应的同列的 X 也得位于同一行。（即构成一个矩形）

注意，我们总可以通过最多各两次行、列交换将一个“矩形”移至整个区域的最外侧从而不再考虑它，作$\frac{n-1}{2}$次这样的操作之后剩下的一个“X”自然位于区域中央，也就构成了目标态。

以上操作的前提是在初始态$2n-1$个“X”要构成$\frac{n-1}{2}$个不共边的“矩形”和一个孤立“X”，这也就是目标态可达的判定条件了。

完整代码:

```
#include <vector>
#include <cstring>
#include <cstdio>

const int N = 55;

char grids[N][N];

int n;
void read() {
    memset(grids, 0, sizeof(grids));
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%s", grids[i]);
}

std::pair<int, int> find2XinRow(int u) {      // Find exactly two 'X' in row u.
    std::pair<int, int> ans;
    int counter = 0;
    for (int i = 0; i < n; i++) {
        if (grids[u][i] == 'X') {
            counter++;
            if (counter > 2) return {-1, -1};
            if (counter == 1) ans.first = i;
            else ans.second = i;
        }
```

```
        }
        if (counter == 0) return {-1, -1};
        if (counter == 1) ans.second = ans.first;
        return ans;
}

int findAnotherXinCol(int u, int v) {
        int ans = -1;
        for (int i = 0; i < n; i++) {
                if (i == u) continue;
                if (grids[i][v] == 'X') {
                        if (ans == -1) ans = i;
                        else return -1; // More than two 'X's in col v.
                }
        }
        return ans;
}

bool solve() {
        for (int i = 0; i < n; i++) {
                int p1, p2, q1, q2;
                std::tie(p1, p2) = find2XinRow(i);
                if (p1 == -1) return false;
                if (p1 == p2) continue;
                q1 = findAnotherXinCol(i, p1);
                q2 = findAnotherXinCol(i, p2);
                if (q1 == -1 || q2 == -1 || q1 != q2) return false;
        }
        return true;
}

int main() {
        freopen("bin.txt", "r", stdin);
        freopen("bout.txt", "w", stdout);
        int t;
        scanf("%d", &t);
        for (int i = 0; i < t; i++) {
                read();
                printf("Case #%d: %s\n", i + 1, solve() ? "POSSIBLE" : "IMPOSSIBLE");
        }
}
```

3.3 Kickstart 2017 Round C

Problem C. Magical Thinking (纯数学)

题述：

你和你的N个朋友刚刚参加了魔法学校的入学 B.A.T(Binary Answer Test)测试。试卷上有Q道题，每道题1分，你还没有魔法，所以你只能蒙答案并寄希望于蒙对尽可能多的题。

考试结果已经由信鸽寄出。尽管你的信鸽还没有到，但你的朋友们已经告诉了你他们的答案和分数。你记录了自己的答案，作为乐观主义者你希望自己表现良好。

你知道Q道判断题都有唯一的标准答案，在知道N个朋友的答案和得分的情况下，你可能获得的最大分数是多少？

输入：

第一行为测试样例数量T。接着是T个测试样例，每个测试样例开始一行为两个整数N和Q，接着的有$N+1$行，第i行为第i个朋友的答案A_i，长度为Q，每个字符为T或F（True 和 False 的缩写）。A_{i+1}是你的答案。最后一行为N个整数，其中第i个整数S_i为第i个朋友的得分。（注意你的成绩不在上面）

输出：

对于每一个测试样例，输出一行$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是你可能的最高得分。

参数限制：

$1 \le T \le 100$

$|A_i| = Q,\ \ i = 1, 2, \dots, N$

A_i中的每个字符为 T 或 F

$0 \le S_i \le Q$

保证标准答案存在性

小数据集：

$N = 1$

$1 \le Q \le 10$

大数据集：

$1 \le N \le 2$

$1 \le Q \le 50$

测试样例：

Input	Output
3 1 2 TF FF 1 1 3 TTT TTF 0 2 3 TTF FTF TTT 1 2	Case #1: 2 Case #2: 1 Case #3: 2

注：最后一个测试样例不会出现在小数据集中。

Case #1 中，朋友的答案是 TF 你的答案是 FF，朋友的两道题中有一道是正确的，如果第二道正确，那么标准答案是 FF，你的两道题都正确，得2分，不会有更好的情况了。
Case #2 中，朋友的答案全部错误，标准答案为 FFF，你得1分，不存在其他情况。
Case #3 中，标准答案只可能为 FTT 和 FFF，对于这两种情况，FTT 会让你得到最高的2分。

WeiYong1024 的数学分析及$T\sim O(1)$解:

可供参照的朋友数量为1和2，故直观想应该存在各自$T\sim O(1)$的解。

首先分析以1个朋友 A 的答案和得分作为参考的情况。凝练问题：**一共Q道题，A 答对了K道，我和 A 有H道答案相同，问我可能答对问题的最大数量**。以让我和 A 答案相同的题尽量为A答对的题为原则，分类如下:

· 若$H \le K$，那么最好的情况是我和 A 答案相同的题都是 A 答对的，则 A 答错的题我也都答对了，即我的最高得分为$H+(Q-K)$。

· 若$H > K$，那么最好的情况是 A 答对的问题我都答对了，那么 A 答错的题中我有$(H-K)$道题和他一起错了，另外$(Q-K)$道题我是对的，即我的最高得分为$K+(Q-H)$。

综上，以1个朋友 A 作为参考我的最高得分为$Q-|K-H|$。

下面考虑以2个朋友 A 和 B 的答案和得分作为参考的情形。凝练问题：**Q道题中，我知道 A 的Q个答案并且其中有K_A道正确，知道 B 的答案并且其中有K_B道正确，问我的答案可能的最大正确数量**。不妨设$K_A \le K_B$，则 B 比 A 多答对了(K_B-K_A)道题，这些题必位于 A 和 B 答案不同的那些题中。用L表示 A 和 B 答案不同的题的数量，这其中 B 比 A 多答对(K_B-K_A)题，故在这L道 A、B 答案不同的题中 A 答对了$\frac{L-(K_B-K_A)}{2}$，B 答对了$\frac{L+(K_B-K_A)}{2}$道题；在 A 和 B 答案相同的$(Q-L)$道题中，A 答对了$K_A-\frac{L-(K_B-K_A)}{2}$道，于是可将原问题分上述两种情况分别考虑求解:

· 在 A 和 B 答案不同的 L 道题中，A 答对了$\frac{L-(K_B-K_A)}{2}$道，我和 A 有H_1道的答案相同，故这部分中我最多可能答对$L-|\frac{L-(K_B-K_A)}{2}-H_1|$道。

· 在 A 和 B 答案相同的$(Q-L)$道题中，A 答对了$K_A-\frac{L-(K_B-K_A)}{2}$道，我和 A 有$H_2$道的答案相同，故这部分中我最多可能答对$(Q-L)-\left|\left[K_A-\frac{L-(K_B-K_A)}{2}\right]-H_2\right|$道。

综上，我的最高得分为$L-\left|\frac{L-(K_B-K_A)}{2}-H_1\right|+(Q-L)-\left|\left[K_A-\frac{L-(K_B-K_A)}{2}\right]-H_2\right|$。

根据表达式统计出L, K_A, K_B, Q, H_1, H_2的值即可。复杂度$T\sim O(NQ)$

完整代码:

```
#include <iostream>

int n, q;
std::string A, B, me;
int k, ka, kb;
void read() {
    std::cin >> n >> q;
    if (n == 1) {
        std::cin >> A >> me;
        std::cin >> k;
```

```
    }
    else {
        std::cin >> A >> B >> me;
        std::cin >> ka >> kb;
    }
}

int solve() {
    if (n == 1) {
        int h = 0;
        for (int i = 0; i < q; i++)
            if (A[i] == me[i]) h++;
        return q - abs(k - h);
    } else {
        if (ka > kb) {
            std::swap(ka, kb);
            std::swap(A, B);
        }
        int l = 0, h1 = 0, h2 = 0;
        for (int i = 0; i < q; i++) {
            if (A[i] != B[i]) l++;
            if (A[i] != B[i] && A[i] == me[i]) h1++;
            if (A[i] == B[i] && A[i] == me[i]) h2++;
        }
        int ans1 = l - abs((l - (kb - ka)) / 2 - h1);
        int ans2 = (q - l) - abs(ka - (l - (kb - ka)) / 2 - h2);
        return ans1 + ans2;
    }
}

int main() {
    freopen("cin.txt", "r", stdin);
    freopen("cout.txt", "w", stdout);
    std::ios::sync_with_stdio(false);
    int t;
    std::cin >> t;
    for (int i = 0; i < t; i++) {
        read();
        printf("Case #%d: %d\n", i + 1, solve());
    }
}
```

3.3 Kickstart 2017 Round C

Problem D. The 4M Corporation (纯数学，奇偶分析)

题述：

4M 公司聘请你来管理部门并分配人力。你至少要成立一个部门，并为每个部门分配正数个员工。这个工作可不简单，你的四个上级分别提出了各自的要求：

1. 人数最少的部门有 **MINIMUM** 个员工。
2. 人数最多的部门有 **MAXIMUM** 个员工。
3. 所有部门员工人数的平均数必须为 **MEAN**。
4. 所有部门员工人数的中位数必须为 **MEDIAN**。（这里中位数定义：当所有部门按照人数非降序排列时，位于中间的部门人数或者位于中间的两个部门人数的平均数）

还有，为了便于管理，成立的部门数越少越好。那么在满足四个上级各自要求的情况下，最少许要成立几个部门呢？

输入：

第一行为测试样例数量T。紧接着是T个测试样例，每一个包含四个整数，依次为 **MINIMUM, MAXIMUM, MEAN, MEDIAN**。

输出：

对于每个测试样例，输出一行$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是最少需要的部门数量，或者不存在解时为 IMPOSSIBLE。

参数限制：

$1 \leq T \leq 100$

小数据集：

$1 \leq \mathbf{MINIMUM} \leq 8$

$1 \leq \mathbf{MAXIMUM} \leq 8$

$1 \leq \mathbf{MEAN} \leq 8$

$1 \leq \mathbf{MEDIAN} \leq 8$

小数据集的结果保证不超过14，或者为 IMPOSSIBLE。

大数据集：

$1 \leq \mathbf{MINIMUM} \leq 10000$

$1 \leq \mathbf{MAXIMUM} \leq 10000$

$1 \leq \mathbf{MEAN} \leq 10000$

$1 \leq \mathbf{MEDIAN} \leq 10000$

测试样例：

Input	Output
5 6 4 5 1 7 7 8 8 2 2 2 2 3 7 5 5 1 4 3 4	Case #1: IMPOSSIBLE Case #2: IMPOSSIBLE Case #3: 1 Case #4: 2 Case #5: 3

Case #1 中，MINIMUM 的值比 MAXIMUM 大故不可能。

Case #2 中，MEDIAN 的值比 MAXIMUM 大故不可能。

Case #3 中，1个员工数为2的部门即可满足。

Case #4 中，2个员工数分别为3和7的部门即可满足。

Case #5 中，3个员工数分别为$1, 4, 4$的部门即可满足。

参考 Analysis / 吕甘霖的$T\sim O(1)$解:

首先，若四个统计量之间本身存在大小上的矛盾，即若以下五个条件：$\text{MIN} \le \text{MAX}$; $\text{MIN} \le \text{MEAN}$; $\text{MIN} \le \text{MED}$; $\text{MEAN} \le \text{MAX}$; $\text{MED} \le \text{MAX}$ 其中之一不被满足，则无解。

若四个统计量之间本身不存在大小冲突，那么以下两种情况的答案很显然:

· 若$\text{MIN} = \text{MAX}$，则结果为1。

· 若$\text{MIN} < \text{MAX}$且$\text{MIN} + \text{MAX} = 2\text{MEAN}$且$\text{MIN} = \text{MED}$，则结果为$2$。

若非以上两种情况，最终答案若存在则一定大于2。按照中位数的定义方式分类，首先考虑总部门数量为奇数的情况。直接可以确定的是存在三个人数分别为$\text{MIN}, \text{MED}, \text{MAX}$的部门。不失一般性地，假设三者的均值$\frac{1}{3}(\text{MIN} + \text{MED} + \text{MAX}) < \text{MEAN}$（为了避免浮点误差实现中用$\text{MIN} + \text{MED} + \text{MAX}$和$3 \times \text{MEAN}$比较），令$D = 3\text{MEAN} - (\text{MIN} + \text{MED} + \text{MAX})$。接下来每次添加2个人数分别为MED和MAX的部门，直到所有部门的人数总和**追上**$\text{MEAN} \times$当前部门数。为此做如下讨论，令$d = \text{MED} + \text{MAX} - 2\text{MEAN}$，于是

· 若$d \le 0$，则所有部门人数总和永远无法**追上**$\text{MEAN} \times$当前部门数，无解。

· 否则（$d > 0$），只需添加$\left\lceil \frac{D}{d} \right\rceil$次，相应的答案为$3 + 2\left\lceil \frac{D}{d} \right\rceil$。具体地，前$\left\lceil \frac{D}{d} \right\rceil - 1$次每次添加一对人数分别为MED和MAX的部门，最后一次添加一对人数和为$2\text{MEAN} + D\%d$的部门（**总能找到**，尾附证明）。

这样得到的结果是奇数个部门情况下最少的部门数量，用ans_odd表示。

利用类似的方法求出总共偶数个部门情况下最少的部门数量，用ans_even表示。这里值得注意的是，总共偶数个部门的情况下可确定的初始态是存在四个人数分别为$\text{MIN}, \text{MED}, \text{MED}, \text{MAX}$的部门。为什么中间的两个数不是$\text{MED} - 1$和$\text{MED} + 1$呢？因为这样只会减少可供我们选择添加新部门对儿的人数范围，从而延缓追赶目标均值的进程。

最终结果为$ans = \min(anx_odd, ans_even)$。$T\sim O(1)$

完整代码:

```
#include <algorithm>
#include <cstdio>

const int INF = 1111111111;

int MIN, MAX, MEAN, MED;
void read() {
    scanf("%d %d %d %d", &MIN, &MAX, &MEAN, &MED);
}

int solve_odd() {
    if (MIN + MED + MAX == 3 * MEAN) return 3;

    if (MIN + MED + MAX < 3 * MEAN) {
        int D = 3 * MEAN - (MIN + MED + MAX);
        int d = (MED + MAX) - 2 * MEAN;
        // IMPOSSIBLE.
        if (d <= 0) return INF;
        // Add upp_bound(D / d) times ans finally there will be 3 + 2 * upp_bound(D / d) departments.
        if (D % d == 0) return 3 + 2 * (D / d);
        else return 3 + 2 * (D / d + 1);
```

```
    } else {      // MIN + MED + MAX > 3 * MEAN
        int D = (MIN + MED + MAX) - 3 * MEAN;
        int d = 2 * MEAN - (MIN + MED);
        if (d <= 0) return INF;
        if (D % d == 0) return 3 + 2 * (D / d);
        else return 3 + 2 * (D / d + 1);
    }
}

int solve_even() {
    if (MIN + MED + MED + MAX == 4 * MEAN) return 4;

    if (MIN + MED + MED + MAX < 4 * MEAN) {
        int D = 4 * MEAN - (MIN + MED + MED + MAX);
        int d = (MED + MAX) - 2 * MEAN;
        if (d <= 0) return INF;
        if (D % d == 0) return 4 + 2 * (D / d);
        else return 4 + 2 * (D / d + 1);
    } else {      // MIN + MED + MED + MAX > 4 * MEAN
        int D = (MIN + MED + MED + MAX) - 4 * MEAN;
        int d = 2 * MEAN - (MIN + MED);
        if (d <= 0) return INF;
        if (D % d == 0) return 4 + 2 * (D / d);
        else return 4 + 2 * (D / d + 1);
    }
}

int solve() {
    if (!(MIN <= MAX && MIN <= MEAN && MIN <= MED && MEAN <= MAX && MED <= MAX)) return
INF;
    if (MIN == MAX) return 1;
    if (MIN < MAX && MED == MEAN && 2 * MEAN == MIN + MAX) return 2;
    int ans_odd = solve_odd(), ans_even = solve_even();
    return std::min(ans_odd, ans_even);
}

int main() {
    freopen("din.txt", "r", stdin);
    freopen("dout.txt", "w", stdout);
    int t;
    scanf("%d", &t);
    for (int i = 0; i < t; i++) {
        read();
        int ans = solve();
        if (ans == INF) printf("Case #%d: IMPOSSIBLE\n", i + 1);
```

```
            else printf("Case #%d: %d\n", i + 1, ans);
        }
}
```

“**总能找到**”的证明:

首先，在[MIN, MED]和[MED, MAX]内各选一个数，二者的和最小为(MIN + MED)，最大为(MED + MAX)，且可取到二者之间任意整数。

目标和为$(2 \times MEAN + D\%d) < 2 \times MEAN + d = (MIN + MED)$——上界。

又因为$(2 \times MEAN + D\%d) \geq 2MEAN > (MIN + MED)$——下界。

故目标和可达。

【证毕】

3.4 Kickstart 2017 Round D

Problem A. Go Sightseeing (动态规划，背包问题，二进制决策链)

题述：

一次旅行中，你希望游览尽可能多的城市，但这并不总能实现，因为你需要赶发往下一个城市的公交车。为了最大化旅行乐趣，你决定写一个程序来规划行程。

初始你在0时刻位于城市1，你计划顺次游览城市2到城市N。有从第i个城市到第$i+1$个城市公交车，第i路公交车的时刻表由3个整数S_i, F_i, D_i表征，分别为首车时间、发车间隔和单程时间。具体地，这意味着每到时刻$S_i + xF_i$就有一班从城市i发往城市$i+1$的车，单程行驶时间为D_i。

在编号为1到$N-1$的任意城市，你可以选择用T_s的时间游览该城市，然后等待发往下一个城市的汽车，或者直接等待发往下一个城市的汽车。每个城市只能游览一次，假设上车下车本身不花费时间。你必须在T_f时刻之前赶到城市N。（注意即便你提前到达城市N，你也不能游览该城市，因为这时这里还什么都没有！XD）

那么你能游览最多多少城市呢？

输入：

第一行一个整数T，为测试样例的数量。接下来是T个测试样例。

每个测试样例开始一行为3个整数N, T_s, T_f——分别表示城市数、游览所需要的时间和到达城市N的最晚时间。

接下来的$N-1$行每行包含3个整数S_i, F_i, D_i——分别表示从城市i发往城市$i+1$的公交车的首车时间、发车频率和单程时间。

输出：

对于每一个测试样例，输出一行包括$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是你在保证不晚于T_f赶到城市N的前提下可以游览的最大城市数量。如果无法在T_f之前赶到城市N，输出$Case\ \#x$: IMPOSSIBLE。

参数限制：

$1 \le T \le 100$.

小数据集：

$2 \le N \le 16$.

$1 \le S_i \le 5000$.

$1 \le F_i \le 5000$.

$1 \le D_i \le 5000$.

$1 \le T_s \le 5000$.

$1 \le T_f \le 5000$.

大数据集：

$2 \le N \le 2000$.

$1 \le S_i \le 10^9$.

$1 \le F_i \le 10^9$.

$1 \le D_i \le 10^9$.

$1 \le T_s \le 10^9$.

$1 \le T_f \le 10^9$.

测试样例：

Input	Output
4 4 3 12 3 2 1 6 2 2 1 3 2 3 2 30 1 2 27	Case #1: 2 Case #2: 0 Case #3: IMPOSSIBLE Case #4: 4

3 2 1 4 1 11 2 1 2 4 1 5 8 2 2 5 10 5000 14 27 31 27 11 44 30 8 20 2000 4000 3	

测试样例 1 中。你可以游览城市1，乘坐于时刻3发出的汽车于时刻4到达城市2。你可以继续游览城市2，乘坐于时刻8发出的汽车并于时刻10到达城市3，然后你直接乘坐于时刻10发出的汽车，恰好于时刻12到达城市4。

（适用于小数据集）WeiYong1024 基于二叉决策树的深搜+备忘+剪枝算法:

该问题要找到一个旅行策略，具体地就是确定N个城市中哪些城市游览、哪些城市不游览。搜索空间为一个二叉决策树的所有叶节点。根据是否游览当前层次所代表的城市进行深搜，采用适当的备忘和剪枝策略加速。

对于小数据集，决策树叶结点的规模为2^{15}，可以通过遍历所有叶节点得到。对于城市$i+1$，其到达时间只由在城市i开始等车的时间T_w决定。通过调用$arrive(i+1, T_w)$函数得到所有叶节点的到达时间。取所到达时间满足条件的叶结点中的游览城市数量的最大值即为所求。

完整代码:

```
#include <algorithm>
#include <iostream>

const int N = 2005;

int n, ts, tf;
int s[N], f[N], d[N];
void read() {
    scanf("%d %d %d", &n, &ts, &tf);
    for (int i = 1; i < n; i++)
        scanf("%d %d %d", &s[i], &f[i], &d[i]);
}

int arrive(int city_idx, int tw) {
    if ((std::max(tw, s[city_idx - 1]) - s[city_idx - 1]) % f[city_idx - 1] == 0) return std::max(tw, s[city_idx - 1]) +
d[city_idx - 1];
    else return std::max(tw, s[city_idx - 1]) + (f[city_idx - 1] - (std::max(tw, s[city_idx - 1]) - s[city_idx - 1]) %
f[city_idx - 1]) + d[city_idx - 1];
}

int ans;
void dfs(int city_idx, int arr_time, int city_vis) {
    if (city_idx == n) {
        ans = arr_time <= tf ? std::max(ans, city_vis) : ans;
        return;
```

```
    }
    dfs(city_idx + 1, arrive(city_idx + 1, arr_time), city_vis);
    dfs(city_idx + 1, arrive(city_idx + 1, arr_time + ts), city_vis + 1);
}

void solve() {
    ans = -1;
    dfs(1, 0, 0);
}

int main() {
    freopen("asin.txt", "r", stdin);
    freopen("asout.txt", "w", stdout);
    std::ios::sync_with_stdio(false);
    int T;
    scanf("%d", &T);
    for (int i = 0; i < T; i++) {
        read();
        solve();
        if (ans == -1) printf("Case #%d: IMPOSSIBLE\n", i + 1);
        else printf("Case #%d: %d\n", i + 1, ans);
    }
}
```

（适用于大数据集）的动态规划算法:

对于大数据集，城市规模达到2000，决策树的叶节点规模为2^{2000}，纯 dfs 遍历所有节点的 Brute Force 方法不可行。由于城市i的到达时间由在城市$i-1$开始等车的时间唯一确定，这启发我们设计一个动态规划算法。

根据原问题的形式，作决策链前缀上的动态规划，首先想到的子问题是求解并用$dp[i][j]$记录最晚于j时刻到达城市i的条件下可以游览的最大的城市数目。但时间的取值范围为10^9，以其划分子问题不现实。于是反过来（将子问题的返回值和索引对调），我们可以求解并用$dp[i][j]$存储在游览j个城市的条件下，到达城市i的最早时刻。利用小数据集中的$arrive()$函数，按照前一个城市是否游览，建立如下最优子结构／状态转移方程：$dp[i][j] = min(arrive(i, dp[i][j]), arrive(i, dp[i][j-1] + T_s))$。

该动态规划的:

· 平凡解：$dp[2][0] = arrive(2, 0), dp[2][1] = arrive(2, T_s)$。

· 边界状态转移：$dp[i][0] = arrive(i, dp[i-1][0]), dp[i][i-1] = arrive(i, dp[i-1][j-1] + T_s)$。

· 最终答案：满足$dp[n][j] <= T_f$的最大j。

· 复杂度$T/S \sim O(N^2)$

完整代码:

```
#include <algorithm>
#include <cstring>
#include <cstdio>

using ll = long long;

const int maxn = 2005;
```

```
int n, ts, tf;
int s[maxn], f[maxn], d[maxn];
void read() {
	scanf("%d %d %d", &n, &ts, &tf);
	for (int i = 1; i < n; i++)
		scanf("%d %d %d", &s[i], &f[i], &d[i]);
}

ll arrive(int i, ll tw) {
	ll set_off;
	if (tw <= s[i - 1]) set_off = s[i - 1];
	else if ((tw - s[i - 1]) % f[i - 1] == 0) set_off = tw;
	else set_off = s[i - 1] + ((tw - s[i - 1]) / f[i - 1] + 1) * f[i - 1];
	return set_off + d[i - 1];
}

ll dp[maxn][maxn];

int solve() {
	memset(dp, 0, sizeof(dp));
	dp[2][0] = arrive(2, 0), dp[2][1] = arrive(2, ts);
	for (int i = 3; i <= n; i++)
		dp[i][0] = arrive(i, dp[i - 1][0]);
	for (int i = 3; i <= n; i++)
		for (int j = 1; j < i; j++) {
			dp[i][j] = j == i - 1 ? arrive(i, dp[i - 1][j - 1] + ts) : std::min(arrive(i, dp[i - 1][j]), arrive(i, dp[i - 1][j - 1]
+ ts));
//				if (dp[i][j] > tf) break;
		}

	if (dp[n][0] > tf) return -1;
	else {
		int ans = 0;
		while (dp[n][ans + 1] <= tf && ans + 1 < n)
			ans++;
		return ans;
	}
}

int main() {
	freopen("alin.txt", "r", stdin);
	freopen("alout.txt", "w", stdout);
	int T;
	scanf("%d", &T);
```

```
    for (int t = 1; t <= T; t++) {
        read();
        int ans = solve();
        if (ans == -1) {
            printf("Case #%d: IMPOSSIBLE\n", t);
            fprintf(stderr, "Case #%d / %d: IMPOSSIBLE\n", t, T);
        } else {
            printf("Case #%d: %d\n", t, ans);
            fprintf(stderr, "Case #%d / %d: %d\n", t, T, ans);
        }
    }
}
```

注：上述代码前几次提交时由于存在如下三点细节问题：

· 少考虑一种边界状态转移——当$j = i - 1$时，$dp[i-1][j]$无意义，$dp[i][j]$的值仅由$dp[i-1][j-1]$决定。

· 在 solve()函数由备忘录生成结果的部分中，由于考虑不当出现了$ans = n$的情况。

· 代码中红色加粗的一行本义为剪枝加速，然而这样做会导致下面的普通状态转移出现问题。

总结：

编写程序时，在复杂度满足要求的情况下，严格按照填表顺序生成备忘录最保险，添加额外的 trick 虽有少量加速效果但由于破坏原算法整体布局维护成本／风险都太高。

背包问题／二维动态规划问题，自己在草纸上模拟出小数据情况下的填表顺序。一定要对**备忘录的生成顺序**和**最终答案的组成**有非常清晰的整体认知。例如本题中备忘录的填表顺序如下：

$dp[i][j]$	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
$i=0$							
$i=1$							
$i=2$	$arrive(2,0)$	$arrive(2,T_s)$					
$i=3$	$dp[3][0]$	$dp[3][1]$	$dp[3][2]$				
$i=4$	$dp[4][0]$	$dp[4][1]$	$dp[4][2]$	$dp[4][3]$			
$i=5$	$dp[5][0]$	$dp[5][1]$	$dp[5][2]$	$dp[5][3]$	$dp[5][4]$		
$i=6$	$dp[6][0]$	$dp[6][1]$	$dp[6][2]$	$dp[6][3]$	$dp[6][4]$	$dp[6][5]$	

1. 初始红色
2. 黄色边界状态转移
3. 黑色普通状态转移（考虑紫色边界状态转移）

3.4 Kickstart 2017 Round D

Problem B. Sherlock and The Matrix Game (代码量很大的二分搜索，容斥原理)

题述：

今天 Sherlock 和 Watson 上了一门矩阵论入门的课程。Sherlock 不是一个特别热衷于矩阵论的程序员，但他今天却给 Watson 出了一道关于矩阵论的问题。

Sherlock 给了 Watson 两个一维数组A和B，长度都是N。他首先要 Watson 生成一个N行N列的矩阵，其中位于第i行第j列的元素为A中第i个元素和B中第j个元素的乘积。

令(x, y)表示位于矩阵第x行（从最上面的一行开始从0编号）第y列（从最左面的一行开始从0编号）的元素。于是一个子矩阵分别由其左下和右上单元——(a, b)和(c, d)表征，其中$a \geq c, d \geq b$，子矩阵中由所有满足$c \leq i \leq a, b \leq j \leq d$的单元$(i, j)$构成。子矩阵的元素和定义为子矩阵中所有元素的代数和。

为了挑战 Watson，Sherlock 给了 Watson 一个整数K并要求 Watson 输出所有子矩阵元素和中第K大的值，其中排位值K按照降序从1开始计数。（可能存在多个K值对应同一个矩阵元素和的情况，也就是说可能有多个子矩阵元素和相同）。

你能帮助 Watson 吗？

输入：

第一行为测试样例数量T，接下来是T个测试样例。每个测试样例为一行包含九个整数$N, K, A_1, B_1, C, D, E_1, E_2$和$F$。其中$N$是数组$A, B$的长度，$K$是 Watson 所求子矩阵元素和的降序排位数，$A_1$和$B_1$分别是数组$A$和$B$的首元，剩下的五个数字用于生成数组$A, B$：

首先定义$x_1 = A_1, y_1 = B_1, r_1 = 0, s_1 = 0$，然后用下面的方式递归地生成$x_i$和$\mathrm{y_i}(2 \leq \mathrm{i} \leq \mathrm{N})$。

· $\mathrm{x_i} = (\mathrm{C} * \mathrm{x_{i-1}} + \mathrm{D} * \mathrm{y_{i-1}} + \mathrm{E_1})\%\mathrm{F}$

· $\mathrm{y_i} = (\mathrm{D} * \mathrm{x_{i-1}} + \mathrm{C} * \mathrm{y_{i-1}} + \mathrm{E_2})\%\mathrm{F}$

进一步，用下式生成r_i和$s_i(2 \leq \mathrm{i} \leq \mathrm{N})$

· $\mathrm{r_i} = (\mathrm{C} * \mathrm{r_{i-1}} + \mathrm{D} * \mathrm{s_{i-1}} + \mathrm{E_1})\%2$

· $\mathrm{s_i} = (\mathrm{D} * \mathrm{r_{i-1}} + \mathrm{C} * \mathrm{s_{i-1}} + \mathrm{E_2})\%2$

我们定义$\mathrm{A_i} = (-1)^{\mathrm{r_i}} * \mathrm{x_i}, \mathrm{B_i} = (-1)^{\mathrm{s_i}} * \mathrm{y_i}$，其中$2 \leq \mathrm{i} \leq \mathrm{N}$。

输出：

对于每个测试样例，输出一行Case #x: y，其中x是从1开始的测试样例序号，y是所有子矩阵元素和中第K大的数。

参数限制：

$1 \leq \mathrm{T} \leq 20$.

$1 \leq \mathrm{K} \leq \min(10^5, |\{\text{submatrices}\}|)$.

$0 \leq \mathrm{A_1} \leq 10^3$.

$0 \leq \mathrm{B_1} \leq 10^3$.

$0 \leq \mathrm{C} \leq 10^3$.

$0 \leq \mathrm{D} \leq 10^3$.

$0 \leq \mathrm{E_1} \leq 10^3$.

$0 \leq \mathrm{E_2} \leq 10^3$.

$0 \leq \mathrm{F} \leq 10^3$.

小数据集：

$1 \leq \mathrm{N} \leq 200$.

大数据集：

$1 \leq \mathrm{N} \leq 10^5$.

测试样例：

Input	Output
4	Case #1: 6

2 3 1 1 1 1 1 1 5 1 1 2 2 2 2 2 2 5 2 3 1 2 2 1 1 1 5 9 8 7 6 5 4 3 2 1	Case #2: 4 Case #3: 1 Case #4: 42

测试样例 1 中，用生成方法得到数组A, B分别为$[1, -3]$和$[1, -3]$，故矩阵为：

$$\begin{bmatrix} 1 & -3 \\ -3 & 9 \end{bmatrix}$$

所有子矩阵元素和的降序排列为$[9, 6, 6, 4, 1, -2, -2, -3, -3]$，由$K = 3$得到答案为6。

测试样例 2 中，用生成方法得到数组A, B分别为$[2]$和$[2]$，故矩阵为：

$[4]$

由$K = 1$得到答案为4。

测试样例 3 中，用生成方法得到数组A, B分别为$[1, 0]$和$[2, -1]$，故矩阵为：

$$\begin{bmatrix} 2 & -1 \\ 0 & 0 \end{bmatrix}$$

所有子矩阵元素和的降序排列为$[2, 2, 1, 1, 0, 0, 0, -1, -1]$，由$K = 3$得到答案为1。

作者给出利用容斥原理计算子矩阵元素和的小数据集算法:

对于小数据集，一种直观的想法是生成矩阵，然后迭代遍历所有子矩阵，并计算每个子矩阵的元素和。子矩阵的数量为$O(N^4)$，直接计算每个子矩阵的元素和对于 20 个测试样例的运算量为$T \sim O(TN^6) \approx 1.28 \times 10^{15}$。如果能在常数$T \sim O(1)$时间内计算每个子矩阵的元素和，则复杂度降为$T \sim O(TN^4) \approx 3.2 \times 10^{10}$，对于现代计算机可以在分钟级别内计算出结果。

利用容斥原理（Inclusion-Exclusion）在常数时间内得到子矩阵元素和:

为了在常数时间内得到矩阵M每个子矩阵的元素和，使用容斥原理。维护另一个矩阵P，其中$P(i, j)$记录以$M(0, 0), M(i, j)$为左上-右下元的子矩阵的元素和，则以$M(a_1, b_1) - M(a_2, b_2)$为左上-右下元的子矩阵的元素和可以表示为$sum = P(a_2, b_2) - P(a_1 - 1, b_2) - P(a_2, b_1 - 1) + P(a_1 - 1, b_1 - 1)$。而P本身可以递归地生成：$P(i, j) = P(i - 1, j) + P(i, j - 1) - P(i - 1, j - 1) + P(i, j)$。

注：按照小数据集的参数限制，子矩阵元素和的最大值可以达到$10^3 \times 10^3 \times 200 \times 200 = 4 \times 10^{10}$，故中间数据应该用 64 整型存储。

完整代码:

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

const int N = 205;

int n, k, a1, b1, c, d, e1, e2, f;
int A[N], B[N];
int x[N], y[N], r[N], s[N];
void read() {
    scanf("%d %d %d %d %d %d %d %d %d", &n, &k, &A[1], &B[1], &c, &d, &e1, &e2, &f);

    x[1] = A[1], y[1] = B[1], r[1] = s[1] = 0;
    for (int i = 2; i <= n; i++) {
        x[i] = (c * x[i - 1] + d * y[i - 1] + e1) % f;
        y[i] = (d * x[i - 1] + c * y[i - 1] + e2) % f;
```

```
            r[i] = (c * r[i - 1] + d * s[i - 1] + e1) % 2;
            s[i] = (d * r[i - 1] + c * s[i - 1] + e2) % 2;
        }
        for (int i = 2; i <= n; i++) {
            A[i] = (r[i] & 1) ? -x[i] : x[i];
            B[i] = (s[i] & 1) ? -y[i] : y[i];
        }
}

std::vector<long long> val;
long long P[N][N];
long long solve() {
        val.clear();
        val.reserve(n * n * n * n);

        P[1][1] = A[1] * B[1];
        for (int i = 2; i <= n; i++) {
            P[1][i] = P[1][i - 1] + A[1] * B[i];
            P[i][1] = P[i - 1][1] + A[i] * B[1];
        }
        for (int i = 2; i <= n; i++)
            for (int j = 2; j <= n; j++)
                P[i][j] = P[i - 1][j] + P[i][j - 1] - P[i - 1][j - 1] + A[i] * B[j];

        for (int i1 = 1; i1 <= n; i1++)
            for (int j1 = 1; j1 <= n; j1++)
                for (int i2 = i1; i2 <= n; i2++)
                    for (int j2 = j1; j2 <= n; j2++)
                        val.push_back(P[i2][j2] - P[i1 - 1][j2] - P[i2][j1 - 1] + P[i1 - 1][j1 - 1]);

        std::sort(val.begin(), val.end());

        return val[val.size() - k];
}

int main() {
        clock_t start, finish;
        start = clock();

        freopen("bsin.txt", "r", stdin);
        freopen("bsout.txt", "w", stdout);
        int T;
        scanf("%d", &T);
        for (int i = 0; i < T; i++) {
            read();
```

```
        long long ans = solve();
        printf("Case #%d: %lld\n", i + 1, ans);
        fprintf(stderr, "Case #%d: %lld\n", i + 1, ans);
    }

    finish = clock();
    double totle_time = (double)(finish - start) / CLOCKS_PER_SEC;
    fprintf(stderr, "Totle time: %lf s\n", totle_time);
}
```

注：上述代码在 C02TR2TEHTD7 上 Dubug/Release 模式下运行时间为 376.411340 s / 291.899425 s。与 WeiYong1024 比赛期间的代码时间复杂度相同。但作为T~O(TN^4)的算法该版本更具一般性。

Hezhu 的两层二分搜索算法（与作者给出的大数据集参考算法完全相同）:

对于大数据集，N的规模达到10^5，在内存中建立矩阵不现实，更不必说（let alone）遍历所有子矩阵的时间了。然而，由于矩阵元素$M_{i,j} = A_i * B_j$，我们可以将以$(a,b)-(c,d)$为左上-右下对角元素的子矩阵的元素和表示成$(A_a + A_{a+1} + \cdots + A_c) * (B_b + B_{b+1} + \cdots + B_d)$，即，$A$和$B$的连续子序列的元素和乘积。

假设我们用有序表U和V分别存储A和B的所有连续子序列的元素和，那么最终目标就是找到对于所有(i,j)，$U_i * V_j$的值中第K大的那一个。注意到K的值不超过10^5，我们无须考虑U,V中所有的元素组合，而只需要考虑U和V中各自最大的和最小的K个元素（注意由于存在负值故需要考虑代数值最小的元素），进而保证最终的目标一定出现在这些元素的乘积之中。至此，我们将最终问题拆分成如下两个子问题:

· 子问题 1:

给定长度为N的序列A，找到其连续子序列元素和中最大的K个。注意到K的值为$O(N)$。如果该子问题被解决，那么找到最小的K个连续子序列元素和只需把原序列元素取反然后运行相同算法即可。

· 解决方案:

使用二分搜索！思路是首先利用二分搜索找到A的连续子序列元素和中第K大的值，然后具体找到所有大于它的子区间元素和。具体地，为了用二分搜索找到子区间元素和中第K大的值SUM_K，对于给定常数tmp，我们需要一个算法能够快速统计元素和大于tmp的子区间的数量。用ps表示序列A的前缀和$ps_i = \sum_{k=1}^{i} A_k$，则对于任意下标$i$我们需要统计满足$ps_j < ps_i - tmp$且$j < i$的下标$j$的数量，为此只需维护前缀和序列$ps$的二进索引树（BIT）即可完在对数时间内完成该操作。

然后是找到所有大于SUM_K的子区间元素和值，对于这个任务，从小到大遍历区间右端点的下标i，对于每个i值，我们需要找到所有满足$j < i$且大于SUM_K的$ps[i] - ps[j]$的值。如果$ps[0], \ldots, ps[i-1]$总是有序排列的那么这个问题就会很容易解决，为此，我们用一个优先队列维护以访问的前缀和的值即可。

解决子问题 1 的时间复杂度为$T\sim O(\log(range\ of\ answer) * N\log(N) + K)$

· 子问题 2:

给定两个长度为$O(K)$的序列A,B，对于所有(i,j)，找到$A_i * B_j$的值中第K大的。

· 解决方案:

同样地，再次利用二分搜索，只需一个快速统计给定X，有多少有序数对(i,j)满足$A_i * B_j > X$的函数。

和子问题 1 的解法一样，思路是迭代序列A的下标i，对于每个i统计有多少j满足$A_i * B_j > X$，对于这个问题，若B有序则可以用二分搜索在对数时间内完成，只需注意A_i的正负性分别讨论即可。

解决子问题 2 的时间复杂度为$T\sim O(\log(range\ of\ answer) * N * \log(N))$

完整代码:

```
#include <algorithm>
#include <set>
```

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

using ll = long long;

const int maxn = 100005;

int x[maxn], y[maxn], r[maxn], s[maxn], a[maxn], b[maxn];
int n, k, a1, b1, c, d, e1, e2, f;

void read() {
	scanf("%d %d %d %d %d %d %d %d %d", &n, &k, &a1, &b1, &c, &d, &e1, &e2, &f);
	x[1] = a1, y[1] = b1;
	r[1] = 0, s[1] = 0;
	a[1] = a1, b[1] = b1;
	for (int i = 2; i <= n; i++) {
		x[i] = (c * x[i - 1] + d * y[i - 1] + e1) % f;
		y[i] = (d * x[i - 1] + c * y[i - 1] + e2) % f;
		r[i] = (c * r[i - 1] + d * s[i - 1] + e1) % 2;
		s[i] = (d * r[i - 1] + c * s[i - 1] + e2) % 2;
		a[i] = r[i] == 1 ? -x[i] : x[i];
		b[i] = s[i] == 1 ? -y[i] : y[i];
	}
}

std::vector<ll> vals;
int bit[maxn];

void init() {
	for (int i = 1; i <= vals.size(); i++)
		bit[i] = 0;
}

void add(int x) {
	while (x <= vals.size())
		bit[x]++, x += x & -x;
}

int query(int x) {
	int ans = 0;
	while (x > 0)
		ans += bit[x], x -= x & -x;
	return ans;
}
```

```
ll ps[maxn];
int idx[maxn];


std::vector<ll> get_maxk_mink(int a[]) {
    for (int i = 1; i <= n; i++)
        ps[i] = ps[i - 1] + a[i];
    vals.clear();
    for (int i = 0; i <= n; i++)
        vals.emplace_back(ps[i]);
    std::sort(vals.begin(), vals.end());
    vals.erase(std::unique(vals.begin(), vals.end()), vals.end());
    for (int i = 0; i <= n; i++)
        idx[i] = (int)(std::lower_bound(vals.begin(), vals.end(), ps[i]) - vals.begin()) + 1;

    ll l = -1e10, r = 1e10;
    while (l < r) {
        ll m = (l + r) >> 1;
        ll cnt = 0;
        init();
        for (int i = 0; i <= n; i++) {
            int t = (int)(std::lower_bound(vals.begin(), vals.end(), ps[i] - m) - vals.begin());
            cnt += query(t);
            add(idx[i]);
        }
        if (cnt < k) r = m;
        else l = m + 1;
    }

    std::vector<ll> ans;
    std::multiset<ll> st;
    for (int i = 0; i <= n; i++) {
        for (ll x : st) {
            if (ps[i] - x <= l) break;
            ans.emplace_back(ps[i] - x);
        }
        st.insert(ps[i]);
    }
    while (ans.size() < k)
        ans.emplace_back(l);
    return ans;
}


std::vector<ll> get_all(int a[]) {
```

```
    std::vector<ll> ans;
    for (int i = 1; i <= n; i++)
        ps[i] = ps[i - 1] + a[i];
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < i; j++)
            ans.emplace_back(ps[i] - ps[j]);
    return ans;
}

ll solve() {
    std::vector<ll> ak, bk;
    ll tot = 1ll * n * (n + 1) / 2;
    if (k * 2 >= tot)
        ak = get_all(a), bk = get_all(b);
    else {
        ak = get_maxk_mink(a), bk = get_maxk_mink(b);
        for (int i = 1; i <= n; i++)
            a[i] = -a[i], b[i] = -b[i];
        std::vector<ll> a_neg = get_maxk_mink(a);
        std::vector<ll> b_neg = get_maxk_mink(b);
        for (ll x : a_neg)
            ak.emplace_back(-x);
        for (ll x : b_neg)
            bk.emplace_back(-x);
    }

    std::sort(bk.begin(), bk.end());
    ll L = -1e18, R = 1e18;
    while (L < R) {
        ll M = (L + R) >> 1;
        ll cnt = 0;
        for (ll x : ak) {
            if (x == 0) {
                if (M < 0) cnt += bk.size();
            } else if (x > 0) {
                int l = 0, r = (int)bk.size();
                while (l < r) {
                    int m = (l + r) >> 1;
                    if (1ll * x * bk[m] > M) r = m;
                    else l = m + 1;
                }
                cnt += bk.size() - l;
            } else {
                int l = 0, r = (int)bk.size();
                while (l < r) {
```

```
                int m = (l + r) >> 1;
                if (1ll * x * bk[m] > M) l = m + 1;
                else r = m;
            }
            cnt += l;
        }
    }
    if (cnt < k) R = M;
    else L = M + 1;
  }
  return L;
}

int main() {
  freopen("blin.txt", "r", stdin);
  freopen("blout.txt", "w", stdout);
  int T;
  scanf("%d", &T);
  for (int t = 1; t <= T; t++) {
    read();
    ll ans = solve();
    printf("Case #%d: %lld\n", t, ans);
    fprintf(stderr, "Case #%d / %d: %lld\n", t, T, ans);
  }
}
```

注：正如作者给出的总结，本题中大量使用二分搜索。

3.4 Kickstart 2017 Round D

Problem C. Trash Throwing (解析几何，二分搜索，三分搜索)

题述：

Bob 是一个杰出的 Googler，他热衷于高效工作，所以他做什么事都是又好又快的。这天，Bob 发现他办公桌旁的垃圾桶不见了！这就意味着他得用另外一个距离他的办公桌比较远的垃圾桶。由于离开座位走过去倒垃圾的过程太费时间，他打算干脆把垃圾团成球扔过去。

不过 Google 办公室里有太多的障碍物，扔出去的垃圾如果打到人、砸到墙上或者碰到任何其他东西都是不礼貌的。所以 Bob 希望找到一种方法使得扔去的垃圾不会碰到任何东西。

为了简化问题，我们只考虑 Bob 和垃圾桶所在的竖直平面。Bob 位于$(0,0)$点，垃圾桶位于$(P,0)$点。另外，办公室内还有N个障碍物，每一个用平面上的一个点(X_i,Y_i)表示。办公室的天棚用平面上的一条水平线$y=H$表示。由于 Bob 所在的办公室是一个高科技悬浮办公室，故在这个问题中不用考虑地面，你不用考虑抛射物和地面之间的碰撞。Bob 扔出的垃圾球半径为R，初始时球心位于$(0,0)$。垃圾球被抛出后，其球心沿抛物线运行，抛物线轨迹为$f(x)=ax(x-P)$，$0\le x\le P$，其中a为任意小于等于0的实数。当垃圾球的球心落在$(P,0)$点时轨迹结束，球体的其他部分碰到$(P,0)$点不算结束。

Bob 好奇，在不碰到障碍物和天棚的前提下他最大可以投掷多大块的垃圾球？也就是说，我们要找到最大的R值，使得存在满足如下条件的a：对于任意$0\le x\le P$，点$(x,f(x))$和点(x,H)的欧氏距离大于R；对于任意i，点$(x,f(x))$和点(X_i,Y_i)的欧氏距离大于等于R。

输入：

第一行一个整数T为测试样例数量。后跟T个测试样例。每个测试样例的第一行为三个整数N,P,H——分别为障碍物数量、垃圾桶的坐标和天棚高度。接下来的N行，其中第i行表示第i个障碍物，其中两个整数X_i,Y_i表示障碍物的坐标。

输出：

对于每个测试样例，输出一行$Case\ \#x:\ y$，其中，x是从1开始的测试样例序号，y是表征最大R值的 double 型变量。绝对或相对误差小于10^{-4}的结果被视为正确。如果需要关于精度要求以及我们接受的答案形式的具体说明请查看 FAQ。

参数限制：

$1\le T\le 50$.

$2\le P\le 1000$.

$2\le H\le 1000$.

$0<X_i<P$.

$0<Y_i<P$.

小数据集：

$N=1$.

大数据集：

$1\le N\le 10$.

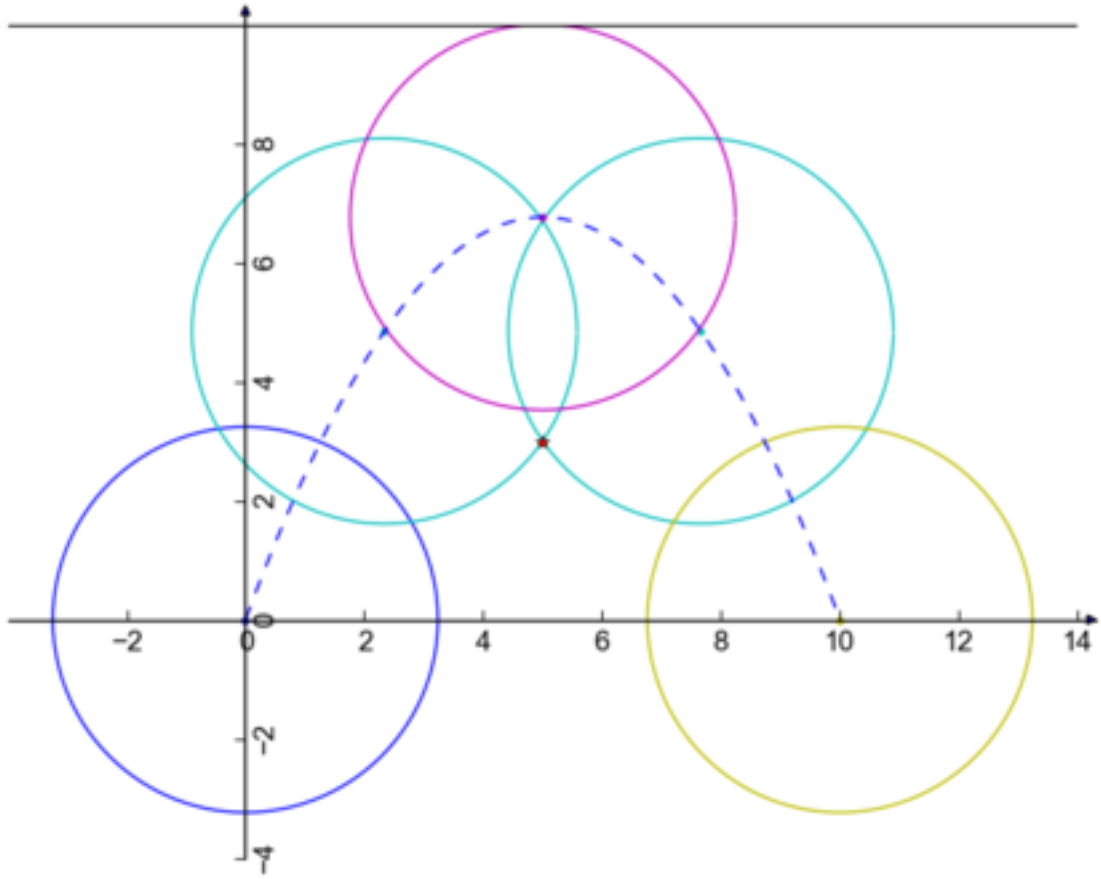
测试样例：

Input	Output
4 1 10 10 5 3 1 10 10 5 4 1 100 10 50 3 2 10 10	Case #1: 3.23874149472 Case #2: 4.00000000000 Case #3: 3.50000000000 Case #4: 2.23145912401

4 2 6 7	

注意最后一个测试样例不会出现在小数据集中。

下图为$Case$ #1的图示。Bob 位于$(0,0)$，垃圾桶位于$(10,0)$，$(5,3)$处有一个障碍物用星号表示。如果 Bob 选择从障碍物上方抛垃圾球，则最大半径R约为3.2387，相应的a为-0.2705。如果选择从障碍物下方抛垃圾球，则最大半径R为3，相应的a为0。所以R的最大值为3.2387。



$Case$ #2中的场景与$Case$ #1相似，但障碍物的位置升高了一个单位。在这种情况下，如果 Bob 从障碍物下方抛垃圾球，则最大半径R为4（相应的a为0）。如果从障碍物上方抛掷，则最大半径仅能够达到约2.8306（相应的$a=-0.4$）。故最大R值为4。

WeiYong1024 同于 Contest Analysis 的直观二分搜索思路:

显然，该问题为寻找满足条件的最小参数值问题。用连续区间的二分搜索算法。那么问题进一步转化为：**给定半径值 R，判断是否存在避开所有障碍物的两点式抛物线**（外层二分搜索判别函数）。

这个问题如何解决？首先对于只存在一个障碍点(x_0,y_0)的情况，球心的轨迹抛物线或者从障碍点上方经过，或者从障碍点下方经过。对于前者，抛物线开口参数a的变化区间为$[-\frac{4(H-R)}{P^2},\frac{y_0}{x_0(x_0-P)}]$，即下至经过障碍点，上至顶点高度为$H-R$。这个区间内，若存在满足条件的$a$，那么$a'<a$的$a'$也满足条件。于是我们又可以利用二分搜索找到一个$a$的阈值$a_1$，以及相应的可行区间$[-\frac{4(H-R)}{P^2},a_1]$。对于抛物线从下方经过的情况一样可以得到一个可行区间$[a_2,0]$。这样就得到了给定障碍点(x_0,y_0)以及球体半径R情况下抛物线开口的可行区间$[-\frac{4(H-R)}{P^2},a_1]\cup[a_2,0]$。

若该区间非空则存在抛物线使得半径为 R 的球体球心以之为轨迹运行的过程中球体不会碰触任何障碍物或天棚。现在问题转化为：**给定抛物线$y = ax(x - P)$、定点(x_0, y_0)和半径长度R，判断点到抛物线的距离是否小于R。**

对于这个问题，Contest Analysis 的思路是将问题转化为：判断抛物线和圆心位于(x_0, y_0)，半径为 R 的圆是否有交点／切点，联立求解圆方程$(x - x_0)^2 + (y - y_0)^2 = R^2$和抛物线方程$y = ax(x - P)$，具体地利用如牛顿法或费拉里法（一元四次方程求根公式）求解。鉴于实现复杂度，WeiYong1024 **在此使用三分搜索求解点到抛物线的距离**，具体依据第一象限的定点到抛物线的距离在定点所在的半区间$[0, \frac{P}{2}]$或$[\frac{P}{2}, P]$上只有一个下凸最值。（注：三分搜索迭代 57 次就可以使目标区间缩小至初始值的10^{-10}）

最后，对于多个障碍点的情形，采用相同的思路，其中判别给定R值是否可行时，只需对每一个障碍点计算 a 的可行区间，然后判断这些区间是否存在非空交集即可。

hqw_huang 的两层三分搜索算法:

首先，天棚的限制可以等效为一个位于$(\frac{P}{2}, H)$的障碍物。

最小值的最大值问题，hqw_huang 的两层三分搜索算法基于该问题的以下三点性质:

1. 设所有$N + 1$个障碍物的高度从低到高排序为$h_1, h_2, \dots, h_{N+1}$。固定与x轴两交点的抛物线由其高度为一决定，设高度为H_p的抛物线到所有障碍物的最近距离为$f_{H_p}(H_p)$，则$f_{H_p}(H_p)$在$H_p \in [0, h_1], [h_1, h_2], \dots, [h_N, h_{N+1}]$这些每一个区间上有至多一个极大值（可以用三分搜索求解）。
2. 抛物线开口比例参数a为抛物线高度H_p的严格单调减函数。单增序列$0, h_1, h_2, \dots, h_{N+1}$唯一对应单减序列$0, a_1, a_2, \dots, a_{N+1}$。
3. 抛物线$y = ax(x - P)$上的点在$x \in [0, P]$区间内到定点(x_0, y_0)的距离有至多一个极小值（可以用三分搜索求解）。

基于上述三点性质，不难得到如下算法：外层用三分搜索找到$f_a(a)$ 在每个小区间$[0, a_1], [a_1, a_2], \dots, [a_N, a_{N+1}]$上的最大值（唯一极大值），则问题的解为$f_a(a)$在所有区间上的最大值（最大值的最大值）；内层函数$f_a(a)$本身的求解算法为：用三分搜索求解抛物线上的点在区间$x \in [0, P]$上到每一个障碍物$(x_1, y_1), (x_2, y_2), \dots, (x_{N+1}, y_{N+1})$的最小距离（唯一极小值），则$f_a(a)$的值为距离所有点的最小值（最小值的最小值）。

完整代码:

```
#include <algorithm>
#include <math.h>
#include <vector>
#include <stdio.h>

const double eps = 1e-10;

struct point {
    double x;
    double y;
    point() {};
    point(double _x, double _y) : x(_x), y(_y) {};
};

int N;
double P, H;
std::vector<point> obs;
void read() {
```

```
	scanf("%d %lf %lf", &N, &P, &H);
	obs.clear();
	for (int i = 0; i < N; i++) {
		point tmp;
		scanf("%lf %lf", &tmp.x, &tmp.y);
		obs.push_back(tmp);
	}
	obs.push_back(point(P / 2, H));
}

bool cmp(point a, point b) { return a.y < b.y; }

double dist(point a, point b) { return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y)); }

double f(double a) {
	double ans = 1e9;
	for (auto v : obs) {
		double l = 0, r = P;
		while (r - l > eps) {
			double m1 = (2 * l + r) / 3;
			double m2 = (l + 2 * r) / 3;
			if (dist(v, point(m1, a * m1 * (m1 - P))) <= dist(v, point(m2, a * m2 * (m2 - P)))) r = m2;
			else l = m1;
		}
		ans = std::min(ans, dist(v, point(r, a * r * (r - P))));
	}
	return ans;
}

double solve() {
	std::sort(obs.begin(), obs.end(), cmp);
	double ans = 0, prev = 0;
	for (int i = 0; i < obs.size(); i++) {
		double l = -obs[i].y * (2 / P) * (2 / P), r = prev;
		while (r - l > eps) {
			double m1 = (2 * l + r) / 3;
			double m2 = (l + 2 * r) / 3;
			if (f(m1) <= f(m2)) l = m1;
			else r = m2;
		}
		ans = std::max(ans, f(r));
		prev = -obs[i].y * (2 / P) * (2 / P);
	}
	return ans;
}
```

```
int main() {
    freopen("c_in.txt", "r", stdin);
    freopen("c_out.txt", "w", stdout);
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        double ans = solve();
        printf("Case #%d: %lf\n", t, ans);
        fprintf(stderr, "Case #%d: %lf\n", t, ans);
    }
}
```

注：hwqhuang 的代码中求解点到抛物线距离使用的是$x \in [0, P]$上的三分搜索。这个地方有漏洞，当抛物线成“瘦高”状时抛物线下方的点到抛物线上的距离存在先减小再增大而后减小最后增大的情况，即有两个极小值和一个极大值，此时三分搜索不再适用。具体的改进方法为用下方 Christinass 代码中求解点到抛物线距离的方法，同样是利用三分搜索，但只在定点所在的半区间$\left[0, \frac{P}{2}\right]$或$\left[\frac{P}{2}, P\right]$上搜索。**点到抛物线的距离在轴的一侧必只有一个极小值。**

ngochai94 的二分搜索算法:

外层对R在$[0, H]$的范围内做二分搜索。其中判断半径R是否可行的函数 bool check(double R)其实现方法为将x轴在$[0, P]$的区间内5000等分。

Q: 5000 等分的方式怎样保证精度符合要求？

完整代码:

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

const double eps = 1e-10;

const int maxN = 15;

int N;
double P, H;
double X[maxN], Y[maxN];
void read() {
    scanf("%d %lf %lf", &N, &P, &H);
    for (int i = 0; i < N; i++)
        scanf("%lf %lf", &X[i], &Y[i]);
}
```

```
std::vector<std::pair<double, int>> v;
bool check(double R) {
	bool ans = false;
	int cnt = 0;
	v.clear();
	for (int i = 0; i < N; i++) {
		if (X[i] * X[i] + Y[i] * Y[i] < R * R) return false;
		if ((X[i] - P) * (X[i] - P) + Y[i] * Y[i] < R * R) return false;
		double minx = std::max(0.001, X[i] - R), maxx = std::min(P - 0.001, X[i] + R);
		int limit = 5000;
		for (int j = 0; j < limit; j++) {
			double x = minx + (maxx - minx) * j / (limit - 1);
			double dx = std::abs(x - X[i]);
			double dy = sqrt(R * R - dx * dx);
			v.push_back({(Y[i] - dy) / (x * (P - x)), 0});
			v.push_back({(Y[i] + dy) / (x * (P - x)), 1});
		}
	}
	std::sort(v.begin(), v.end());
	double lat = -0.0001;
	for (auto p : v) {
		double val = p.first;
		if (p.second) cnt--;
		else cnt++;
		if (!cnt) lat = val;
		if (p.second == 0 && cnt == 1) {
			if (lat <= 0 && 0 <= val) ans = true;
			if (lat >= 0 && lat * (P / 2) * (P / 2) <= H - R) ans = true;
		}
	}
	if (lat * (P / 2) * (P / 2) <= H - R) ans = true;
	return ans;
}

double solve() {
	double low = 0, high = H;
	while (high - low > eps) {
		double mid = (low + high) / 2;
		if (check(mid)) low = mid;
		else high = mid;
	}
	return (low + high) / 2;
}

int main() {
```

```
    freopen("c_in.txt", "r", stdin);
    freopen("c_out.txt", "w", stdout);
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        double ans = solve();
        printf("Case #%d: %lf\n", t, ans);
        fprintf(stderr, "Case #%d: %lf\n", t, ans);
    }
}
```

Christinass 按抛物线从每个点上／下经过分类的算法:

内层函数用三分搜索求点到抛物线距离。

完整代码:

```
#include <algorithm>
#include <cmath>
#include <cstdio>

const int maxN = 15;

const double eps = 1e-12;

struct point {
    double x, y;
    point() {};
    point(double _x, double _y) : x(_x), y(_y) {};
};

int N;
double P, H;
point pt[maxN];
void read() {
    scanf("%d %lf %lf", &N, &P, &H);
    for (int i = 0; i < N; i++)
        scanf("%lf %lf", &pt[i].x, &pt[i].y);
}

double dist(point a, point b) { return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y)); }

double search_R(double a, const point &o) {
    double left = 0, right = P;
    if (o.x >= P / 2) left = P / 2;
```

```
    else right = P / 2;
    while (right - left > eps) {
        double m1 = (2 * left + right) / 3;
        double m2 = (left + 2 * right) / 3;
        if (dist(point(m1, a * m1 * (m1 - P)), o) <= dist(point(m2, a * m2 * (m2 - P)), o)) right = m2;
        else left = m1;
    }
    return dist(point(left, a * left * (left - P)), o);
}

double func_min(double a) {
    double ans = 1e20;
    for (int i = 0; i < N; i++)
        ans = std::min(ans, search_R(a, pt[i]));
    double dis_to_ceil = H - (a * (P / 2) * (-P / 2));
    ans = std::min(ans, dis_to_ceil);
    return ans;
}

double calc(int state) {
    double high = 0, low = - 4 * H / (P * P);
    for (int i = 0; i < N; i++) {
        double tmp = pt[i].y / (pt[i].x * (pt[i].x - P));
        if (state & (1 << i)) high = std::min(high, tmp);
        else low = std::max(low, tmp);
    }
    if (low >= high) return 0;
    while (high - low > eps) {
        double m1 = (2 * low + high) / 3;
        double m2 = (low + 2 * high) / 3;
        if (func_min(m1) <= func_min(m2)) low = m1;
        else high = m2;
    }
    return func_min((low + high) / 2);
}

double solve() {
    double ans = 0;
    for (int i = 0; i < (1 << N); i++)
        ans = std::max(ans, calc(i));
    return ans;
}

int main() {
    freopen("c_in.txt", "r", stdin);
```

```
    freopen("c_out.txt", "w", stdout);
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        double ans = solve();
        printf("Case #%d: %lf\n", t, ans);
        fprintf(stderr, "Case #%d: %lf\n", t, ans);
    }
}
```

hamayanhamayan 的两层三分搜索算法:

与 hqwhuang 的算法类似，外层对a分别在小区间上三分搜索$f(a)$的最大值（其中为所有障碍点到参数为a的抛物线的最短距离），内层$f(a)$本身的算法为三分搜索求解点到抛物线的距离。不同点在于 hamayanhamayan 外层对 a 划分的使用三分搜索时的小区间以经过各障碍物所在点的抛物线为边界，相比 hqwhuang 的分区间方法，这样做是有道理的，所依据的是如下命题:

命题 1：在固定与x轴两焦点的抛物线$y = ax(x - P)$的a值从0变化到$-\frac{4H}{P^2}$的过程中，抛物线会顺次经过N个障碍点，在抛物线经过第i个点和第$i+1$个障碍点的过程中，$f(a)$的值从0开始先增大后减小到0，中间只有一个极大值。

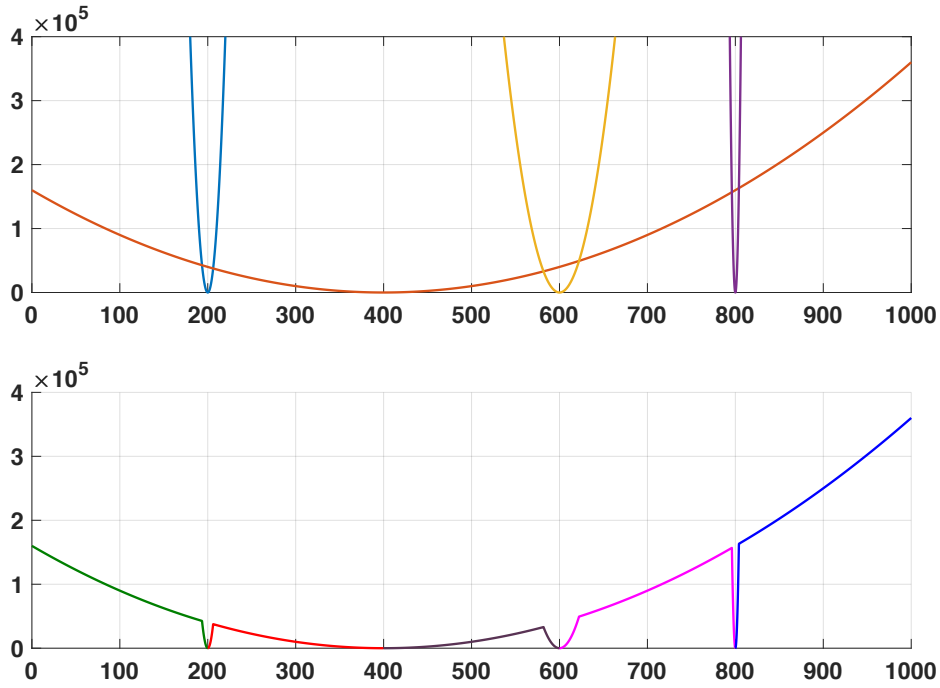
欲证明命题 1 首先证明如下命题:

命题 2：第一象限固定点(x_0, y_0)到抛物线的距离随着抛物线的a值从0变化到$-\infty$的过程中先减小到0再无限趋近于x_0。

命题 2 成立是由于在抛物线顶点上升的过程中，定点(x_0, y_0)始终在抛物线对称轴的同一侧，只考虑定点到这半侧抛物线的距离，当点在抛物线下方时，随着抛物线顶点的上升，定点到抛物线的距离逐渐增大。而当定点在抛物线上方的时候，显然随抛物线上升，定点到抛物线的距离逐渐减小。

命题 3：若干个最小值为 0 的单峰函数的最小值构成的函数在任意两个顶点之间只有一个极大值。

下图形象地给出抛物线顶点在上升过程中抛物线到所有点距离最小值的变化过程，对于每两个零点之间的部分（用不同颜色区分）可用三分搜索定位该区间内最大值的位置。



另外，hamayanhamayan 分别实现了搜索极大值点和极小值点的三分搜索模版。主程序中的两层三分搜索通过调用函数模版实现。

作者不愧为全球第二／中国第一的参赛选手，其 C++功底可见一斑。

完整代码:

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

const int maxN = 15;

int N;
double P, H, X[maxN], Y[maxN];
void read() {
    scanf("%d %lf %lf", &N, &P, &H);
    for (int i = 0; i < N; i++)
        scanf("%lf %lf", &X[i], &Y[i]);
}

#define LOOPMAX 201
template<typename Func>
double findMaxReal(double L, double R, Func f) {
    double left = L, right = R;
    for (int i = 0; i < LOOPMAX; i++) {
        double m1 = (2 * left + right) / 3;
        double m2 = (left + 2 * right) / 3;
        if (f(m1) <= f(m2)) left = m1;
```

```
            else right = m2;
        }
        return (left + right) / 2;
}

template<typename Func>
double findMinReal(double L, double R, Func f) {
        double left = L, right = R;
        for (int i = 0; i < LOOPMAX; i++) {
            double m1 = (2 * left + right) / 3;
            double m2 = (left + 2 * right) / 3;
            if (f(m1) <= f(m2)) right = m2;
            else left = m1;
        }
        return (left + right) / 2;
}

double A;
double func(double x) { return A * x * (x - P); }
int j;
double check(double x) {
        double y = func(x);
        double dx = x - X[j], dy = y - Y[j];
        double r = sqrt(dx * dx + dy * dy);
        return r;
}

double f(double a) {
        A = a;
        double ans = H - func(P / 2);
        for (int i = 0; i < N; i++) {
            j = i;
            double x = findMinReal(0, P, check);
            ans = std::min(ans, check(x));
        }
        return ans;
}

std::vector<double> v;
double solve() {
        double a_min = H / (P / 2 * (P / 2 - P));
        v.clear();
        v.push_back(0), v.push_back(a_min);
        for (int i = 0; i < N; i++) {
            double a_tmp = Y[i] / (X[i] * (X[i] - P));
```

```
            if (a_min <= a_tmp && a_tmp < 0) v.push_back(a_tmp);
        }
        std::sort(v.begin(), v.end());
        double ans = 0;
        for (int i = 1; i < v.size(); i++) {
            double a = findMaxReal(v[i - 1], v[i], f);
            ans = std::max(ans, f(a));
        }
        return ans;
}

int main() {
        freopen("c_in.txt", "r", stdin);
        freopen("c_out.txt", "w", stdout);
        int T;
        scanf("%d", &T);
        for (int t = 1; t <= T; t++) {
            read();
            double ans = solve();
            printf("Case #%d: %lf\n", t, ans);
            fprintf(stderr, "Case #%d: %lf\n", t, ans);
        }
}
```

cchao（D 轮第一名，唯一满分，台湾选手）的代码:

注：如下代码在 C02TR2TEHTD7 上 Debug/Release 模式下的运行时间为 957s/277s（约 16min/5min）。

注 2：如下代码无法通过赛后的练习用大数据集。

```
#include <algorithm>
#include <complex>
#include <iostream>

const int maxN = 15;

int n;
double p, h, x[maxN], y[maxN];

void read() {
        scanf("%d %lf %lf", &n, &p, &h);
        for (int i = 0; i < n; i++)
            scanf("%lf %lf", &x[i], &y[i]);
}

using CD = std::complex<double>;
```

```
inline CD point(double p, double a, double x) {
	return CD(x, a * x * (x - p));
}

double search(double p, double a, double x, double y, double l, double r) {
	double ans = 1e9;
	for (int i = 0; i < 50; i++) {
		double m1 = (2 * l + r) / 3;
		double m2 = (l + 2 * r) / 3;
		CD p1 = point(p, a, m1);
		CD p2 = point(p, a, m2);
		double d1 = std::abs(p1 - CD(x, y)), d2 = std::abs(p2 - CD(x, y));
		if (d1 > d2) l = m1;
		else r = m2;
		ans = std::min(ans, std::min(d1, d2));
	}
	return ans;
}

double dis(double p, double a, double x, double y) {
	return std::min(search(p, a, x, y, 0, p / 2), search(p, a, x, y, p / 2, p));
}

double ans;

double eval(double a) {
	if (a > 0) a = 0;
	double top = a * (p / 2) * (- p / 2);
	double r = h - top;
	for (int i = 0; i < n; i++)
		r = std::min(r, dis(p, a, x[i], y[i]));
	ans = std::max(ans, r);
	return r;
}

void go(double a, double step) {
	if (step < 1e-9) return;
	if (a > 0) a = 0;
	double na = a;
	double v = eval(a);
	for (int i = 0; i < 10; i++) {
		double del = (2.0 * rand() / RAND_MAX - 1.0) * step;
		double ta = a + del;
		double tv = eval(ta);
```

```
            if (tv > v)
                v = tv, na = ta;
        }
        go (na, step * 0.9);
}

double solve() {
        double ub = - h * 4 / (p * p);
        ans = 0;
        for (int j = 0; j < 100; j++)
            go(ub / j, 100);
        for (int j = 0; j < 100; j++)
            go(ub / j, ub / -100);
        return ans;
}

int main() {
        clock_t start, finish;
        start = clock();

        freopen("csin.txt", "r", stdin);
        freopen("csout.txt", "w", stdout);
        int T;
        scanf("%d", &T);
        for (int t = 1; t <= T; t++) {
            read();
            double ans = solve();
            printf("Case #%d: %lf\n", t, ans);
            fprintf(stderr, "Case #%d: %lf\n", t, ans);
        }

        finish = clock();
        double totle_time = (double) (finish - start) / CLOCKS_PER_SEC;
        fprintf(stderr, "Totle time: %lf s\n", totle_time);
}
```

如上代码中的**紫色加粗**部分可以作为生成[−1, 1]之间均匀分布随机数的发生器。

（综合上述所有算法的）最终版两层三分搜索算法:

基于 5 位通过大数据的选手提交的代码以及作者给出的分析，综合以上分析给出最适用于本题的算法版本，该算法保证严谨、高效、容易实现。具体如下:

使用三分搜索。具体地，$f_a(a)$在$[-\frac{4H}{P^2},0]$上的最大值就是本题的答案，以抛物线经过各障碍物为分界将整个区间分成$N+1$段，$\left[-\frac{4H}{P^2},a_1\right],[a_1,a_2],\dots,[a_N,0]$。如上图所示$f_a(a)$在每个小区间上有且仅有一个最大值峰，故用可用三分搜索找到$f_a(a)$在每个区间上的最大值，从而得到$f_a(a)$在整个区间上的最大值。

$f_a(a)$本身怎样求解呢？给定抛物线参数a，则$f_a(a)$的值就是所有点（还有天棚）到抛物线距离的最小值。那么定点到抛物线的距离如何求解呢？还是使用三分搜索，定点到抛物线上点的距离在定点所在抛物线轴的一侧先减小后增大，故可以使用三分搜索求得这个最小距离即为点到抛物线的距离。

整个算法的时间复杂度为$T\sim O(\log(a\text{的精度要求})*N*\log(P\text{的精度要求}))$

完整代码:

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

const int maxN = 15;

int N;
double P, H, X[maxN], Y[maxN];
void read() {
    scanf("%d %lf %lf", &N, &P, &H);
    for (int i = 0; i < N; i++)
        scanf("%lf %lf", &X[i], &Y[i]);
}

#define LOOPMAX 201
template<typename Func>
double findMaxReal(double L, double R, Func f) {
    double left = L, right = R;
    for (int i = 0; i < LOOPMAX; i++) {
        double m1 = (2 * left + right) / 3;
        double m2 = (left + 2 * right) / 3;
        if (f(m1) <= f(m2)) left = m1;
        else right = m2;
    }
    return (left + right) / 2;
}

template<typename Func>
double findMinReal(double L, double R, Func f) {
    double left = L, right = R;
    for (int i = 0; i < LOOPMAX; i++) {
        double m1 = (2 * left + right) / 3;
        double m2 = (left + 2 * right) / 3;
        if (f(m1) <= f(m2)) right = m2;
```

```
            else left = m1;
        }
        return (left + right) / 2;
}

double A;
double func(double x) { return A * x * (x - P); }
int j;
double check(double x) {
        double y = func(x);
        double dx = x - X[j], dy = y - Y[j];
        double r = sqrt(dx * dx + dy * dy);
        return r;
}

double f(double a) {
        A = a;
        double ans = H - func(P / 2);
        for (int i = 0; i < N; i++) {
            j = i;
            double x = X[j] <= P / 2 ? findMinReal(0, P / 2, check) : findMinReal(P / 2, P, check);
            ans = std::min(ans, check(x));
        }
        return ans;
}

std::vector<double> v;
double solve() {
        double a_min = H / (P / 2 * (- P / 2));
        v.clear();
        v.push_back(0), v.push_back(a_min);
        for (int i = 0; i < N; i++) {
            double a_tmp = Y[i] / (X[i] * (X[i] - P));
            if (a_min <= a_tmp && a_tmp < 0) v.push_back(a_tmp);
        }
        std::sort(v.begin(), v.end());
        double ans = 0;
        for (int i = 1; i < v.size(); i++) {
            double a = findMaxReal(v[i - 1], v[i], f);
            ans = std::max(ans, f(a));
        }
        return ans;
}

int main() {
```

```
    freopen("clin.txt", "r", stdin);
    freopen("clout.txt", "w", stdout);
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        double ans = solve();
        printf("Case #%d: %lf\n", t, ans);
        fprintf(stderr, "Case #%d: %lf\n", t, ans);
    }
}
```

3.5 Kickstart 2017 Round E

Problem A. Copy & Paste (动态规划)

题述：

从空串开始，欲构造仅有小写字母构成的目标字符串S。每一步可以进行如下三种操作中的一种：

- 在当前字符串尾部添加任意小写字母。
- 将当前字符串的任意连续子串复制到剪贴板。这一步将覆盖剪贴板中的原有内容，剪贴板初始为空。
- 将剪贴板中的全部内容复制到当前字符串尾部。这一步不改变剪贴板中的内容。

构成目标字符串最少需要多少步？注意要精确构成目标串，不可以存在多余的字符。

输入：

第一行为测试样例的数量T。接下来的T行每行为一个目标字符串S。

输出：

对于每个测试样例，输出一行$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是构造目标字符串的最少步数。

参数限制：

S中的字符只包含 a 到 z 之间的小写字母。

小数据集：

$1 \le T \le 100$

$1 \le |S| \le 6$

大数据集：

$1 \le T \le 100$

$1 \le |S| \le 300$

测试样例：

Input	Output
3 abcabab aaaaaaaaaaa vnsdmvnsnsdmkvdmkvnsdmk	Case #1: 6 Case #2: 7 Case #3: 15

Case #1 中最优的构造步骤是：

1. 输入 a
2. 输入 b
3. 输入 c
4. 复制 ab 到剪贴板
5. 粘贴 ab
6. 粘贴 ab

Case #2 中最优的构造步骤是：

1. 输入 a
2. 输入 a
3. 输入 a
4. 复制 aaa 到剪贴板
5. 粘贴 aaa
6. 复制 aaaaa 到剪贴板
7. 粘贴 aaaaa

caoxiaoran 的动态规划算法：

前缀上的动态规划，主状态为前缀长度，伴随状态为当前剪贴板中的内容。具体地，用$dp[i][j][k]$记录生成S长度为i的前缀且剪贴板中的内容为$s[j \ldots k]$时所需的最小步骤数。

复杂度：$T/S\sim O(N^3)$。

完整代码（惊艳代码）：

```
#include <algorithm>
#include <cstring>
#include <cstdio>

const int maxn = 305;
const int INF = 1e9;

char s[maxn];
void read() {
    scanf("%s", s);
}

int dp[maxn][maxn][maxn];
int solve() {
    for (int i = 0; i < maxn; i++)
        for (int j = 0; j < maxn; j++)
            for (int k = 0; k < maxn; k++)
                dp[i][j][k] = INF;

    int l = (int)strlen(s);
    if (l == 1) return 1;

    dp[0][1][0] = 0;
    for (int i = 0; i < l; i++) {
        int mn = INF;
        for (int j = 0; j < l; j++)
            for (int k = 0; k < l; k++) {
                mn = std::min(mn, dp[i][j][k]);
                dp[i + 1][j][k] = std::min(dp[i + 1][j][k], dp[i][j][k] + 1);
            }
        for (int j = 0; j < i; j++)
            for (int k = j, h = i; k < i && h < l && s[k] == s[h]; k++, h++)
                dp[h + 1][j][k] = std::min(dp[h + 1][j][k], std::min(mn + 1, dp[i][j][k]) + 1);
    }

    int ans = INF;
    for (int j = 0; j < l; j++)
        for (int k = 0; k < l; k++)
            ans = std::min(ans, dp[l][j][k]);
    return ans;
}
```

```
int main() {
    freopen("alin.txt", "r", stdin);
    freopen("alout.txt", "w", stdout);
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        int ans = solve();
        printf("Case #%d: %d\n", t, ans);
        fprintf(stderr, "Case #%d / %d: %d\n", t, T, ans);
    }
}
```

注释（关于备忘录初值设置 dp[0][1][0] = 0 的说明）：备忘录初值的设置只是为了让算法在搜索 dp[i][..][..]的最小值时能够搜到一个正确的值。实质上，这里**dp[..][1][0]的物理意义为：剪贴板为空的状态**，也即从未经过复制步骤一步一个字符地增加。当然，这一步是人为设置的，如果设置任意满足$i > j$的 dp[..][i][j]为空剪贴板状态都是可以的。

Tian.Xie 的动态规划算法:

dp[i]记录 s[0...i]的最小构成步骤数。用 LCP[i][j]记录目标字符串中下标为 i 和 j 开始的最长相同连续子串的长度。

复杂度：$T\sim O(N^4), S\sim O(N^2)$

（Tian.xie.在本轮测试中以 71 分的成绩排名全球第九，恭喜他！希望以后能和他成为同事）

完整代码:

```
#include <algorithm>
#include <string>
#include <iostream>

const int maxn = 305;
const int INF = 1e9;

std::string s;
void read() {
    std::cin >> s;
}

int dp[maxn];
int LCP[maxn][maxn];
int solve() {
    for (int i = 0; i < s.size(); i++)
        for (int j = 0; j < s.size(); j++) {
            int tmp = 0;
            while (i + tmp < s.size() && j + tmp < s.size() && s[i + tmp] == s[j + tmp])
                tmp++;
```

```
                LCP[i][j] = tmp;
            }

        for (int i = 0; i < maxn; i++)
            dp[i] = INF;

        dp[0] = 1;
        for (int i = 0; i < s.size(); i++) {
            for (int j = i + 1; j < s.size(); j++)
                dp[j] = std::min(dp[j], dp[i] + j - i);
            for (int l = 0; l <= i; l++)
                for (int r = l; r <= i; r++) {
                    int len = r - l + 1;
                    int visited = 0, last = i;
                    for (int j = i + 1; j < s.size(); j++) {
                        int head = j - len + 1;
                        if (LCP[head][l] >= len && head > last)
                            visited++, last = j;
                        dp[j] = std::min(dp[j], dp[i] + 1 + j - i - (len - 1) * visited);
                    }
                }
        }

        return dp[s.size() - 1];
}

int main() {
        freopen("alin.txt", "r", stdin);
        freopen("alout.txt", "w", stdout);
        int T;
        scanf("%d", &T);
        for (int t = 1; t <= T; t++) {
            read();
            int ans = solve();
            printf("Case #%d: %d\n", t, ans);
            fprintf(stderr, "Case #%d / %d: %d\n", t, T, ans);
        }
}
```

3.5 Kickstart 2017 Round E

Problem B. Trapezoid Counting (计数问题，频次表)

题述：

在这个问题中，我们考虑有且仅有一对平行边的凸四边形。进一步，如果非平行边的长度相同，我们称这样的凸四边形为等腰梯形。

现有若干长度不同的木条，你要从中取出四根构成等腰梯形的边。那么有多少组四根木条的组合符合要求呢？注意这里长度相同的两根木条算作不同的两只。木条不能弯曲或折断。

输入：

第一行为测试样例数量T，后跟T个测试样例。对每个测试样例两行，第一行为一个整数N，表示木条的数量，接下来的一行为N个整数，其中第i各整数L_i表示第i根木条的长度。

输出：

对于每个测试样例，输入一行$Case\ x: y$，其中x是从1开始的测试样例序号，y如上所述为可构成等腰梯形边的木条四元组数。

参数限制：

$1 \leq T \leq 100$

$1 \leq L_i \leq 10^9$

小数据集：

$1 \leq N \leq 50$

大数据集：

$1 \leq N \leq 5000$

测试样例：

Input	Output
5 5 2 3 3 4 3 4 1 5 3 1 4 2 2 3 3 4 999999998 999999999 999999999 1000000000 9 3 4 1 4 2 5 3 1 3	Case #1: 5 Case #2: 0 Case #3: 0 Case #4: 1 Case #5: 73

样例 Case #1 中，木条四元组有五个，其中任何一个都可以构成等腰梯形的四边

样例 Case #2 中，四元组$\{1, 1, 3, 5\}$无法构成等腰梯形，尽管有一对木条长度相同。

样例 Case #3 中，四元组$\{2, 2, 3, 3\}$可以构成一个矩形，但在本题中举行不算等腰梯形。

Caoxiaoran 基于频次表的计数方法:

找到所有可以构成等腰梯形的木条四元组，对等腰梯形有两条相同边和有三条相同边的情况分别进行统计。

将原始数据按长度出现次数存表 map<ll, ll> mp，迭代所有长度值。对于出现次数不少于两次的长度，累计以该长度值为腰长且上下底长度都不为该值的等腰梯形数量。对于出现次数不少于三次的长度值，累计以该长度为两腰和一个底边长度的等腰梯形数量。

完整代码:

```
#include <algorithm>
```

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

using ll = long long;

int n;
std::map<ll, ll> mp;
void read() {
	scanf("%d", &n);
	mp.clear();
	for (int i = 0; i < n; i++) {
		int tmp;
		scanf("%d", &tmp);
		mp[tmp]++;
	}
}

ll C(ll n, ll m) {	// Combination number.
	ll x = 1, y = 1;
	for (ll i = n, j = m; j > 0; i--, j--)
		x *= i, y *= j;
	return x / y;
}

ll solve() {
	ll ans = 0;
	for (auto e : mp) {
		if (e.second >= 2) {
			ll cc = C(e.second, 2);
			std::vector<std::pair<ll, ll>> b;
			for (auto x : mp)
				if (x.first != e.first) b.emplace_back(x);
			ll sum = 0;
			std::sort(b.begin(), b.end());
			for (int i = 0, j = 0; i < b.size(); i++) {
				sum -=b[i].second;
				while (j < b.size() && b[j].first < b[i].first + e.first * 2)
					sum += b[j++].second;
				ans += cc * sum * b[i].second;
			}
		}
		if (e.second >= 3) {
			ll cc = C(e.second, 3);
			for (auto x : mp)
```

```
                if (x.first != e.first && x.first < 3 * e.first) ans += cc * x.second;
            }
        }
        return ans;
}

int main() {
        freopen("bsin.txt", "r", stdin);
        freopen("bsout_cmp.txt", "w" ,stdout);
        int T;
        scanf("%d", &T);
        for (int t = 1; t <= T; t++) {
            read();
            ll ans = solve();
            printf("Case #%d: %lld\n", t, ans);
            fprintf(stderr, "Case #%d / %d: %lld\n", t, T, ans);
        }
}
```

算法编程心得: 有一些操作书写代码可能需要和当前已有数量相仿的代码，然而却能使问题更加明晰，数据组织更加合理。比如这道题 Caoxiaoran 对于每个可以作为腰长的木棒长度，都重新建立一个新的不包含该长度的序列 b，**这个简单的操作虽然消耗了$T/S\sim O(N)$的复杂度，然而却在不构成关键路径的情况下使得数据组织更清晰**，算法书写更容易。

对于通过不构成程序关键路径的操作使得算法更容易书写的情况，要积极编写这样的代码。

3.5 Kickstart 2017 Round E

Problem C. Blackhole (二分搜索，平面几何)

题述：

Alice 努力防止黑洞威胁地球。目前，在空间三个不同的点上存在三个黑洞。Alice 要创造三个异空间，这三个异空间半径相同。异空间之间互不影响，所以他们可以重叠。

Alice 要让每个黑洞被至少一个异空间涵盖，另外，为了维持稳定性，三个异空间必须构成一个连通区域。

Alice 想要创造异空间的成本最低，那么满足条件的最小异空间半径是多少呢？

输入：

第一行一个整数T表示测试样例数量，后跟T个测试样例。每个测试样例由三行组成，其中每行为三个整数X_i, Y_i, Z_i——代表第i个黑洞的空间位置。

输出：

对于每个测试样例，输出一行$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是一个有理数(rantional)，代表 Alice 解决该问题所使用的最小半径长度。当y值的绝对 / 相对误差小于10^{-6}时答案被视作正确。

参数限制：

$1 \le T \le 100.$

$-1000 \le X_i \le 1000, for\ all\ i.$

$(X_j, Y_j, Z_j) \ne (X_k, Y_k, Z_k), for\ all\ j \ne k.$

小数据集：

$Y_i = 0, for\ all\ i.$

$Z_i = 0, for\ all\ i.$

大数据集：

$-1000 \le Y_i \le 1000, for\ all\ i.$

$-1000 \le Z_i \le 1000, for\ all\ i.$

测试样例：

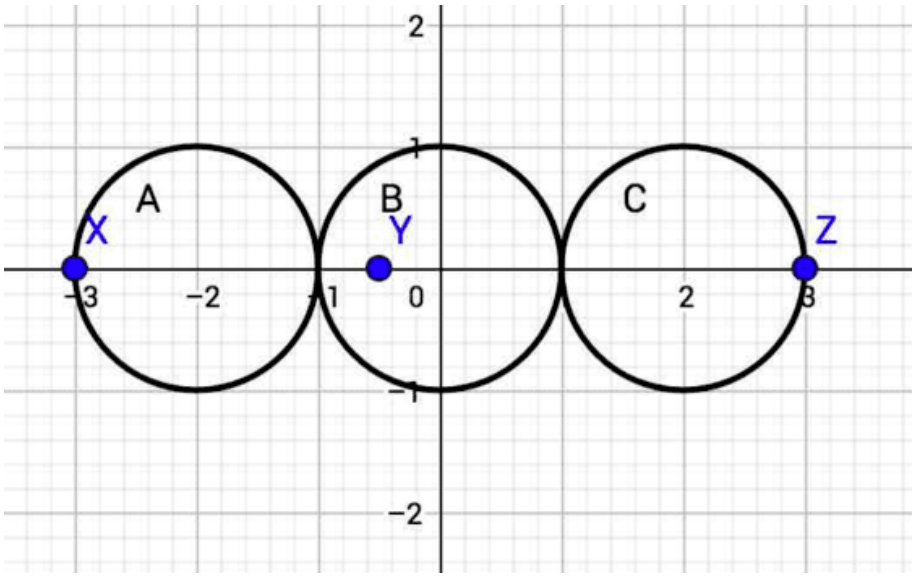
Input	Output
4 0 0 0 1 0 0 -1 0 0 4 0 0 5 0 0 -2 0 0 0 0 0 1 1 1 -1 -1 -1 -4 2 -2 5 1 -4 0 4 -9	Case #1: 0.3333333333 Case #2: 1.1666666667 Case #3: 0.5773502692 Case #4: 2.1373179212

注意最后两个测试样例不会出现在小数据集中。

样例 Case #1 中，可以使用的最小半径是$\frac{1}{3}$，三个异空间中心分别位于$\left(-\frac{2}{3}, 0, 0\right), (0, 0, 0), (\frac{2}{3}, 0, 0)$。

官方解析:

对于小数据集，黑洞三点共线，最优覆盖方案就是三个球心共线、相互外切。

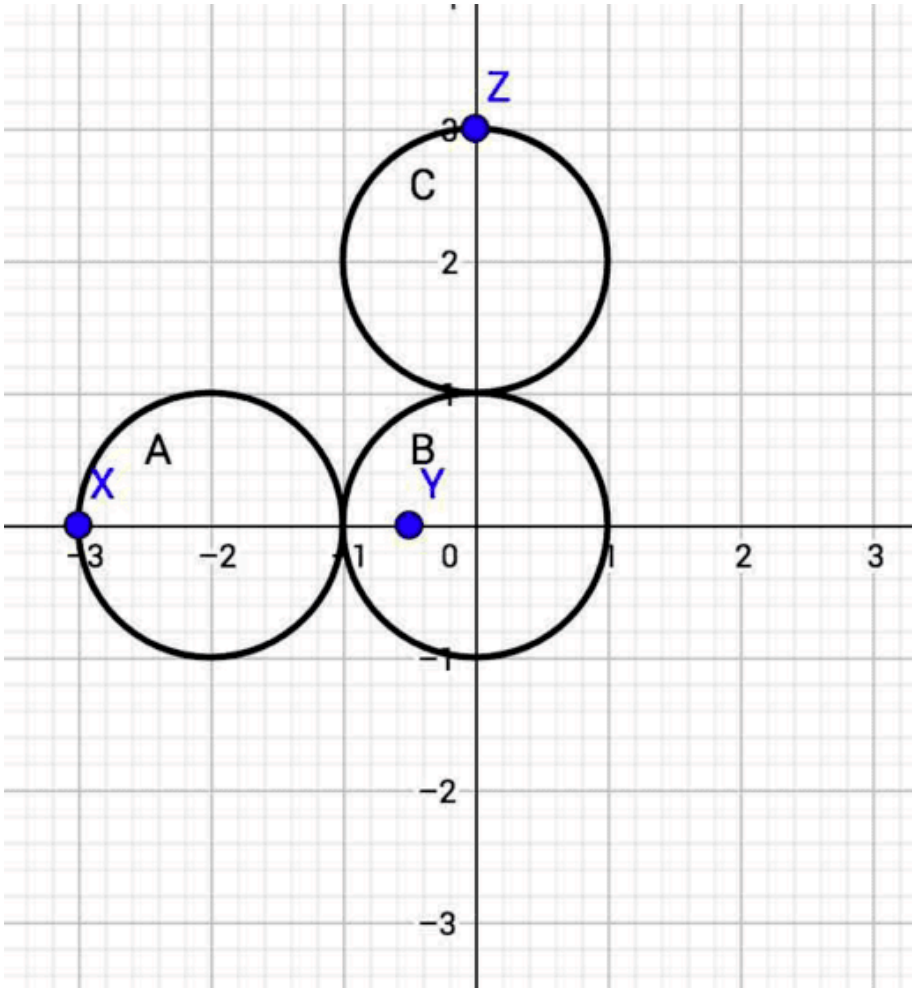


满足覆盖的最小半径为边上两点距离的六分之一，不解释。

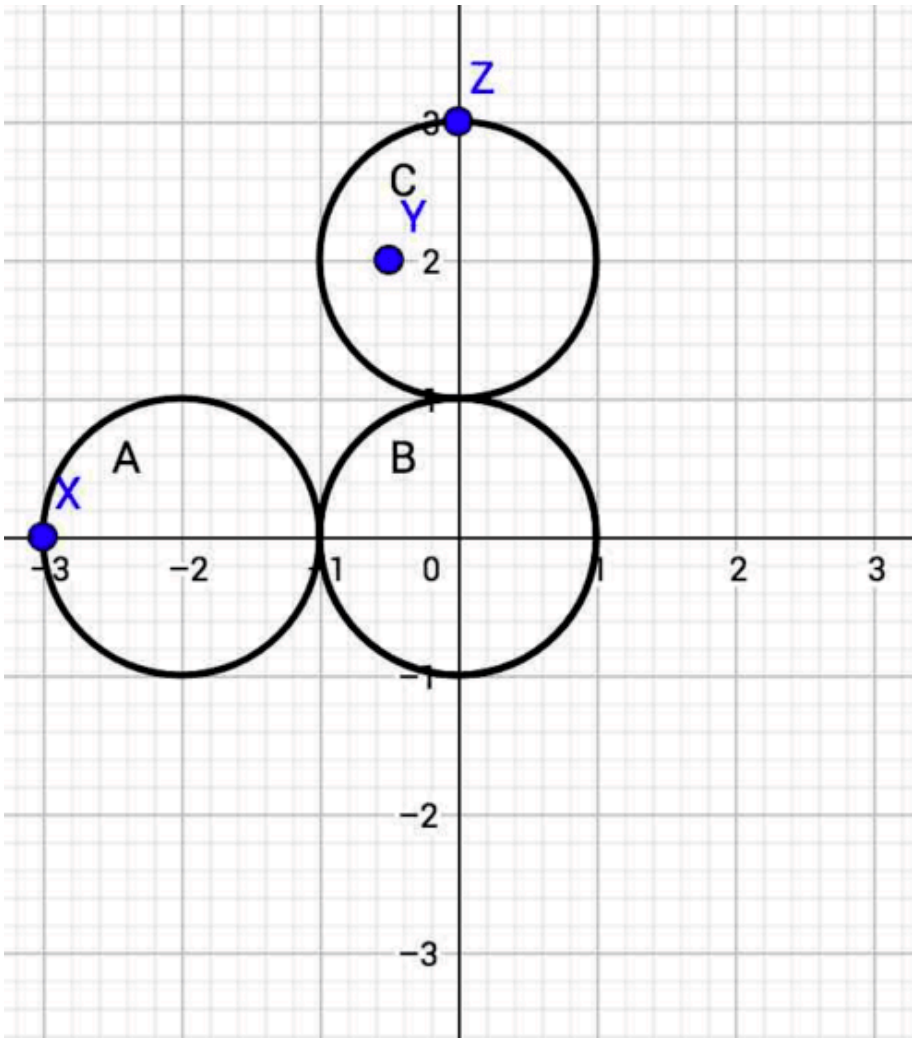
对于大数据集，很显然这个 3D 问题本质上是一个 2D 问题。即在三点所确定的平面上分析：如何利用三个半径相等且相互连通的圆围住平面上的三个定点。

假设我们有圆 A,B,C，其中圆 A 和圆 B 连通，圆 B 和圆 C 连通，圆 A 和圆 C 不一定连通。那么有如下两种方法来覆盖三个指定点:

· 第一种，三个圆各覆盖一个指定点:



· 第二种，圆 A 覆盖一个点，圆 C 覆盖两个点。



其他的覆盖方式都不是最优的。具体而言，如果三个点都在一个圆内，那么就浪费了另外两个圆；如果两个点位于第一个圆内，另外一个点位于第二个圆内，且第三个圆不同时和前两个圆直接相连，那么总可以通过把第三个圆放在这两个圆中间来缩小覆盖所需的半径。

外层用二分搜索找到满足覆盖的最小半径，那么现在需要一个函数，给定半径值R，判断是否有以R为半径且相互连通的三个圆能够把三个定点覆盖住。

· 对于第一种覆盖方式，其特征点为圆B中心，用三个半径为R的圆能够以方式一覆盖三个定点等价于存在符合下述条件的圆心B:

—— “圆心B和三个定点中的一个距离不超过R，与另外两个的距离不超过$3R$。”

具体的，这个判据等价于：“以三个定点为圆心，半径分别为$R, 3R, 3R$的圆所围区域是否有交集。”

· 对于第二种覆盖方式，其特征点为圆C中心。用三个半径为R的圆能够以方式二覆盖三个定点等价于存在符合下述条件的圆心C:

—— “圆心C和三个定点中的两个距离不超过R，与另外一个的距离不超过$5R$。”

具体的，这个判据等价于：“以三个定点为圆心，半径分别为$R, R, 5R$的圆所围区域是否有交集。”

至此，最后剩下的一个问题是如何判定三个圆是否有交集。这里采用如下方法：首先，若存在两个圆相离，则无交集。其次，若存在两个圆相互包含（包括内切），则判断内圆和第三个圆是否相离即可。对于三个圆两两相交（包括外切）的情况，共有三对共六个交点（可能重合）。若三个圆有公共部分，则公共部分一定包含这六个交点之一（**证明见后**），那么只需求出这六个交点判断其是否在另外一个圆内即可。

时间复杂度$T\sim O(\log(range\ of\ answer\ \&\ precision))$。

完整代码:

```
#include <algorithm>
```

```
#include <cmath>
#include <iostream>

const double eps = 1e-10;

struct point {
	double x, y;
	point() {};
	point(double _x, double _y) : x(_x), y(_y) {};
};

struct circle {
	point o;
	double r;
	circle() {};
	circle(point _o, double _r) : o(_o), r(_r) {};
};

double x[3], y[3], z[3];

void read() {
	for (int i= 0; i < 3; i++)
		scanf("%lf %lf %lf", &x[i], &y[i], &z[i]);
}

point u, v, w;

void convert_3d_2d() {
	double la, lb, lc;
	double e[3];
	e[0] = sqrt((x[0] - x[1]) * (x[0] - x[1]) + (y[0] - y[1]) * (y[0] - y[1]) + (z[0] - z[1]) * (z[0] - z[1]));
	e[1] = sqrt((x[0] - x[2]) * (x[0] - x[2]) + (y[0] - y[2]) * (y[0] - y[2]) + (z[0] - z[2]) * (z[0] - z[2]));
	e[2] = sqrt((x[1] - x[2]) * (x[1] - x[2]) + (y[1] - y[2]) * (y[1] - y[2]) + (z[1] - z[2]) * (z[1] - z[2]));
	std::sort(e, e + 3);
	la = e[0], lb = e[1], lc = e[2];
	double cosa = (lb * lb + lc * lc - la * la) / (2 * lb * lc);
	double sina = sqrt(1 - cosa * cosa);
	u.x = 0, u.y = 0;
	v.x = lc, v.y = 0;
	w.x = lb * cosa, w.y = lb * sina;
}

double dist(point a, point b) {
	return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}
```

```
bool disjoint(circle a, circle b) { // Whether circle a and b are disjoint from each other.
    if (dist(a.o, b.o) > a.r + b.r) return true;
    else return false;
}

bool cover(circle a, circle b) {    // Whether circle a covers circle b.
    if (a.r >= b.r && dist(a.o, b.o) <= a.r - b.r) return true;
    else return false;
}

double sqr(double x) {
    return x * x;
}

std::pair<point, point> get_cross(circle A, circle B) { // Get the two cross points of cirlce A and B,
    double x1 = A.o.x, y1 = A.o.y, x2 = B.o.x, y2 = B.o.y, r1 = A.r, r2 = B.r;

    if (std::abs(x1 - x2) < std::abs(y1 - y2)) {
        double k = (x1 - x2) / (y1 - y2);
        double b = (sqr(r2) - sqr(r1) + sqr(x1) - sqr(x2) + sqr(y1) - sqr(y2)) / (2 * (y1 - y2));

        double a2 = 1 + sqr(k);
        double a1 = - 2 * x1 - 2 * k * b + 2 * k * y1;
        double a0 = sqr(x1) + sqr(y1) + sqr(b) - 2 * b * y1 - sqr(r1);

        point p1, p2;
        double delta = sqr(a1) - 4 * a2 * a0;
        p1.x = (-a1 + sqrt(delta)) / (2 * a2);
        p2.x = (-a1 - sqrt(delta)) / (2 * a2);
        p1.y = - k * p1.x + b;
        p2.y = - k * p2.x + b;
        return {p1, p2};
    } else {
        double k = (y1 - y2) / (x1 - x2);
        double b = (sqr(r2) - sqr(r1) + sqr(x1) - sqr(x2) + sqr(y1) - sqr(y2)) / (2 * (x1 - x2));

        double a2 = 1 + sqr(k);
        double a1 = - 2 * y1 - 2 * k * b + 2 * k * x1;
        double a0 = sqr(x1) + sqr(y1) + sqr(b) - 2 * b * x1 - sqr(r1);

        point p1, p2;
        double delta = sqr(a1) - 4 * a2 * a0;
        p1.y = (-a1 + sqrt(delta)) / (2 * a2);
        p2.y = (-a1 - sqrt(delta)) / (2 * a2);
```

```
            p1.x = - k * p1.y + b;
            p2.x = - k * p2.y + b;
            return {p1, p2};
        }
}

bool tri_cross(circle a, circle b, circle c) {    // Whether circle a, b and c share a common part.
        // If there is any disjoint pair.
        if (disjoint(a, b) || disjoint(b, c) || disjoint(a, c)) return false;

        // If there is any covering pair.
        if (cover(a, b)) return !disjoint(b, c);
        if (cover(b, a)) return !disjoint(a, c);
        if (cover(a, c)) return !disjoint(c, b);
        if (cover(c, a)) return !disjoint(a, b);
        if (cover(b, c)) return !disjoint(c, a);
        if (cover(c, b)) return !disjoint(b, a);

        // Each pair cross each other.
        point p1, p2;
        std::tie(p1, p2) = get_cross(a, b);
        if (dist(p1, c.o) <= c.r || dist(p2, c.o) <= c.r) return true;
        std::tie(p1, p2) = get_cross(a, c);
        if (dist(p1, b.o) <= b.r || dist(p2, b.o) <= b.r) return true;
        std::tie(p1, p2) = get_cross(b, c);
        if (dist(p1, a.o) <= a.r || dist(p2, a.o) <= a.r) return true;

        return false;
}

bool check(double r) {
        // Type I.
        if (tri_cross(circle(u, r), circle(v, 3 * r), circle(w, 3 * r))) return true;
        if (tri_cross(circle(u, 3 * r), circle(v, r), circle(w, 3 * r))) return true;
        if (tri_cross(circle(u, 3 * r), circle(v, 3 * r), circle(w, r))) return true;
        // Type II.
        if (tri_cross(circle(u, 5 * r), circle(v, r), circle(w, r))) return true;
        if (tri_cross(circle(u, r), circle(v, 5 * r), circle(w, r))) return true;
        if (tri_cross(circle(u, r), circle(v, r), circle(w, 5 * r))) return true;
        return false;
}

double solve() {
        convert_3d_2d();
```

```
    double l = 0, r = 2000;
    while (r - l > eps) {
        double m = (l + r) / 2;
        if (check(m)) r = m;
        else l = m;
    }
    return (l + r) / 2;
}

int main() {
    freopen("clin.txt", "r", stdin);
    freopen("clout.txt", "w", stdout);
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        double ans = solve();
        printf("Case #%d: %.10lf\n", t, ans);
        fprintf(stderr, "Case #%d / %d: %.10lf\n", t, T, ans);
    }
}
```

命题：三个圆两两相交的情况下，存在两个圆的交点在第三个圆内⇔三个圆存在交集

证明：

⟹显然。

⟸说明如下：在两两相交的三个圆中，任取两个圆，二者相交的到一个“眼形”，三个圆的交集为该“眼形”和第三个圆的交集。这个“眼形”和第三个圆的公共区域一定有“尖点”，否则第三个圆应被前两个圆相交的到的“眼形”包含，这与三个圆两两相交矛盾。由于这个尖点既是三圆交集边界上的点，又是某两个圆周的交点，故⟸得证。

算法编程心得：对于给定问题，**通过小案例探究遍所有可能性**，针对每种可能性分别设计算法解决。

3.6 Kickstart 2017 Round F

A. Kicksort (小数据量尝试，线性策略)

题述：

Kickstart 团队的成员是著名算法——Quicksort 的粉丝，这个算法在序列中选择一个轴点，然后把其他所有元素根据其与轴点值的相对大小移动到两个新的序列中，然后递归地处理这两个新的序列。然而，若轴点选择不当可能会导致所有其他的元素被分配到两个序列中的某一个，这样违背了分治策略的初衷。我们称这样的轴点为“最坏轴点”。

为了避免这个问题，我们给出了一种备选方案——Kicksort。有些人说把序列中点选作轴点可以达到较好的效果，于是我们的算法设计如下：

```
Kicksort(A): // A is a 0-indexed array with E elements

   If E ≤ 1, return A.

   Otherwise:

     Create empty new lists B and C.

     Choose A[floor((E-1)/2)] as the pivot P.

     For i = 0 to E-1, except for i = floor((E-1)/2):

       If A[i] ≤ P, append it to B.

       Otherwise, append it to C.

   Return the list Kicksort(B) + P + Kicksort(C).
```

实践中，我们将 Kicksort 用于1到N的排列的排序。很不幸，Kicksort 目测也存在 Quicksort 的问题——存在每个轴点都是最坏轴点的情形。

例如，考虑序列$1, 4, 3, 2$。Kicksort 选择4为轴点，所有剩下的值$1, 3, 2$将进入新序列其中一个。然后对序列$1, 3, 2$调用 Kicksort 时，3被选作轴点，所有剩下的值$1, 2$进入两个新序列其中之一。最后，对序列$1, 2$调用 Kicksort 时，1为轴点，另外一个值2进入两个新序列之一。在每一步，算法都选择了“最坏轴点”。（注意当 Kicksort 被调用在长度为 0 或1的序列时不选择轴点）。

请帮助我们进一步分析！当 Kicksort 作用于一个1到N的排列时，判断是否每次都是最坏轴点。

输入：

第一行一个整数T为测试样例的数量，后跟T个测试样例。每个测试样例包含两行，第一行一个整数N为排列长度，第二行为N个整数A_i，表示排列的元素。

输出：

对于每个测试样例，输出一行包含$Case: \#x: y$，其中x是从1开始的测试样例序号，如果 Kicksort 每次选择的轴点都是“最坏轴点”则y为 YES，否则为 NO。

参数限制：

A_i的值为1到N的排列。

小数据集：

$1 \le T \le 32$

$2 \le N \le 4$

大数据集：

$1 \le T \le 100$

$2 \le N \le 10000$

测试样例：

Input	Output
4 4 1 4 3 2 4 2 1 3 4 2 2 1 3 1 2 3	Case #1: YES Case #2: NO Case #3: YES Case #4: NO

测试样例 1 为题述中的例子。

测试样例 2 中，第一个轴点是1，这是一个“最坏轴点”，因为剩下的元素2, 3, 4将被移入两个新序列中的同一个。因为。然而当 Kicksort 对序列2, 3, 4调用时轴点为3，这不是一个“最坏轴点”，因为剩下的元素2和4将被移入不同的序列。

测试样例 3 中，Kicksort 选择的第一个轴点是2，此后再无选择轴点的操作。

测试样例 4 中，Kicksort 选择的第一个轴点是2，这不是一个“最坏轴点”。

WeiYong1024 的分析与算法：

尝试一些长度较小的排列，试图找到规律。

首先，若每一步都选择最坏轴点，则每一步选择轴点的数量都是1，即完成排序需要恰好n次函数调用。

可以发现，本题中的 Kicksort 算法每次递归在选择完轴点之后将剩余元素移动至两个新序列的过程中并不改变新序列中元素在原序列中的相对位置（这一点与本地操作的快速排序不同，唐发根《数据结构》中的快速排序是本地不稳定的）。这就导致如果每次选择的都是最坏轴点，即每次轴点之外的元素都移动至轴点的同一侧，那么每一步所选定的一个轴点在原序列中的顺序是固定的，具体的，

· 对于n为奇数的情况以7为例：

数组	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
第几个被选为轴点	6	4	2	1	3	5	7

· 对于n为偶数的情况以6为例：

数组	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$
第几个被选为轴点	5	3	1	2	4	6

每次都选到最坏轴点⇔每次归类都只归到一侧⇔每次所选的元素在原排列中从中间向两侧扩散，那么只需按照最坏轴点下标顺序检查位于该位置的元素是否为剩余元素的轴点（最大值或最小值）即可。

时间复杂度$T\sim O(N)$

完整代码:

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

const int maxn = 10005;

int a[maxn];
int n;
void read() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++)
		scanf("%d", &a[i]);
}

std::vector<int> idx;
bool solve() {
	idx.clear();
	if (n & 1) {
		idx.emplace_back((n - 1) >> 1);
		for (int i = 1; i <= (n - 1) >> 1; i++) {
			idx.emplace_back((n - 1) / 2 - i);
			idx.emplace_back((n - 1) / 2 + i);
		}
	} else {
		for (int i = (n - 1) / 2; i >= 0; i--) {
			idx.emplace_back(i);
			idx.emplace_back(n - 1 - i);
		}
	}

	int mn = 1, mx = n;
	for (int x : idx) {
		if (a[x] == mn) mn++;
		else if (a[x] == mx) mx--;
		else return false;
	}
	return true;
}

int main() {
	freopen("alin.txt", "r", stdin);
	freopen("alout.txt", "w", stdout);
```

```
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        bool ans = solve();
        if (ans) {
            printf("Case #%d: YES\n", t);
            fprintf(stderr, "Case #%d: YES\n", t);
        } else {
            printf("Case #%d: NO\n", t);
            fprintf(stderr, "Case #%d: NO\n", t);
        }
    }
}
```

3.6 Kickstart 2017 Round F

B. Dance Battle (线性策略，贪心法)

题述：

你的团队要在斗舞赛上展示实力！起初，你的队伍有E点能量点，0点荣耀值。共有N个团队作为斗舞对手，其中第i个团队在队列中位于第i位，舞技值为S_i

在每轮斗舞的中，你们团队需要迎战对头的团队，你可以选择下属四种操作中的一种：

1. 斗舞：你的团队消耗等同于对手团队舞技值的能量点，对手团队不再回到队列，你的团队获得1点荣耀值。若该操作后你团队的能量点变为0或以下，则该操作无法进行。
2. 延期：你可以说："我们的鞋带没系好！"，则对手团队移至队尾，你团队的能量点和荣耀值不变。
3. 停战：和对手团队宣布停战，则对手团队不再回到队列，你团队的能量点和荣耀值不变。
4. 招募：将对手团队招募进你的团队，则对手团队不再进入队列，你的团队获得等同于该团队舞技值的能量点，损失1点荣耀值。若该操作后你的荣耀值为负则不能进行该操作。

当队列中不再存在对手团队时斗舞赛结束。那么如果使用最优决策链，最终可以获得的最大荣耀值是多少？

输入：

第一行一个整数T为测试样例数量，后跟T个测试样例。每个测试样例包含两行，第一行两个整数E和N分别表示你团队的初始能量点和对手团队数量。第二行N个整数S_i，其中第i个表示初始队列中第i个团队的舞技值。

输出：

对于每个测试样例，输出一行包含$Case\ \#x: y$，其中x是从1开始的测试样例序号，y是采用最优决策链时斗舞结束时的最大荣耀值。

参数限制：

$1 \leq T \leq 100$

$1 \leq E \leq 10^6$

$1 \leq S_i \leq 10^6$

小数据集：

$1 \leq N \leq 5$

大数据集：

$1 \leq N \leq 1000$

测试样例：

Input	Output
2 100 1 100 10 3 20 3 15	Case #1: 0 Case #2: 1

测试样例 1 中，只有一个对手团队，其舞技值和你团队的能量点相同，无法斗舞。又由于初始荣耀值为 0 故无法招募。延期操作对状态没有任何改变。故唯一的选择是停战，最终荣耀值为 0。

测试样例 2 中，一个可选的决策链如下：

1. 让第一个团队延期，第一个对手团队移至队尾。
2. 和第二个团队斗舞，能量点降至7，荣耀值升至1。
3. 招募第三个团队，能量点升至22，荣耀值降至0。
4. 和第一个团队（此刻又出现在对头）斗舞，能量点降至2，荣耀值升至1。

最终荣耀值为1。

WeiYong1024 的线性时间解:

首先发现：由于延期操作是对队列的循环左移，且最终只要求最大荣耀值而非具体决策链。故利用延期操作可以以任意顺序对队列中的任意对手团队做任意操作。

基于上述发现，原队列中元素的排列顺序不再重要。我们思考以怎样的方式处理队列可以使最终荣耀值达到最大。很容易发现以下几条准则：

· 若能斗舞，则和当前队列中舞技值最小的团队斗舞以增长荣耀值。

· 若不能斗舞，则尝试招募队列中舞技值最大的团队，并且招募的数量要尽可能少，以最小化荣耀值消耗。

· 若不能斗舞且不能招募（荣耀值为0），则剩下的团队只能停战。

基于上述三条准则，给出如下算法：

Step1：将N个对手按照舞技值从小到大排序。

Step3：按照从易到难的顺序和当前队列中的团队斗舞，直到所剩能量点不足以和队列中舞技值最小的团队斗舞，更新最大荣耀值。

Step3：若当前荣耀值大于零，则招募一个当前队列中舞技值最大的团队，回到 Step2；否则结束算法（因为剩下的团队只能停战）。

时间复杂度$T\sim O(N\log N)$

完整代码：

```
#include <algorithm>
#include <iostream>

const int maxn = 1005;
const int INF = 1e9;

int s[maxn];
int e, n;
void read() {
    scanf("%d %d", &e, &n);
    for (int i = 0; i < n; i++)
        scanf("%d", &s[i]);
}

int solve() {
    std::sort(s, s + n);
    int l = 0, r = n - 1;
    int honor = 0;
    int mx = -INF;
    while (1) {
        while (e > s[l] && l <= r) {
            e -= s[l];
            honor++;
            l++;
        }
        mx = std::max(mx, honor);
        if (honor > 0) {
            honor--;
            e += s[r];
            r--;
```

```
                if (l >= r) break;
            } else break;
        }
        return mx;
}

int main() {
        freopen("blin.txt", "r", stdin);
        freopen("blout.txt", "w", stdout);
        int T;
        scanf("%d", &T);
        for (int t = 1; t <= T; t++) {
            read();
            int ans = solve();
            printf("Case #%d: %d\n", t, ans);
            fprintf(stderr, "Case #%d: %d\n", t, ans);
        }
}
```

3.6 Kickstart 2017 Round F

C. Catch Them All (动态规划，概率论，Floyd 最短路算法)

题述：

Codejamon Go 发布后，你像朋友们一样，走上城市的街道去捕获尽可能多的毛茸茸的小精灵，这个游戏的目标就是到达城市中所有 Codejamon 出现的位置并捕获它们。你好奇把它们都捉住需要多长时间。

你所居住的城市由N个区域构成，编号1到N。你初始位于1号区域，有M条双向路径，编号1到M。其中第i条路连接两个不同区域(U_i, V_i)，在两个方向上通过该条路径用时都是D_i分钟。拓扑保证从1号区域开始通过若干路径可以到达任何其他区域。

在0时刻，一个 Codejamon 以均匀概率出现在除1号区域外的其他某个区域。均匀概率意味着 Codejamon 位于除1号区域外其他$N-1$个区域中任何一个的概率都是$\frac{1}{N-1}$。一旦 Codejamon 出现，你就可以立刻前往捕获之。当你到达一个 Codejamon 所在的区域，你就可以立刻捉住它，随之下一个 Codejamon 就会以均匀概率随机出现在除当前区域以外其他$N-1$个区域中的任意一个，以此类推。注意在任何时刻有且只有一个 Codejamon，而且你必须在捕获当前 Codejamon 之后才能去捕获下一个。

给定你所居住的城市布局，请计算捕获P个 Codejamon 的期望时间，注意在任意两个区域之间移动时你总是走最短路径。

输入：

第一行一个整数T代表测试样例数量，随后是T个测试样例。

每个测试样例开始一行包含三个整数N, M, P分别表示区域数、道路数和待捕获 Codejamon 的数量。

接下来每个测试样例包含M行。其中第i行三个整数U_i, V_i, D_i分别表示第i条路径所连接的两个区域U_i和V_i，以及从两个方向通过这条路径需要D_i分钟时间。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y是捕获P个 Codejamon 的期望分钟数。所提交答案的绝对或相对误差在10^{-4}之内时被视作正确。FAQ 这里有关于我们接受实数格式的解释。

参数限制：

$1 \le T \le 100$.

$N - 1 \le M \le \frac{N(N-1)}{2}$.

对所有i，$1 \le D_i \le 10$.

对所有i，$1 \le U_i, V_i \le N$.

对所有$i \ne j$的i和j有$U_i \ne U_j$且 / 或$V_i \ne V_j$。（每对区域之间最多有1条路径）

输入保证从 1 号区域出发经过若干条路径可以到达任意区域。

小数据集：

$2 \le N \le 50$.

$1 \le P \le 200$.

大数据集：

$2 \le N \le 100$.

$1 \le P \le 10^9$.

测试样例：

Input	Output
4 5 4 1 1 2 1 2 3 2	Case #1: 2.250000 Case #2: 1000.000000 Case #3: 5.437500 Case #4: 1.500000

1 4 2 4 5 1 2 1 200 1 2 5 5 4 2 1 2 1 2 3 2 1 4 2 4 5 1 3 3 1 1 2 3 1 3 1 2 3 1	

测试样例 1 中，逮捕获的 Codejamon 数量只有1个。它以均匀概率出现在2,3,4,5号区域，与1号区域的距离分别为1,3,2,3，故捕获该 Codejamon 的期望时间为$\frac{1+3+2+3}{4}$

测试样例 2 中，只有一条路径连接两个区域。每次 Codejamon 出现都在你所在区域对面的那个，我们就要经过这条唯一的路径去捕获之。故我们经过这条路径200次，每次5分钟，一共1000分钟。

测试样例 3 中，地图和测试样例 1 相同。两个 Codejamon 出现位置的有序数对共有16种可能，经过计算得到其期望为$\frac{87}{16} = 5.4375$分钟。

测试样例 4 中，待捕获的 Codejamon 等可能地出现在2号或3号区域，最短路径长度分别为2和1，平均用时1.5分钟。

WeiYong1024 基于状态转移矩阵的动态规划解:

首先，所有区域两两之间的最短路径可以由 Floyd 算法在$T\sim O(N\text{^}3)$得到。

我们要求随机均匀P步移动的距离期望，由期望的定义，若将每一步移动的目标视为一次决策，则应求得所有长度为P的决策链时长的均值。然而长度为P的决策链数量为$(N-1)^P$，用晶体管计算机来遍历跟开玩笑一样。于是思考问题的递推性质（寻找动态规划解）。

注意到P长决策链中，其中任意一个决策都可能是$(N-1)*(N-1)$条路径中的一条，那么该决策的耗时期望为$E[t] = \sum_{i=1}^{N}\sum_{j=1}^{N} p_{ij} * sp_{ij}$。其中$sp_{ij}$为从区域$i$到区域$j$最短路径的长度，而$p_{ij}$则是该决策为$(i,j)$的概率。最短路径由 Floyd 算法一次性求出，那么接下来的问题是如何求解p_{ij}呢？

设第u次决策前的区域为s_u，$p_{ij}^u = p((s_u, s_{u+1}) = (i,j)) = p(s_u = i) * p(s_{u+1} = j | s_u = i)$。其中$p(s_{u+1} = j | s_u = i) = \begin{cases} 0, & j = i \\ \frac{1}{N-1}, & j \neq i \end{cases}$，而$p(s_u = i)$可以由初始状态和状态转移概率计算得到。具体地，初始状态为$p_s^{(1)} = [1,0,0,\dots,0]$，状态转移矩阵为$P_t = \begin{bmatrix} 0 & \frac{1}{N-1} & \cdots & \frac{1}{N-1} \\ \frac{1}{N-1} & 0 & \cdots & \frac{1}{N-1} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{1}{N-1} & \frac{1}{N-1} & \cdots & 0 \end{bmatrix}$。由移动概率的均匀性，有$p_s^{(u)} = p_s^{(1)} \times P_t^{u-1}$。

另一方面，当移动步数超过100以后，状态概率趋于稳态，实验证明第200步与第199步的增量差异扩大10^9倍仍然满足10^{-4}的绝对精度要求。

整个算法的时间复杂度为$T\sim O(TPN^2)$。

完整代码:

```
#include <vector>
```

```
#include <iostream>

using ll = long long;

const int maxn = 105;
const int INF = 1e9;

double sp[maxn][maxn];
int n, m, p;
void read_get_sp() {       // Input data and get shortest pathes using Floyd.
      for (int i = 0; i < maxn; i++)
            for (int j = 0; j < maxn; j++)
                  sp[i][j] = INF;
      for (int i = 0; i < maxn; i++)
            sp[i][i] = 0;

      scanf("%d %d %d", &n, &m, &p);
      for (int i = 0; i < m; i++) {
            int u, v;
            double d;
            scanf("%d %d %lf", &u, &v, &d);
            u--, v--;
            sp[u][v] = sp[v][u] = d;
      }

      for (int k = 0; k < n; k++)
            for (int i = 0; i < n; i++)
                  for (int j = 0; j < n; j++)
                        if (sp[i][j] > sp[i][k] + sp[k][j]) sp[i][j] = sp[i][k] + sp[k][j];
}

double pt[maxn][maxn];   // State-transfering-matrix.
double ps[maxn];     // State-prob-vector.
void initialization() { // Get state-transfering-matrix pt ans set value for state-prob-vector.
      for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                  if (i != j) pt[i][j] = (double)1 / (n - 1);
                  else pt[i][j] = 0;
      memset(ps, 0, sizeof(ps));
      ps[0] = 1;
}

double ps_tmp[maxn];
void upd() {       // ps = ps * pt.
      for (int i = 0; i < n; i++)
```

```
            ps_tmp[i] = ps[i];
        memset(ps, 0, sizeof(ps));
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                ps[i] += ps_tmp[j] * pt[j][i];
}

double get_ans(int p) { // Calculate exception time of p steps under condition p <= 200.
        initialization();     // Get state-transfering-matrix.

        double ans = 0;
        for (int i = 0; i < p; i++) {
            for (int j = 0; j < n; j++)
                for (int k = 0; k < n; k++)
                    ans += sp[j][k] * ps[j] / (n - 1);
            upd();
        }

        return ans;
}

double solve() {
        if (p <= 200) return get_ans(p);
        else {
            double ans199 = get_ans(199);
            double ans200 = get_ans(200);
            double ans = ans200 + (ans200 - ans199) * (p - 200);
            return ans;
        }
}

int main() {
        freopen("clin.txt", "r", stdin);
        freopen("clout.txt", "w", stdout);
        int T;
        scanf("%d", &T);
        for (int t = 1; t <= T; t++) {
            read_get_sp();
            double ans = solve();
            printf("Case #%d: %.6lf\n", t, ans);
            fprintf(stderr, "Case #%d: %.6lf\n", t, ans);
        }
}
```

3.6 Kickstart 2017 Round F

D. Eat Cake (预处理)

题述：

Wheatley 来到了世界上最棒的 Party：这里有吃不完的蛋糕！每块蛋糕都被做成边长为整数（厘米）的正方形。Party 上所有尺寸的蛋糕都不限量供应，所有蛋糕的高度都是一样的，所以我们只关心蛋糕的面积。

Wheatley 计划吃掉总面积为Ncm² 的蛋糕。由于他是一个健身爱好者，他希望吃掉尽量少的蛋糕。你能帮助他计算一下最少需要吃掉几块蛋糕吗？

输入：

第一行一个整数T为测试样例数量，接下来是T个测试样例，每个测试样例包含一个整数N，表示他计划吃掉的蛋糕的总面积。

输出：

对于每个测试样例输出一行 Case #x: y，其中x是从1开始的测试样例序号，y是 Wheatley 为达到总共Ncm² 的面积需要吃掉的最少蛋糕数。

参数限制：

小数据集：

$1 \le T \le 50$

$1 \le N \le 50$

大数据集：

$1 \le T \le 100$

$1 \le N \le 10000$

测试样例：

Input	Output
3 3 4 5	Case #1: 3 Case #2: 1 Case #3: 2

测试样例 1 中，Wheatley 唯一的策略就是吃掉三块边长为1的蛋糕。

测试样例 2 中，Wheatley 可以吃掉一块边长为2的蛋糕。这比吃掉四块边长为1的蛋糕更优。

测试样例 3 中，最优策略是吃掉一块边长为2的蛋糕和一块边长为1的蛋糕。

WeiYong1024 的生成答案表的算法:

问题抽象为函数$f(x)$——将x拆成数量最少的平方数和，$f(x)$是最少平方数的数量。其中x为整数且取值在$[1, 10000]$之间。

生成函数取值表，在计算$f(x)$时假设已经求出$f(1) \dots f(x-1)$，那么对于$f(x)$，如果 x 是完全平方数，则$f(x) = 1$。否则，$f(x)$的值为$f(i) + f(x-i), i = 1, \dots x-1$的最小值。

算法复杂度$T \sim O(N^2 + T)$

完整代码:

```
#include <cmath>
#include <iostream>

const int maxn = 10005;
const int INF = 1e9;

int f[maxn];
void pre_process() {
```

```
	f[1] = 1;
	for (int i = 2; i <= 10000; i++) {
		int tmp = sqrt(i);
		if (i == tmp * tmp) {
			f[i] = 1;
			continue;
		}

		int mn = INF;
		for (int j = 1; j <= i / 2; j++)
			if (mn > f[j] + f[i - j]) mn = f[j] + f[i - j];
		f[i] = mn;
	}
}

int n;
void read() {
	scanf("%d", &n);
}

int solve() {
	return f[n];
}

int main() {
	freopen("dlin.txt", "r", stdin);
	freopen("dlout.txt", "w", stdout);
	int T;
	scanf("%d", &T);
	pre_process();
	for (int t = 1; t <= T; t++) {
		read();
		int ans = solve();
		printf("Case #%d: %d\n", t, ans);
		fprintf(stderr, "Case #%d: %d\n", t, ans);
	}
}
```

3.7 Kickstart 2017 Round G

A. Huge Numbers (大数计算，快速幂)

题述：

今天 Shekhu 教授给 Akki 提出了一个问题。他给了 Akki 三个正整数A, N和P，并让他计算$A^{N!}$模P的余数。如平常所见，$N!$代表前N个正整数的乘积。

输入：

第一行一个整数T为测试样例数量，接下来是T个测试样例。每行包含三个整数A, N和P，意义如前所述。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y为答案。

参数限制：

$1 \leq T \leq 100$.

小数据集：

$1 \leq A \leq 10$.

$1 \leq N \leq 10$.

$1 \leq P \leq 10$.

大数据集：

$1 \leq A \leq 10^5$.

$1 \leq N \leq 10^5$.

$1 \leq P \leq 10^5$.

测试样例：

Input	Output
2 2 1 2 3 3 2	Case #1: 0 Case #2: 1

测试样例 1 中，答案为$2^{1!}\%2 = 2\%2 = 0$。

测试样例 2 中，答案为$3^{3!}\%2 = 3^6\%2 = 729\%2 = 1$。

Author 的算法:

乍一看这道题可能需要一个数论算法。但如果用快速幂算法则很快可以在$T \sim O(N\log N)$内完成。首先，模P意义下A^N利用快速幂算法可以在$T \sim O(\log N)$时间内完成，然后只需迭代计算（$A^{N!} = (A^{(N-1)!})^N$）指数阶乘即可。

完整代码:

```
#include <cstdio>

int A, N, P;
void read() {
    scanf("%d %d %d", &A, &N, &P);
}

int pw(int a, int b) {   // Fast exponentiation: a^b.
    int ans = 1;
    while (b) {
        if (b & 1) ans = 1ll * ans * a % P;
        a = 1ll * a * a % P;
        b >>= 1;
    }
```

```
	return ans;
}

int solve() {
	int ans = A;
	for (int i = 1; i <= N; i++)
		ans = pw(ans, i);
	return ans;
}

int main() {
	freopen("alin.txt", "r", stdin);
	freopen("alout.txt", "w", stdout);
	int T;
	scanf("%d", &T);
	for (int t = 1; t <= T; t++) {
		read();
		int ans = solve();
		printf("Case #%d: %d\n", t, ans);
		fprintf(stderr, "Case #%d / %d: %d\n", t, T, ans);
	}
}
```

3.7 Kickstart 2017 Round G

B. Cards Game (博弈策略，贪心法，最小生成树)

题述：

Shekhu 教授是计算机科学领域早期研究博弈论的著名科学家。现在，Shekhu 教授正在开发一种新的游戏。具体地，箱子中有N张卡片，其中每一张卡片在其正反两面分别印有一个蓝色的数字和一个红色的数字，这两个数字都是正整数。游戏规则如下：

- 玩家一开始积分为0，游戏目标为以最小积分结束。
- 只要箱子中的卡片数量大于等于2，玩家就要重复如下步骤：
  - 从箱子中取出两张卡片，选择一张卡片的蓝色数字B和另一张卡片的红色数字R。
  - 将R^B的值加入积分，其中^是按位亦或。
  - 将一张卡片放回箱子中，丢弃另一张卡片。
- 当箱子中只剩一张卡片的时候游戏结束（此时无法再进行上述步骤了）。

Shekhu 教授叫他最优秀的学生 Akki 来解决这个博弈问题。考虑游戏所有可能的进行方式，你能帮助 Akki 找到最小可能的积分吗？

输入：

第一行一个整数T为测试样例数量，接下来是T个测试样例。每个测试样例包含三行：

1. 第一行为一个正整数N：箱子中初始卡片数量。
2. 第二行包含N个正整数，其中第i个整数R_i代表印在第i张卡片上的红色数字。
3. 第二行包含N个正整数，其中第i个整数B_i代表印在第i张卡片上的蓝色数字。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y是 Akki 采用最优策略能够得到的最小积分。

参数限制：

$1 \leq T \leq 100$.

$1 \leq R_i \leq 10^9$.

$1 \leq B_i \leq 10^9$.

小数据集：

$2 \leq N \leq 5$.

大数据集：

$2 \leq N \leq 100$.

测试样例：

Input	Output
2 2 1 2 3 3 3 1 101 501 3 2 3	Case #1: 1 Case #2: 5

测试样例 1 中，Akki 只会进行一步，有如下两种选择：

1. 选择第一张卡片上的红色数字和第二张卡片上的蓝色数字，最终积分为1^3 = 2。
2. 选择第一张卡片上的蓝色数字和第二张卡片上的红色数字，最终积分为2^3 = 1。

其中第 2 种策略更优，最终结果为1。

测试样例 2 中，一种最优的策略为：首先取第一张卡片上的红色数字和第二张卡片上的蓝色数字，得到积分1^2 = 3，并将第一张卡片放回箱中。然后取第一张卡片上的红色数字和第三张卡片上的蓝色数字，讲1^3 = 2加入积分，并将任意一张卡片放回箱中，最终积分为5。

Author 的算法:

对于小数据集卡片数量N不超过5，可以采用**纯模拟**的方法。遍历所有可能的游戏路径找到最优解，C++代码如下。

核心代码:

```
std::vector<int> remove(std::vector<int> &v, int r) {    // Return a copy of v after removing the r^th element.
    std::vector<int> ans;
    ans.reserve(v.size());
    for (int i = 0; i < v.size(); i++) {
        if (i == r) continue;
        ans.emplace_back(v[i]);
    }
    return ans;
}

int game(std::vector<int> r, std::vector<int> b) {   // Return answer for input arrays r and b.
    if (r.size() <= 1) return 0;
    int l = (int)r.size();
    int ans = INF;
    for (int i = 0; i < l; i++)
        for (int j = i + 1; j < l; j++) {
            int tmp = std::min(r[i] ^ b[j], r[j] ^ b[i]);
            tmp += std::min(game(remove(r, i), remove(b, i)), game(remove(r, j), remove(b, j)));
            ans = std::min(ans, tmp);
        }
    return ans;
}
```

上述算法的复杂度为$T\sim O(2^N \cdot N^3)$。然而，通过维护一个 bitmask 而不是整个序列作为 game()函数传递的参数，时间复杂度可以将至$T\sim O(2^N \cdot N^2)$。

上述算法可以通过小数据集。对于大数据集需要更高效的算法。注意到，卡片之间的消除操作具有以下性质:

· 每次消除操作涉及到两张卡片——一去一留;

· 有且仅有$N-1$次消除操作;

· 整个过程每张卡片都被比较过。

以上描述和**无向树**的定义多么相似！我们大胆作出以下启发式假设：N张卡片构成N个顶点，每两张卡片之间的边权重为两种数字异或结果中的较小值$\min(R_i \text{^} B_j, R_j \text{^} B_i)$。那么如果我们找到连接这$N$个顶点的一颗最小生成树（MST），我们就可以按照下述规则的到一个合法的操作序列:

规则: 从最小生成树开始，每次去掉一个只连接一条边的顶点，并将因此消失的边的权重累加，则最终结果就是最小生成树的总权重。

进一步，所有的最小生成树及其根据上述规则所催生的所有操作序列就是所有合法的最优操作序列了。

复杂度方面，最小生成树问题的最优复杂度$T\sim O(E\log V)$，进而有$T\sim O(N^2\log N)$。由于原图全连通，下面给出利用邻接矩阵复杂度为$T\sim O(V^2)=O(N^2)$ 的 MST 经典实现。

完整代码:

```
#include <algorithm>
#include <cstdio>

using ll = long long;

const int INF = 2e9;
const int maxn = 105;

int n;
int R[maxn], B[maxn];
void read() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++)
		scanf("%d", &R[i]);
	for (int i = 0; i < n; i++)
		scanf("%d", &B[i]);
}

int G[maxn][maxn];

int next_vertex(int low_cost[], bool mst_set[]) {
	int mn = INF, mn_idx = -1;
	for (int i = 0; i < n; i++)
		if (!mst_set[i] && low_cost[i] < mn)
			mn = low_cost[i], mn_idx = i;
	return mn_idx;
}

ll prim_MST() {
	int par[n];   // Store which inner vertex to connect for outer vertexes.
	int low_cost[n]; // lowest cost for outer vertexes to connect to a inner vertex.
	bool mst_set[n]; // Mark inner vertices.

	// Initially put 1st vertex into mst_set.
	mst_set[0] = true, par[0] = -1;
	for (int i = 1; i < n; i++)
		low_cost[i] = G[0][i], mst_set[i] = false, par[i] = 0;

	// There will be n - 1 steps constructing MST.
	for (int i = 0; i < n - 1; i++) {
		int u = next_vertex(low_cost, mst_set);
		mst_set[u] = true;
		for (int v = 0; v < n; v++)
			if (!mst_set[v] && G[u][v] < low_cost[v])
```

```
                par[v] = u, low_cost[v] = G[u][v];
        }

        ll ans = 0;
        for (int i = 1; i < n; i++)
                ans += G[i][par[i]];
        return ans;
}

ll solve() {
        for (int i = 0; i < n; i++)
                G[i][i] = INF;
        for (int i = 0; i < n; i++)
                for (int j = i + 1; j < n; j++)
                        G[i][j] = G[j][i] = std::min(R[i] ^ B[j], R[j] ^ B[i]);
        return prim_MST();
}

int main() {
        freopen("blin.txt", "r", stdin);
        freopen("blout.txt", "w", stdout);
        int T;
        scanf("%d", &T);
        for (int t = 1; t <= T; t++) {
                read();
                ll ans = solve();
                printf("Case #%d: %lld\n", t, ans);
                fprintf(stderr, "Case #%d/%d: %lld\n", t, T, ans);
        }
}
```

3.7 Kickstart 2017 Round G

C. Matrix Cutting (博弈策略，动态规划，深搜+备忘)

题述：

Shekhu 教授手头有一个N行M列的矩阵，其中行从上向下从0到$N-1$编号，列从左到右从0到$M-1$编号。每个矩阵单元包含一个正整数。

他希望通过水平和竖直切割将这个矩阵分割成$N*M$个$1*1$的小矩阵，每一刀只沿水平的两行或竖直的两列之间切割。

Shekhu 教授请他最优秀的学生 Akki 来解决这个问题，他提出了一个有趣的要求：每次 Akki 在一个子矩阵中切一刀之前，他就会获得那个子矩阵中最小值的积分。注意，每切一刀，子矩阵的数量都会增加。另外，所切割的一刀在经过的所有子矩阵中分别累加积分，Akki 在每个子矩阵切割的过程中独立获得积分。

现在 Akki 有很多方法来进行切割，你能帮他最大化最终得分吗？

输入：

第一行一个整数T为测试样例数量，后跟T个测试样例。对于每个测试样例，第一行为两个整数N和M，意义如前所述。接下来的N行每行包含M个正整数，表示矩阵。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从一开始的测试样例序号，y是 Akki 在采取最优策略的情况下可以获得的最大积分。

参数限制：

$1 \le T \le 100$.

1 ≤矩阵元素≤ 10^5.

小数据集：

$N = 1$

$1 \le M \le 10$

大数据集：

$1 \le N \le 40$

$1 \le M \le 40$

测试样例：

Input	Output
3 2 2 1 2 3 4 2 3 1 2 1 2 3 2 1 2 1 2	Case #1: 5 Case #2: 7 Case #3: 1

测试样例 1 中，Akki 有两种切割方法：

1. 先横切一刀，积分为1，再在两个子矩阵（$[1,2],[3,4]$）中分别竖切一刀，分别积分1和3。
2. 先竖切一刀，积分为1，再在两个子矩阵（$[1,3]^T,[2,4]^T$）中分别横切一刀，分别积分 1 和 2。

第一种策略更好，积分最大值为5。

测试样例 2 中，Akki 最多可以得到7分。一种方法为：先横切一刀，积分为1。然后在上方子矩阵$[1,2,1]$中，先在第一列右侧切一刀积1分，然后在第二列右侧切一刀，积1分。类似地，对于下方子矩阵$[2,3,2]$，Akki 首先在第一列右侧切一刀积2分，然后在第二列右侧切一刀积1分。总共积7分。

测试样例 3 中，只能切一刀。

Author 的算法:

问题本身具有递归属性——很显然，我们定义一个函数返回任意一个由（左上，右下）元胞对定义的子矩阵所能获得的最大积分。那么对于任意一个矩阵求解其上的最大积分，只要它不是不可切割的（即1×1），则它的答案为：对于所有切割位置，被切开的两个子矩阵上最大积分和的最大值，加上矩阵本身的最小元素。而对于1×1矩阵，其上无法再因切割获得积分，返回零即可，这也是递归问题的探底条件。

显然，原始问题的递归求解过程存在大量的重复计算，故用一个四维备忘录加速。

复杂度：$T/S\sim O(N^2M^2)$

完整代码:

```
#include <algorithm>
#include <cstdio>

const int INF = 2e9;
const int maxn = 45;
const int maxm = 45;

int n, m;
int mat[maxn][maxm];
void read() {
    scanf("%d %d", &n, &m); // Read matrix's dims.
    for (int i = 0; i < n; i++) // Read matrix.
        for (int j = 0; j < m; j++)
            scanf("%d", &mat[i][j]);
}

int dp[maxn][maxm][maxn][maxm]; // dp[x1][y1][x2][y2] records the max score in sub-matrix with left-up cell (x1, y1) ans right-down cell (x2, y2).
int max_score(int x1, int y1, int x2, int y2) {
    if (dp[x1][y1][x2][y2] != -1) return dp[x1][y1][x2][y2];      // If answer was already calculated.
    if (x1 == x2 && y1 == y2) return 0; // If there is only one cell (therefore no cut can be made).

    int mn = INF;
    for (int i = x1; i <= x2; i++)
        for (int j = y1; j <= y2; j++)
            mn = std::min(mn, mat[i][j]);
    int ans = 0;
    for (int xm = x1; xm < x2; xm++)      // Cut vertically.
        ans = std::max(ans, max_score(x1, y1, xm, y2) + max_score(xm + 1, y1, x2, y2));
    for (int ym = y1; ym < y2; ym++)     // Cut horizontally.
        ans = std::max(ans, max_score(x1, y1, x2, ym) + max_score(x1, ym + 1, x2, y2));
    ans += mn;

    return dp[x1][y1][x2][y2] = ans;
}
```

```
int solve() {
	for (int x1 = 0; x1 < n; x1++)
		for (int y1 = 0; y1 < m; y1++)
			for (int x2 = 0; x2 < n; x2++)
				for (int y2 = 0; y2 < m; y2++)
					dp[x1][y1][x2][y2] = -1;
	return max_score(0, 0, n - 1, m - 1);
}

int main() {
	freopen("clin.txt", "r", stdin);
	freopen("clout.txt", "w", stdout);
	int T;
	scanf("%d", &T);
	for (int t = 1; t <= T; t++) {
		read();
		int ans = solve();
		printf("Case #%d: %d\n", t, ans);
		fprintf(stderr, "Case #%d / %d: %d\n", t, T, ans);
	}
}
```

3.8 Kickstart 2018 Round A

A. Even Digits

题述：

Supervin 有一个独特的计算器，这个计算器只有一个显示器，一个加一按钮和一个减一按钮。此时此刻显示器上的数字为N。

每按一次加一按钮会让显示器上的数字增加1，减一按钮的功能类似，数字不显示前缀零。例如，若当前现实数字为100，则按一次减一按钮后数字将变成99。

Supervin 不喜欢奇数，因此，他希望显示器上的数字全部由偶数数字构成。他希望能通过最少的按键操作达到这个状态。

请帮助 Supervin 计算使得当前现实的数字变成全偶数构成所需的操作数量。

输入：

输入第一行一个整数T表示测试样例的数量，接下来是T个测试样例。对于每个测试样例，用一行包含一个整数N表示当前显示器上的数字。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y是使显示器上的数字全部为偶数所需的最小操作步数。

参数限制：

$1 \le T \le 50$

小数据集：

$1 \le N \le 10^5$

大数据集：

$1 \le N \le 10^{16}$

测试样例：

Input	Output
4 42 11 1 2018	Case #1: 0 Case #2: 3 Case #3: 1 Case #4: 2

对于测试样例 1，当前数字全部为偶数构成，无需按键。

对于测试样例 2，连按三次减一按钮将数字变成8，是最好的方式。

对于测试样例 3，一次加一变成2或一次减一变成0均可。

对于测试样例 4，连按两次加一按钮将数字变成2020，是最好的方式。

weiyong1024 的纯数学算法：

本题的问题重述如下：给定一个正整数N，求N与和它最近的全部由偶数数字构成的数的距离。

显然，与N最近的全部为偶数的数字与N的最高奇数位有关，设10^p位是最高奇数位，分别分析比N大的第一个合法数字和比N小的第一个合法数字有什么规律。

N的下一个全部由偶数组成的数字为其最高奇数为加一后面全是零构成，比654321大的第一个合法数字为660000。注意到如果最高奇数位是9，则情况比较特殊，但可以知道大于其的第一个合法数字与之距离必然距离大于10^{p+1}，因为至少需要让10^{p+1}位原来的数字（可能是0）加2，显然不如比N小的第一个合法数字距离更近。

比N小的第一个合法数字为10^p位减一，后续所有低位为8的情形，不加证明。

算法复杂度：$T \sim O(lgN)$。

完整代码：

```
#include <algorithm>
```

```
#include <vector>
#include <cstdio>

using ll = long long;

ll n;
void read() {
	scanf("%lld", &n);
}

ll qpw(ll base, ll idx) {
	ll ans = 1;
	while (idx) {
		if (idx & 1) ans *= base;
		base *= base;
		idx >>= 1;
	}
	return ans;
}

std::vector<ll> v;
ll solve() {
	v.clear();
	ll tmp = n;
	while (tmp != 0) {
		v.emplace_back(tmp % 10);
		tmp /= 10;
	}
	std::reverse(v.begin(), v.end());

	int p = -1;
	for (int i = 0; i < v.size(); i++)
		if (v[i] & 1) {
			p = i;
			break;
		}
	if (p == -1) return 0;   // Already all even.
	if (p == v.size() - 1) return 1;       // The only odd num is the lastest bit.

	ll base = 0;
	for (int i = p; i < v.size(); i++)
		base += v[i] * qpw(10, v.size() - 1 - i);
	ll below = (v[p] - 1) * qpw(10, v.size() - 1 - p);
	for (int i = p + 1; i < v.size(); i++)
		below += 8 * qpw(10, v.size() - 1 - i);
```

```
    if (v[p] == 9) return base - below;

    ll above = (v[p] + 1) * qpw(10, v.size() - 1 - p);
    return std::min(above - base, base - below);
}

int main() {
    freopen("alin.txt", "r", stdin);
    freopen("alout.txt", "w", stdout);

    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        ll ans = solve();
        printf("Case #%d: %lld\n", t, ans);
        fprintf(stderr, "Case #%d / %d: %lld\n", t, T, ans);
    }
}

// Notice, float point error will make pow() not working on big dateset.
```

注意:

长整形的幂运算用快速幂自己实现，如使用 cmath 中的 pow 函数会导致浮点误差累计过1。

3.8 Kickstart 2018 Round A

B. Lucky Dip（概率 DP）

题述：

你在参加 Kickstart 幸运大抽奖活动，奖品很有诱惑力。

在这次大抽奖中，奖池中有N件奖品，其中第i件的价值为V_i，你将手伸入奖池随机抽出一件，奖池中所有奖品被抽出的可能性相同。为了增加抽奖过程的戏剧性，主办方制定如下规则：当你拿出一个奖品，你可以选择留下它或者放回重新抓取，你一共有K次放回的机会，一旦K次达到，你将必须保留第$K+1$次抽到的奖品。

如果你使用最佳策略，那么你最终获得奖品价值的期望是多少？

输入：

输入第一行为一个整数T表示测试样例的数量，后跟T个测试样例。

每个测试样例包含两行，第一行两个整数N和K分别代表奖品的数量和允许放回的最大次数。第二行N个整数表示每个商品的价格。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y是最终奖品价值的期望。绝对或相对误差上限为10^{-6} 。

参数限制：

$1 \le T \le 100$

$1 \le V_i \le 10^9$

$1 \le N \le 20000$

小数据集：

$0 \le K \le 1$

大数据集：

$1 \le K \le 50000$

测试样例：

Input	Output
5 4 0 1 2 3 4 3 1 1 10 1 3 15 80000 80000 80000 1 1 10 5 3 16 11 7 4 1	Case #1: 2.500000 Case #2: 6.000000 Case #3: 80000.000000 Case #4: 10.000000 Case #5: 12.358400

测试样例 1 中，不能放回，故期望为商品价值的平均值。

测试样例 2 中，最好的策略是如果抽到 10 则留下，否则放回。

测试样例 3 中，所有的商品价值相同，故期望也是这个数。

测试样例 4 不会出现在小数据集中。

weiyong1024 的概率 DP 算法：

K次有放回抽奖的价值期望，显然是一个概率 DP 问题。直观上看放回的机会越多，最终拿到的奖品价值期望越高。

对于还剩K次放回机会的抽取，最优策略如下：如果抽到的奖品价值大于有$K-1$次放回机会的价值期望E_{K-1}，则留下，否则继续抽。于是还剩K次放回机会时最终得到商品的期望如下：

$$E_K = P(\text{抽到奖品价值} > E_{K-1}) * \text{价值大于}E_{K-1}\text{的奖品均价} + P(\text{抽到奖品价值} \le E_{K-1}) * E_{K-1}$$

若将奖品价值降序排列，然后计算前缀均值，则可以在$O(lgN)$的时间内计算每次迭代。故总复杂度$T\sim O(KlgN)$

完整代码：

```
#include <algorithm>
#include <cmath>
#include <cstdio>

const int maxn = 20005;
const double eps = 1e-10;

double v[maxn];
int n, k;
void read() {
    scanf("%d %d", &n, &k);
    for (int i = 0; i < n; i++)
        scanf("%lf", &v[i]);
}

double v_avg[maxn];

int get_idx(double base) { // Return the last idx i that v[i] > base.
    int l = 0, r = n - 1;
    while (l < r - 1) {
        int m = (l + r) >> 1;
        if (v[m] > base) l = m;
        else r = m;
    }
    return l;
}

double get_exp(int k) {
    if (k == 0) return v_avg[n - 1];
    double exp_km1 = get_exp(k - 1);
    int idx = get_idx(exp_km1);
    return v_avg[idx] * (double(idx - 0 + 1) / n) + exp_km1 * (double(n - idx - 1) / n);
}

double solve() {
    std::sort(v, v + n);
    std::reverse(v, v + n);
    double sum = 0;
    for (int i = 0; i < n; i++) {
        sum += v[i];
```

```
            v_avg[i] = sum / (i + 1);
        }

        return get_exp(k);
}

int main() {
        freopen("blin.txt", "r", stdin);
        freopen("blout.txt", "w", stdout);

        int T;
        scanf("%d", &T);
        for (int t = 1; t <= T; t++) {
            read();
            double ans = solve();
            printf("Case #%d: %.6lf\n", t, ans);
            fprintf(stderr, "Case #%d / %d: %.6lf\n", t, T, ans);
        }
}
```

注：虽然将本体分类为概率 DP，但实质上最优子结构中只有一个重叠子问题。

3.8 Kickstart 2018 Round A

C. Scrambled words（频次统计，前缀和统计，模式集合）

题述：

Scramble 教授在批阅论文的时候发现了一些错位词，但她同时发现这些错位词并不影响文章理解。经过一些调研之后，她发现了一篇 article 有如下内容：

英国某大学一项研究表明，单词中字幕的顺序并不重要。只要首字母和尾字母在正确的位置上，就算其他的字母完全乱序也不影响你阅读文章。这是因为人脑在阅读文章的时候并不关注每个字母，而是将单词当作一个整体。

Scramble 教授想进一步研究这个问题，她开始编辑一些包含相似错位词的句子并计划投稿给行业期刊。很不幸，由于一次误操作她删掉了文本中的所有空格。她请求你帮助她统计有多少字典中的单词以子串或错位子串的形式出现在了这个长字符串中（其中错位子串的含义为字符集合相同，首尾字符不变，中间字符以任意顺序排列）。

作为 Scramble 教授最得意的学生，你能帮助她统计有多少字典中的单词作为子串或错位子串出现在长文本中了吗（同一个单词的多次出现只算做一次）？

输入：

输入第一行一个整数T表示测试样例的数量，后跟T个测试样例。每个测试样例包括三行，第一行一个整数L，第二行有L个由小写字符构成的单词，这些单词构成了字典，第三行包含两个字符S_1, S_2和五个整数N, A, B, C, D。S_1, S_2是长字符串的前两个字符，N是字符串的长度，其他的四个整数用来生成字符串S剩下的部分：

首先定义 ord(c)表示在字符 c 的十进制数值，char(n)表示十进制数 n 所代表的字符。举例而言，ord(a)=97, char(97)=a，字符按照 ASCII table 编码。

定义$x1 = \text{ord}(S1), x2 = \text{ord}(S2)$，然后使用如下公式递归生成$xi, i = 3 \dots N$：

$$x_i = (A * x_{i-1} + B * x_{i-2} + C) \text{ module } D$$

而S_i定义为 char$(97 + (x_i \text{ module } 26))$，$i = 3 \dots N$。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中 x 是从 1 开始的测试样例序号，y 是字典里的单词中，以子串或错位子串出现在长字符串中的单词数量。

参数限制：

$1 \le T \le 20$

字典中的单词各不相同

字典中每个单词的长度在2到10^5之间

字典中所有单词的长度不超过10^5

S_1, S_2是小写的英文字母

$0 \le A \le 10^9$

$0 \le B \le 10^9$

$0 \le C \le 10^9$

$1 \le D \le 10^9$

小数据集：

$1 \le L \le 1000$

$2 \le N \le 1000$

大数据集：

$1 \le L \le 20000$

$2 \le N \le 10^6$

测试样例：

Input	Output
1	Case #1: 4

5 axpaj apxaj dnrbt pjxdn abd a a 50 1 1 1 30	

在测试样例 1 中，所生成的字符串 S 为 aapxjdnrbtvldptfzbbdbbzxtndrvjblnzjfpvhdhhpxjdnrbt 字典中单词的错位出现如下：

axpaj: aapxjdnrbtvldptfzbbdbbzxtndrvjblnzjfpvhdhhpxjdnrbt

apxja: aapxjdnrbtvldptfzbbdbbzxtndrvjblnzjfpvhdhhpxjdnrbt

dnrbt: aapxjdnrbtvldptfzbbdbbzxtndrvjblnzjfpvhdhhpxjdnrbt

pjxdn: aapxjdnrbtvldptfzbbdbbzxtndrvjblnzjfpvhdhhpxjdnrbt

abd 未出现

weiyong1024 基于单词长度的遍历搜索算法:

朴素的方法是对于每一个单词，维护一个与其长度相同的滑窗在长字符串中匹配模式，复杂度$T\sim O(NL)$。该算法在 MacbookPro 上运行 8min-9min 之间。这样的做法存在两点不够高效的地方:

· 单词可以规约到由其首尾字符和字母频次表征的形式，所有单词用频次表表示才没有冗余信息;

· 相同长度的滑窗在遍历场字符串的时候搜索到的频次序列是相同的。

于是可以将上述方法做如下改进：首先单词由其字母频次表+首尾字符表征，然后对于每种长度的滑窗只进行一次遍历操作。

复杂度：$T\sim O(\sqrt{M}NlogL)$

完整代码:

```
#include <iostream>
#include <map>
#include <set>
#include <string>
#include <vector>

using namespace std;
using ll = long long;

using scramble_word = pair<vector<int>, pair<char, char>>;  // Oridinary word can transforms to this form.
map<scramble_word, int> words;  // Number of occurance of each scramble_word.
set<int> lengths;
string tex; // The Text.
int l, n;
void read() {
    lengths.clear();
    words.clear();
    scanf("%d", &l);
    for (int i = 0; i < l; i++) {
        string tmp_s;
        cin >> tmp_s;
        lengths.insert((int)tmp_s.size());

        vector<int> tmp_v(26, 0);
        for (int j = 0; j < tmp_s.size(); j++)
```

```
            tmp_v[tmp_s[j] - 'a']++;
        scramble_word tmp_word = make_pair(tmp_v, make_pair(tmp_s[0], tmp_s[tmp_s.size() - 1]));
        words[tmp_word]++;
    }

    tex.clear();
    int a, b, c, d;
    char c1, c2;
    cin >> c1 >> c2 >> n >> a >> b >> c >> d;
    tex.resize(n);
    tex[0] = c1, tex[1] = c2;
    ll x1 = tex[0], x2 = tex[1];
    for (int i = 2; i < n; i++) {
        ll x = (a * x2 + b * x1 + c) % d;
        tex[i] = 97 + x % 26;
        x1 = x2, x2 = x;
    }
}

vector<vector<int>> sum;
int solve() {
    sum.clear();
    sum.resize(n, vector<int>(26, 0));
    sum[0][tex[0] - 'a']++;
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < 26; j++)
            sum[i][j] = sum[i - 1][j];
        sum[i][tex[i] - 'a']++;
    }

    int ans = 0;
    for (auto length : lengths) {
        for (int i = 0; i + length - 1 < n; i++) {
            vector<int> tmp_v(26);
            if (i == 0) {
                for (int j = 0; j < 26; j++)
                    tmp_v[j] = sum[i + length - 1][j];
            } else {
                for (int j = 0; j < 26; j++)
                    tmp_v[j] = sum[i + length - 1][j] - sum[i - 1][j];
            }
            pair<vector<int>, pair<char, char>> tmp_word = make_pair(tmp_v, make_pair(tex[i], tex[i + length -
1]));
            if (words.find(tmp_word) != words.end()) {
                ans += words[tmp_word];
```

```
                    words.erase(tmp_word);
                }
            }
        }
        return ans;
}

int main() {
        freopen("clin.txt", "r", stdin);
        freopen("clout.txt", "w", stdout);

        int T;
        scanf("%d", &T);
        for (int t = 1; t <= T; t++) {
            read();
            int ans = solve();
            printf("Case #%d: %d\n", t, ans);
            fprintf(stderr, "Case #%d: %d\n", t, ans);
        }
}
```

上述程序在 MacbookPro(15-inch, 2017)上运行的时间 193s。

3.9 Kickstart 2018 Round B

A. No Nine（数位 DP）

题述：

No Nine 是一个休闲娱乐的小游戏。在这个游戏中你需要找出合法的数字，一个数字被称为“合法”当且仅当以下条件全部成立：

- 正整数
- 该数字的十进制表示中不含有数字9
- 该数不能被9整除

举例来说，16和17是合法的，而$18, 19, 17.2, -17$则是不合法的。

游戏的第一步你需要说出一个合法的数字F，接下来的每一步你要报出下一个合法的数字。例如，如果你从$F = 16$开始游戏，那么接下来你所报的数字为$16, 17, 20, 21, 22$等等。

Alice 很善于这个游戏而且从不出错。她记得自己一次游戏从F开始并以L结束，她想知道自己报了多少个数。

输入：

输入第一行一个整数T为测试样例的数量，后跟T个测试样例，每个测试样例有一行包含两个数字F和L——第一个和最后一个报的数。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y是报数的数量。

参数限制：

$1 \le T \le 10$

F和L都是合法的

小数据集：

$1 \le F < L \le 10^6$

大数据集：

$1 \le F < L \le 10^{18}$

测试样例：

Input	Output
2 16 26 88 102	Case #1: 9 Case #2: 4

测试样例 1 中，报数依次为：$16, 17, 20, 21, 22, 23, 24, 25, 26$。

测试样例 2 中，报数依次为：$88, 100, 101, 102$

Invin.Amit 的数位 dp 深搜解：

深搜函数 dfs(pos, upb, num)，加上低位上的动规备忘录 dp[pos][upb][num]:

· pos: 当前处理的数位;

· upb: 高位是否已达上界（若高位未达上界则当前位可以任意取值，否则只能取到上界在该数位的值）;

· num: 构造中的数字（由于只关心其是否能被9整除，我们取其模9的余数所形成的9种状态）。

完整代码：

```
#include <cstdio>
#include <cstring>

using namespace std;
using ll = long long;

const int maxn = 20;
```

```
ll f, l;
void read() {
	scanf("%lld %lld", &f, &l);
}

int a[maxn];
int n;
ll dp[maxn][2][9];
ll dfs(int pos, int upb, int num) {
	if (pos == -1) return num == 0 ? 0 : 1;
	if (dp[pos][upb][num] != -1) return dp[pos][upb][num];

	ll &ans = dp[pos][upb][num];
	ans = 0;
	for (int i = 0; i < 9; i++) {
		if (upb && i > a[pos]) break;
		ans += dfs(pos - 1, upb && i == a[pos], (num * 10 + i) % 9);
	}
	return ans;
}

ll legal_num(ll x) {
	n = 0;
	while (x)
		a[n++] = x % 10, x /= 10;
	memset(dp, -1, sizeof(dp));
	return dfs(n - 1, 1, 0);
}

ll solve() {
	return legal_num(l) - legal_num(f) + 1;
}

int main() {
	freopen("alin.txt", "r", stdin);
	freopen("alout.txt", "w", stdout);

	int T;
	scanf("%d", &T);
	for (int t = 1; t <= T; t++) {
		read();
		ll ans = solve();
		printf("Case #%d: %lld\n", t, ans);
		fprintf(stderr, "Case #%d: %lld\n", t, ans);
```

```
    }
}
```

注：GCC 中对一个地址取偏移（a[offset]）本身编译不会报错，然而预先声明数组大小就是保证了开辟连续合法的内存空间。否则可能出现 EXC_BAD_ACCESS.

3.9 Kickstart 2018 Round B

B. Sherlock and Bit Strings（数位 DP，顺序统计量）

题述：

生成指定长度为N的二进制串$S(S1, S2, \dots, SN)$，要求满足K个限制条件，每个条件由三个整数A_i, B_i, C_i表示，意思是$S_{A_i}, \dots, S_{B_i}$中 1 的数量为C_i。

保证存在至少一个符合要求的二进制串S，请返回所有符合要求的二进制串中按字典升序排在第P位的那一个。

输入：

第一行一个整数T表示测试样例的数量，返回T个测试样例。对于每个测试样例，第一行三个整数N, K, P，加下来的K行每行三个整数A_i, B_i, C_i的意义如上。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y是符合所有K个限制的二进制串中按字典序排在第P位的那一个。

参数限制：

$1 \le T \le 100$

$1 \le N \le 100$

$1 \le K \le 100$

$1 \le P \le \min\{10^{18}$，符合条件二进制串的数量$\}$

$1 \le A_i \le B_i \le N \text{ for } 1 \le i \le K$

$0 \le C_i \le N \text{ for } 1 \le i \le K$

$(A_i, B_i) \ne (A_j, B_j) \text{ for } 1 \le i \le j \le K$

小数据集：

$A_i = B_i \text{ for } 1 \le i \le K$

大数据集：

$B_i - A_i \le 15 \text{ for } 1 \le i \le K$

测试样例：

Input	Output
2 3 1 2 2 2 1 4 3 1 1 2 1 2 3 1 3 4 1	Case #1: 011 Case #2: 0101

第二个测试样例不会出现在小数据集中。

第一个测试样例中，所有符合条件的二进制串按字典序排列为[010, 011, 110, 111].

第二个测试样例中，所有符合条件的二进制串按字典徐排列为[0101, 1010].

Benq 的数位 DP 解法:

满足限制条件的二进制串中第K大的那一个，乍一看是一个复杂的问题，从两个独立的方面递进考虑：1. 怎样生成字符串空间的顺序统计量；2. 怎样描述限制条件？

对于第一个问题，第P大的顺序统计量采用从高位到低位的顺序逐位判断取值。具体地，怎样判断最高位的取值呢？如果除了最高位以外的数位构成的合法数字的数量大于等于P，则可以由此推断最高位的取值为0，否则为1。接下来的每一位取值都可以采用类似的方法得到，但需要注意的是，对于某一位取1的情况，向下一位迭代之前要将顺序统计量P值减去低位可以构成的合法串的数量。下面的问题是如何计算低位可以构成的合法串的数量。

第二个问题要解决计算由低位组成的二进制串中符合条件的数量。维护一个长度为 16 的 bitmask，这里使用一个 int32 变量的低 16 位来满足这个要求：通过不断右移一位并将高位 0/1 来模拟长度 16 的 bitmask 在二进制串上的移动。具体地，在考虑长度为n的二进制串的最高位的时候，维护以当前最高位作为末位的长度为 16 的 bitmask，判断这个 bitmask 中记录的信息局部信息是否满足所有以当前最高位为区间右端点的那些限制条件即可。

复杂度：时间复杂度就是填表的时间，故时、空复杂度都是$S\sim O(2^{16}N)=O(65536N)$

完整代码:

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

using namespace std;
using ll = long long;
using ld = long double;

const int maxn = 105;

ll n, k, p;
ld num[maxn][1 << 16];
bool done[maxn][1 << 16];
vector<pair<int, int>> need[maxn];
void read() {
    memset(done, 0, sizeof(done));
    for (int i = 0; i < maxn; i++)
        need[i].clear();
    cin >> n >> k >> p;
    for (int i = 0; i < k; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        need[b].push_back({a, c});
    }
}

ld dfs(int a, int b) {
    if (done[a][b]) return num[a][b];
    done[a][b] = true;
    for (auto x : need[a])
        if (__builtin_popcount(b >> (15 - a + x.first)) != x.second) return num[a][b] = 0;
    if (a == n) return num[a][b] = 1;
    return num[a][b] = dfs(a + 1, b / 2) + dfs(a + 1, b / 2 + (1 << 15));
}

string solve() {
    string cur;
    int b = 0;
    for (int i = 0; i < n; i++)
        if (dfs(i + 1, b >> 1) >= p)
```

```
            cur += '0', b >>= 1;
        else
            cur += '1', p -= dfs(i + 1, b >> 1), b = (b >> 1) + (1 << 15);
    return cur;
}

int main() {
    freopen("bsin.txt", "r", stdin);
    freopen("bsout.txt", "w", stdout);

    int T;
    cin >> T;
    for (int t = 1; t <= T; t++) {
        read();
        string ans = solve();
        cout << "Case #" << t << ": " << ans << endl;
        clog << "Case #" << t << ": " << ans << endl;
    }
}
```

3.9 Kickstart 2018 Round B

C. King's Circle（平面几何，移动扫除线，区间树）

题述：

给定平面上的N个互不相同的点$(V_1, H_1), \ldots, (V_N, H_N)$，取其中的三个点，要求存在一个四个顶点坐标都为整数且边平行于坐标轴的正方形，让这三个点落在其边缘上。这样的点三元组有多少个？

输入：

输入第一行一个整数位测试样例数的数量，后跟 T 个测试样例。每个测试样例有一行包含是个整数：$N, V_1, H_1, A, B, C, D, E, F, M$。

N为点的数量，第一个点的坐标为(V_1, H_1)。对于剩余的点$i = 2, \ldots, N$的坐标可由以下公式生成：

$V_i = (AV_{i-1} + BH_{i-1} + C) \text{module} M$

$H_i = (DV_{i-1} + EH_{i-1} + F) \text{module} M$

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y是满足条件的点三元组的数量。

参数限制：

$1 \le T \le 100$

$0 \le A < M$

$0 \le B < M$

$0 \le C < M$

$0 \le D < M$

$0 \le E < M$

$0 \le F < M$

$0 \le V_1 < M$

$0 \le H_1 < M$

$(V_i, H_i) \ne (V_j, H_J), \ i \ne j$

小数据集：

$3 \le N \le 1000$

$2 \le M \le 1000$

大数据集：

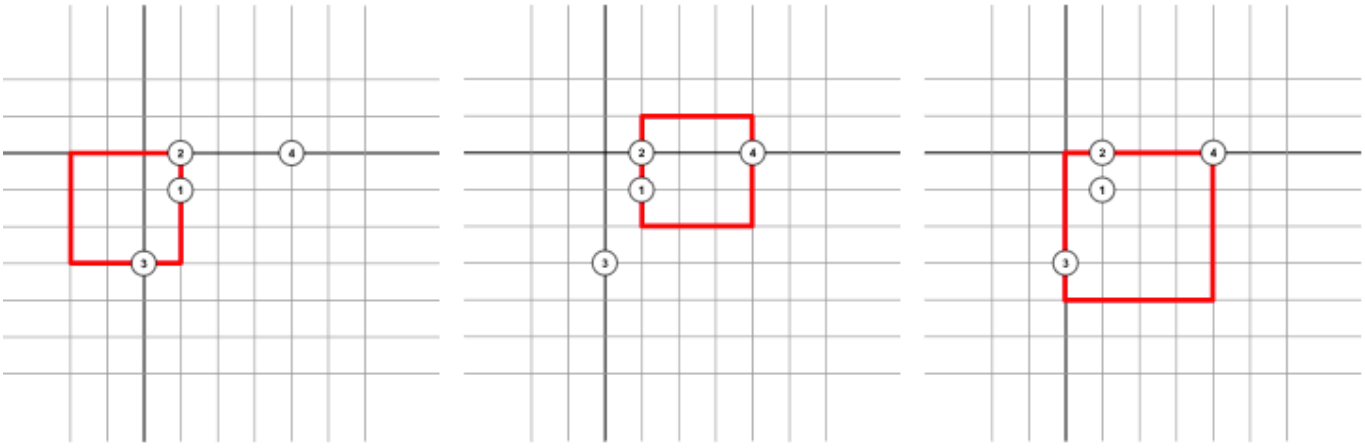
$3 \le N \le 5 \times 10^5$

$2 \le M \le 10^6$

测试样例：

Input	Output
3 4 1 1 4 1 1 4 2 4 5 6 3 1 1 0 1 0 1 0 9 3 7 24 34 11 17 31 15 40 50	Case #1: 3 Case #2: 20 Case #3: 0

测试样例 1 中，有4个点分别位于$(1,1), (1,0), (0,3), (4,0)$，如下图所示，存在穿过123, 124, 234号点的 axis-aligned 正方形，但不存在穿过134的正方形。



测试样例 2 中，所有6个点位于同一条水平线上，故其中任意三点符合题意，最终结果为$C_6^3 = 20$。
测试样例 3 中，三个点分别位于$(7,24),(19,17),(0,34)$，不存在符合条件的正方形穿过它们，故答案为0.
注：对于本体大数据集，不推荐使用解释形/较慢的语言。

Benq 的移动扫除线+区间树算法:

首先解决第一个子问题：给定三个点，是否存在一个 axis-aligned 的正方形恰好经过它们。经过若干尝试后不难发现，这样的正方形存在，当且仅当三个点都落在其包裹框（包含三个点的面积最小的矩形）的边上。

注意到由任意两个点所确定的包裹框以这两个点为对角，那么这两个点与所有**严格位于其包裹框内部**的点组成的三元组不符合条件，而与所有其他的点构成符合条件的三元组。于是我们只需要统计出对于每一个包裹框，其严格内部有多少点并累加就得到了所有不符合条件的三元组的数量，包裹框内部的点的数量可以用 2D 累加数组在$O(1)$时间内得到。这样的方法复杂度为$T\sim O(M^2+N^2)$，满足小数据集的要求。

对于大数据集，我们换一种方式统计不符合条件三元组的数量，我们以内部点为出发点，累计对于每个点，以它作为严格内部点的包裹框的数量，包裹框由位于其左上和右下的一对点或位于其右上和左下的一对点构成，数量为$n_{lu}n_{rd}+n_{ru}n_{ld}$。于是我们接下来就是要统计每个点的$n_{lu},n_{ld},n_{ru},n_{rd}$四个数量的值，这一步是用区间树作为扫除线状态的移动扫除线算法可以在$T\sim O(NlogM)$的时间内完成。

完整代码:

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

using namespace std;
using ll = long long;

const int maxn = 1005;
const int maxm = 1005;

struct Point {
    int x, y;
    Point() {}
    Point(int _x, int _y) : x(_x), y(_y) {}
};

bool comp(Point &a, Point &b) {
    return a.x < b.x || (a.x == b.x && a.y < b.y);
}
```

```
template<typename T> class SGT {
public:
	SGT() {}
	SGT(int _sz) : sz(_sz) {
		sgt = vector<T>(sz << 1, 0);
	}
	T query(int l, int r) {
		T ans = 0;
		for (l += sz, r += sz; l < r; l >>= 1, r >>= 1) {
			if (l & 1) ans += sgt[l++];
			if (r & 1) ans += sgt[--r];
		}
		return ans;
	}
	void add(int x, T val) {
		for (sgt[x += sz] += val; x > 1; x >>= 1)
			sgt[x >> 1] = sgt[x] + sgt[x ^ 1];
	}
private:
	int sz;
	vector<T> sgt;
};

Point points[maxn];
int n;
ll a, b, c, d, e, f, m;
void read() {
	cin >> n >> points[0].x >> points[0].y >> a >> b >> c >> d >> e >> f >> m;
	for (int i = 1; i < n; i++) {
		points[i].x = int((a * points[i - 1].x + b * points[i - 1].y + c) % m);
		points[i].y = int((d * points[i - 1].x + e * points[i - 1].y + f) % m);
	}
}

ll lu[maxn], ld[maxn], ru[maxn], rd[maxn];
void sweeplr() {
	SGT<ll, maxm> sgt;
	int i = 0;
	while (i < n) {
		int l = i, r = i + 1;
		while (r < n && points[r].x == points[l].x)
			r++;
		for (int j = l; j < r; j++) {
			lu[j] = sgt.query(points[j].y + 1, m);
			ld[j] = sgt.query(0, points[j].y);
```

```
            }
            for (int j = l; j < r; j++)
                sgt.add(points[j].y, 1);
            i = r;
        }
    }

    void sweeprl() {
        SGT<ll, maxm> sgt;
        int i = n - 1;
        while (i >= 0) {
            int l = i - 1, r = i;
            while (points[l].x == points[r].x)
                l--;
            for (int j = r; j > l; j--) {
                ru[j] = sgt.query(points[j].y + 1, m);
                rd[j] = sgt.query(0, points[j].y);
            }
            for (int j = r; j > l; j--)
                sgt.add(points[j].y, 1);
            i = l;
        }
    }

    ll solve() {
        sort(points, points + n, comp);
        for (int i = 0; i < n; i++) {
            assert(points[i].x < m);
            points[i].x++;
        }
        sweeplr();
        sweeprl();
        ll ans = n * (n - 1) * (n - 2) / 6;
        for (int i = 0; i < n; i++)
            ans -= lu[i] * rd[i] + ru[i] * ld[i];
        return ans;
    }

    int main() {
        freopen("csin.txt", "r", stdin);
        freopen("csout.txt", "w", stdout);

        int T;
        cin >> T;
        for (int t = 1; t < T + 1; t++) {
```

```
        read();
        ll ans = solve();
        cout << "Case #" << t << ": " << ans << endl;
        clog << "Case #" << t << ": " << ans << endl;
    }
}
```

3.10 Kickstart 2018 Round C

A. Planet Distance（图论、DFS 找环、BFS 算距离）

题述：

无向连通图的有N个顶点和N条边，图中有且仅有一个环，请给出所有顶点到该环的距离。

输入：

第一行一个整数T为测试样例的数量，后跟T组测试样例。每个测试样例的第一行为一个整数N，顶点编号为$1 \ldots N$，接下来的N行没行两个整数x_i, y_i表示第i条边的两个端点编号。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y是N个由空格分隔的整数，表示每个顶点到图中环的距离。

参数限制：

$1 \le T \le 100$

$1 \le x_i, y_i \le N$

$x_i \ne y_i$

图中有且仅有一个环。

小数据集：

$3 \le N \le 30$

大数据集：

$3 \le N \le 1000$

测试样例：

Input	Output
2 5 1 2 2 3 3 4 2 4 5 3 3 1 2 3 2 1 3	Case #1: 1 0 0 0 1 Case #2: 0 0 0

测试样例 1 中，顶点$2, 3, 4$构成环，所以这三个顶点到环的距离为0，顶点$1, 2$之间有一条边，顶点$3, 5$之间有一条边，故顶点$1, 5$到环的距离为1。

测试样例 2 中，所有顶点都在环上，故它们到环的距离都是0。

weiyong1024 的算法:

这道题的思路很明确，首先 DFS 确定图中的环，然后从环上的点开始 BFS 计算所有点到环的距离。时间复杂度$T \sim O(V + E) = O(N)$。

完整代码:

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

using namespace std;

const int maxn = 1005;
```

```
vector<int> g[maxn];
int n;
void read() {
	for (int i = 0; i < maxn; i++)
		g[i].clear();
	cin >> n;
	for (int i = 0; i < n; i++) {
		int u, v;
		cin >> u >> v;
		u--, v--;
		g[u].emplace_back(v);
		g[v].emplace_back(u);
	}
}

bool vis[maxn];
vector<int> path;
bool found;
void dfs(int cur) {
	vis[cur] = true;
	path.emplace_back(cur);
	for (int i = 0; i < g[cur].size(); i++) {
		if (path.size() > 1 && g[cur][i] == path[path.size() - 2]) continue;
		if (vis[g[cur][i]]) {
			path.emplace_back(g[cur][i]);
			found = true;
			return;
		} else dfs(g[cur][i]);
		if (found) return;
	}
	vis[cur] = false;
	path.pop_back();
}

vector<int> find_circle() {
	found = false;
	path.clear();
	memset(vis, 0, sizeof vis);
	dfs(0);
	int start = 0;
	while (path[start] !=  path[path.size() - 1])
		start++;
	vector<int> ret;
	for (int i = start + 1; i < path.size(); i++)
```

```
            ret.emplace_back(path[i]);
        return ret;
}

int ans[maxn];
void solve() {
        vector<int> circle = find_circle();
        memset(ans, -1, sizeof ans);
        queue<int> que;
        for (int x : circle)
                que.push(x), ans[x] = 0;
        while(!que.empty()) {
                int tmp = que.front();
                que.pop();
                for (int i = 0; i < g[tmp].size(); i++)
                        if (ans[g[tmp][i]] == -1)
                                que.push(g[tmp][i]), ans[g[tmp][i]] = ans[tmp] + 1;
        }
}

int main() {
        freopen("alin.txt", "r", stdin);
        freopen("alout.txt", "w", stdout);
        int T;
        cin >> T;
        for (int t = 1; t <= T; t++) {
                read();
                solve();
                cout << "Case #" << t << ":";
                for (int i = 0; i < n; i++)
                        cout << " " << ans[i];
                cout << endl;
                clog << "Case #" << t << ":";
                for (int i = 0; i < n; i++)
                        clog << " " << ans[i];
                clog << endl;
        }
}
```

3.10 Kickstart 2018 Round C

B. Fairies and Witches（巧妙设计 DFS）

题述：

现有N支魔法棒，他们之间可能有共同的端点，于是构成了一张图——图的顶点为魔法棒的端点，连接两个顶点的一条边代表这两个顶点为一支魔法棒的两端。每当取走一支魔法棒，与之共享端点的魔法棒就会消失。现希望取出若干支魔法棒组成一个面积非零的凸多边形，请问总共有多少种取法？

输入：

第一行一个整数T表示测试样例的数量，后跟T个测试样例。每个测试样例开始一行包含一个整数N，表示有魔法棒所构成的图的顶点数量，接下来的N行没行包含N个整数$L_{i,j}$，表示以顶点i和顶点j为端点的魔法棒的长度，若为零则表示这两个顶点之间没有魔法棒。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y为合法的魔法棒子集的数量。

参数限制：

$1 \leq T \leq 100$

$1 \leq L_{i,j} \leq 100$

$1 \leq L_{i,i} \leq 100$

$L_{i,j} = L_{j,i}$

小数据集：

$N = 6$

大数据集：

$6 \leq N \leq 15$

测试样例：

Input	Output
5 6 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 6 0 2 0 0 0 0 2 0 0 0 0 0 0 0 0 3 0 0 0 0 3 0 0 0 0 0 0 0 0 4 0 0 0 0 4 0 6 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 2 0 0 0 0 2 0 0 0 0 0 0 0 0 4	Case #1: 1 Case #2: 1 Case #3: 0 Case #4: 0 Case #5: 5

0 0 0 0 4 0 6 0 1 1 1 1 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 8 0 5 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 5 0	

注意最后一组测试样例不会出现在小数据集中。

测试样例 1 中，图中有5条长度相同的边：1-2, 2-3, 3-4, 4-5, 5-6。构成凸多边形至少需要3条边，取出三条边的唯一方法选择就是{1-2, 3-4, 5-6}。

测试样例 2 中，图中共有3支魔法棒：1-2, 3-4, 5-6，正好可以构成一个三角形。

测试样例 3 中，图中共有3支魔法棒：1-2, 3-4, 5-6，但他们的长度关系无法构成一个三角形。

测试样例 4 中，取走任意一支魔法棒都会使得其他所有魔法棒消失。

测试样例 5 中，图中共又4支等长的魔法棒：1-2, 3-4, 5-6, 7-8，任取三支构成三角形的方法有4种，构成四边形的方法有一种，共5种选择。

nhho（本场第四名）的纯模拟搜索算法:

dfs(a, sm, mx, num)函数没有返回值，每次迭代考虑如下状态信息：基于已有信息（已经取出的魔法棒长度和、最大长度、数量），在以顶点$\{a,\dots,n-1\}$为端点的魔法棒中继续取棒。用一个标记数组 flag[maxn]记录对于每个顶点，以该点为端点的魔法棒是否还可以被取出。探底条件——剩余搜索域为空，即$a=n$。

算法的复杂度为纯模拟的复杂度，必然是最优的。

完整代码:

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

using namespace std;

const int maxn = 20;

int l[maxn][maxn];
int n;
void read() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
			scanf("%d", &l[i][j]);
```

```
}

int ans;
bool flag[maxn];
void dfs(int a, int sm, int mx, int num) {
	if (a == n) {
		if (num >= 3 && sm - mx > mx) ans++;
	} else {
		if (!flag[a])
			for (int i = a + 1; i < n; i++)
				if (!flag[i] && l[a][i] > 0) {
					flag[i] = true;
					dfs(a + 1, sm + l[a][i], max(mx, l[a][i]), num + 1);
					flag[i] = false;
				}
		dfs(a + 1, sm, mx, num);
	}
}

int solve() {
	ans = 0;
	dfs(0, 0, 0, 0);
	return ans;
}

int main() {
	freopen("blin.txt", "r", stdin);
	freopen("blout.txt", "w", stdout);
	int T;
	cin >> T;
	for (int t = 1; t <= T; t++) {
		read();
		int ans = solve();
		cout << "Case #" << t << ": " << ans << endl;
		clog << "Case #" << t << ": " << ans << endl;
	}
}
```

3.10 Kickstart 2018 Round C

C. Kickstart Alarm（快速幂、1~n逆元表）

题述：

Kick 闹钟会响K次，由一串长度为N的数字$A_1, A_2, \ldots, A_N$作为指令控制。其中第i次响铃的功率P_i由指令串所有子串的i−指数和得到。具体地，子串$A_j, \ldots, A_k$的i−指数和为$S_{jk}^i = A_j \times 1^i + A_{j+1} \times 2^i + \cdots + A_k \times (k-j+1)^i$，进而$P_i = \sum_{1 \le j \le k \le N} S_{jk}^i$。

举例而言，如果$i = 2, A = [1, 4, 2]$，那么P_i的计算过程如下：

- [1]：$1 \times 1^2 = 1$
- [4]：$4 \times 1^2 = 4$
- [2]：$2 \times 1^2 = 2$
- [1, 4]：$1 \times 1^2 + 4 \times 2^2 = 17$
- [4, 2]：$4 \times 1^2 + 2 \times 2^2 = 12$
- [1, 4, 2]：$1 \times 1^2 + 4 \times 2^2 + 2 \times 3^2 = 35$

如上结果的和为$P_i = 71$。

给定 Kick 闹钟的响铃次数和控制序列，请你来计算每次响铃功率的和$P_1 + P_2 + \cdots + P_K$。

输入：

第一行一个整数T为测试样例的数量，后跟T个测试样例。每个测试样例包含一行九个整数$N, K, x_1, y_1, C, D, E_1, E_2, F$，其中$N$是指令串$A$的长度，$K$是响铃次数，接下来的参数用于计算指令串，对于$i \ge 2$：

- $x_i = (C \times x_{i-1} + D \times y_{i-1} + E_1)\%F$
- $y_i = (D \times x_{i-1} + C \times y_{i-1} + E_2)\%F$

定义$A_i = (x_i + y_i)\%F$，$1 \le i \le N$。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是从1开始的测试样例序号，y是K次响铃的功率和，结果对1000000007取模。

参数限制：

$1 \le T \le 100$

$1 \le x_1 \le 10^5$

$1 \le y_1 \le 10^5$

$1 \le C \le 10^5$

$1 \le D \le 10^5$

$1 \le E_1 \le 10^5$

$1 \le E_2 \le 10^5$

$1 \le F \le 10^5$

小数据集：

$1 \le N \le 100$

$1 \le K \le 20$

大数据集：

$1 \le N \le 10^6$

$1 \le K \le 10^4$

测试样例：

Input	Output
2 2 3 1 2 1 2 1 1 9 10 10 10001 10002 10003 10004 10005 10006 89273	Case #1: 52 Case #2: 739786670

参考 pr3pony（本场第五名）的亚线性时间算法:

大数据集中N最大可以取到$1e6$，所以我们需要一个亚线性时间的算法直接得到最终表达式$\sum_{i=1}^{K} P_i$中每个指令字符A_i的系数b_i，先来N较小的情形有什么规，令$f(x)=\sum_{i=1}^{K} x^i$

· $N=1$:

$$\sum_{i=1}^{K} P_i = A_1 \times f(1)$$

· $N=2$:

$$\sum_{i=1}^{K} P_i = A_1 \times 2 \times [f(1)] + A_2 \times 1 \times [f(1)+f(2)]$$

· $N=3$:

$$\sum_{i=1}^{K} P_i = A_1 \times 3 \times f(1) + A_2 \times 2 \times [f(1)+f(2)] + A_3 \times 1 \times [f(1)+f(2)+f(3)]$$

· $N=4$:

$$\sum_{i=1}^{K} P_i = A_1 \times 4 \times f(1) + A_2 \times 3 \times [f(1)+f(2)] + A_3 \times 2 \times [f(1)+f(2)+f(3)] + A_4 \times 1 \times [f(1)+f(2)+f(3)+f(4)]$$

· $N=5$:

$$\sum_{i=1}^{K} P_i = A_1 \times 5 \times f(1) + A_2 \times 4 \times [f(1)+f(2)] + A_3 \times 3 \times [f(1)+f(2)+f(3)] + A_4 \times 2 \times [f(1)+f(2)+f(3)+f(4)] + A_5 \times 1 \times [f(1)+f(2)+f(3)+f(4)+f(5)]$$

…

可以总结出（也很好证明），假设$\sum_{i=1}^{k} P_i = \sum_{i=1}^{N} A_i \times b_i$，则指令串$A$第$i$个字符$A_i$的系数$b_i$的值为:

$$b_i = (N-i+1)[f(1)+f(2)+\cdots+f(i)] = (N-i+1)s_i$$

其中，$s_i = f(1)+f(2)+\cdots+f(i)$。

下面我们的任务是在$T\sim O(N)$时间内算出$s_1, s_2, \ldots, s_n$，注意到$s_i = f(1)+\cdots+f(i)$，可得如下递推公式:

$$s_i = s_{i-1} + f(i)$$

于是有:

$$s_i = s_{i-1} + \sum_{j=1}^{K} i^j$$

两边同乘i:

$$i \times s_i = i \times s_{i-1} + \sum_{j=2}^{K+1} i^j = i \times s_{i-1} + \sum_{j=1}^{K} i^j + i^{K+1} - i = i \times s_{i-1} + (s_i - s_{i-1}) + i^{K+1} - i$$

移项得:

$$(i-1) \times s_i = (i-1) \times s_{i-1} + i^{K+1} - i$$

两边同除以$(i-1)$:

$$s_i = s_{i-1} + (i^{K+1} - i) \times (\mathrm{i}-1)^{-1}$$

该式可以在$T\sim O(\log K)$时间内得到，初始值$s_1 = K$，问题得解！

总复杂度$T\sim O(N\log K)$。

完整代码:

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

using namespace std;
```

```
using ll = long long;

const int maxn = 1000005;
const ll MOD = 1000000007;

ll qpw(ll a, ll b) {
	ll ans = 1;
	while (b) {
		if (b & 1) ans = ans * a % MOD, b--;
		a = a * a % MOD;
		b >>= 1;
	}
	return ans;
}

ll inv[maxn + 1];
void pre_process() {
	inv[1] = 1;
	for (int i = 2; i <= maxn; i++)
		inv[i] = (MOD - MOD / i) * inv[MOD % i] % MOD;
}

ll a[maxn];
ll n, k, x, y, c, d, e1, e2, f;
void read() {
	cin >> n >> k >> x >> y >> c >> d >> e1 >> e2 >> f;
	a[1] = (x + y) % f;
	for (int i = 2; i <= n; i++) {
		ll x_tmp = (c * x + d * y + e1) % f;
		ll y_tmp = (d * x + c * y + e2) % f;
		a[i] = (x_tmp + y_tmp) % f;
		x = x_tmp, y = y_tmp;
	}
}

ll s[maxn];
ll solve() {
	s[1] = k;
	for (int i = 2; i <= n; i++)
		s[i] = s[i - 1] + (qpw(i, k + 1) + MOD - i) * inv[i - 1], s[i] %= MOD;
	ll ans = 0;
	for (int i = 1; i <= n; i++)
		ans += a[i] * s[i] % MOD * (n - i + 1) % MOD, ans %= MOD;
	return ans;
}
```

```
int main() {
	freopen("clin.txt", "r", stdin);
	freopen("clout.txt", "w", stdout);
	pre_process();

	int T;
	cin >> T;
	for (int t = 1; t <= T; t++) {
		read();
		ll ans = solve();
		cout << "Case #" << t << ": " << ans << endl;
		clog << "Case #" << t << ": " << ans << endl;
	}
}
```

CodeJam 2018 Round 1b

A. Rounding Error（贪心法，四舍五入）

题述：

调研N个受访者最下换的编程语言并做以统计，已经统计其中一部分受访者的选择，剩下的受访者则可能在任意已有的语言或新语言中做出选择。你计划在调研最后给出上报的语言中每种的的票统计，采用四舍五入到小数点后一位的百分制表示。舍入误差的存在可能会使得最终得票率的加和不正好为100。那么基于已有的的票统计，最终得票率的加和最大可能为多少？

输入：

第一行一个整数T为测试样例的数量，后跟T个测试样例。每个测试样例有两行，第一行两个整数N和L：分别表示受访者总人数和已经被投票的语言种类数。第二行包含L个整数C_i，表示已经被投票的语言每种具体的票的数量。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中 x 是从 1 开始的测试样例序号，y 是当前情形下可能达到最大百分数和。

参数限制：

$1 \leq T \leq 100$

$1 \leq L < N$

$1 \leq C_i$

$$\sum_{i=1}^{l} C_i < N$$

每个测试集限时 10 秒

内存限制：1GB

测试集 1（可见）

$2 \leq N \leq 25$

测试集 2（可见）

$2 \leq T \leq 250$

测试集 3（隐藏）

$2 \leq N \leq 10^5$

测试样例：

Input	Output
4 3 2 1 1 10 3 1 3 2 6 2 3 1 9 8 1 1 1 1 1 1 1 1	Case #1: 100 Case #2: 100 Case #3: 101 Case #4: 99

测试样例 1 中，有两个受访者已经做出了不同的选择，还有一个受访者没有投票。如果这个人投了已有语言，则百分率加和为 33+33+33=99，否则为 33+67=100。故 100 为所求。

测试样例 2 中，无论剩下四人怎样投票，所有语言得票率必为 10 的倍数，故最终加和比为 100。

测试样例 3 中，一种最优解为：剩下俩个人各自选择一个没有出现过的语言，则最终得票率和为 50+17+17+17=101。

测试样例 4 中，无论剩下一个人怎样选择，最终得票率的和都是 99。

overtroll 的贪心算法:

虽然可选语言的种类没有数量限制，但最终可能的语言种类数最多为$(N-\sum_{i=1}^{l}C_i)+l$。考虑这么多种语言选项，没有被选择的语言得票数为视为0不会影响百分比加和的结果。

首先解决四舍五入的表示方式，对于任意整数a，其被N除的结果a/N四舍五入相当于$a+N/2$被N除取整数部分。

希望尽可能多的语言的票数被进一，那么对于加下来的$(N-\sum_{i=1}^{l}C_i)$票，我们这样操作：每次都投给已有的票数最容易进位的那一种语言。整个过程的时间复杂度为$T\sim O(N\log N)$。

完整代码:

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

using namespace std;

const int p = 100;
int n, l;
vector<int> c;
void read() {
	cin >> n >> l;
	c.resize(l);
	for (int &x : c)
		cin >> x;
}

int solve() {
	priority_queue<pair<int, int>> pq;
	int sum = 0;
	for (int i = 0; i < l; i++) {
		sum += c[i];
		c[i] = c[i] * p + n / 2;
		pq.emplace(c[i] % n, i);
	}
	for (int i = sum; i < n; i++) {
		c.emplace_back(n / 2);
		pq.emplace(n / 2, i - sum + l);
	}
	while (sum++ < n) {
		int tmp = pq.top().second;
		pq.pop();
		c[tmp] += p;
		pq.emplace(c[tmp] % n, tmp);
	}
	int ans = 0;
	for (int x : c)
		ans += x / n;
```

```
    return ans;
}

int main() {
    int T;
    cin >> T;
    for (int t = 1; t <= T; t++) {
        read();
        cout << "Case #" << t << ": " << solve() << endl;
    }
}
```

CodeJam 2018 Round 1b

B. Mysterious Road Signs（分治法）

CodeJam 2018 Round 1b

C. Transmutation（优化问题转判别问题，过程模拟）

题述：

有M种金属，每种金属a可以由另外两种金属b和c等量合成，即最小合成操作是用1个单位的b和1个单位的c合成1个单位的a（只能一个单位一个单位地合成）。给定M种金属的合成公式和初始存量，求最终可以获得最多多少1号金属。

输入：

第一行一个整数T表示测试样例的数量，后跟T个测试样例。每个测试样例的第一行为一个整数M：周期表中金属的数量，接下来的M行每行包含两个整数R_{i1}, R_{i2}，表示可以由1单位的金属R_{i1}和1单位的金属R_{i2}提炼1单位的金属i。最后一行包含M个有空格分隔的整数$G_1, G_2, \ldots, G_M$，表示每种金属的存量。

输出：

对于每个测试样例，输出一行 Case #x: y，其中x是从个1开始的测试样例序号，y是可以获得的最多的1号金属的数量。

参数限制：

$1 \leq T \leq 100$

$1 \leq R_{i1} < R_{i2} \leq M$

每个测试集限时 5 秒

内存限制：1GB

测试集 1（可见）

$2 \leq M \leq 8$

$0 \leq G_i \leq 8$

测试集 2（隐藏）

$2 \leq M \leq 100$

$0 \leq G_i \leq 100$

测试集 3（隐藏）

$2 \leq M \leq 100$

$0 \leq G_i \leq 10^9$

测试样例：

Input	Output
3 3 2 3 1 3 1 2 5 2 3 5 3 4 3 4 4 5 3 5 1 3 0 8 6 2 4 4 3 4 2 3	Case #1: 7 Case #2: 4 Case #3: 0

2 3 2 3 0 1 1 0	

测试样例 1 中，最优策略是用2单位的 2 号金属和2单位的 3 号金属合成2单位的 1 号金属，最终总共有7单位的 1 号金属。
测试样例 2 中，最优策略首先用2单位 3 号金属和2单位 5 号金属合成2单位 4 号金属，然后用4单位 3 号金属和4单位 4 号金属合成4单位 1 号金属。
测试样例 3 中，注意到由于合成过程最少为1单位，故不存在合成 1 号金属的途径。

muye67 的二分+深搜解

直接看最大的数据范围，将优化问题转化为判别问题：给定整数L，能否基于现有资源储备合成L单位的 1 号金属？

模拟合成金属的过程，如果 1 号金属储备非零，那么就消耗当前储备。如果储备不足，那么就将剩下的需求转移给其两个原料金属，这样的过程可以用深搜来模拟，直到需求被完全满足或者遇到一个当前调用栈中存在的金属则终止合成返回失败，注意维护当前原料库的状态数组。（注意储备为负证明该金属已在调用栈中，而储备为零则说明该金属不在当前调用栈中）

该算法的时间复杂度为：

完整代码：

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

using namespace std;
using ll = long long;

const int maxm = 105;

ll g[maxm];
pair<int, int> r[maxm];
int m;
void read() {
    cin >> m;
    for (int i = 0; i < m; i++) {
        cin >> r[i].first >> r[i].second;
        r[i].first--, r[i].second--;
    }
    for (int i = 0; i < m; i++)
        cin >> g[i];
}

ll g_tmp[maxm];

bool dfs(int cur, ll need) {
    if (g_tmp[cur] < 0) return false;
    g_tmp[cur] -= need;
    if (g_tmp[cur] >= 0) return true;
    bool valid = dfs(r[cur].first, -g_tmp[cur]) && dfs(r[cur].second, -g_tmp[cur]);
```

```
    if (valid) g_tmp[cur] = 0;
    return valid;
}

ll solve() {
    ll total = 0;
    for (int i = 0; i < m; i++)
        total += g[i];
    ll l = 0, r = total + 1;
    while (l < r - 1) {
        ll mid = (l + r) >> 1;
        memcpy(g_tmp, g, sizeof g);
        if (dfs(0, mid)) l = mid;
        else r = mid;
    }
    return l;
}

int main() {
    int T;
    cin >> T;
    for (int t = 1; t <= T; t++) {
        read();
        ll ans = solve();
        cout << "Case #" << t << ": " << ans << endl;
    }
}
```

CodeJam 2018 Round 2

A. Falling Balls（贪心法）

题述：

现有一个落球玩具，可由二维矩阵表征，矩阵的部分网格中有 \ 和 / 两种方向导引装置，小球在网格中的下落轨迹遵循如下规律：

- 若当前网格为空，则下落至正下方的网格；
- 若当前网格为 /，则下落至左下方的网格；
- 若当前网格为 \，则下落至右下方的网格；
- 小球的下路过程不会相互干扰，每个网格可以容纳很多小球。

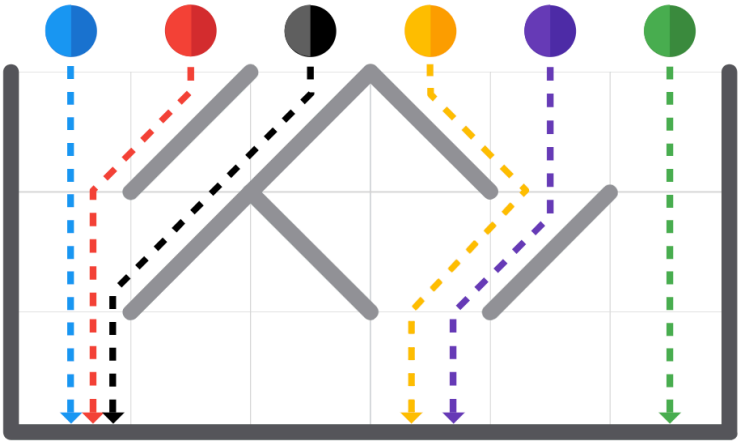
矩阵的最左列、最右列和最下行的网格必须为空，另外为了不让小球出现中途卡顿的情形，不能有 \ 的网格和 / 的网格顺序左右相邻。

现从矩阵的第一行的每个网格释放一个小球，给定最终所有小球落在玩具最下方行中每个网格内的数量分布，请问如何构造一个**行数最少**的落球玩具或判定不可能？

举例而言，如果给定最终底层网格的球数分布为3 0 0 2 1，则一个可行解如下：

```
.//\..
./\./.
......
```

轨迹的形象表示：



官方的贪心算法:

考虑小球经过玩具可能的路径，注意到只要两条路经中有共同点，则这两个小球必定落到同一个底层网格内。另一方面，**由于不存在 \／ 的相邻网格，故路径之间无法交叉**。由此可知，对于终态最左边网格的K个小球，其必须是最左侧的K列下落的小球。（反证：假如最左侧网格的K个小球中其中一个小球来自左起第$K+i$列，那么这个小球必然与左侧的$K+i-1$列下落的小球轨迹重合，那么这$K+i$个小球必然终止到相同的点——最左侧点，与假设矛盾）

与上面的分析类似，终态从最左侧网格开始，接下来每一个落求数量非零的网格i中的B_i个小球必然来自第一行接下来的B_i个连续列。（这样才能保证所有轨迹不交叉），于是我们有了一个底层每个小球数非零的网格到其中的小球对应区间的映射。以终态为3 2 0 0 0 0 2 1为例，我们将第一列映射到$[1,3]$，将第二列映射到$[4,5]$，将第七列映射到$[6,7]$，将第八列映射到$[8,8]$。这个过程耗时$T\sim O(C)$。

接下来具体构建落求玩具，从左向右扫描映射，对于每个落求数非零的底层列i，若其对应区间的左端点位于其左侧则从顶层左端点开始向右下方修建连续的 \，直到第$i-1$列；再从右向左扫描映射，对于每个落求数非零的底层列i。若其对应区间的右端点位位于其右侧则从顶层右断点开始向左下方修建连续的 /，知道第$i+1$列。取所有连续 \ 或 / 的最大长度加1（底层不能有导引装置）就是该玩具的深度。这个过程耗时$T\sim O(C)$。

这样的到的玩具是最小深度吗？是的，上文已经证明映射的必然性，由于每下落一层只能水平方向移动1位，那么深度不会小于起点和终点的最大水平绝对距离+1。什么样的情况下不可行？由于最左侧和最右侧的列不能有导引装置，故底层左右两端的网格至少有一个从该列下落的小球，这两个值为零是不可以的。
完整代码:

CodeJam 2018 Round 2

B. Graceful Chainsaw Jugglers

题述：

现有R个红球和B个蓝球，将这些球分给若干人，要求每个人至少分得一个球并且任意两个人手中的红球和蓝球数量不分别相同。问最多可以分给几个人？

输入：

分析:

问题要求每个人的红、蓝球数量不同时相同，以每种可能的红蓝数量对作为一个货物，**这就构成了一个零一背包问题**，只不过现在背包有两个槽。可能的物品取值有$(R+1)(B+1)-1$个（由于$(0,0)$不合法，我们在最终的结果中减去1），按任意顺序排列它们然后做前缀上的动态规划:

按红高位蓝低位的顺序排列，则第i种货物对应的红、蓝球数量对为$(\left\lfloor \frac{i}{B+1} \right\rfloor, i\%(B+1))$，其中$0 \le i \le RB+R+B$。令$f(i,x,y)$表示背包的红、蓝槽容量分别为$x$和$y$时，从前$i$种选择中可以装入背包的最大物品数，有最优子结构:

$$f(0,x,y)=0 \qquad 0 \le x \le R, 0 \le y \le B$$

$$f(i,x,y)=\begin{cases}\max\big(f(i-1,x-r[i],y-b[i])+1, f(i-1,x,y)\big) & x \ge r[i] \text{且} y \ge b[i] \\ f(i-1,x,y) & \text{otherwise}\end{cases} \qquad i>1$$

初始化$i=0$的所有子问题的答案都为0，最终答案为$f(RB+R+B,R,B)$。重叠子问题的备忘录大小为$S\sim O((RB)^2)$，整个动态规划就是求解这些重叠子问题得过程，故时间复杂度也是$T\sim O((RB)^2)$，可以应付小数据集$0 \le R,B \le 50$的规模。

对于大数据集

CodeJam 2018 Round 2

C. Custume Change（二分图的最大匹配）

题述：

有一个$N \times N$的矩阵，矩阵中每个元素的取值范围为$1 \ldots N$和$-1 \ldots -\mathrm{N}$共$2N$个整数。我们希望矩阵内的元素取值具有如下性质：

• 不存在两个相同取值的元素位于同一行内或同一列内。

给定矩阵元素的初始取值，请问最少改变多少个元素的值可以是的矩阵满足如上要求。

输入：

官方解析的最大二分图匹配算法:

对于这个问题，我们首先关心怎样找到原矩阵中行列不存在重复数字的最大点集。对于剩下的每个点，由于与其同行或同列的元素最多有$2(N-1)$个，而可选取值则有$2N$种，故总可以选取一个取值使得其与行列中的每个元素都不重复。

一种粗鲁的方法是对于所有的子集（2^{N^2}个）分别判断是否满足位于同一行、同一列的点值不重复（N^2）。故这种方法的总复杂度为$T\sim O(N^2 2^{N^2})$，可以应付小数据集。

对于大数据集，我们需要一个更高效的方式找到行列不重复的最大点集。为了求解这个问题，我们可以分别求解对于每个数字x，矩阵元素中行列不重复的最大子集的大小$\mathrm{f(x)}$，则最终答案为$\sum_{\mathrm{i}=-\mathrm{N},\mathrm{i}\neq 0}^{\mathrm{N}} \mathrm{f(i)}$。而对于$\mathrm{f(x)}$的求解本质上是一个二分图的最大匹配问题:

· 令$\{\mathrm{A}_1, \mathrm{A}_2, \ldots, \mathrm{A}_\mathrm{N}\}$为所有行的集合;

· 令$\{\mathrm{B}_1, \mathrm{B}_2, \ldots, \mathrm{B}_\mathrm{N}\}$为所有列的集合;

· 对于原矩阵中数字x的每次出现的位置(i, j)，在$\mathrm{A_i}$和$\mathrm{B_j}$之间连接一条线。

那么寻找行列不重复的值为x的点的子集，就是在寻找上述二分图的一个最大匹配。最大二分匹配算法的运行时间为$\mathrm{T\sim O(VE)}$，在本问题中$\mathrm{V=2N, E=O(N^2)}$，故总时间复杂度为$\mathrm{T\sim O(N^3)}$，可以应付大数据集。

完整代码:

CodeJam 2018 Round 2

D. Gridception

3.5 Code Jam 2017 Round 2

A. Fresh Chocolate

题述：

你是一个巧克力生产商的公关经理。很不幸，最近公司的形象蒙上了污点，消费者普遍认为公司老板过于吝啬。于是你计划通过组织一次工厂开放日的免费品尝活动来挽回公司形象。

项目开始不久后，你就明白了公司老板的臭名声真是来得真是理所应当：他只同意你以最低的成本提供免费巧克力。免费提供的巧克力一个包装内有P块。你本希望为每一个观光团开启一包新的巧克力，但老板坚持如果已经开封的包装还有剩余，必须将它们全部发给新来的观光团后才能开启一个新的包装。

例如，假设每个包装内含$P = 3$块巧克力，此时来了一个5人观光团。你打开两个包装给每人分一块巧克力后，剩下了一块。假设之后又来了一个6人团。他们中的一个人会先被分得剩下的一块，然后你将会打开两个新的包装分给其他的人，于是你又剩下了一块巧克力。如果接下来莅临参观的是连续的两个4人团，第一组将会得到那块剩下的巧克力外加一包新巧克力，而第二组得到的样品将全部来自两个新开启的包装。请注意，即便你想立刻打开一个新包装，如果当前仍有剩余巧克力未被分发，你就不能这样做。

在上例中，4组中的2组（第一组和最后一组）得到的巧克力样品全部来自新的包装。另外的两组得到的样品中有的来自新的包装有的则来自已开封的包装。你很清楚把已开封的巧克力样品分发给来访者不会起到很好的挽回公司在消费者心中吝啬形象的作用，但你又必须向当前体系妥协以使你吝啬的老板支持你的开放日项目。尽管事情不尽如人意，你还是竭力做好这项工作。

你收到了来自N个观光团的参观意向，每个观光团都都指定了参与活动的人数。团体将一次进入。你希望给这些观光团指定观光顺序以最大化所得巧克力样品全部来自新包装的组的数量。你不可以拒绝某个观光团，也不能让某个观光团参观超过一次，你必须给每组中的每个人恰好一块巧克力作为样品。

在上例中，如果前来参观的顺序不是$5, 6, 4, 4$而是$4, 5, 6, 4$，则共有3组（除5人组外的其余三组）观光团将会得到全部新鲜的样品。对于这个例子中的观光团集合，不存在更优的排序，因为没有一种排序能使得所有观光团的巧克力样品全部来自新的包装。

输入：

输入第一行给定测试用例数量T。随后是具体T个测试用例。每个测试用例包含两行，第一行包含两个整数N（观光团数量）和P（一个包装内的巧克力块数）。第二行包含N个整数$G_1, G_2, \dots, G_N$，每个观光团的人数。

输出：

对于每个测试样例，输出一行包含 Case #x: y，其中x是测试样例序号（从1开始），y是采用最优排序后被分得的巧克力样品全部来自新包装的观光团数量。

参数范围：

$1 \le T \le 100$

$1 \le N \le 100$

$1 \le G_i \le 100$

小数据集：

$2 \le P \le 3$

大数据集：

$2 \le P \le 4$

测试样例：

Input	Output
3 4 3 4 5 6 4 4 2 4 5 6 4	Case #1: 3 Case #2: 4 Case #3: 1

```
3 3
1 1 1
```

Case 1 就是题述中的例子。除上文给出的最优顺序之外，其他的顺序如 6, 5, 4, 4 同样使得到新鲜样品的组数最大化，尽管具体是哪些组未必相同。注意我们只关心得到全部新鲜样品的组的数量，而不关心它们共有了多少人。

Case 2 中，观光团集合与 Case 1 相同，但每个包装内含有 2 块巧克力。在这个 Case 中，存在一些（如 4 ,4 ,6 ,5）顺序使所有组都只得到新鲜样品。

Case 3 中，每个观光团内都只有一个人，他们总是会被分得来自同一个包装的巧克力。当然，只有排在第一位的观光团得到了新鲜样品。

3.5 Code Jam 2017 Round 2

B. Roller Coaster scheduling

题述：

你开了一家新的过山车游乐园正待开张。列车由单排的连续N个座位组成，从车头到车尾编号$1 \ldots N$。当然，越靠近车头的座位价值越高。游客已经购买了开放日门票，每张门票允许特定的游客到列车上的指定位置。有的游客买了超过一张的门票，她们是希望把每张门票都用于一次乘坐的。

你要统计开放日一共需要开放多少座次。对于每一班车，每个游客坐在一个座位上，车上的某些座位将被空下。你不能在一趟车上给一个游客安排超过一个座位，也不能把两个不同的游客安排在同一个座位上。

你希望安排做少的车次来兑现所有门票以及节省运行开支。为了尽可能减少车次，你可以升级已经卖出的票，升级意味着用一张价值更高的票替换一张已经卖出的票（即座位号更小）。你希望升级尽可能少的票，因为如果升级太多可能会助长游客贪婪的心态进而在未来要求更多的升级。

给定已售车票的座位和购买者，则在以最优的方式进行必要升级的情况下，最少安排多少趟车次就能兑现所有已售车票？在最少车次下升级的最少票数是多少？注意举例来说把四号位的票升级到二号位被视作一次升级，而不是两次。

输入：

输入第一行给出测试用例的数量T，随后是具体T个测试用例。每个测试用例第一行有三个整数：N（每趟车上的座位数），C（潜在游客数），M（卖出的票数），游客按$1 \ldots C$编号。随后是M行，每行有两个整数：P_i（第i张车票对应的座位号）和B_i（购买第i张车票的游客号）。

输出：

对于每个测试用例，输出一行 Case #x: y z。其中，x是从1开始的测试用例序号，y是采用最优规划下需要安排的车次数，z是相应的升级数。

参数范围：

$1 \leq T \leq 100$

$2 \leq N \leq 1000$

$1 \leq M \leq 1000$

$1 \leq P_i \leq N$

$1 \leq B_i \leq C$

小数据集：

$C = 2$

大数据集：

$2 \leq C \leq 1000$

测试样例：

Input

```
5
2 2 2
2 1
2 2
2 2 2
1 1
1 2
2 2 2
1 1
2 1
1000 1000 4
```

Output

```
Case #1: 1 1
Case #2: 2 0
Case #3: 2 0
Case #4: 2 1
Case #5: 2 1
```

```
3 2
2 1
3 3
3 1
3 3 5
3 1
2 2
3 3
2 2
3 1
```

请注意样例中的后两个不会出现在小数据集中。

Case #1 中，两个游客都购买了2号位的票。可以用一趟车兑现所有已售票，只需要把其中一张票升级到1号位即可。

Case #2 情景类似，但两个游客购买的都是1号位的票。由于你无法升级1号位的车票，所以你只能安排两趟车（每人一趟）以兑现所有已售票。

Case #3 中，同一个游客购买了两张票，由于你必须为该游客安排两次座位，故无需升级票位。

Case #4 中，游客和座位都存在未被关联到票上的。在这个情形下，3号位的票售出了三张。举例来说，如果你将2号游客的票位升级到2号位，你可以安排两趟车，第一趟车中1号游客坐2号位，3号游客坐3号位，第二趟车中2号游客坐1号位，1号乘客坐3号位。至此额外的升级也无法再减少过山车的总趟数，因为1号乘客买了两张票，你必须为他安排至少两趟车。

Case #5 中，一种最优规划是将一张$(3,1)$票升级为$(1,1)$票。

3.5 Code Jam 2017 Round 2

C. Beaming With Joy

题述：

Joy 即将开始一段长假，所以她雇了技术人员在家里安装基于红外光束的安全系统。技术人员给了她一张图纸将她的房子用R行C列的二维网格分割。每个网格有如下单元之一：

- /：左下-右上放置的双面反射镜。
- \：右下-左上放置的双面反射镜。
- -：水平放置的双向红外线发射器。
- |：竖直放置的双向红外发射器。
- #：墙壁。（注意物资四周并不一定有墙壁包围，这也是她安装安全系统的原因）
- .：空房间。

红外线沿直线传播并且会穿过所有空房间，光束遇到平面镜会反射90°。向右、左、下、上传播的光束遇到/镜时分别向上、下、左、右反射。类似地，向右、左、下、上传播的光束遇到\镜时分别向下、上、右、左反射。当一束光打在墙上或房屋边界时其会终止。光束之间的交叉不会影响各自的传播，如果光束打在发射器上则发射器会损坏（也包括发射器发出的光束打在自己身上的情况）。

Joy 希望保证每个空房间都有至少一条光束经过，并且没有发射器被损坏，因为那只会浪费金钱！只可惜，技术人员已经安装好了整个系统，而 Joy 能做的只是将部分发射器旋转90°。也就是说，她可以把任意数量的-改为|，反之亦然。

请问能否找到一种满足 Joy 要求的旋转方法，或者给出不可行的结论？注意不需要被旋转的平面镜数量最小。

输入：

输入第一行给出测试用例的数量T，接下来是具体的T个测试用例。每个 Case 开始一行为两个整数R和C（房子被网格分出的行数和列数），接下来的R行每行为一个由‘/\-|#.’构成的C长字符串。

输出：

对于每一个测试用例，输出一行 Case #x: y。其中x是从1开始的测试用例序号，y的内容如下：若不存在可行解输出 IMPOSSIBLE，或者存在可行解时输出 POSSIBLE。接下来，如果解存在，则按照与输入相同的格式输出调整后的表征房间内容的网格矩阵

如果存在多个解，输出任意一个即可。

参数范围：

$1 \le T \le 100$

$1 \le C \le 50$

每个字符取自‘/\-|#.’中的字符。

‘-’和‘|’的总数（即发射器的数量）在1到100之间，包括1和100。

至少存在一个空房间。

小数据集：

$1 \le R \le 5$

网格中不存在/或\（即无反射镜）。

大数据集：

$1 \le R \le 50$

测试样例：

Input	Output
5 1 3 -.-	Case #1: IMPOSSIBLE Case #2: POSSIBLE #.##

```
3 4
#.##
#--#
####
2 2
-.
#|
4 3
.|.
-//
.-.
#\/
3 3
/|\
\\/
./#
```

```
#||#
####
Case #3: POSSIBLE
|.
#|
Case #4: POSSIBLE
.-.
|//
.|.
#\/
Case #5: IMPOSSIBLE
```

请注意后两个测试样例不会出现在小数据集中。

Case #1 中，任何一个发射器如果将光束射向空房间，则一定会摧毁另一个发射器。故答案为 IMPOSSIBLE。

Case #2 中，左边的发射器必须旋转90°以覆盖空房间，左侧的发射器也必须旋转90°以避免摧毁左侧发射器。

Case #3 中，现有发射器已经满足覆盖所有空房间并且不会相互摧毁，故直接输出输入矩阵可以成为一个解。然而，值得注意的是我们给出的输出同样满足题意。

Case #4 中，一个可行解是旋转所有发射器，然而，请注意以下解同样可行；

```
.-.
|//
.-.
#\/
```

因为装镜子的房间无需光束通过，再说谁又会去头一个能连接房间对角那么大的玻璃镜子呢？

Case #5 中，无论发射器摆在什么方向上都会摧毁自己。故结果为 IMPOSSIBLE。

3.5 Code Jam 2017 Round 2

D. Shoot the Turrets

题述：

星际入侵解放战刚刚结束，所有人都很高兴，爱与和平重回地球。

城市用R行C列的网格表征。一些网格是建筑物（可以挡住任何人的视线、射击并无法穿过），另外一些是街道（任何人都可以经过，可以射击并且视线可通过）。很不幸，战争期间外星人在城市中设置了自动化安全塔楼，这些塔楼都安置在街道上（不在建筑物内），对市民构成了威胁。但幸运的是，街道上出了外星人的塔楼外，还有人类抵抗军（同样不在建筑物内）。开始时，抵抗军士兵与塔楼不处在同一网格。

外形塔楼不会移动。它们体积很小，所以不会挡住视线和弹道。士兵不能通过存在活跃塔楼的网格，但在其被摧毁后可以经过。当塔楼可以发现其水平和垂直方向视线内的士兵，当一个士兵进入到这样一个网格中时，塔楼不会攻击，但当士兵企图离开这样一个网格时，塔楼会攻击他。幸运的是，士兵在这样的网格中仍可以射击，塔楼不会视该动作为移动。这意味着事实上不会有士兵死亡，因为最不济他们可以一直在那里不动，等待救援（这个过程可能需要很长时间）。也许不久后你将有机会营救他们。

每个士兵一共可以移动M步，每一步只能移至水平或竖直方向的相邻网格。士兵不会挡住其他士兵的去路，也不会挡住其他士兵或塔楼的视线。每个士兵又一发子弹。当士兵的水平或竖直视线上出现塔楼时，士兵可以射击并摧毁它。每发子弹只能摧毁一个塔楼，但我们的士兵素质过硬，可以绕过前面的士兵和塔楼来摧毁更远方的某个塔楼。

给定一张地图矩阵（塔楼和士兵所在位置被标注），问士兵可以摧毁最多多少塔楼？

输入：

输入第一行给出测试用例数量T，接下来是具体的T个测试用例。每个测试用例第一行为三个整数C（地图宽度），R（地图高度）和M（每个士兵移动步数上限）。接下来的R行每行为一个长度为C的字符串，其中. 表示街道，#表示建筑，S表示士兵，T表示塔楼。

输出：

对于每个测试用例，输出一行 Case #x: y。其中x为从1开始的测试用例序号，y为士兵可以摧毁的最大塔楼数。接下来的y行，每行包含两个整数s_i和t_i表征第i个被摧毁的塔楼是t_i，摧毁它的是士兵s_i（无需给出士兵是如何移动的）。如果存在很多策略，你可以输出其中任意一个。

参数范围：

$1 \leq T \leq 100$

$1 \leq M \leq C * R$

小数据集：

$1 \leq C \leq 30$

$1 \leq R \leq 30$

士兵数量在1到10之间。

塔楼数量在1到10之间。

大数据集：

$1 \leq C \leq 100$

$1 \leq R \leq 100$

士兵数量在1到100之间。

塔楼数量在1到100之间。

测试样例：

Input

```
4
2 2 1
#S
T.
2 6 4
.T
.T
.T
S#
S#
S#
5 5 4
.....
SS#.T
SS#TT
SS#.T
.....
3 3 8
S.#
.#.
#.T
```

Output

```
Case #1: 1
1 1
Case #2: 3
3 3
1 1
2 2
Case #3: 3
1 2
2 1
6 3
Case #4: 0
```

Case #2 中，一种可行解为让3号士兵上移三格摧毁3号塔楼。然后1号士兵可以上移一格并右移一格（来到3号塔楼之前的位置）摧毁2号塔楼后面的1号塔楼。最后，2号士兵上移三格摧毁2号塔楼。

Case #3 中，1号士兵上移一格并右移三格摧毁2号塔楼。然后2号士兵可以上移一格并右移三个摧毁1号塔楼。最后，6号士兵下移一格并右移三个摧毁3号塔楼。其他士兵的可移动步数不足以摧毁其他的塔楼。

Case #4 中，士兵无法移动到塔楼所在的行或列，故塔楼无法被摧毁。

3.8 Hacker Cup 2017 Round 3

A. Salient Strings

题述：

许多人热衷于报纸上的填字游戏，但你不是。我的意思是，这个游戏的形式太糟糕了，不是吗？你只能使用英文单词填空，而且只要受过教育的人都能通过查辞典得到答案。再者设计一个谜题太复杂了，多浪费时间啊。

你给报社编辑部写了一封信来描述一种新的文字游戏。为这种新游戏设计一个谜题非常简单，你只需要给答题者一个前N个正整数的排列$P_{1\ldots N}$，然后由答题者给出一个与之相对应的 Salient string。

排列$P_{1\ldots N}$的 Salient string 是一个由N个大写字母构成的字符串，要求当其N个非空后缀按字典序排列时，以第i个字符开头的后缀在其中排第P_i位。一个排列可能没有 Salient string。

你的信中需要包含若干谜题样例。你已经有了一些排列，所以你需要做的只是给每个排列提供一个参考答案。（如果答案存在）

输入：

开始一行为一个整型T，表示排列的个数。对于每个排列，第一行为一个整型N，第二行的内容为由空格分隔的N个整型P_i 表示排列的第i个数（保证$1 \ldots N$中每个数在P中出现且只出现一次）。

注：原题中所有数字都在一列，此处为节省空间稍作改动。

输出：

对第i个排列，在一行内打印“Case #i: ”后跟输入排列的一个 Salient string，如果不存在返回-1。

参数范围：

$1 \le T \le 2000,\ 1 \le N \le 1000,\ 1 \le P_i \le N$

样例解释：

对 Case 1 而言，字符串 FACEBOOK 所有后缀的字典序为：

1. ACEBOOK
2. BOOK
3. CEBOOK
4. EBOOK
5. FACEBOOK
6. K
7. OK
8. OOK

按长度降序读上表中的序号得到一个排列51342876，于是“FACEBOOK”是该排列的一个 Salient string，也一个可接受的答案。

样例输入：

```
5
8
5 1 3 4 2 8 7 6
5
2 4 5 1 3
9
6 9 3 5 8 2 4 7 1
60
54 17 49 46 60 58 14 55 51 37 11 9 10 44 56 43 36 38 34 33 28 59 26 57 6 48 29 25 12 23 40 45 50 2 20 52 47 16 7 21 1 13 42 22 3 32 35 15 19 39 27 24 8 30 53 31 41 18 4 5
45
35 21 9 44 20 28 24 32 43 18 42 36 1 19 11 7 37 30 39 5 29 34 12 8 40 13 17 10 6 33 45 26 16 3 2 23 25 4 27 22 14 31 41 15 38
```

样例输出：

Case #1: FACEBOOK

Case #2: FOXEN

Case #3: HUEHUEHUE

Case #4: -1

Case #5: PNEUMONOULTRAMICROSCOPICSILICOVOLCANOCONIOSIS

Official Solution: 令$S[i]$为排列P的逆索引即$P[S[i]] = i$

3.8 Hacker Cup 2017 Round 3

B. Sluggish Security

题述：

N对朋友在两个N长队列中等待机场安检，第一列中的第i个人是第A_i对朋友中的一员，第二列中的第i个人是第B_i对朋友中的一员。在$A_{1\ldots N}$和$B_{1\ldots N}$共$2N$个整数中$1\ldots N$中每一个数出现且只出现两遍。一对朋友中的两个人可能站在同一列或分别占在两列中。

每分钟两个位于队头的人中有一个进入安检，如果一队为空，则另一队头经过安检。整个过程持续$2N$分钟直到两个队列都空。由于有些乘客在脱鞋、解皮带，将笔记本电脑单独装箱等繁琐操作的过程比较缓慢，每分钟经过安检口的乘客来自哪一队预先是不确定的。

当一个乘客经过安检时会在检查点等待其朋友，若其朋友已经在检查点等候则二人会立刻去他们的登机口。例如，如果一队朋友分别在a和b分钟后经过安检，则他们会在$\max(a,b)$分钟时一起去登机口。然而，大家都不喜欢过长时间的等待，如果有某对朋友相互等待的时间超过了2分钟，即$\max(a,b)-\min(a,b)>2$，他们就会气愤地闹事使机场关闭。

假设所有乘客都尽可能不闹事地经过安检，按照 N 对朋友赴往登机口的顺序构成一个朋友对序号的排列，那么这样的排列共有多少个？（在模 1000000007 意义下）

为减小输入规模，给定A_1，则$A_{2\ldots N}$可通过如下方式计算：$A_i=A_{i-1}+D_{A,i-1}$。欲计算$D_{A,1\ldots(N-1)}$，给定另外K_A个序列，其中第i个序列包含$L_{A,i}$个元素$S_{A,i,1\ldots L_{A,i}}$，以及一个重复数$R_{A,i}$。序列$D_{A,1\ldots(N-1)}$可由这K_A个序列串接而成：

第i个元素为$S_{A,i,1\ldots L_{A,i}}$的序列重复$R_{A,i}$次。（保证这样得到的序列有$(N-1)$个元素，即$\sum_{i=1}^{k_A}L_{A,i}R_{A,i}=N-1$）

类似地，给定B_1，则$B_{2\ldots N}$可由$D_{B,1\ldots(N-1)}$生成，$D_{B,1\ldots(N-1)}$则由K_B个序列拼接得到，其中第i个由连续的$R_{B,i}$个$S_{B,i,1\ldots L_{B,i}}$构成。

输入：

开始一行为一个整型T，表示机场个数。对于每一个机场，第一行为一个整型N，接下来一行为两个由空格分隔的整形A_1和K_A。接下来的K_A行第i行开头为两个整型$R_{A,i}$和$L_{A,i}$，后跟$L_{A,i}$个由空格分隔的整型，其中第j个为$S_{A,i,j}$。类似地，接下来一行为由空格分隔的两个整型B_1和K_B。接下来的K_B行第i行开头为两个整型$R_{B,i}$和$L_{B,i}$，后跟$L_{B,i}$个由空格分隔的整型，其中第j个为$S_{B,i,j}$。

输出：

对第i个机场，打印一行“Case #i:”，后跟可能的排列数量，对 1000000007 取模。如果不存在所有人不闹事通过安检的可能性，输出 0。

参数范围：

$1\le T\le 400$

$2\le N\le 2{,}000{,}000$

$1\le A_i,B_i,K_A,K_B,R_{A,i},R_{B,i},L_{A,i},L_{B,i}\le N$

$-N\le D_{A,i},D_{B,i},S_{A,i,j},S_{B,i,j}\le N$

样例解释：

Case 1 中，$A=[1,3,2]$，$B=[3,1,2]$。两种可能的朋友对排列为[1, 3, 2]和[3, 1, 2]。前者对应的一种安检序为：1,2,2,1,1,2（相应朋友对序号为 1,3,1,3,2,2）。

Case 2 中，$A=B=[1,2,3,4,5]$。只有一种可能的朋友对序：$[1,2,3,4,5]$ 。

Case 3 中，$A=[2,12,5,12,5,7,1,7,4,4,15,10,15,10,14]$，$B=[6,6,8,8,11,2,11,13,9,13,9,3,3,1,14]$。

Case 4 中，$A=[1,2,3,4,5]$，$B=[5,4,3,2,1]$。没有使所有乘客不闹事通过安检的可能性。

Case 5 中，A=[1, 1, 2, 2, …, 4999, 4999, 5000, 5000]，B=[5001, 5001, 5002, 5002, …, 9999, 9999, 10000, 10000]。

样例输入：

```
5
3
1 1
```

1 2 2 -1
3 2
1 1 -2
1 1 1
5
1 1
4 1 1
1 1
4 1 1
15
2 1
1 14 10 -7 7 -7 2 -6 6 -3 0 11 -5 5 -5 4
6 1
1 14 0 2 0 3 -9 9 2 -4 4 -4 -6 0 -2 13
5
1 1
4 1 1
5 1
4 1 -1
10000
1 2
4999 2 0 1
1 1 0
5001 2
4999 2 0 1
1 1 0

样例输出：

Case #1: 2
Case #2: 1
Case #3: 12
Case #4: 0
Case #5: 2413012

3.8 Hacker Cup 2017 Round 3

C. Pie Packages

题述：

威尔逊，尽人皆知的算法竞赛活跃选手，最终决定是时候退休了。至于原因，经过数轮工作变化和奇迹般的晋升，他不仅已经变得非常富有，而且终于脱离了自己最不喜欢的行政工作。

威尔逊目前在一家派送餐公司担任账房主管，他被委派了退休前的最后一项任务——计算送餐卡车的汽油用量。他本人其实宁愿去做那些卡车司机中的一个……

派送餐公司为$N+1$个城镇构成的区域提供服务，这个区域本身就像一个派（或者更像一把伞…）。其中N个城镇在区域外层围成一个圈，顺时针从1到N编号。它们之间由N条公路相连，每条公路都是双向通车。这些公路中的第i条连接第i号和第$i+1$号城镇，通过需要O_i升汽油。（第N条公路连接第N和第1个城镇）

最后一个城镇位于区域的中心，编号$N+1$，在它和其他N个城镇之间设有公路，每条公路双向通车。这些公路中的第i条连接第i号和第$N+1$号城镇，通过需要R_i升汽油。

送货表中共有$\frac{N(N+1)}{2}$单任务，每一单都在两个城镇间运送。具体地，对于城镇i，有$i-1$单要从它出发分别送往城镇$1 \ldots i-1$。由于汽油费用要卡车司机自己支付，所以每一单的卡车司机总是会走起点和终点之间耗油最少的路径。威尔逊的任务是累计所有$\frac{N(N+1)}{2}$笔订单运输的汽油总用量，公司要求在模 1000000007 意义下计算。

给定O_1和常数A_o, B_o, C_o, D_o，则$O_{2\ldots N}$可以通过下式计算：$O_i = (A_o * O_{i-1} + B_o)\%C_o + D_o$。类似地，给定$R_1$和常数$A_r, B_r, C_r, D_r$，则$R_{2\ldots N}$可通过下式计算：$R_i = (A_r * R_{i-1} + B_r)\%C_r + D_r$。

输入：

开始一行为一个T，表示城镇区域数。对于每一个城镇区域，第一行为一个整型N，接下来的一行为由空格分隔的 5 个整型O_1, A_o, B_o, C_o, D_o，再下一行为由空格分隔的五个整型R_1, A_r, B_r, C_r, D_r。

输出：

对第i张图，打印一行“Case #i: ”，后跟所有订单的总耗油量。（模 1000000007）

参数范围：

$1 \le T \le 40$

$3 \le N \le 1{,}000{,}000$

$1 \le O_1, C_o, D_o \le 1{,}000{,}000$

$0 \le A_o, B_o \le 1{,}000{,}000$

$1 \le R_1, C_r, D_r \le 1{,}000{,}000$

$0 \le A_r, B_r \le 1{,}000{,}000$

样例解释：

Case 1 中，经过外圈公路分别需要$1, 3, 5$升汽油，与此同时，经过径向公路分别需要$1, 2, 2$升汽油。运送6比订单的耗油量分别为$[1, 1, 2, 2, 3, 3]$升，总共12升。

样例输入：

```
5
3
1 1 1 100 1
1 0 1 100 1
3
1 1 1 100 1
50 1 0 100 1
5
2 3 4 13 3
4 3 2 13 3
```

10
23 42 32 98 29
19 43 36 67 21
1000
123 456 789 123456 789
789 456 123 654321 987

样例输出：

Case #1: 12
Case #2: 161
Case #3: 93
Case #4: 4288
Case #5: 292394669

3.8 Hacker Cup 2017 Round 3

D. Broken Bits

题述：

你有一种新的钟表，表盘由一行N个灯组成。每个灯或亮或灭，它们的状态构成了一个N位二进制数。第一个灯表示最高位，第N个灯表示最低位，每个灯亮代表1灭代表0。

此刻你开始观察这块表，你发现每秒钟这块表显示的数字会自增1，并通过灯的亮灭用二进制表示当前数字。当前数字若为$2^N - 1$（即N位全部为1），则下一秒归0并重新开始计数。

然而，0个或多个灯是坏的。你不知道具体哪些灯是坏的，但你知道坏的灯一直处于熄灭状态，即便在应该被点亮的时候。

此时此刻，若$L_i = 1$则第i个灯为点亮状态，否则（$L_i = 0$）为熄灭状态。据知情者透露当前理论上应该有K个灯处于点亮状态。（保证K不小于当前处于点亮状态的灯的数量）

假设你就在这儿观察钟表的变化，则最少经过多长时间你就能确定哪些灯在哪些时候应该点亮？有可能你立马就能确定，即只用0秒。也可能你将无法确定，无论等待多长时间。

输入：

开始一行为一个整型T，钟表的数量。对每一块钟表，第一行为两个由空格分隔的整形N和K，接下来的一行为N个由空格分隔的数字，其中第i个为L_i。

输出：

对于第i个钟表，打印一行“Case #i: ”，直到你确定当前所有灯中哪些应该处于点亮状态所经过的秒数，如果永远无法确定，输出-1。

参数限制：

$1 \le T \le 5{,}000$

$1 \le N \le 60$

$0 \le K \le N$

$1 \le L_i \le 1$

样例解释：

Case 1 中，开始时右侧两个灯之一应该处于点亮状态，所以当前钟表应该显示101(5)或110(6)。一秒以后，钟表应该显示110(6)或111(7)。假设钟表右侧两灯全部坏掉了，那么此时都会显示100，所以你无法分辨具体应该显示二者中的哪一个。再过一秒，钟表应该显示111(7)或000(0)，通过观察最高位的状态，你就可以确定当前钟表应该显示111还是000。

Case 2 中，如果最左边的两个灯全部损坏，那么你将无法通过观察确定当前应显示的数字。

样例输入：

```
5
3 2
1 0 0
3 2
0 0 1
7 4
0 1 0 0 1 0 0
20 15
1 1 0 0 0 0 1 0 1 1 0 0 0 1 1 1 1 0 0 1
20 19
0 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 1 0 1 0
```

样例输出：

```
Case #1: 2
Case #2: -1
```

Case #3: 19

Case #4: 217601

Case #5: 8193

3.8 Hacker Cup 2017 Round 3

E. Steadfast Snakes

题述：

从前，你是一个骄傲的主人，拥有许多梯子和蛇。不幸的是，迫于政策你放弃了其中绝大部分，梯子和蛇都只剩下一个。很快，你仅剩的那条蛇也爬走了…

但这一切马上就要改变了。你刚刚得到消息称不久前通过了一条新的行政令，允许你重新保留任意多的蛇！作为准备，你兴高采烈地造了一连串N个梯子作为即将到来的蛇群的住处。不幸的是，直到这时你才意识到一个巨大的错误——蛇的饲养费真是 TMD 太贵了。

这N个梯子在地上排成一条线，每个梯子竖直放置。相邻的两个梯子间隔1米，从左向右数第i个梯子高度为H_i米。作为一个爬行动物行为模式专家，你确信一定数量的蛇将会入住你的这些资产。具体地，每个无序梯子对将被一条蛇认领，也就是说总共$\frac{N(N-1)}{2}$条蛇将会出现。如果一条蛇认领了一对梯子i和j，它将把自己的身体抻长以覆盖两个梯子以及其间的地面。因此，这条蛇的长度将会达到$H_i + |i - j| + H_j$米。

你极度希望尽快削减某些梯子的高度以吸引长度更短的蛇入住从而节省饲养开支。你预计在蛇群到来之前将有K分钟用于改建。每一分钟，你将选择一个梯子砍掉其上方的一些阶，将其高度降低1米。当某个梯子高度达到1米时，就不能再降低了。在每一分钟里你也可以选择不砍掉任何一个梯子的高度。

众所周知，一条蛇每天的饲养费与其长度成正比。话虽如此，你所关心的却并不是每日用于饲养的总开销，而是在某一条蛇身上花费的最大开销。因此，你希望确定采用最优改建策略时所有$\frac{N(N-1)}{2}$条蛇的最大长度的最小值。

给定H_1和常数A, B, C，则$H_{2\ldots N}$可通过下式计算：$H_i = (A * H_{i-1} + B)\%C + 1$。

输入：

开始一行为一个整型T，表征不同梯子集的数量。对每个梯子集，第一行包含由空格分隔的两个整型N和K，接下来一行为四个由空格分隔的整型H_1, A, B, C。

输出：

对第i个梯子集，打印一行“Case #i: ”，后跟所有蛇中最大长度的最小值。

参数范围：

$1 \le T \le 20$

$2 \le N \le 800{,}000$

$0 \le K \le 1e15$

$1 \le H_1, C \le 1{,}000{,}000{,}000$

$0 \le A, B \le 1{,}000{,}000{,}000$

样例解释：

Case 1 中，梯子的高度分别为$4, 5, 6$米。削减最后一个梯子三次将会产生高度$4, 5, 3$米。此时长度最大的蛇为认领前两个梯子的那一条，其长度为$4 + 1 + 5 = 10$米。你也可以削减第二个梯子一次，第三个梯子两次使最终高度为 4, 4, 4。在这种情况下长度最大的蛇为认领第$1, 3$号梯子的蛇，其长度为$4 + 2 + 4 = 10$米。

Case 2 中，你有足够的时间把所有的梯子都裁剪到1米。此时长度最大的蛇为认领首末两个梯子的蛇，其长度为$1 + 4 + 1 = 6$米。

Case 3 中，梯子的高度为$[6, 16, 36, 2, 8, 20, 7, 18]$。

样例输入：

```
5
3 3
4 1 0 100
5 100
1 2 3 4
```

8 8

6 2 3 37

1000 30000

10 10 10 1337

25000 888888

987 654 321 123456789

样例输出：

Case #1: 10

Case #2: 6

Case #3: 51

Case #4: 2692

Case #5: 246727827

3.8 Hacker Cup 2017 Final Round

A. Fox Patros (计数方法，快速幂)

题述：

在一个隐蔽的山谷中生活着一个兴盛的神秘动物族群——Foxen！然而，让栖息地对外界（例如虎视眈眈的人类）保持隐蔽是很必要的。为此，一队 Foxen 被派往边界巡逻。

在巡逻的路上，小 Foxen 们知道它们将要穿过的是一片神奇的矩形森林，森林可以被建模为一个R行C列的矩阵。矩阵的行自北向南编号1到R，列自西向东编号1到C。每个单元的中心长有一棵树，树的高度为正整数（单位为米）且不超过H。

如果小 Foxen 们从森林北侧观望，那么位于同一列单元格中的树就会相互重叠遮挡。那么事实上，小 Foxen 们只能画出从那个方向看来森林的“轮廓线”。从北侧看到的轮廓线可以表示为一个长度为C的正整数序列，其中第i个数为第i列R棵树中最高一棵的高度。

类似地，如果从西侧观望，也能获得一条“轮廓线”。从西侧看的轮廓线可以表示为一个长度为R的正整数序列，其中第i个数为第i行C棵树中最高者的高度。

在前往森林的路上，小 Foxen 们好奇轮廓线会是什么样子。通过事先调查他们知道森林的维度数R和C，以及树的最大可能高度H，但它们不知道任何一棵树的具体高度。它们想知道最终它们可能会看到多少种“外貌不同”的森林。一个森林为一个R行C列的树矩阵，两个森林被视作“外貌不同”当且仅当它们的“北轮廓线”和“南轮廓线”不同时相同。

请帮助小 Foxen 们确定“外貌不同”的森林数量，由于数量巨大，只关心该值对1000000007取模后的结果。

输入：

开始一个整数T表示森林的数量。对于每个森林，用一行三个有空格分隔的整数R, C, H表征。

输出：

对于第i个森林，输出一行“$Case\ \#i:$”后跟可能的“外貌不同”的森林数量对1000000007取模后的结果。

参数限制：

$1 \le T \le 30$

$1 \le R, C, H \le 500{,}000$

样例解释：

测试样例 1 中，共有10种“外貌不同”的2×2森林，每棵树的高度为1m 或2m。例如，下面的两个森林就有不同的“外貌”（西轮廓相同，北轮廓不同），故二者都应被计数。

```
1 2    2 1
1 1    1 1
```

再比如，下面的两个森林无论从西侧还是北侧看都具有相同的“外貌”：

```
1 2    2 2
2 1    2 1
```

测试样例：

Input	Output
5 2 2 2 3 3 3 10 10 2 10 10 3 12345 54321 98765	Case #1: 10 Case #2: 411 Case #3: 1046530 Case #4: 367947134 Case #5: 151251795

Jacob Plachta（作者）的算法：

让我们从分析一个北侧轮廓线$N[1 \dots C]$和一个西侧轮廓线$W[1 \dots R]$是否匹配的决定因素开始。一对轮廓线如果相互匹配，意味着存在一个森林分别以二者为两侧轮廓。一个必要条件是N中的最大值一定和W中的最大值相等。

事实上，这也是一对轮廓线匹配的充分条件。设N中最大值为$N[i]$（如果最大值不唯一，那么考虑其中任意一个即可），W中最大值为$W[j]$，那么相应的森林可以按如下方式构建：给第j行的树赋高度$N[1 \dots C]$，给第i列的树赋高度$W[1 \dots R]$，其他的树赋高度1即可。这样一来，位于(i,j)的树同时为两侧轮廓中的最高值，而两侧轮廓不会相互影响，也不会被其他的树遮挡。

至此，这个问题本质上是求解匹配轮廓线的对数，基于以上分析这个问题用排列组合知识不难解决。考虑树林最大高度h，对于1到H之间的每个h值，以此为最大树高的森林数等于以此为最大高度的北轮廓数量乘以一次为最大高度的西轮廓数量。即长度为k（k为R或C）、每个元素值在1到h之间且至少有一个元素为h的序列数量，这样的序列有$h^k-(h-1)^k$个。利用模1000000007意义下的快速幂算法指数可以在$T\sim O(\log k)$时间内完成，总复杂度为$T\sim O(H(\log R+\log C))$

完整代码:

```
#include <iostream>

using ll = long long;

const int MOD = 1000000007;

int n, m, h;
void read() {
    scanf("%d %d %d", &n, &m, &h);
}

int pw(ll a, int b) {    // a^b.
    int ans = 1;
    while (b) {
        if (b & 1) ans = ans * a % MOD;
        a = a * a % MOD;
        b >>= 1;
    }
    return ans;
}

ll cnt(int n, int h) {
    return (pw(h, n) - pw(h - 1, n) + MOD) % MOD;
}

int solve() {
    int ans = 0;
    for (int i = 1; i <= h; i++)
        ans = (ans + cnt(n, i) * cnt(m, i)) % MOD;
    return ans;
}

int main() {
    freopen("fox_patrols_in.txt", "r", stdin);
    freopen("fox_patrols_out.txt", "w", stdout);
```

```
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        int ans = solve();
        printf("Case #%d: %d\n", t, ans);
        fprintf(stderr, "Case #%d / %d: %d\n", t, T, ans);
    }
}
```

3.8 Hacker Cup 2017 Final Round

B. Fox Moles (图论建模，顶点着色，最小覆盖，最大匹配)

题述：

晚餐时间到了！一只由N只 Foxen 组成的狩猎小队潜伏在一片区域中，悄悄等待食物的出现。这片区域可以用一个无限长数轴表征。其中，第i只 Foxen 站在P_i的位置，任意两只 Foxen 的站位不同。在数轴上的每一个整数点处有一个鼹鼠洞，每个鼹鼠洞都通向一个鼹鼠巢穴，Foxen 们知道这些可口的小动物或早或晚将跳出洞口。

Foxen 有一种鲜为人知的能力，它们除了常规感官敏锐以外，还拥有类似声纳的器官能够利用普通人无法察觉的声波探测超远距离之外的物体。第i只 Foxen 把其波长调整到R_i，以使它能够检测到距离恰好为R_i的鼹鼠洞的动静（也就是位于$P_i + R_i$和$P_i - R_i$的鼹鼠洞）。

一刹那间，一定数量的鼹鼠探出地面来！探出地面的鼹鼠满足如下几条约束：没有鼹鼠从有 Foxen 蹲点儿的洞口冒出，不存在两只鼹鼠从同一个洞中冒出，每只鼹鼠都被至少一只 Foxen 发现。更进一步，每只编号为i的 Foxen 都发现恰好1只鼹鼠出现在与它距离为R_i的位置。（与之相对的，还可能发现0只或2只）

在初始事件之后，又陆陆续续出现了一些骚动（commontion）。一些原来探出头的鼹鼠回到地下，一些新的鼹鼠钻出地面，这一切以某种顺序发生。每时每刻，探出地面的鼹鼠还遵循初始时的那几条约束，但有一点不同——每只编号为i的 Foxen 能够确信**至少有一只**鼹鼠在与它距离为R_i的位置出现，但无法肯定具体是1只还是2只。

经过一段时间之后，Foxen 们决定要开始扑食，“邀请”当前地面上的一部分鼹鼠充当它们的晚餐。不幸的是，这些 Foxen 过分沉浸于跟踪那些冒出地面的鼹鼠，而忘记去统计地面上到底有多少只鼹鼠。假设 Foxen 们初始时的观测是正确的，经过一段时间鼹鼠们冒出地面 / 回到地下之后，请帮助 Foxen 确定最终在地面上的鼹鼠数量的有多少种可能性。

如果 Foxen 初始时的观测本身就是（in the first place）错误的，输出-1。

输入：

输入开始一个整数T表示狩猎区域的数量。对每一个狩猎区域，第一行为一个整数N，接下来的N行中第i行为两个整数P_i和R_i。

输出：

对于第i片狩猎区域，输出一行“$Case\ \#i:$”后跟一个整数，最终位于地面上的鼹鼠不同数量的可能性数，亦或若初始观测是不准确的则输出-1。

参数限制：

$1 \le T \le 30$

$1 \le N \le 5{,}000$

$0 \le P_i \le 1{,}000{,}000{,}000$

$1 \le R_i \le 1{,}000{,}000{,}000$

样例解释：

第一个测试样例中，鼹鼠的数量最终可能为1个（在坐标为1或-1处）或2个（在坐标为1和-1处）。不可能是0个，因为 Foxen 必须能检测到1个。也不可能多于2个，因为这样一来唯一的一只 Foxen 就无法把它们都检测到。

第三个测试样例中，鼹鼠的初始状态无法满足让每只 Foxen 恰好检测到其中1只。

测试样例：

Input	Output
5 1 0 1 3 4 3 8 3 11 4 6	Case #1: 2 Case #2: 3 Case #3: -1 Case #4: 4 Case #5: 10

11 5 12 5 8 1 20 4 18 3 13 4 5 25 1 22 1 26 3 13 1 19 5 17 29 1 53 5 13 8 51 6 60 2 26 3 31 26 34 24 37 8 41 11 8 37 2 8 21 7 49 39 17 12 42 2 33 19	

Jacob Plachta（出题人）的算法:

题中所述鼹鼠和 Foxen 之间的约束关系可以用一个图来表征，每个顶点表示$O(N)$个被关心的位置（Foxen 所在的位置以及 Foxen 能探测到的位置）中的一个。对于每只 Foxen，在表征其所探测的两个位置的顶点之间建立一条边。

现给每个顶点着白色（若该位置有鼹鼠）或黑色（若该位置无鼹鼠）。对于初始鼹鼠探出地面的情况，顶点着色有两条限制——[1] 表征 Foxen 所在位置的顶点必须着黑色，[2] 每条边所连接的两个顶点着不同颜色。对上述图进行 2-着色的存在性判定和构造性算法，是一个存在贪心解的经典问题。这里值得注意的是如何处理那些必须着黑色的顶点（例如，可以先从这些顶点开始递归着色）。

假设上述 2-着色方案存在，那么对表征最终鼹鼠集合的顶点着色时第二条限制条件被放宽——即，对于每一条边，其两端点至少一个着白色，而不是之前的必须一黑一白。在这种情况下，着白色（有鼹鼠）顶点的最大可能数量为非必需着黑色的顶点数量。因此，如果我们能确定着白色顶点的最小数量，则最终答案就是二者之间的所有整数数量（包括二者本身），这是因为着白色的顶点数量为二者之间任意值也是可行的（把白色顶点逐个涂黑的过程不会破坏着色的合法性）。

着白色顶点的最小数量问题实质上等价于图的最小顶点覆盖，要求每条边都至少有一个白色顶点作为端点，尽管这里必须着黑色的顶点仍要特殊考虑（一种可能性是去掉这些点以及由其出发的边，然后处理剩下的图）。计算任意图的最小顶点覆盖是**NP-完全**的，但由第一步着色的结果可知该图具有二分性（因为 2-着色存在），根据 König 定理，这意味着最小顶点覆盖的大小（magnitude）等于最大匹配的大小。而欲求的最大匹配利用 Hopcroft-Karp 算法可以在$T\sim O(N\sqrt{N})$内得到，$T\sim O(N^2)$的算法也可以满足要求。

完整代码:

```
#include <queue>
#include <vector>
#include <map>
#include <iostream>

const int maxn = 5005 * 3;
const int INF = 1e9;

int K, A, B;
std::vector<int> con[maxn], con2[maxn];
bool col0[maxn];      // Mark those nodes which forced to be coloured black / 0.
int col[maxn], ind[maxn], p1[maxn], p2[maxn], dist[maxn];

std::map<int, int> mp;   // Node's position to node's index.
int get_index(int x) {   // Point position to corresponding node index.
    if (mp.find(x) != mp.end()) return mp[x];
    else return mp[x] = K++;
}

bool dfs(int i, int c) {
    // Conflict?
    if ((col0[i] && c > 0) || (col[i] >= 0 && c != col[i])) return false;
    // Already coloured?
    if (col[i] >= 0) return true;
    // Colour node and its neighbours.
    col[i] = c;
    for (int j = 0; j < con[i].size(); j++)
        if (!dfs(con[i][j], 1 - c)) return false;
    return true;
}

bool match_bfs() {
    std::queue<int> q;
    for (int i = 0; i < maxn; i++)
        dist[i] = INF;
    for (int a = 0; a < A; a++)
        if (p1[a] < 0) dist[a] = 0, q.push(a);
    while (!q.empty()) {
        int a = q.front();
```

```
            q.pop();
            if (dist[a] < dist[A])
                for (int i = 0; i < con2[a].size(); i++) {
                    int b = con2[a][i];
                    int a2 = p2[b];
                    if (dist[a2] >= INF) dist[a2] = dist[a] + 1, q.push(a2);
                }
        }
        return dist[A] < INF;
}

bool match_dfs(int a) {
        if (a == A) return true;
        for (int i = 0; i < con2[a].size(); i++) {
            int b = con2[a][i];
            int a2 = p2[b];
            if (dist[a2] == dist[a] + 1 && match_dfs(a2)) {
                p1[a] = b, p2[b] = a;
                return true;
            }
        }
        dist[a] = INF;
        return false;
}

int max_matching() {
        for (int i = 0; i < maxn; i++)
            p1[i] = -1;
        for (int b = 0; b < B; b++)
            p2[b] = A;
        int ans = 0;
        while (match_bfs())
            for (int a = 0; a < A; a++)
                if (p1[a] < 0 && match_dfs(a)) ans++;
        return ans;
}

void init() {
        for (int i = 0; i < K; i++)
            con[i].clear();
        for (int i = 0; i < A; i++)
            con2[i].clear();
        K = 0;
        mp.clear();
        for (int i = 0; i < maxn; i++)
```

```
        col0[i] = false, col[i] = ind[i] = p1[i] = p2[i] = -1;
}

void read() {
    init();
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        int p, r;
        scanf("%d %d", &p, &r);
        int u = get_index(p), v = get_index(p - r), w = get_index(p + r);
        col0[u] = true;
        con[v].emplace_back(w);
        con[w].emplace_back(v);
    }
}

int solve() {
    // Attempt to 2-color the graph (initially just from forced nodes).
    for (int z = 0; z < 2; z++)
        for (int i = 0; i < K; i++)
            if (col[i] < 0) {
                if (z == 0 && !col0[i]) continue;
                if (!dfs(i, 0)) return -1;
            }
    // Construct the bipartite graph, ignoring forced nodes.
    A = B = 0;
    for (int i = 0; i < K; i++)
        if (!col0[i]) {
            bool flag = false;
            for (int j = 0; j < con[i].size(); j++)
                if (col0[con[i][j]]) flag = true;
            if (!flag) {
                if (col[i] == 0) ind[i] = A++;
                else ind[i] = B++;
            }
        }
    for (int i = 0; i < K; i++)
        if (ind[i] >= 0 && col[i] == 0) {
            int a = ind[i];
            for (int j = 0; j < con[i].size(); j++) {
                int b = ind[con[i][j]];
                if (b >= 0) con2[a].emplace_back(b);
            }
        }
```

```
    return A + B - max_matching() + 1;
}

int main() {
    freopen("fox_moles_in.txt", "r", stdin);
    freopen("fox_moles_out.txt", "w", stdout);
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        int ans = solve();
        printf("Case #%d: %d\n", t, ans);
        fprintf(stderr, "Case #%d / %d: %d\n", t, T, ans);
    }
}
```

注 1：STL 中的关联容器 set, multiset, map, multimap 内部采用了一种十分高效的平衡二叉搜索树：红黑树（RB 树）。RB 树的统计性能要好于一般的平衡二叉树（如自平衡二叉查找树——AVL 树）。与之相对的，STL 中的 unordered_map 内部采用 hash 函数实现映射。

注 2（深搜着色函数）：

注 3（König 定理）：二分图的最小顶点覆盖等于最大匹配。

3.8 Hacker Cup 2017 Final Round

C. Fox Strolls (计数方法，计数问题)

题述：

森林中生长着N棵树，恰好每棵树的树顶住着一只 Fox！树从1到N编号，树根排列在一条直线上，相邻的编号为i和$i+1$的树根之间的距离为1米，第i棵树的高度为H_i米。

所有 Foxen 之间互相都是好朋友，经常出门溜达到别人家串门。相比于在树干之间跳跃，它们更喜欢让爪子附着在某些东西的表面上，比如树干或地面。如此一来，从第i棵树的树顶到第j棵树的树顶之间的最短路径需要爬下第i棵树，沿着地面爬到第j棵树的树根，然后在爬到树顶，总共的路程长度为$H_i + |j-i| + H_j$米。

然而，Foxen 们觉得它们当前需要走的路程太长了。由于需要频繁出门，它们决定投资搭建一些连接树干之间的桥来缩短出行路程。它们只愿意搭建长度为1米的、水平的桥，每座桥可以连接任意一对相邻的树i和$i+1$，并且可以搭建在树干上的任意高度，只要两端都能和树干接触（也就是说，最大高度不能超过$\min(H_i, H_{i+1})$）。相邻两棵树之间可以搭建高度不同的若干座桥。一旦桥被搭建，Foxen 一般更愿意走桥以省去爬到地上的时间。

共有$\frac{N(N-1)}{2}$对 Foxen 朋友(i,j)需要相见，那么就要从第i棵树的树顶走到第j棵树的树顶的路径（反之亦然）。

Foxen 们希望最小化所有$\frac{N(N-1)}{2}$条路径的长度之和，请帮助他们确定这个和的最小值。这里假设只修建达到要求所需数量的桥，也就是说，Foxen 们不希望把时间都用在建桥上，因而又产生了一个问题，它们希望知道为了达到这个路程距离和的最小值需要修建的桥的最小数量是多少。

输入：

开始一行一个整数T，表示森林的数量。每个森林由两行表述，第一行一个整数N为树的数量，第二行N个由空格分隔的整数，其中第i个整数为第i棵树的高度H_i。

输出：

对于第i个森林，输出“Case #x: ”，后跟两个整数：所有点对之间最短路径长度的和的最小值（以米为单位），以及为了达到这个最小值最少需要修建的桥的数量。

参数限制：

$1 \le T \le 30$

$2 \le N \le 500{,}000$

$1 \le H_i \le 10{,}000{,}000$

所有T个测试样例中的N值的和不超过$6{,}000{,}000$

样例解释：

测试样例 1 中，

测试样例：

Input	Output

5 3 30 20 10 5 1 2 3 2 1 7 8 3 7 11 8 8 6 7 6279148 9485166 9832201 1 8916058 9059232 7182836 15 172 472 972 472 493 736 487 384 918 856 124 975 376 846 173	Case #1: 44 2 Case #2: 30 6 Case #3: 158 9 Case #4: 213877494 8 Case #5: 67828 26

作者 Jacob Plachta 的算法:

这个问题可以被分解为两个独立的部分——计算成对距离和的最小值，以及计算为达到这一最小值所需修建最少的桥数。

对于第一部分，考虑有序数对(i,j)之间的最短路径（$i<j$），需要将高度下降到$[i,j]$之间最低的树高$h(i,j)$，而不用更低（假设已经修建了足够多的桥），因此这对树顶之间路径的长度为$d(i,j)=H_i+H_j-2h(i,j)+j-i$。上式对所有点对$(i,j)$求和$\sum_{(i,j)} d(i,j)$的结果除了$-2h(i,j)$这一项之外其他部分可以按序号迭代在$T\sim O(N)$内完成，但计算$-\sum_{(i,j)} 2h(i,j)$这一项中每棵树的高度$H_i$出现的频次是一个棘手的问题。一种方法是按照树的高度从低到高非降序迭代，同时维护一个已访问编号的集合。在处理树i的时候，找到在i的左边或右边最近的已访问过的树的编号a和b（则(a,b)之间的所有树高都大于等于树i），注意到对于那些左端点$p\in[a+1,i-1]$，右端点$q\in[i+1,b-1]$的点对(p,q)，其间路径的最低高度为$h(p,q)=H_i$（为便于表述，如果a,b不存在，则将其分别置为0或$N+1$）。基于这一观察，我们的做法是在将序号i插入集合之前在结果中减去$H_i*(i-a)*(b-i)$。整个过程可以在$T\sim O(N\log N)$内完成。

简而言之，对于第一部分:

· $d(i,j)=H_i+H_j-2h(i,j)+(j-i)$ for all $i<j$.

· $ans1=\sum_{i<j} d(i,j)$

问题的第二部分有多种解决方案。一种解决方案是从左到右迭代所有树木，并维护一个“下包络”用以表征从当前位置到左侧所有树顶的最短路径所必须沿直线（垂直或水平）直接到达的位置。这个包络可以用一个点(x,y)的堆栈表示，具体地，每当遇到一棵高于等于前树的树时，就要构造一座连接前树顶端和当前树的桥并相应地扩展包络，否则，删除包络中所有高于当前树顶的点，并用当前树顶高度的桥替换。整个过程可以在$T\sim O(N)$内完成。

完整代码:

```
#include <algorithm>
#include <set>
#include <stack>
#include <iostream>

using ll = long long;

const int maxn = 500005;
```

```
int n;
int H[maxn];
std::pair<int, int> p[maxn];
void read() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++) {
		scanf("%d", &H[i]);
		p[i] = std::make_pair(H[i], i);
	}
}

std::set<int> st;
std::stack<std::pair<int, int>> stk;
std::pair<ll, ll> solve() {
	// Part1: sum of distances.
	ll ans1 = 0;
	// Get base sum assuming no bridges.
	for (int i = 0; i < n; i++)
		ans1 += H[i];
	ans1 *= n - 1;
	for (int i = 1; i < n; i++)
		ans1 += 1ll * i * (n - i);
	// Process trees bottom to top.
	std::sort(p, p + n);
	st.clear();
	st.insert(-1), st.insert(n);
	for (int i = 0; i < n; i++) {
		int a, b, h, j;
		std::tie(h, j) = p[i];
		// Find interval of paths having this as their shortest tree.
		auto tmp = st.lower_bound(j);
		b = *tmp - 1;
		tmp--;
		a = *tmp + 1;
		// Substract total distance saved by bridges.
		ans1 -= 1ll * h * 2 * (1ll * (j - a) * (b - j) + (b - a));
		// Insert this tree.
		st.insert(j);
	}

	// Part2: number of bridges.
	ll ans2 = 0;
	// Process trees left to right.
	while (!stk.empty()) stk.pop();
	stk.push(std::make_pair(0, 0));
```

```
    for (int i = 1; i < n; i++)
        if (H[i] >= H[i - 1]) {
            // At least as high as envelope's highest point, so build single bridge backwards.
            if (stk.top().first == H[i - 1]) stk.pop();
            stk.push(std::make_pair(H[i - 1], i));
            ans2++;
        } else {
            // Clip off higher parts of envelope.
            int j = i - 1;
            while (stk.top().first > H[i])
                j = stk.top().second, stk.pop();
            // Build row of bridges back across, and update envelope.
            if (!stk.top().first) ans2 += i - j;
            else {
                ans2 += i - stk.top().second;
                int h = stk.top().first; stk.pop();
                if (H[i] != h) stk.push(std::make_pair(h, j));
            }
            stk.push(std::make_pair(H[i], i));
        }

    return std::make_pair(ans1, ans2);
}

int main() {
    freopen("fox_strolls_in.txt", "r", stdin);
    freopen("fox_strolls_out.txt", "w", stdout);
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; t++) {
        read();
        ll ans1, ans2;
        std::tie(ans1, ans2) = solve();
        printf("Case #%d: %lld %lld\n", t, ans1, ans2);
        fprintf(stderr, "Case #%d/%d: %lld %lld\n", t, T, ans1, ans2);
    }
}
```

注 1（关于线性时间计数法）：一般来说当我们要统计一个组记录的综合，一种提升运算速度的做法是识别记录的类别和频次，然后在类别数量的时间内求的结果。例如，本题中求解$\sum_{i<j} j-i$这一步，问题空间的大小为n以内数对(i,j)的数量（其中$i<j$），即$C_n^2=\frac{n(n-1)}{2}$，故遍历求和的时间复杂度为$T\sim O(n^2)$。如果注意到问题空间的构成为：

· $n-1$个1

· $n-2$个2

…

· 1个$n-1$

则可以用取值乘频次的方法快速得到求和结果$\sum_{i=1}^{n-1} i(n-i)$，时间复杂度为取值类别的线性项$T\sim O(n)$。

第四章：面试篇

——科学备战 SDE 技术面试

4.1 技术面试需要掌握的知识
4.2 设计一份性感的简历

4.1 技术面试准备

1. 你需要掌握的知识

大多数面试官都不会问你平衡二叉树具体算法或其他复杂问题。老实说，离开学校这么多年，恐怕他们自己也记不清这些算法了。

一般来说，你只要掌握基本知识即可，下面这份清单列出了必须掌握的知识：

数据结构	算法	概念
链表	广度优先搜索	位操作
二叉树	深度优先搜索	单例设计模式
单词查找数（Trie）	二分搜索	工厂设计模式
栈	归并排序	内存（栈和堆）
队列	快速排序	递归
向量/数组列表	树的插入/查找等	大O时间
散列表		

对于上述各主题，请务必掌握他们的具体实现和用法、应用场景、空间和时间复杂度。

对于其中的数据结构和算法，你还要练习如何从无到有地实现。面试官可能会要求你直接实现一种数据结构或算法，或对其进行修改。不管怎样吗，你越熟悉具体实现，把握就越大。

其中，散列表一项特别重要。你会发现，解决面试问题时，经常会用到散列表。

2. 2 的幂表

下面这张表内的信息要很熟悉，回答可扩展性问题时，这张表用处很大，借助它可以快速算出一组数据占用多少时间。

指数	准确值	近似值	字节转换成 MB、GB 等
7	128		
8	256		
10	1024	Kilo	1K
16	65,536		64K
20	1,048,576	Million	1MB
30	1,073,741,824	Billion	1GB
32	4,294,967,296		4GB
40	1,099,511,627,776	Trillion	1TB

有了这张表，就可以做速算。例如，一个将每个32位整数映射为布尔值的散列表可以把一台计算机的内存填满（4GB）。

X. 我想问面试官的问题

我就想做机器学习软件工程师，请给我讲述这个职位的工作和指责。

附录

附录 A:

在 Xcode 上的程序计时:

```
int main() {
    clock_t start, finish;
    start = clock();

    ...

    finish = clock();
    double totle_time = (double)(finish − start) ／ CLOCKS_PER_SEC;
    printf("Totle time: %lf s\n”  , totle_time);
}
```

附录 B: 各 OJ 平台的精度要求。

CodeJam:

绝对或相对误差不超过10^{-6}时答案被接受。

用于检查绝对或相对误差的 Python 代码:

```
def IsApproximatelyEqual(x, y, epsilon):
"""Returns True iff y is within relative or absolute 'epsilon' of x.

By default, 'epsilon' is 1e-6.
"""
# Check absolute precision.
if -epsilon <= x - y <= epsilon:
return True

# Is x or y too close to zero?
if -epsilon <= x <= epsilon or -epsilon <= y <= epsilon:
return False

# Check relative precision.
return (-epsilon <= (x - y) / x <= epsilon
or -epsilon <= (x - y) / y <= epsilon)
```

Codeforces:

当且仅当绝对或相对误差不超过10^{-6}时答案被视为正确的。具体地，假设所提交的答案为a，参考答案为b，则满足$\frac{|a-b|}{\max(1,b)} \leq 10^{-6}$时答案被接受。

附录 C: C++标准库中的常用函数

C1. 二分搜索（#include <algorithm>）

在**有序容器**中的二分搜索函数:

C1.1 搜索下界

声明:

std::lower_bound

· template<class ForwardIterator, class T>

ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const T& val);

· template<class ForwardIterator, class T, class Compare>

ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const T& val, Compare comp);

返回指向区间[first, last)内第一个不小于 val 的元素的迭代器（指向下界的迭代器）。

范围中的元素应已用相应比较方法（operator<或 comp）排序，或至少根据相对 val 的大小分隔。

参数:

· first, last:

指向一个有序（或合理分边）序列首尾的前向迭代器，函数作用区间为[first, last)。

· val:

区间内搜索值的下界;

对于第一种实现，T 应为可以与[first, last)之间的数据类型通过 operator<比较（T 作为右运算元）的数据类型。

· comp:

接收两个参数（第一个由 ForwardIterator 指向，第二个总是 val）的二分函数，返回值可转换为 bool 型。返回第一个参数排序是否位于第二个参数前面。

函数不改变输入参数;

可以为函数指针或函数对象。

返回值:

返回区间内指向 val 的下界的迭代器。

如果区间内元素全部小于 val，则返回 last。

C1.2 搜索上界

声明:

std::upper_bound

· template<class ForwardIterator, class T>

ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const T& val);

· template<class ForwardIterator, class T, class Compare>

ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const T& val, Compare comp);

返回指向区间[first, last)内第一个大于 val 的元素的迭代器（指向上界的迭代器）。

范围中的元素应已用相应比较方法（operator<或 comp）排序，或至少根据相对 val 的大小分隔。

参数:

· first, last:

指向一个有序（或合理分边）序列首尾的前向迭代器，函数作用区间为[first, last)。

· val:

区间内搜索值的上界;

对于第一种实现，T 应为可以与[first, last)之间的数据类型通过 operator<比较（T 作为左运算元）的数据类型。

· comp:

接收两个参数（第一个由 ForwardIterator 指向，第二个总是 val）的二分函数，返回值可转换为 bool 型。返回第一个参数排序是否位于第二个参数前面。

函数不改变输入参数;

可以为函数指针或函数对象。

返回值:

返回区间内指向 val 的上界的迭代器。

如果区间内元素全部不大于（小于等于）val，则返回 last。

C1.2 查找（二分搜索）

声明:

std::binary_search

· template<class ForwardIterator, class T>

bool binary_search(ForwardIterator first, ForwardIterator last, const T& val);

· template<class ForwardIterator, class T, class Compare>

bool binary_search(ForwardIterator first, ForwardIterator last, const T& val, Compare comp);

返回 true 如果区间[first, last)中有元素和 val equivalent，否则返回 false。

个体元素使用 operator<或 comp 进行比较。两个元素 a, b 视作 equivalent if (!(a < b) && !(b < a))或 if (!comp(a, b) && !comp(b, a))。

范围中的元素应已用相应比较方法（operator<或 comp）排序，或至少根据相对 val 的大小分隔。

参数:

· first, last:

指向一个有序（或合理分边）序列首尾的前向迭代器，函数作用区间为[first, last)。

· val

区间内待搜索的值;

对于第一种实现，T 应为可以作为左/右运算元与[first, last)之间的数据类型通过 operator<进行比较的数据类型。

· comp:

接收两个参数（一个由 ForwardIterator 指向，另一个为 val）的二分函数，返回值可转换为 bool 型。返回第一个参数排序是否位于第二个参数前面。

函数不改变输入参数;

可以为函数指针或函数对象。

返回值:

true: 如果[first, last)中有元素与 val equivalent，否则 false。

C2. 全排列生成器（#include <algorithm>）

C2.1.1 下一排列

声明:

std::next_permutation

· template<class bidirectional_iterator>

bool next_permutation(bidirectional_iterator first, bidirectional_iterator last);

· template<class bidirectional_iterator, class compare>

bool next_permutation(bidirectional_iterator first, bidirectional_iterator last, compare comp);

*N*个元素的排列有*N*!种可能，其中不同的排列按照“字典顺序”相互比较大小，其中最小的排列为升序排列，最大的则是降序排列。

个体元素之间使用 operator<（第一种）比较大小或者使用 comp 比较大小。函数将两个迭代器之间的元素重排成字典序更大的下一个排列并返回 true，如果输入迭代器之间的元素已经为降序排列（具有最大字典序），则函数将元素重排为升序（具有最小字典序）并返回 false 表示进入新一轮循环。

参数:

· first, last:

指向序列起止的双向迭代器，表示范围[first, last)，bidirectional_iterator 应为 std::swap 函数定义数据类型的迭代器。

· comp:

接收两个由 bidirectional_iterator 所指向类型参数的二分类函数，其返回值可转化为 bool 型。其返回值表示了按其所定义的严格若排序第一个参数是否在第二个参数前面。

该函数不改变输入参数;

该参数可以为函数指针或函数对象。

返回值:

· true: 如果函数可以将范围内元素重排为字典序较大的下一个排列;

· false: 函数将范围内元素重排为字典序最小的排列（升序排列）。

C2.1.2 前一排列

声明:

std::next_permutation

· template<class bidirectional_iterator>

bool prev_permutation(bidirectional_iterator first, bidirectional_iterator last);

· template<class bidirectional_iterator, class compare>

bool prev_permutation(bidirectional_iterator first, bidirectional_iterator last, compare comp);

*N*个元素的排列有*N*!种可能，其中不同的排列按照“字典顺序”相互比较大小，其中最小的排列为升序排列，最大的则是降序排列。

个体元素之间使用 operator<（第一种）比较大小或者使用 comp 比较大小。函数将两个迭代器之间的元素重排成字典序更小的下一个排列并返回 true，如果输入迭代器之间的元素已经为升序排列（具有最小字典序），则函数将元素重排为降序（具有最大字典序）并返回 false 表示进入新一轮循环。

参数:

· first, last:

指向序列起止的双向迭代器，表示范围[first, last)，bidirectional_iterator 应为 std::swap 函数定义数据类型的迭代器。

· comp:

接收两个由 bidirectional_iterator 所指向类型参数的二分类函数，其返回值可转化为 bool 型。其返回值表示了按其所定义的严格若排序第一个参数是否在第二个参数前面。

该函数不改变输入参数;

该参数可以为函数指针或函数对象。

返回值:

· true: 如果函数可以将范围内元素重排为字典序较小的下一个排列;

· false: 函数将范围内元素重排为字典序最大的排列（降序排列）。

时间复杂度的上界为$T\sim O(n)$。

附录 D: C++中的随机数生成函数:

· rand() 生成成[0, INT32_MAX]之间均与分布的随机整数——#include <stdlib.h>

· srand(unsigned) 设置随机数种子一般为 time(NULL)

常用分布生成方式:

1. 生成$[-1, 1]$之间均匀分布的随机数: 2.0 * rand() / RAND_MAX − 1.0

——精度为$\frac{1}{\text{INT32_MAX}}$。

2. 生成$[1, 10]$之间均匀分布的整数: rand() % 10 + 1