Standard Code Library

Aestas16

Huazhong University of Science and Technology

November 29, 2024

Contents

工具

Linux 下对拍

```
make gen && make test && make std
while true; do
    ./gen > data.in
    ./test < data.in > test.out
    ./std < data.in > std.out
    if diff -w test.out std.out; then
        printf "AC\n"
    else
        printf "WA\n"
        exit 0
    fi
done
```

缺省源

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

using namespace std;

template <class T> void fr(T &a, bool f = 0, char ch = getchar()) {
    for (a = 0; ch < '0' || ch > '9'; ch = getchar()) ch == '-' ? f = 1 : 0;
    for (; ch >= '0' && ch <= '9'; ch = getchar()) a = a * 10 + (ch - '0');
    a = f ? -a : a;
}
template <class T, class... Y> void fr(T &t, Y &... a) { fr(t), fr(a...); }
int fr() { int a; return fr(a), a; }

signed main() {

    return 0;
}
```

基础算法

哈希

树哈希

dmy 牛逼！参考自 dmy 的博客。

考虑这样一种哈希方式。对于一棵以 u 为根的子树，定义子树的哈希 $\mathrm{Hash}(u) = 1 + \sum_{(u,v)} f(\mathrm{Hash}(v))$。其中 f 为一个待定函数。

一种比较优秀的 f 函数如下：

```
ull h(ull x) { return x * x * x * 672328094 + 22637261; }
ull f(ull x) { return h((x << 32) >> 32) + h(x >> 32); }
```

例题：UOJ #763 树哈希。

```
const int N = 1e6;

int n;
ull hsh[N + 10];
vector<int> e[N + 10];

void adde(int x, int y) { e[x].push_back(y); }

ull h(ull x) { return x * x * x * 672328094 + 22637261; }
ull f(ull x) { return h((x << 32) >> 32) + h(x >> 32); }

void dfs(int u, int p) {
    hsh[u] = 307;
    for (auto v : e[u]) if (v != p) dfs(v, u), hsh[u] += f(hsh[v]);
}
```

```
signed main() {
    fr(n);
    for (int i = 1, u, v; i < n; i++) fr(u, v), adde(u, v), adde(v, u);
    dfs(1, 0), sort(hsh + 1, hsh + 1 + n), printf("%d\n", unique(hsh + 1, hsh + 1 + n) - hsh - 1);
    return 0;
}
```

二分

整体二分

顾名思义，把多个询问放在一起二分答案。

记 $\text{solve}(l, r, Q)$ 表示当前正在处理集合 Q 中的询问，且集合 Q 中的询问答案在 $[l, r]$ 内，我们对于 Q 中每个询问查询 $\text{mid} = \dfrac{l+r}{2}$ 是否合法，然后就可以将 Q 分为 Q_0, Q_1，其中 Q_0 的答案值域为 $[l, \text{mid})$，Q_1 的答案值域为 $[\text{mid}, r]$。

时间复杂度 $\Theta(Q \cdot a \log \text{maxans} + \text{maxans} \cdot b)$，其中查询的时间为 $\Theta(a)$，修改的时间为 $\Theta(b)$。

例题：YZOJ P5680 [FOI 2022 多校联训 Round 12] 帝国防卫（校内题）。

```
#define int unsigned long long

const int N = 3e5;

int n, m, k, tim, ans[N + 10], tmp[N + 10], qwq[N + 10], cval[N + 10], L[N + 10], R[N + 10], val[N + 10];
vector<int> vec[N + 10];

int lowbit(int x) { return x & -x; }
void update(int pos, int val) {
    for (int i = pos; i <= m; i += lowbit(i)) cval[i] += val;
}
int qsum(int pos) {
    int ret = 0;
    for (int i = pos; i; i -= lowbit(i)) ret += cval[i];
    return ret;
}
void modify(int l, int r, int val) {
    if (l <= r) update(l, val), update(r + 1, -val);
    else update(1, val), update(r + 1, -val), update(l, val);
}

void solve(int l, int r, vector<int> Q) {
    if (Q.empty()) return;
    if (l == r) {
        for (auto i : Q) ans[i] = l;
        return ;
    }
    int mid = (l + r) >> 1; vector<int> ql, qr;
    while (tim < mid) tim++, modify(L[tim], R[tim], val[tim]);
    while (tim > mid) modify(L[tim], R[tim], -val[tim]), tim--;
    for (auto i : Q) {
        tmp[i] = 0;
        for (auto j : vec[i]) tmp[i] += qsum(j);
        if (tmp[i] >= qwq[i]) ql.push_back(i);
        else qr.push_back(i);
    }
    solve(l, mid, ql), solve(mid + 1, r, qr);
}

signed main() {
    fr(n, m); vector<int> Q;
    for (int i = 1; i <= m; i++) vec[fr()].push_back(i);
    for (int i = 1; i <= n; i++) fr(qwq[i]), Q.push_back(i);
    fr(k);
    for (int i = 1; i <= k; i++) fr(L[i], R[i], val[i]);
    k++, L[k] = 1, R[k] = m, val[k] = 0x3f3f3f3f, solve(1, k, Q);
    for (int i = 1; i <= n; i++)
        if (ans[i] == k) puts("NIE");
        else printf("%llu\n", ans[i]);
```

```
    return 0;
}
```

wqs 二分

高级的东西。

如果有一个凸函数 $f(x)$ $(x \in A)$，而我们能在 $\Theta(m)$ 的时间内求出它的最大值（对于上凸函数）或最小值（对于下凸函数），那么我们就能 $\Theta(m \log |A|)$ 求出 $f(x)$ 在 p 处的取值。

我们发现，对于一个凸函数，它的导函数一定是单调增/减的，也就是说我们可以不断二分凸壳上的切点斜率，直到切到点 p 为止。

那么我们要怎么求出某个斜率 k 切到凸壳上的哪个点呢？注意到对于一个上凸壳，直线 $y = kx + b$ 与其相切时，b 最大，下凸壳则 b 最小。然后我们有 $b = y - kx$，那么我们把与 x 相关的量扣去 k 的贡献，然后对其求最大值即可。下凸壳同理可得。

这就是 wqs 二分！注意编写时的细节：由于 $f(x)$ 可能包含多点共线，也就是说 $(p, f(p))$ 可能不会被切到，而是切到与其共线的点上，因此每次二分时更新答案都应按照切到点 p 来更新答案。

关于 $\log |A|$：其实斜率的范围应该不完全等于值域范围？做题时得具体分析斜率范围。

例题：洛谷 P2619 [国家集训队]Tree I。

```
#define int long long

const int N = 5e4, M = 1e5;

int n, m, nd, fa[N + 10];
struct Edge { int u, v, w, col; } e[M + 10];

int find(int u) { return fa[u] == u ? u : fa[u] = find(fa[u]); }

pair<int, int> solve(int k) {
    for (int i = 1; i <= m; i++) if (e[i].col == 0) e[i].w -= k;
    sort(e + 1, e + 1 + m, [] (Edge a, Edge b) { return a.w == b.w ? a.col < b.col : a.w < b.w; });
    for (int i = 1; i <= n; i++) fa[i] = i;
    int cnt = 0, ans = 0;
    for (int i = 1, tot = 0; tot != n - 1 && i <= m; i++) {
        int x = find(e[i].u), y = find(e[i].v);
        if (x == y) continue;
        tot++, cnt += (e[i].col == 0), ans += e[i].w, fa[x] = y;
    }
    for (int i = 1; i <= m; i++) if (e[i].col == 0) e[i].w += k;
    return {cnt, ans};
}

signed main() {
    fr(n, m, nd);
    for (int i = 1; i <= m; i++) fr(e[i].u, e[i].v, e[i].w, e[i].col), e[i].u++, e[i].v++;
    int l = -200, r = 200, ans = 0;
    while (l <= r) {
        int mid = (l + r) >> 1; auto ret = solve(mid);
        if (ret.first >= nd) ans = ret.second + mid * nd, r = mid - 1;
        else l = mid + 1;
    }
    return printf("%lld\n", ans), 0;
}
```

分治

CDQ 分治

用于给问题降维。如高维点对计数、计算高维点对权值、优化多维偏序 DP 过程、动态问题转静态等。

分治过程：假设当前在处理 $[l, r]$ 内的点对，那么我们只计算跨过 $\text{mid} = \dfrac{l + r}{2}$ 的点对贡献，剩下的递归到 $[l, \text{mid}]$、$[\text{mid} + 1, r]$ 内处理。下面介绍一些具体的例子：

三维偏序：将第一维排序后开始 CDQ 分治，我们需要计算跨过 mid 的点对数量，那么我们现在将 $[l, \text{mid}], [\text{mid} + 1, r]$ 内均按第一维排序，然后双指针将满足 $b_l < b_r$ 的位置 r 扔进树状数组，然后用树状数组进行第三维偏序的计数即可。时间复杂度 $\Theta(n \log^2 n)$。

优化 1D/1D 偏序 DP：假设我们有转移方程：

$$f_i = 1 + \max_{j=1}^{i-1}\{f_j \mid a_j < a_i \wedge b_j < b_i\}$$

这显然是一个三维偏序的转移，第一维 i 已默认排好序，直接 CDQ 分治：先递归处理 $[l, \text{mid}]$ 的转移，再处理跨过 mid 的转移（同样使用双指针 + 树状数组维护前缀最大值），最后递归处理 $[\text{mid}+1, r]$ 的转移。注意这里处理过程必须按照中序遍历的顺序，原因显然。时间复杂度 $\Theta(n \log^2 n)$。

动态问题转静态：将所有的操作多加一维时间作为第一维，然后使用 CDQ 分治处理，此时所有的修改都在询问之前就已完成，我们只需要处理静态的问题即可。例如矩阵加与矩阵求和操作，使用 CDQ 分治后就变为静态的矩形面积并，线段树维护即可，不需要树套树。实现了问题的「降维」。

再说一个技巧：**CDQ 分治套 CDQ 分治**。

我们上述问题中计算跨过 mid 的贡献时都使用了数据结构，但此时我们其实可以转换问题，再使用一个内层的 CDQ 分治解决。

以三维偏序为例，我们考虑计算跨过 mid 的贡献时问题转换为了什么：1. 将第一维在 $[l, \text{mid}]$ 中的点加入到点集中；2. 对于每个第一维在 $[\text{mid}+1, r]$ 中的点，计算有点集内有多少个点的第二维第三维都比它小。

然后考虑用 CDQ 分治解决这个问题，将第二维排序，问题变为顺序枚举每个点，然后做如下操作：1. 遇到第一维在 $[l, \text{mid}]$ 中的点，将其加入到点集中；2. 遇到第一维在 $[\text{mid}+1, r]$ 中的点，计算有点集内有多少个点的第三维比它小。

也就是说我们需要维护一个动态的一维偏序，化用上面的**动态问题转静态**套路，我们使用 CDQ 分治处理这个问题后，只需要解决静态的一维偏序，前缀和即可。

CDQ 分治套 CDQ 分治的时间复杂度仍为 $\Theta(n \log^2 n)$。

例题：洛谷 P3157 [CQOI2011] 动态逆序对，洛谷 P2487 [SDOI2011] 拦截导弹。

三维偏序：

```cpp
const int N = 1e5;

struct Node {
    int val, c, b, a;
} arr[(N << 1) + 10];

long long ans[N + 10];
int n, m, atot, pos[N + 10], a[N + 10];

namespace BIT {
    int bval[N + 10];
    int lowbit(int x) { return x & -x; }
    void add(int pos, int val) {
        for (int i = pos; i <= n; i += lowbit(i)) bval[i] += val;
    }
    int qsum(int pos) {
        int ret = 0;
        for (int i = pos; i; i -= lowbit(i)) ret += bval[i];
        return ret;
    }
} // namespace BIT
using namespace BIT;

bool cmp2d(Node x, Node y) { return x.b < y.b; }

void cdqdiv(int l, int r) {
    if (l == r) return;
    int mid = (l + r) / 2, j = l;
    cdqdiv(l, mid), cdqdiv(mid + 1, r), sort(arr + l, arr + mid + 1, cmp2d), sort(arr + mid + 1, arr + r + 1, cmp2d);
    for (int i = mid + 1; i <= r; i++) {
        while (arr[j].b <= arr[i].b && j <= mid) add(arr[j].c, arr[j].val), j++;
        ans[arr[i].a] += arr[i].val * (qsum(n) - qsum(arr[i].c));
    }
    for (int i = l; i < j; i++) add(arr[i].c, -arr[i].val);
    j = mid;
    for (int i = r; i > mid; i--) {
        while (arr[j].b >= arr[i].b && j >= l) add(arr[j].c, arr[j].val), j--;
```

```
        ans[arr[i].a] += arr[i].val * qsum(arr[i].c - 1);
    }
    for (int i = mid; i > j; i--) add(arr[i].c, -arr[i].val);
}

int main() {
    fr(n), fr(m);
    for (int i = 1; i <= n; i++) fr(a[i]), pos[a[i]] = i, arr[++atot] = {1, a[i], i, 0};
    for (int i = 1, x; i <= m; i++) fr(x), arr[++atot] = {-1, x, pos[x], i};
    cdqdiv(1, atot);
    for (int i = 1; i <= m; i++) ans[i] += ans[i - 1];
    for (int i = 0; i < m; i++) printf("%lld\n", ans[i]);
    return 0;
}
```

优化 DP:

```
const int N = 5e4;

struct Node {
    int a, b, c, f;
    double p;
} arr[N + 10];

int n, ans, f1[N + 10], f2[N + 10];
double k, p1[N + 10], p2[N + 10];

namespace BIT {
    int fval[N + 10];
    double pval[N + 10];
    int lowbit(int x) { return x & -x; }
    void add(int pos, int fv, double pv) {
        for (int i = pos; i <= n; i += lowbit(i))
            if (fval[i] < fv)
                fval[i] = fv, pval[i] = pv;
            else if (fval[i] == fv)
                pval[i] += pv;
    }
    void erase(int pos) {
        for (int i = pos; i <= n; i += lowbit(i)) fval[i] = pval[i] = 0;
    }
    int qfmax(int pos) {
        int ret = 0;
        for (int i = pos; i; i -= lowbit(i)) ret = max(ret, fval[i]);
        return ret;
    }
    double qpsum(int pos, int fv) {
        double ret = 0;
        for (int i = pos; i; i -= lowbit(i))
            if (fval[i] == fv) ret += pval[i];
        return ret;
    }
} // namespace BIT
using namespace BIT;

bool cmp1d1(Node x, Node y) {
    if (x.a == y.a) {
        if (x.b == y.b) return x.c < y.c;
        return x.b > y.b;
    }
    return x.a > y.a;
}
bool cmp1d2(Node x, Node y) {
    if (x.a == y.a) {
        if (x.b == y.b) return x.c < y.c;
        return x.b < y.b;
    }
    return x.a < y.a;
}
bool cmp2d1(Node x, Node y) { return x.b > y.b; }
bool cmp2d2(Node x, Node y) { return x.b < y.b; }

```

```
bool cmp(int j, int i, int opt) {
    if (opt == 0) return arr[j].b >= arr[i].b;
    return arr[j].b <= arr[i].b;
}

void cdqdiv(int l, int r, int opt) {
    if (l == r) return;
    int mid = (l + r) / 2, j = l;
    cdqdiv(l, mid, opt);
    if (opt == 0) sort(arr + l, arr + mid + 1, cmp2d1), sort(arr + mid + 1, arr + r + 1, cmp2d1);
    else sort(arr + l, arr + mid + 1, cmp2d2), sort(arr + mid + 1, arr + r + 1, cmp2d2);
    for (int i = mid + 1; i <= r; i++) {
        while (cmp(j, i, opt) && j <= mid) add(arr[j].c, arr[j].f, arr[j].p), j++;
        int tmp = qfmax(arr[i].c) + 1;
        if (arr[i].f < tmp) arr[i].f = tmp, arr[i].p = qpsum(arr[i].c, tmp - 1);
        else if (arr[i].f == tmp) arr[i].p += qpsum(arr[i].c, tmp - 1);
    }
    for (int i = l; i < j; i++) erase(arr[i].c);
    if (opt == 0) sort(arr + mid + 1, arr + r + 1, cmp1d1);
    else sort(arr + mid + 1, arr + r + 1, cmp1d2);
    cdqdiv(mid + 1, r, opt);
}

int main() {
    fr(n);
    for (int i = 1; i <= n; i++) fr(arr[i].a), fr(arr[i].b), arr[i].c = i, arr[i].f = arr[i].p = 1;
    sort(arr + 1, arr + n + 1, cmp1d1), cdqdiv(1, n, 0);
    for (int i = 1; i <= n; i++) f1[arr[i].c] = arr[i].f, p1[arr[i].c] = arr[i].p, ans = max(ans, arr[i].f);
    printf("%d\n", ans);
    for (int i = 1; i <= n; i++) arr[i].c = n - arr[i].c + 1, arr[i].f = arr[i].p = 1;
    sort(arr + 1, arr + n + 1, cmp1d2), cdqdiv(1, n, 1);
    for (int i = 1; i <= n; i++) {
        f2[n - arr[i].c + 1] = arr[i].f, p2[n - arr[i].c + 1] = arr[i].p;
        if (arr[i].f == ans) k += arr[i].p;
    }
    for (int i = 1; i <= n; i++)
        if (f1[i] + f2[i] == ans + 1) printf("%.5lf ", p1[i] * p2[i] / k);
        else printf("0.00000 ");
    return puts(""), 0;
}
```

字符串

Manacher

大致思想是维护一个最长回文半径以及它的中心，然后由于回文的对称性，可以加速回文半径的转移。

```
scanf("%s", str + 1); int ans = 0, n = 0; s[0] = '$', s[++n] = '#';
for (int i = 1; str[i]; i++) s[++n] = str[i], s[++n] = '#';
for (int i = 1, sr = 0, smid = 0; i <= n; i++) {
    if (i < sr) p[i] = min(sr - i, p[smid * 2 - i]);
    while (s[i + p[i] + 1] == s[i - p[i] - 1]) p[i]++;
    if (i + p[i] > sr) sr = i + p[i], smid = i;
    ans = max(ans, p[i]);
}
```

最小表示法

吗的，每一次都记不住。太冷门了这东西。

维护两个指针 i, j: - 如果 $a_i \neq a_j$，保留小的指针，将大的指针移动一位。- 如果 $a_i = a_j$，那么假设 $a_{i:i+k}$ 与 $a_{j:j+k}$ 均相等，而 a_{i+k+1} 与 a_{j+k+1} 不相等，同样保留小的指针，但是此时大的指针可以移动 $k+1$ 位，原因显然。- 怎么维护 k 呢？直接维护即可。

```
// 记得将 a 在末尾复制一遍。
int minarr() {
    int i = 1, j = 2, k = 0;
    while (i <= n && j <= n && k < n)
        if (a[i + k] == a[j + k]) k++;
        else {
```

```
            if (a[i + k] > a[j + k]) i += k + 1;
            else j += k + 1;
            if (i == j) i++;
            k = 0;
        }
    return min(i, j);
}
```

AC 自动机 & Fail 树

多模式串单文本串匹配。

先把所有模式串插进 trie 里，然后建出 trie 图和 fail 指针就得到 ACAM 了！

怎么建：- bfs Trie 树，如果 u 有出边 i，则 $ch_{u,i}$ 的 fail 就赋值为 u 的 fail 的出边 i，然后把出点加进队列里 bfs 用；- 反之我们直接更新 trie 的结构让其变为 trie 图，具体地，我们令 $ch_{u,i} = ch_{fail_u,i}$。

```
void Build() {
    Hd = 1, Tl = 0;
    for (int i = 0; i < 26; i++)
        if (ch[0][i]) que[++Tl] = ch[0][i];
    while (Hd <= Tl) {
        int u = que[Hd++];
        for (int i = 0; i < 26; i++) {
            if (ch[u][i])
                fail[ch[u][i]] = ch[fail[u]][i], que[++Tl] = ch[u][i];
            else
                ch[u][i] = ch[fail[u]][i];
        }
    }
}
```

然后怎么匹配文本串呢？暴力的做法是文本串每在 Trie 图上走到一个新的节点就不断跳 fail 指针把路径中所有的串都计入答案，但是这样太慢了很容易 T 飞，于是我们考虑优化这个过程，发现每个点只有一个 fail 指针，所以 fail 指针构成了一个树的结构，而不断跳 fail 指针则是在树上从一个点跳到根节点。

有了这个性质我们就可以建出 fail 树然后在树上用一些东西来维护答案。没做多少题所以不懂啥技巧，考场上现场推吧。

字符串哈希

```
int Hash(int l, int r) { return (sum[r] - (1ll * sum[l - 1] * Pow[r - l + 1] % P) + P) % P; }

Pow[0] = 1;
for (int i = 1; i <= n; i++) sum[i] = (1ll * sum[i - 1] * BASE % P + h(str[i])) % P, Pow[i] = 1ll * Pow[i - 1] * BASE
 ↪  % P;
```

KMP

记 $\text{Preffix}(S, i)$ 为字符串 S 中长度为 i 的前缀，$\text{Suffix}(S, i)$ 为字符串 S 中长度为 i 的后缀。

对于一个字符串 S 和 $0 \le k \le |S|$，其中最大的 k 满足 $\text{Preffix}(S, k)$ 与 $\text{Suffix}(S, k)$ 相等，则称 $\text{Preffix}(S, k)$ 为 S 的 border。

对于一个长度为 n 的字符串 S，前缀函数 $\pi(i)$ 即为 $\text{Preffix}(S, i)$ 的 border 长度。

前缀函数 π 即为 KMP 算法中的 next 或 kmp 数组。

假设我们要求 $\pi(i)$，且对于所有的 $j(1 \le j < i)$，$\pi(j)$ 均已计算完成。

显然若 S_i 与 $S_{\pi(i-1)+1}$ 相等，则 $\pi(i) = \pi(i-1) + 1$。

反之，我们令 $j = \pi(i-1)$。

则我们要求出一个最大的 $k(0 \le k < j)$，满足 $\text{Preffix}(\text{Preffix}(S, i-1), k) = \text{Suffix}(\text{Preffix}(S, i-1), k)$，然后判断 S_{k+1} 是否与 S_i 相等，若相等则 $\pi(i) = k + 1$，否则令 $j = k$ 并**重复此过程**直至满足条件为止。

那么该如何寻找 k 呢?

由于 $k < \pi(i-1)$,所以可以由 π 的定义得:- $\text{Preffix}(\text{Preffix}(S, i-1), k) = \text{Preffix}(\text{Preffix}(S, \pi(i-1)), k)$;- $\text{Suffix}(\text{Preffix}(S, i-1), k) = \text{Suffix}(\text{Preffix}(S, \pi(i-1)), k)$。

又因为 $\text{Preffix}(\text{Preffix}(S,i-1),k)=\text{Suffix}(\text{Preffix}(S,i-1),k)$,所以 $\text{Preffix}(\text{Preffix}(S,\pi(i-1)),k)=\text{Suffix}(\text{Preffix}(S,\pi(i-1)),k)$。

有没有似曾相识的感觉？没错，k 其实就是 $\text{Preffix}(S,\pi(i-1))$ 的 border 长度，也就是说，k **等于** $\pi(\pi(i-1))$！

那么求 $\pi(i)$ 的时候，若不能成功匹配，只需要一直令 $k=\pi(k)$ 直到能够成功匹配即可。

这样子我们就可以写出如下的一份代码：

```
int n = strlen(s + 1);
for (int i = 2, j = 0; i <= n; i++) {
    while (j && s[i] != s[j + 1]) j = kmp[j];
    if (s[i] == s[j + 1]) j++;
    kmp[i] = j;
}
```

这份代码的时间复杂度是多少呢？我们可以发现，时间复杂度与 j 跳 kmp 数组的次数有关，j 最多增加 $|S|$ 次，因此 j 也最多减少 $|S|$ 次（即最多跳 kmp 数组 $|S|$ 次），所以时间复杂度是 $\Theta(|S|)$ 的。

计算每一个前缀的出现次数

回顾刚刚求 π 的过程，可以发现，每跳一次 kmp 数组，就代表一个前缀重复出现了一次，那我们统计对于每一个 j **最多**可以跳几次 kmp 数组即可。

直接在计算 π 的代码上面修改就行？不对，要求的是**最多**的次数，而计算 π 的过程中并没有把 kmp 数组跳到底。

因此我们考虑另一种方式：倒序枚举 $1\sim|S|$，然后将每一个位置 j 的出现次数累加到跳跃位置 $\text{kmp}(j)$ 上即可。

```
for (int i = 0; i <= n; i++) cnt[i] = 1;
for (int i = n; i; i--) cnt[kmp[i]] += cnt[i];
```

查找子串

用于判断一个串 S 是否在 T 中出现。

对 S 求出 π，然后按位匹配，失配就跳 kmp 数组即可。

```
int m = strlen(t);
for (int i = 1, j = 0; i <= m; i++) {
    while (j && t[i] != s[j + 1]) j = kmp[j];
    if (t[i] == s[j + 1]) j++;
    if (j == m) cout << i - m + 1 << endl, j = kmp[j];
}
```

子序列自动机

一种能够接受一个字符串 S 的所有子序列的自动机。

构建也很简单，记 $\text{nxt}(i,c)$ 为串 S 中第 i 位之后的第一个字符 c 的出现位置，那么我们从后往前遍历字符串，$\text{nxt}(i-1)$ 继承 $\text{nxt}(i)$ 的状态，补上 $\text{nxt}(i-1,S_i)=i$ 即可。

时空复杂度 $\Theta(|S||\Sigma|)$，其中 Σ 为字符集。

```
int len = strlen(s + 1);
for (int i = len; i; i--) memcpy(nxt[i - 1], nxt[i], sizeof(nxt[i])), nxt[i - 1][s[i]] = i;
```

数学

数论

裴蜀定理

小结论：若 $(a,b)=1$，则对于所有 $n\in[0,ab-a-b]$，$ax+by=n$ 和 $ax+by=ab-a-b-n$ 恰有一个有解。

线性求逆元

```
inv[1] = 1;
for (int i = 2; i <= n; i++) inv[i] = 1ll * (P - (P / i)) * inv[P % i] % P;
```

线性筛任意积性函数

假如一个积性函数 f 满足：对于任意质数 p 和正整数 k，可以在 $\Theta(1)$ 时间内计算 $f(p^k)$，那么可以在 $\Theta(n)$ 时间内筛出 $f(1), f(2), \ldots, f(n)$ 的值。

设合数 n 的质因子分解是 $\prod_{i=1}^{k} p_i^{\alpha_i}$，其中 $p_1 < p_2 < \cdots < p_k$ 为质数，我们在线性筛中记录 $g_n = p_1^{\alpha_1}$，假如 n 被 $x \cdot p$ 筛掉（p 是质数），那么 g_n 满足如下递推式：

$$g_n = \begin{cases} g_x \cdot p & x \bmod p = 0 \\ p & \text{otherwise} \end{cases}$$

假如 $n = g_n$，说明 n 就是某个质数的次幂，可以 $\Theta(1)$ 计算；否则，$f(n) = f(\frac{n}{g_n}) \cdot f(g_n)$。

杜教筛

求一个积性函数 $f(n)$ 的前缀和 $S(n)$。

构造一个函数 g 使得函数 $h = f * g$ 的前缀和能够快速计算，则有：

$$\begin{aligned} \sum_{i=1}^{n} h(i) &= \sum_{i=1}^{n} \sum_{d|i} g(d) f\left(\frac{i}{d}\right) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{\lfloor \frac{n}{i} \rfloor} g(i) f(j) \\ &= \sum_{i=1}^{n} g(i) S\left(\left\lfloor \frac{n}{i} \right\rfloor\right) \end{aligned}$$

因此有

$$g(1) S(n) = \sum_{i=1}^{n} h(i) - \sum_{i=2}^{n} g(i) S\left(\left\lfloor \frac{n}{i} \right\rfloor\right)$$

递归求解即可。为了使复杂度更加优秀，我们预处理前 $n^{\frac{2}{3}}$ 个 S 的取值以及加上记忆化，可以使时间复杂度达到 $\Theta(n^{\frac{2}{3}})$。

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

using namespace __gnu_pbds;

gp_hash_table<int, int> Smu;
gp_hash_table<int, ull> Sphi;

void Init() {
    mu[1] = phi[1] = 1;
    for (int i = 2; i <= N; i++) {
        if (isnp[i] == 0) pnum[++ptot] = i, mu[i] = -1, phi[i] = i - 1;
        for (int j = 1; j <= ptot && i * pnum[j] <= N; j++) {
            isnp[i * pnum[j]] = 1;
            if (i % pnum[j] == 0) { mu[i * pnum[j]] = 0, phi[i * pnum[j]] = phi[i] * pnum[j]; break; }
            else mu[i * pnum[j]] = -mu[i], phi[i * pnum[j]] = phi[i] * (pnum[j] - 1);
        }
        mu[i] += mu[i - 1], phi[i] += phi[i - 1];
    }
}

int muSum(uint n) {
    if (n <= N) return mu[n];
    if (Smu[n]) return Smu[n];
    int ret = 0;
```

```
    for (uint l = 2, r; l <= n; l = r + 1) r = n / (n / l), ret += (r - l + 1) * muSum(n / l);
    return Smu[n] = 1 - ret;
}
ull phiSum(uint n) {
    if (n <= N) return phi[n];
    if (Sphi[n]) return Sphi[n];
    ull ret = 0;
    for (uint l = 2, r; l <= n; l = r + 1) r = n / (n / l), ret += 1ull * (r - l + 1) * phiSum(n / l);
    return Sphi[n] = 1ull * n * (n + 1) / 2 - ret;
}
```

μ 函数

useful trick：当计算 $\gcd = 1$ 的答案时，我们可以先舍去 $\gcd = 1$ 的限制，然后我们枚举值域内的每一个 $\mu(i) \neq 0$ 的数 i，只计算集合中是 i 的倍数的数的贡献，然后乘上系数 $\mu(i)$ 后加和就可以得到 $\gcd = 1$ 的答案。证明显然。

BSGS

BSGS 用于在 $\Theta(\sqrt{p})$ 时间内求解 $a^x \equiv b \pmod p$。a, p 互质。

由质数的性质我们有 $x \in [0, p)$，因此假设 $x = k\lceil\sqrt{p}\rceil - m\ (k, m \in [0, \lceil\sqrt{p}\rceil])$，然后我们就有

$$a^{k\lceil\sqrt{p}\rceil} \equiv b \cdot a^m \pmod p$$

然后直接暴力枚举所有 m 把 $b \cdot a^m$ 扔进哈希表里，再暴力枚举所有 k 查询 $a^{k\lceil\sqrt{p}\rceil}$ 是否在表里即可。

进阶版本：求解 $x^a \equiv b \pmod p$。

求出 p 的原根 g，令 $x = g^c$，则有 $g^{ac} \equiv b \pmod p$，套用上述方法求出 t 使得 $ac \equiv t \pmod{\varphi(p)}$，使用 exgcd 即可求出所有的 c。

```
#define int long long

map<int, int> vis;

int BSGS(int a, int b, int P) {
    int t = sqrt(P), ap = 1; vis.clear();
    for (int i = 0; i < t; i++) vis[b * ap % P] = i, ap = ap * a % P;
    for (int i = 0, app = 1; i <= t; i++) {
        if (vis.count(app)) {
            printf("%lld\n", (i * t % P - vis[app] + P) % P);
            return 0;
        }
        app = app * ap % P;
    }
}

signed main() {
    fr(P), fr(a), fr(b);
    int t = sqrt(P), ap = 1;
    for (int i = 0; i < t; i++) vis[b * ap % P] = i, ap = ap * a % P;
    for (int i = 0, app = 1; i <= t; i++) {
        if (vis.count(app)) {
            printf("%lld\n", (i * t % P - vis[app] + P) % P);
            return 0;
        }
        app = app * ap % P;
    }
    puts("no solution");
    return 0;
}
```

exBSGS

处理 a, p 不互质的情况。

```
#define int long long

map<int, int> vis;
```

```cpp

int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }

void exgcd(int a, int b, int &x, int &y) {
    if (b == 0) return (void)(x = 1, y = 0);
    exgcd(b, a % b, y, x), y -= a / b * x;
}
int inv(int a, int P) {
    int x, y;
    exgcd(a, P, x, y);
    return (x + P) % P;
}

int qpow(int a, int b, int P) {
    int ret = 1;
    for (; b; b >>= 1, a = a * a % P) b & 1 ? ret = ret * a % P : 0;
    return ret;
}

int exBSGS(int a, int b, int P) {
    vis.clear(), b %= P;
    if (b == 1 || P == 1) return 0;
    int d = 1, g = gcd(a, P), k = 0;
    while (g != 1) {
        if (b % g) return -1;
        b /= g, P /= g, d = d * (a / g) % P, k++, g = gcd(a, P);
        if (d == b) return k;
    }
    int up = ceil(sqrt(P)), ap = 1;
    b = b * inv(d, P) % P;
    for (int i = 0; i < up; i++, ap = ap * a % P) vis[b * ap % P] = i;
    for (int i = 1, app = ap; i <= up; i++, app = app * ap % P)
        if (vis.find(app) != vis.end()) return i * up - vis[app] + k;
    return -1;
}

signed main() {
    int a = fr(), P = fr(), b = fr();
    while (a || P || b) {
        int ans = exBSGS(a, b, P);
        if (ans != -1) printf("%lld\n", ans);
        else puts("No Solution");
        fr(a, P, b);
    }
    return 0;
}
```

阶与原根

阶 $\delta_p(a)$ 一定是 $\varphi(p)$ 的因数，$\Theta(\sqrt{\varphi(p)})$ 枚举所有 $\varphi(p)$ 的因数 d 判断其是否满足 $a^d \equiv 1 \pmod p$ 即可，对所有满足条件的 d 取 min 即得到 $\delta_p(a)$。

原根：正整数 g 是正整数 n 的原根，当且仅当 $1 \le g \le n-1$，且 g 模 n 的阶为 $\varphi(n)$。只有 $2, 4, p^k, 2p^k$ 有原根，其中 p 为奇素数。

记最小原根为 g，则所有的原根都为 g^k，其中 $(k, \varphi(n)) = 1$，也就是说求出最小原根后我们可以在 $\Theta(\varphi(n))$ 时间内求出所有的原根。

现在的问题转化为求最小原根，有一个性质是最小原根一定不超过 $n^{0.25}$ 因此我们暴力枚举每个数 check 它的阶是否是 $\varphi(n)$ 即可。时间复杂度 $\Theta(n^{0.25}\sqrt{\varphi(n)})$。

但是可以更快，由于我们只需要知道是否存在小于 $\varphi(n)$ 的数 a 使得 $g^a \equiv 1 \pmod p$，也就是说我们其实只需要枚举 $\varphi(n)$ 的所有真因子。

那么我们将 $\varphi(n)$ 分解质因数后 check $\frac{\varphi(n)}{p_i}$（p_i 为 $\varphi(n)$ 的质因子）是否满足条件，若满足条件则阶一定不是 $\varphi(n)$。这样做时间复杂度优化为 $\Theta(n^{0.25}\log\varphi(n))$。

```cpp
#define int long long

const int N = 1e6;

int P, ptot, ftot, anstot, fact[N + 10], ans[N + 10], pnum[N + 10], phi[N + 10];
```

```cpp
bool isnp[N + 10], hasrt[N + 10];

void Init() {
    isnp[1] = hasrt[2] = hasrt[4] = 1;
    for (int i = 2; i <= N; i++) {
        if (isnp[i] == 0) pnum[++ptot] = i, phi[i] = i - 1;
        for (int j = 1; j <= ptot && i * pnum[j] <= N; j++) {
            isnp[i * pnum[j]] = 1;
            if (i % pnum[j] == 0) { phi[i * pnum[j]] = phi[i] * pnum[j]; break; }
            phi[i * pnum[j]] = phi[i] * (pnum[j] - 1);
        }
    }
    for (int i = 2; i <= ptot; i++) {
        for (int j = pnum[i]; j <= N; j *= pnum[i]) hasrt[j] = 1;
        for (int j = 2 * pnum[i]; j <= N; j *= pnum[i]) hasrt[j] = 1;
    }
}
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }
int qpow(int a, int b) {
    int ret = 1;
    for (; b; b >>= 1, a = a * a % P) b & 1 ? ret = ret * a % P : 0;
    return ret;
}
void factor(int x) {
    ftot = 0;
    for (int i = 2; i * i <= x; i++)
        if (x % i == 0) {
            fact[++ftot] = i;
            while (x % i == 0) x /= i;
        }
    if (x > 1) fact[++ftot] = x;
}
bool chk(int x) {
    if (qpow(x, phi[P]) != 1) return 0;
    for (int i = 1; i <= ftot; i++) if (qpow(x, phi[P] / fact[i]) == 1) return 0;
    return 1;
}
int getrt() {
    for (int i = 1; i < P; i++) if (chk(i)) return i;
    return 0;
}
void getallrt(int rt) {
    anstot = 0;
    for (int i = 1, tmp = rt; i <= phi[P]; tmp = tmp * rt % P, i++) if (gcd(i, phi[P]) == 1) ans[++anstot] = tmp;
}

signed main() {
    Init();
    for (int _ = fr(), d; _--;) {
        fr(P, d);
        if (hasrt[P]) {
            factor(phi[P]), getallrt(getrt()), sort(ans + 1, ans + 1 + anstot), printf("%lld\n", anstot);
            for (int i = 1; i <= anstot / d; i++) printf("%lld ", ans[i * d]);
            puts("");
        } else puts("0\n");
    }
    return 0;
}
```

exCRT

可以代替 CRT，而且个人感觉比 CRT 好记好写。

用于求解线性同余方程组（模数之间可互质可不互质）。考虑对于两个同余方程 $x \equiv a_1 \pmod{m_1}$ 和 $x \equiv a_2 \pmod{m_2}$，我们设 $x = m_1 p + a_1 = m_2 q + a_2$，则有 $m_1 p - m_2 q = a_2 - a_1$，如果 (m_1, m_2) 不是 $a_2 - a_1$ 的因数则无解。

反之我们使用 exgcd 求出一组解 (p, q) 后可以将两个同余方程合并为一个新的同余方程 $x \equiv A \pmod{M}$，其中 $A = m_1 p + a_1, M = \operatorname{lcm}(m_1, m_2)$。

useless trick：由于 CCF 开放了对 `__int128` 的限制，所以可以使用 `__int128` 来实现 $\Theta(1)$ 快速乘。不用再记那个 `long double`

实现的 $\Theta(1)$ 快速乘辣！

```cpp
#define int long long

int a1, m1;

int exgcd(int a, int b, int &x, int &y) {
    if (b == 0) return x = 1, y = 0, a;
    int ret = exgcd(b, a % b, y, x);
    return y -= (a / b) * x, ret;
}

void merge(int a2, int m2) {
    int x, y, a = m1, b = m2, c = a2 - a1, g = exgcd(a, b, x, y), cg = c / g, M = a / g * b;
    if (c % g) puts("-1"), exit(0);
    x = (__int128)x * cg % M, a1 = ((__int128)m1 * x + a1) % M, m1 = M;
}

signed main() {
    int n = fr(); fr(m1, a1);
    for (int i = 1, a2, m2; i < n; i++) fr(m2, a2), merge(a2, m2);
    printf("%lld\n", (a1 + m1) % m1);
    return 0;
}
```

狄利克雷前/后缀和

用于求快速求 $g(i)=\sum\limits_{d|i}f(d)$ 或 $g(i)=\sum\limits_{i|d}f(d)$。

一个小 trick，直接上代码，很好理解。

前缀和：

```cpp
for (int i = 1; i <= n; i++) g[i] = f[i];
for (int i = 1; i <= ptot && pnum[i] <= n; i++) for (int j = 1; j * pnum[i] <= n; j++) g[j * pnum[i]] += g[j];
```

后缀和：

```cpp
for (int i = 1; i <= n; i++) g[i] = f[i];
for (int i = 1; i <= ptot && pnum[i] <= n; i++) for (int j = n / pnum[i]; j; j--) g[j] += g[j * pnum[i]];
```

狄利克雷卷积 & 莫比乌斯反演

狄利克雷卷积：定义一种卷积运算 $h=f*g$：

$$h(n)=\sum_{d|n}f(d)g\left(\frac{n}{d}\right)$$

狄卷满足如下性质：1. 交换律：$f*g=g*f$；2. 结合律：$(f*g)*h=f*(g*h)$；3. 分配律：$f*(g+h)=f*g+f*h$；4. 单位元：$f*\varepsilon=f$；5. 若 f,g 为积性函数，则 $f*g$ 也为积性函数。

常见的狄卷：1. $\varepsilon=\mu*1$；2. $\sigma=\mathrm{id}*1$；3. $\varphi=\mathrm{id}*\mu$；4. $\tau=1*1$；5. $\sigma_k=\mathrm{id}_k*1$；6. $\mathrm{id}=\varphi*1$。

莫比乌斯反演：莫比乌斯反演有三种形式，你可都知道吗？

1. $$\varepsilon=\mu*1$$
2. $$f(n)=\sum_{d|n}g(d)\iff g(n)=\sum_{d|n}\mu(d)f\left(\frac{n}{d}\right)$$
3. $$f(n)=\sum_{n|d}g(d)\iff g(n)=\sum_{n|d}\mu\left(\frac{d}{n}\right)f(d)$$

后两个要求 f, g 为数论函数，但是大家好像都是用第一种。

二三两种形式的莫反其实也是能用的，有的时候可以设一个 g 然后化简 g 之后再用莫反把它回代到 f 里。

useless trick：能加速推式子，但效果很有限。

求

$$\sum_{i=1}^{n}\sum_{j=1}^{m} f(\gcd(i,j))$$

的时候，我们可以构造一个 g 使得 $f = g * 1$，例如 $f = \varepsilon$ 时取 $g = \mu$，$f = \mathrm{id}$ 时取 $g = \varphi$。则有：

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=1}^{m} f(\gcd(i,j)) &= \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{d\mid(i,j)} g(d)\\ &= \sum_{d=1}^{\min(n,m)} g(d)\sum_{i\mid n}\sum_{j\mid m} 1\\ &= \sum_{d=1}^{\min(n,m)} g(d)\left\lfloor\frac{n}{d}\right\rfloor\left\lfloor\frac{m}{d}\right\rfloor\end{aligned}$$

整除分块即可快速求值。

再记个结论：

$$\tau(ij) = \sum_{x\mid i}\sum_{y\mid j}[(i,j)=1]$$

莫比乌斯反演扩展：1. 对于数论函数 f, g 和完全积性函数 t 满足 $t(1) = 1$，有：

$$f(n) = \sum_{i=1}^{n} t(i) g\left(\left\lfloor\frac{n}{i}\right\rfloor\right) \Longleftrightarrow g(n) = \sum_{i=1}^{n}\mu(i)t(i) f\left(\left\lfloor\frac{n}{i}\right\rfloor\right)$$

不会证明。2. 定义一种新的卷积 $h = f \star g$：

$$h(n) = \prod_{d\mid n} f(d)^{g(\frac{n}{d})}$$

其满足 $(f\star g)\star h = f\star(g*h)$，因此有一种新的反演：

$$f = g\star 1 \Longleftrightarrow g = f\star\mu$$

例题：洛谷 P3704 [SDOI2017] 数字表格。

Miller Rabin

快速判断一个大整数是否是素数。

```cpp
#define int long long

const int tnum[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

int qmul(int a, int b, int p) {
    int w = (long double)a * b / p + 0.5, r = a * b - w * p;
    return r < 0 ? r + p : r;
}
int qpow(int a, int b, int P) {
    int ret = 1;
    for (; b; b >>= 1, a = qmul(a, a, P)) b & 1 ? ret = qmul(ret, a, P) : 0;
    return ret;
}

```

```cpp
bool MR(int x) {
    if (x == 1) return 0;
    for (int i = 0; i < 10; i++)
        if (x % tnum[i] == 0) return x == tnum[i];
    int r = x - 1, t = 0;
    while ((r & 1) == 0) r >>= 1, t++;
    for (int i = 0; i < 10; i++) {
        int a = qpow(tnum[i], r, x);
        for (int j = 0; j < t && a > 1; j++) {
            int b = qmul(a, a, x);
            if (b == 1 && a != x - 1) return 0;
            a = b;
        }
        if (a != 1) return 0;
    }
    return 1;
}

signed main() {
    for (int x; ~scanf("%lld", &x);) puts(MR(x) ? "Y" : "N");
    return 0;
}
```

多项式与生成函数

拉格朗日插值

我们要构造一个函数 $f(x)$ 过点 $P_i(x_i, y_i)$ $(\forall 1 \le i \le n)$, 那么我们考虑构造 n 个函数 $f_i(x)$ 过点 $P_i(x_i, y_i)$ 以及 $P_j(x_j, 0)$ $(\forall j \ne i)$, 则有 $f(x) = \sum_{i=1}^{n} f_i(x)$。

问题变为构造 $f_i(x)$，很容易想到一种构造 $f_i(x) = y_i \prod_{j \ne i} \frac{x - x_j}{x_i - x_j}$。

因此 $f(x) = \sum_{i=1}^{n} y_i \prod_{j \ne i} \frac{x - x_j}{x_i - x_j}$。

线性插值：对于给出的点值连续的情况，可以做到线性拉格朗日插值。

```cpp
for (int i = 1; i <= n; i++) fr(x[i]), fr(y[i]);
for (int i = 1, anss, ansm; i <= n; i++) {
    anss = y[i], ansm = 1;
    for (int j = 1; j <= n; j++)
        if (i != j) anss = anss * ((k - x[j] + P) % P) % P, ansm = ansm * ((x[i] - x[j] + P) % P) % P;
    ans = (ans + anss * qpow(ansm, P - 2) % P) % P;
}
```

线性:

```cpp
#define int long long

const int N = 3e3 + 10, P = 1234567891;

int ifac[N + 10], fac[N + 10], lf[N + 10], rf[N + 10];

int LagrangeIP(int *val, int k, int p) {
    int ret = 0; lf[0] = rf[k + 2] = 1;
    for (int i = 1; i <= k + 1; i++) lf[i] = lf[i - 1] * ((p - i + P) % P) % P;
    for (int i = k + 1; i; i--) rf[i] = rf[i + 1] * ((p - i + P) % P) % P;
    for (int i = 1, anss, ansm; i <= k + 1; i++) {
        anss = val[i] * lf[i - 1] % P * rf[i + 1] % P, ansm = ifac[i - 1] * ifac[k + 1 - i] % P;
        if ((k + 1 - i) & 1) ret = (ret - anss * ansm % P + P) % P;
        else ret = (ret + anss * ansm % P) % P;
    }
    return ret;
}

signed main() {
    fac[0] = fac[1] = ifac[0] = ifac[1] = 1;
```

```
    for (int i = 2; i <= N; i++) ifac[i] = (P - P / i) * ifac[P % i] % P, fac[i] = fac[i - 1] * i % P;
    for (int i = 2; i <= N; i++) ifac[i] = ifac[i - 1] * ifac[i] % P;
    return 0;
}
```

FWT

用于解决子集卷积 / 子集和 / 超集和。

或卷积：对两个数组分别做子集和 FWT，然后对应位相乘，然后再做子集和 IFWT 得到答案。

与卷积：对两个数组分别做超集和 FWT，然后对应位相乘，然后再做超集和 IFWT 得到答案。

异或卷积：难以描述。不会现推，只能硬记。

代码的循环部分尤其需要记忆：- 把一个长为 $2k$ 的段切开来变成两个长度为 k 的段，那么前半段和后半段有子集/超集关系。

例题：洛谷 P4717 【模板】快速莫比乌斯/沃尔什变换 (FMT/FWT)。

```
#define int long long

const int N = 1 << 17, P = 998244353;

int n, A[N + 10], B[N + 10], a[N + 10], b[N + 10], c[N + 10];

void OR(int *f, int v) {
    for (int o = 2, k = 1; o <= n; o <<= 1, k <<= 1)
        for (int i = 0; i < n; i += o)
            for (int j = 0; j < k; j++) f[i + j + k] = (f[i + j + k] + v * f[i + j] + P) % P;
}
void AND(int *f, int v) {
    for (int o = 2, k = 1; o <= n; o <<= 1, k <<= 1)
        for (int i = 0; i < n; i += o)
            for (int j = 0; j < k; j++) f[i + j] = (f[i + j] + v * f[i + j + k] + P) % P;
}
void XOR(int *f, int v) {
    for (int o = 2, k = 1; o <= n; o <<= 1, k <<= 1)
        for (int i = 0; i < n; i += o)
            for (int j = 0, x, y; j < k; j++) x = (f[i + j] + f[i + j + k]) % P, y = (f[i + j] - f[i + j + k] + P) %
 ↪  P, f[i + j] = v * x % P, f[i + j + k] = v * y % P;
}

signed main() {
    n = 1 << fr();
    for (int i = 0; i < n; i++) fr(A[i]);
    for (int i = 0; i < n; i++) fr(B[i]);
    for (int i = 0; i < n; i++) a[i] = A[i], b[i] = B[i];
    OR(a, 1), OR(b, 1);
    for (int i = 0; i < n; i++) c[i] = a[i] * b[i] % P;
    OR(c, -1);
    for (int i = 0; i < n; i++) printf("%lld%c", c[i], " \n"[i == n - 1]);
    for (int i = 0; i < n; i++) a[i] = A[i], b[i] = B[i];
    AND(a, 1), AND(b, 1);
    for (int i = 0; i < n; i++) c[i] = a[i] * b[i] % P;
    AND(c, -1);
    for (int i = 0; i < n; i++) printf("%lld%c", c[i], " \n"[i == n - 1]);
    for (int i = 0; i < n; i++) a[i] = A[i], b[i] = B[i];
    XOR(a, 1), XOR(b, 1);
    for (int i = 0; i < n; i++) c[i] = a[i] * b[i] % P;
    XOR(c, (P + 1) / 2);
    for (int i = 0; i < n; i++) printf("%lld%c", c[i], " \n"[i == n - 1]);
    return 0;
}
```

博弈论

SG 函数和 SG 定理

有向图游戏：在一个有向无环图中，只有一个起点，上面有一个棋子，两个玩家轮流沿着有向边推动棋子，不能走的玩家判负。

则对于一个状态 u 和其所有的后继状态 $v_1, v_2, \dots, v_k$，我们有 $\mathrm{SG}(u) = \mathrm{mex}\{\mathrm{SG}(v_1), \mathrm{SG}(v_2), \dots, \mathrm{SG}(v_k)\}$。

SG 定理：对于 n 个有向图游戏组成的组合游戏，设第 i 个有向图游戏的初始状态为 s_i，则当且仅当 $\bigoplus_{i=1}^{n} \mathrm{SG}(i) \neq 0$ 时先手必胜。

组合数学

Lucas 定理

用于计算组合数对小模数取模的结果。

$$\binom{n}{m} \equiv \binom{\lfloor \frac{n}{p} \rfloor}{\lfloor \frac{m}{p} \rfloor} \binom{n \bmod p}{m \bmod p} \pmod{p}$$

换句话说，假设 $(n)_{10} = (n_1 n_2 \dots n_k)_p$，$(m)_{10} = (m_1 m_2 \dots m_k)_p$，我们有：

$$\binom{n}{m} \equiv \prod_{i=1}^{k} \binom{n_i}{m_i} \pmod{p}$$

然后考虑在 mod2 意义下的 Lucas 定理，就可以得到：$\binom{i}{j} \equiv 1 \pmod{2}$ 当且仅当 $i \operatorname{and} j = j$，其中 and 为按位与运算。

```cpp
const int N = 2e5;

int n, m, P, fac[N + 10], ifac[N + 10];

void InitC() {
    fac[0] = fac[1] = ifac[0] = ifac[1] = 1;
    for (int i = 2; i <= N; i++) ifac[i] = (P - P / i) * ifac[P % i] % P;
    for (int i = 2; i <= N; i++) fac[i] = fac[i - 1] * i % P, ifac[i] = ifac[i] * ifac[i - 1] % P;
}

int C(int x, int y) {
    if (y > x || x < 0 || y < 0) return 0;
    return fac[x] * ifac[y] % P * ifac[x - y] % P;
}

int Lucas(int x, int y) {
    if (y == 0) return 1;
    return C(x % P, y % P) * Lucas(x / P, y / P) % P;
}

signed main() {
    for (int T = fr(); T--;) {
        fr(n), fr(m), fr(P), InitC();
        printf("%lld\n", Lucas(n + m, m));
    }
    return 0;
}
```

exLucas

用于处理模数 P 不为质数的情况。

```cpp
int a[30], p[30];

int qpow(int a, int b, int P) {
    int ret = 1;
    for (; b; b >>= 1, a = a * a % P) b & 1 ? ret = ret * a % P : 0;
    return ret;
}

void exgcd(int a, int b, int &x, int &y) {
    if (b == 0) return (void)(x = 1, y = 0);
    exgcd(b, a % b, y, x), y -= a / b * x;
}
int inv(int a, int P) {
    int x, y;
```

```
    exgcd(a, P, x, y);
    return (x + P) % P;
}

int CRT(int n) {
    int P = 1, ans = 0;
    for (int i = 1; i <= n; i++) P *= p[i];
    for (int i = 1; i <= n; i++) {
        int x = P / p[i];
        ans = (ans + a[i] * x % P * inv(x, p[i]) % P) % P;
    }
    return ans;
}

int solve(int n, int P, int Pk) {
    if (n == 0) return 1;
    int ret = 1;
    for (int i = 1; i <= Pk; i++)
        if (i % P) ret = ret * i % Pk;
    ret = qpow(ret, n / Pk, Pk);
    for (int i = n / Pk * Pk + 1; i <= n; i++)
        if (i % P) ret = i % Pk * ret % Pk;
    return ret * solve(n / P, P, Pk) % P;
}
int C(int n, int m, int P, int Pk) {
    int cnt = 0;
    for (int i = n; i; i /= P) cnt += i / P;
    for (int i = m; i; i /= P) cnt -= i / P;
    for (int i = n - m; i; i /= P) cnt -= i / P;
    return qpow(P, cnt, Pk) * solve(n, P, Pk) % Pk * inv(solve(m, P, Pk), Pk) % Pk * inv(solve(n - m, P, Pk), Pk) % Pk;
}
int exLucas(int n, int m, int P) {
    int tot = 0;
    for (int i = 2; i * i <= P; i++)
        if (P % i == 0) {
            p[++tot] = 1;
            while (P % i == 0) p[tot] *= i, P /= i;
            a[tot] = C(n, m, i, p[tot]);
        }
    if (P > 1) p[++tot] = P, a[tot] = C(n, m, P, P);
    return CRT(tot);
}

signed main() {
    int n = fr(), m = fr(), P = fr();
    printf("%lld\n", exLucas(n, m, P));
    return 0;
}
```

网格图路径计数 - 翻折法

useful trick。以 Catalan 数为例：

我们要计算 $(0,0)$ 到 (n,n) 的不穿过直线 $y=x$ 的非降路径数，先考虑容斥，总方案数为 $\binom{2n}{n}$，接下来只需要计算不合法的方案数即可。

考虑每一种不合法的方案数都一定碰到直线 $y=x+1$，那么我们将不合法的方案在第一个碰到直线 $y=x+1$ 位置关于直线翻折，就得到一条 $(0,0)$ 到 $(n-1,n+1)$ 的非降路径。可以证明这样翻折之后的路径与原来的路径一一对应，因此不合法的方案数即为 $(0,0)$ 到 $(n-1,n+1)$ 的非降路径数 $\binom{2n}{n+1}$。

所以 Catalan 数第 n 项为 $\binom{2n}{n}-\binom{2n}{n+1}$。

二项式反演

二项式反演有四种形式：1.

$$g(n)=\sum_{i=0}^{n}\binom{n}{i}f(i)\iff f(n)=\sum_{i=0}^{n}(-1)^{n-i}\binom{n}{i}g(i)$$

2.

$$g(n)=\sum_{i=0}^{n}(-1)^{i}\binom{n}{i}f(i)\iff f(n)=\sum_{i=0}^{n}(-1)^{i}\binom{n}{i}g(i)$$

3.

$$g(n)=\sum_{i=n}^{N}\binom{i}{n}f(i)\iff f(n)=\sum_{i=n}^{N}(-1)^{i-n}\binom{i}{n}g(i)$$

4.

$$g(n)=\sum_{i=n}^{N}(-1)^{i}\binom{i}{n}f(i)\iff f(n)=\sum_{i=n}^{N}(-1)^{i}\binom{i}{n}g(i)$$

套路的用法是做「钦定」和「恰好」的转换，当然也可以直接用于推式子。

以错位排列为例：1. 设 $f(i)$ 为长度为 i 的错排数，那么我们有：

$$n!=\sum_{i=0}^{n}\binom{n}{i}f(i)$$

使用二项式反演，得：

$$f(n)=\sum_{i=0}^{n}(-1)^{n-i}\binom{n}{i}i!$$

2. 设 $f(k)$ 为长度为 n 的排列中恰好有 k 个位置错位的方案数，$g(k)$ 为长度为 n 的排列中钦定有 k 个位置错位的方案数，则有：

$$g(k)=\sum_{i=k}^{n}\binom{i}{k}f(i)$$

反演得：

$$f(k)=\sum_{i=k}^{n}(-1)^{i-k}\binom{i}{k}g(i)$$

考虑 $g(k)$ 的组合意义，它等价于选出 k 个不动点然后剩下随便填的方案数，也即：

$$g(k)=\binom{n}{k}(n-k)!$$

代入到 $f(k)$ 中，得：

$$f(k)=\sum_{i=k}^{n}(-1)^{i-k}\binom{i}{k}\binom{n}{i}(n-i)!$$

线性代数

高斯消元

```cpp
for (int i = 1; i <= n; i++) {
    int r = i;
    for (int j = 1; j <= n; j++)
        if (cabs(a[j][i]) > cabs(a[r][i])) r = j;
    if (cabs(a[r][i]) < eps) puts("No Solution"), exit(0);
    if (r != i)
        for (int j = 1; j <= n + 1; j++) cswp(a[r][j], a[i][j]);
    double inv = a[i][i];
    for (int j = 1; j <= n; j++) {
        if (j == i) continue;
        double tmp = a[j][i] / inv;
        for (int k = i + 1; k <= n + 1; k++) a[j][k] -= a[i][k] * tmp;
    }
}
```

行列式

记一个矩阵 A 的行列式：

$$\det(A)=\sum_{\sigma\in S_n}\operatorname{sgn}(\sigma)\prod_{i=1}^{n}a_{i,\sigma(i)}$$

其中 S_n 是长度为 n 的排列集合，当 σ 的逆序对数为奇数时 $\operatorname{sgn}(\sigma)$ 为 -1，否则为 1。

行列式有如下性质：- 矩阵转置，行列式不变；- 矩阵行（列）交换，行列式取反；- 矩阵行（列）相加或相减，行列式不变；- 矩阵行（列）所有元素同时乘以数 k，行列式等比例变大。

也就是说我们可以用类似高斯消元的方式消出对角矩阵后，再计算对角矩阵的行列式即可。时间复杂度 $\Theta(n^3)$。

```
int det(int A[N + 5][N + 5]) {
    memcpy(mat, A, sizeof(mat));
    int ret = 1;
    for (int i = 1; i <= n; i++)
        for (int j = i + 1; j <= n; j++) {
            while (mat[i][i]) {
                int t = mat[j][i] / mat[i][i];
                for (int k = i; k <= n; k++) mat[j][k] = (mat[j][k] - mat[i][k] * t % P + P) % P;
                swap(mat[i], mat[j]), ret = -ret;
            }
            swap(mat[i], mat[j]), ret = -ret;
        }
    for (int i = 1; i <= n; i++) ret = ret * mat[i][i] % P;
    return (ret + P) % P;
}
```

Matrix Tree 定理

用于计算一个图的所有生成树边权之积的和，即：

$$\sum_{T\subset G}\prod_{(u,v,w)\in T}w$$

对于无向图：定义邻接矩阵 A 和度数矩阵 D，满足：- $A_{i,j}$ 为 (i,j) 这条边的边权；- $D_{i,j}=0\ (i\neq j)$；- $D_{i,i}=$ 所有与 i 相连的边的边权和。

定义矩阵 $K=D-A$，将矩阵 K 去掉第 k 行第 k 列（k 为 $[1,n]$ 内任意整数）后得到矩阵 K'，$\det(K')$ 即为这个无向图的所有生成树边权之积的和。

对于有向图：定义邻接矩阵 A 和入/出度矩阵 $D^{\text{in/out}}$，满足：- $A_{i,j}$ 为 (i,j) 这条边的边权；- $D^{\text{in/out}}_{i,j}=0\ (i\neq j)$；- $D^{\text{in}}_{i,i}=$ 所有连向 i 的边的边权和；- $D^{\text{out}}_{i,i}=$ 所有从 i 连出的边的边权和。

定义内向树：所有的边方向指向根；外向树：所有的边方向指向叶子。令 $K^{\text{in/out}}=D^{\text{in/out}}-A$，$K^{\text{in/out}}_i$ 为 $K^{\text{in/out}}$ 去掉第 i 行第 i 列的矩阵。

则该有向图的所有以 i 为根的内向生成树边权之积的和为 $\det(K^{\text{out}}_i)$；该有向图的所有以 i 为根的外向生成树边权之积的和为 $\det(K^{\text{in}}_i)$。

```
#define int long long

const int N = 3e2, P = 1e9 + 7;

int n, m, type, a[N + 5][N + 5], mat[N + 5][N + 5];

int MatrixTree(int A[N + 5][N + 5], int rt) {
    memcpy(mat, A, sizeof(mat));
    int ret = 1;
    for (int i = 1; i <= n; i++) if (i != rt)
        for (int j = i + 1; j <= n; j++) if (j != rt) {
            while (mat[i][i]) {
                int t = mat[j][i] / mat[i][i];
                for (int k = i; k <= n; k++) if (k != rt) mat[j][k] = (mat[j][k] - mat[i][k] * t % P) % P;
                swap(mat[i], mat[j]), ret = -ret;
            }
            swap(mat[i], mat[j]), ret = -ret;
        }
    for (int i = 1; i <= n; i++) if (i != rt) ret = ret * mat[i][i] % P;
    return (ret + P) % P;
```

```
}

signed main() {
    fr(n, m, type);
    for (int i = 1, u, v, w; i <= m; i++) {
        fr(u, v, w);
        if (type) a[u][v] = (a[u][v] - w) % P, a[v][v] = (a[v][v] + w) % P;
        else a[u][u] = (a[u][u] + w) % P, a[v][v] = (a[v][v] + w) % P, a[u][v] = (a[u][v] - w) % P, a[v][u] =
 ↪  (a[v][u] - w) % P;
    }
    return printf("%lld\n", MatrixTree(a, 1)), 0;
}
```

useful trick：求解

$$\sum_{T\subset G}\sum_{(u,v,w)\in T} w$$

可以把边权构造为一个多项式 $wx+1$，然后求边权之积的和之后，取结果的一次项系数即可。

线性基

很厉害的东西！

定义：不会线代的严谨定义，说个人话版本的：「线性基内的子集异或和」集合与「原集合里所有子集的异或和」集合相同。

完全不会线代所以我不理解构造，后面有空一定补一下线代原理，现在来不及了只能硬记了。放个 insert：

```
bool insert(int x) {
    for (int i = L; ~i; i--)
        if ((x >> i) & 1) {
            if (b[i]) x ^= b[i];
            else {
                for (int j = 0; j < i; j++) if ((x >> j) & 1) x ^= b[j];
                for (int j = i + 1; j <= L; j++) if ((b[j] >> i) & 1) b[j] ^= x;
                return b[i] = x, bsz++, 1;
            }
        }
    if (x == 0) zero = 1;
    return 0;
}
```

- 最大异或和：线性基里全部异或起来；
- 最小异或和：如果可以异或出 0 则答案为 0，反之为线性基里的最小数；
- k-th 异或和：把线性基里面的 0 删掉，然后如果 k 的二进制下第 i 位是 1 就异或上线性基里的第 i 个。同样记得特判 0。
- 线性基能异或出来的数数量：$2^{\text{线性基大小}}-1+[0\text{ 可以被异或出来}]$。

矩阵求逆

给原矩阵后面加一个单位矩阵，然后对它高斯消元得到的后半部分矩阵即为逆矩阵。

康托展开

```
#define int long long

const int N = 1e6, P = 998244353;

int n, ans, a[N + 10], fac[N + 10];

namespace BIT {
    int cval[N + 10];
    int lowbit(int i) { return i & -i; }
    void update(int pos) {
        for (int i = pos; i <= N; i += lowbit(i)) cval[i]++;
    }
    int qsum(int pos) {
        int ret = 0;
        for (int i = pos; i; i -= lowbit(i)) ret += cval[i];
        return ret;
```

```
    }
} // namespace BIT
using namespace BIT;

signed main() {
    fr(n), fac[0] = 1;
    for (int i = 1; i <= n; i++) fac[i] = fac[i - 1] * i % P, fr(a[i]);
    for (int i = 1; i <= n; i++) {
        int tmp = a[i] - qsum(a[i] - 1) - 1;
        ans = (ans + tmp * fac[n - i] % P) % P, update(a[i]);
    }
    printf("%lld\n", ans + 1);
    return 0;
}
```

动态规划

决策单调性优化 DP & 四边形不等式优化 DP & 分治优化 DP

区间包含单调性：对于所有 $l_1 \le l_2 \le r_2 \le r_1$，有 $f(l_1, r_1) \ge f(l_2, r_2)$ 则称 f 有区间包含单调性。

四边形不等式：对于所有 $l_1 \le l_2 \le r_2 \le r_1$，有 $f(l_1, r_2) + f(l_2, r_1) \le f(l_1, r_1) + f(l_2, r_2)$ 则称 f 满足四边形不等式。

区间 DP 上的四边形不等式优化：

$$f_{l,r} = \min_{k=l}^{r-1}\{f_{l,k} + f_{k+1,r}\} + w(l, r) \qquad (1 \le l < r \le n)$$

若 w 满足区间包含单调性和四边形不等式，则 f 满足决策单调性，可使用四边形不等式优化加速 DP。

怎么加速？记录 $f_{i,j}$ 的最优决策点 $s_{i,j}$。对于 $f_{l,r}$ 有它的最优决策点在 $s_{l,r-1}$ 和 $s_{l+1,r}$ 之间，在这个范围内枚举决策点。时间复杂度 $\Theta(n^2)$。

例题：石子合并。

基于分治的决策单调性优化：

$$f_{i,j} = \min_{k \le j}\{f_{i-1,k}\} + w(k, j) \qquad (1 \le i \le n, 1 \le j \le m)$$

暴力做时间复杂度是 $\Theta(nm^2)$ 的，但若 f 满足决策单调性则可以使用分治优化。

假设当前我们要计算 $f_{i,[l,r]}$，而它们的决策点在 $[L, R]$ 内，那么我们计算 f_{mid} 的决策点 MID，就有 $f_{[l,\text{mid})}$ 的决策点在 $[L, \text{MID}]$ 内；$f_{(\text{mid},r]}$ 的决策点在 $[\text{MID}, R]$ 内，不断分治下去即可，时间复杂度 $\Theta(nm \log m)$。

useful trick：在基于分治的决策单调性优化中，如果计算 w 函数时我们使用了类似于莫队的方法（即暴力移动指针计算答案），那么复杂度是均摊 $\Theta(1)$ 的。不会证明。

例题：CF868F Yet Another Minimization Problem，同时还包含了上述的 trick。

```
#define int long long

const int N = 1e5;

int n, k, ans, L = 1, R, a[N + 10], cnt[N + 10], f[2][N + 10];

void add(int x) { ans += cnt[x], cnt[x]++; }
void del(int x) { cnt[x]--, ans -= cnt[x]; }
int val(int l, int r) {
    while (L > l) add(a[--L]);
    while (L < l) del(a[L++]);
    while (R > r) del(a[R--]);
    while (R < r) add(a[++R]);
    return ans;
}

void solve(int l, int r, int ql, int qr, int tot) {
    if (l > r) return;
    int mid = (l + r) >> 1, qmid = ql;
```

```
    for (int i = ql; i <= min(qr, mid); i++) {
        int tmp = f[(tot & 1) ^ 1][i - 1] + val(i, mid);
        if (tmp < f[tot & 1][mid]) f[tot & 1][mid] = tmp, qmid = i;
    }
    solve(l, mid - 1, ql, qmid, tot), solve(mid + 1, r, qmid, qr, tot);
}

signed main() {
    fr(n, k);
    for (int i = 1; i <= n; i++) fr(a[i]);
    memset(f, 0x3f, sizeof(f)), f[0][0] = 0;
    for (int i = 1; i <= k; i++) solve(1, n, 1, n, i & 1);
    printf("%lld\n", f[k & 1][n]);
    return 0;
}
```

数位 DP

其实感觉很好理解阿，不知道为什么老是忘记。

用于求一个前缀范围内满足某些条件的数的权值和。一般多见的问题是区间形式，两个前缀和相减即可。

怎么求？我们使用 dfs 进行转移，dfs 状态中记录是否含有前导零 ld，这一位是否被限制 lm，当前搜到的位数 pos，此外还可能需要另外记录一些 dp 过程需要的量，视情况而定，然后枚举这一位所有可能放的数字，dfs 下去即可，记得记忆化。

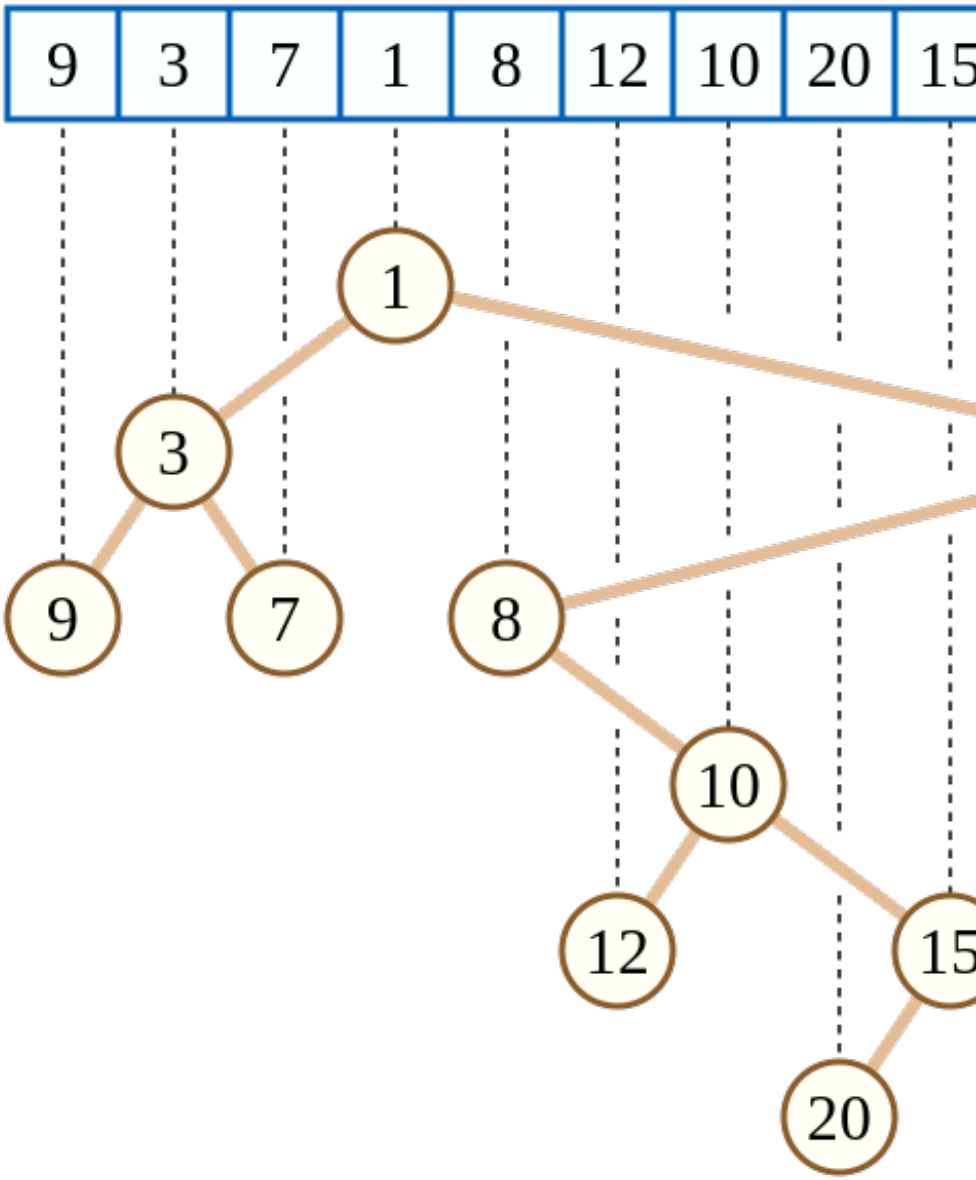
另外需要注意的是当且仅当不存在前导零且这一位也未被限制的时候（即 ld = lm = 0）才能记忆化。

例题：洛谷 P2657 [SCOI2009] windy 数。

```
#define int long long

int len, a[12], f[12][12];

int dfs(int pos, int pre, bool ld, bool lm) {
    if (pos > len) return 1;
    if (ld == 0 && lm == 0 && f[pos][pre] != -1) return f[pos][pre];
    int up = 9, ret = 0;
    if (lm) up = a[pos];
    for (int i = 0; i <= up; i++) {
        if (ld) {
            if (i == 0) ret += dfs(pos + 1, 0, 1, lm && (i == a[pos]));
            else ret += dfs(pos + 1, i, 0, lm && (i == a[pos]));
        } else if (abs(i - pre) >= 2) ret += dfs(pos + 1, i, 0, lm && (i == a[pos]));
    }
    if (ld == 0 && lm == 0) return f[pos][pre] = ret;
    return ret;
}

int solve(int r) {
    len = 0;
    while (r) a[++len] = r % 10, r /= 10;
    reverse(a + 1, a + 1 + len), memset(f, -1, sizeof(f));
    return dfs(1, 0, 1, 1);
}

signed main() {
    int l = fr(), r = fr();
    printf("%lld\n", solve(r) - solve(l - 1));
    return 0;
}
```

数据结构

笛卡尔树



一个满足「中序遍历序列是原序列」的堆,放个图:

怎么建笛卡尔树? 单调栈即可，如果实在不会也可以写多个 log 的 ST 表。怎么用? 由于树上一个节点表示了一个区间的最大/小值，具有良好的性质，比如可以分治做一些东西。

例题：洛谷 P7244 章节划分。

```
const int N = 392699;

int n, k, top, val, mx, rt, a[N + 10], stk[N + 10], ch[N + 10][2];

int solve(int u, int l = 1, int r = n) {
    if (u == 0) return 0;
    if (a[u] % val == 0) return solve(ch[u][0], l, u - 1) + solve(ch[u][1], u + 1, r) + 1;
    int ret = 0;
    if (l != 1) ret = max(ret, solve(ch[u][1], u + 1, r));
    if (r != n) ret = max(ret, solve(ch[u][0], l, u - 1));
    return ret;
```

```
}

signed main() {
    fr(n, k);
    for (int i = 1; i <= n; i++) fr(a[i]), mx = max(mx, a[i]);
    for (int i = 1; i <= n; i++) {
        while (top && a[stk[top]] <= a[i]) ch[i][0] = stk[top--];
        if (top) ch[stk[top]][1] = i;
        stk[++top] = i;
    }
    rt = stk[1];
    for (int i = mx; i; i--) {
        if (mx % i) continue;
        val = i;
        if (solve(rt) >= k) return printf("%d\n", i), 0;
    }
    return 0;
}
```

线段树

李超线段树

考场上手画一下图然后现推「优势线段」的维护就行。

感觉这东西优化 DP 比斜率优化好写多了，就是有的时候比斜率优化多个 log。

```
const int N = 39989, P1 = 39989, P2 = 1e9, INF = 0x7f7f7f7f;

int n, lstans, ltot;

struct Line {
    double k, b; int id;
    Line(double k = 0, double b = -INF, int id = 0) : k(k), b(b), id(id) {}
    double operator [] (const int &x) { return k * x + b; }
};

struct Node {
    Line cur; Node *ch[2];
} mem[(N << 1) + 10], *atot = mem, *rt;

Line max(int x, Line a, Line b) {
    if (a[x] > b[x]) return a;
    if (a[x] < b[x]) return b;
    return a.id < b.id ? a : b;
}

void modify(Node *&u, Line val, int ql, int qr, int l = 1, int r = 39989) {
    if (ql > r || qr < l) return ;
    if (u == 0) u = ++atot;
    int mid = (l + r) >> 1;
    if (ql <= l && qr >= r) {
        if (val[mid] > u->cur[mid]) swap(val, u->cur);
        if (u->cur[l] < val[l]) return modify(u->ch[0], val, ql, qr, l, mid);
        else if (u->cur[r] < val[r]) return modify(u->ch[1], val, ql, qr, mid + 1, r);
        return ;
    }
    modify(u->ch[0], val, ql, qr, l, mid), modify(u->ch[1], val, ql, qr, mid + 1, r);
}
Line qmax(Node *u, int pos, int l = 1, int r = 39989) {
    if (u == 0) return 0;
    Line ret = u->cur; int mid = (l + r) >> 1;
    return max(pos, ret, (pos <= mid) ? qmax(u->ch[0], pos, l, mid) : qmax(u->ch[1], pos, mid + 1, r));
}

void p1(int &x) { x = (x + lstans - 1) % P1 + 1; }
void p2(int &x) { x = (x + lstans - 1) % P2 + 1; }

signed main() {
    fr(n);
    for (int i = 1, opt, k, x0, y0, x1, y1, l, r; i <= n; i++) {
```

```
        fr(opt);
        if (opt == 0) fr(k), p1(k), printf("%d\n", lstans = qmax(rt, k).id);
        else {
            fr(x0), fr(y0), fr(x1), fr(y1), p1(x0), p2(y0), p1(x1), p2(y1), ltot++;
            if (x0 > x1) l = x1, r = x0;
            else l = x0, r = x1;
            if (x0 == x1) modify(rt, Line(0, max(y0, y1), ltot), l, r);
            else {
                double lk = 1.0 * (y1 - y0) / (x1 - x0);
                modify(rt, Line(lk, y0 - x0 * lk, ltot), l, r);
            }
        }
    }
    return 0;
}
```

线段树合并

一般用于快速合并两个数组的信息，多用于树上 DP 时加速转移。可以证明时间复杂度是 $\Theta(n \log n)$ 的。

```
Node *merge(Node *x, Node *y) {
    if (x == null || y == null) return (x == null) ? y : x;
    x->pushdown(), y->pushdown();
    ... // 将 y 的信息合并到 x 上
    x->ch[0] = merge(x->ch[0], y->ch[0]), x->ch[1] = merge(x->ch[1], y->ch[1]), x->pushup();
    return x;
}
```

兔队线段树

useful trick。使用线段树维护前缀最大值的一个 trick。

浅举个例子：查询区间本质不同前缀最大值数量。我们对线段树上的一个节点 $[l, r]$ 维护两个信息：1. 区间最大值 max。2. 仅考虑区间 $[l, r]$ 时，该区间的答案 ans。

然后我们考虑 pushup，会发现 ans 好像不太好合并啊，我们可以直接继承左儿子的答案，但是要计算右子树的答案我们还得考虑左子树的最大值 ☒

于是我们定义一个函数 $\text{calc}(i, pre)$ 表示节点 i 考虑了前面含有最大值 pre 的答案（抄的粉兔博客代码）：

def: $\text{calc}(i, pre)$
  if (i is a leaf node)
    return $[\text{max}[i] > pre]$
  else
    if $(\text{max}[\text{leftchild}[i]] > pre)$
      return $\text{calc}(\text{leftchild}[i], pre) + (\text{ans}[i] - \text{ans}[\text{leftchild}[i]])$
    else
      return $0 + \text{calc}(\text{rightchild}[i], pre)$
    endif.
  endif.
enddef.

绿色为叶子节点的贡献，蓝色为左子树贡献，红色为右子树贡献。

第一块红色那里不太好理解，但是聪明的大家一定能懂！就不解释了！

然后我们发现第一块红色那边要做减法，也就是说我们维护的信息要满足可减性？那好像有点拉啊，我们考虑修改 ans 的定义：- 仅考虑区间 $[l, r]$ 时，右儿子区间的答案 ans。- 叶子节点的 ans 无意义。

这下第一块红色那里就可以改成 $\text{ans}[i]$ 了！不需要可减性了，那么只剩下新定义 ans 的 pushup 问题，我们显然有 $\text{cnt}[i] = \text{calc}(\text{rightchild}[i], \text{max}[\text{leftchild}[i]])$。

问题圆满解决！时间复杂度 $\Theta(n \log^2 n)$。

例题：洛谷 P4198 楼房重建。就是上面说的这个题辣！

```cpp
#define int long long
#define now seg[cur]
#define ls cur << 1
#define rs cur << 1 | 1
#define lch seg[ls]
#define rch seg[rs]

const int N = 1e5;

int n, m, x, y;

struct Node {
    int l, r, cnt;
    double qwq;
    Node(int l = 0, int r = 0) : l(l), r(r) { cnt = 0, qwq = 0; }
} seg[(N << 2) + 10];

void build(int cur, int l, int r) {
    now = Node(l, r);
    if (l == r) return;
    int mid = (l + r) >> 1;
    build(ls, l, mid), build(rs, mid + 1, r);
}

int count(int cur, double &x) {
    if (now.l >= now.r) return now.qwq > x;
    if (lch.qwq <= x) return count(rs, x);
    return now.cnt - lch.cnt + count(ls, x);
}

void pushup(int cur) { now.qwq = cmax(lch.qwq, rch.qwq), now.cnt = lch.cnt + count(rs, lch.qwq); }

void modify(int cur, int &q, double &k) {
    if (now.l > q || now.r < q) return;
    if (now.l >= now.r) return (void)(now.qwq = k, now.cnt = 1);
    modify(ls, q, k), modify(rs, q, k), pushup(cur);
}

signed main() {
    cin >> n >> m, build(1, 1, n);
    while (m--) {
        cin >> x >> y;
        double tmp = (double)y / x;
        modify(1, x, tmp), cout << seg[1].cnt << endl;
    }
    return 0;
}
```

线段树分治

解决一类问题模型：- 有一些仅对一段时间有贡献的操作；- 计算某个点的贡献。

将时间看作一个轴，对时间轴建线段树。将贡献操作扔到线段树上，然后遍历整棵线段树，遇到带贡献的节点就加入贡献，遇到叶子节点计算贡献，退出某个节点时将贡献恢复。

时间复杂度 $\Theta(m \cdot a \log n)$，其中计算贡献和加入贡献的时间复杂度为 $\Theta(a)$。

例题：YZOJ P5394 连连通通（校内题）。

```cpp
const int N = 2e5, M = N << 2;

int n, m, k, top, fa[N + 10], sz[N + 10];

struct Edge {
    int x, y;
} e[N + 10];

struct Node {
    int x, y, val;
} stk[M + 10];

```

```cpp
vector<int> qwq[M + 10];

int find(int x) {
    while (x != fa[x]) x = fa[x];
    return fa[x];
}

void merge(int x, int y) {
    x = find(x), y = find(y);
    if (sz[x] > sz[y]) swap(x, y);
    stk[++top] = {x, y, sz[x]}, fa[x] = y, sz[y] += sz[x];
}

void update(int u, int ql, int qr, int val, int l = 1, int r = k) {
    if (l > qr || r < ql) return;
    if (ql <= l && r <= qr) return (void)(qwq[u].push_back(val));
    int mid = (l + r) >> 1;
    update(u << 1, ql, qr, val, l, mid), update(u << 1 | 1, ql, qr, val, mid + 1, r);
}

void sgtdiv(int u, int l, int r) {
    bool ans = 1;
    int lsttop = top;
    for (int i = 0; i < qwq[u].size(); i++) {
        int x = find(e[qwq[u][i]].x), y = find(e[qwq[u][i]].y);
        if (x == y) {
            for (int k = l; k <= r; k++) puts("No");
            ans = 0;
            break;
        }
        merge(e[qwq[u][i]].x, e[qwq[u][i]].y + n), merge(e[qwq[u][i]].y, e[qwq[u][i]].x + n);
    }
    if (ans) {
        if (l == r)
            puts("Yes");
        else {
            int mid = (l + r) >> 1;
            sgtdiv(u << 1, l, mid), sgtdiv(u << 1 | 1, mid + 1, r);
        }
    }
    while (top > lsttop) sz[fa[stk[top].x]] -= stk[top].val, fa[stk[top].x] = stk[top].x, top--;
    return;
}

int main() {
    fr(n), fr(m), fr(k);
    for (int i = 1, l, r; i <= m; i++) fr(e[i].x), fr(e[i].y), fr(l), fr(r), l++, update(1, l, r, i);
    for (int i = 1; i <= n * 2; i++) fa[i] = i, sz[i] = 1;
    return sgtdiv(1, 1, k), 0;
}
```

可持久化线段树

空间是 $\Theta(n \log m)$ 的。

```cpp
const int N = 2e5;

int n, m, a[N + 10], arr[N + 10];

struct Node {
    int sum;
    Node *ch[2];
    void pushup() { sum = ch[0]->sum + ch[1]->sum; }
} mem[N * 20 + 10], *atot = mem, *rt[N + 10];

void build(Node *&u, int l = 1, int r = arr[0]) {
    u = ++atot;
    if (l == r) return;
    int mid = (l + r) >> 1;
    build(u->ch[0], l, mid), build(u->ch[1], mid + 1, r);
}
```

```
void update(Node *&u, Node *v, int pos, int val, int l = 1, int r = arr[0]) {
    u = ++atot, u->ch[0] = v->ch[0], u->ch[1] = v->ch[1], u->sum = v->sum;
    if (l == r) return u->sum += val, void();
    int mid = (l + r) >> 1;
    (pos <= mid) ? update(u->ch[0], v->ch[0], pos, val, l, mid) : update(u->ch[1], v->ch[1], pos, val, mid + 1, r);
    u->pushup();
}
int qkth(Node *ql, Node *qr, int k, int l = 1, int r = arr[0]) {
    if (l == r) return arr[l];
    int mid = (l + r) >> 1, sz = qr->ch[0]->sum - ql->ch[0]->sum;
    return (k <= sz) ? qkth(ql->ch[0], qr->ch[0], k, l, mid) : qkth(ql->ch[1], qr->ch[1], k - sz, mid + 1, r);
}

signed main() {
    fr(n, m);
    for (int i = 1; i <= n; i++) fr(a[i]), arr[i] = a[i];
    sort(arr + 1, arr + 1 + n), arr[0] = unique(arr + 1, arr + 1 + n) - arr - 1, build(rt[0]);
    for (int i = 1; i <= n; i++) update(rt[i], rt[i - 1], lower_bound(arr + 1, arr + 1 + arr[0], a[i]) - arr, 1);
    for (int i = 1, l, r, k; i <= m; i++) fr(l, r, k), printf("%d\n", qkth(rt[l - 1], rt[r], k));
    return 0;
}
```

莫队

注意移动指针的时候应当先扩大区间，再缩小区间。很重要!!! 不这么写的莫队是假的!!!

```
const int N = 5e5, S = 710;

int n, m, blo, Max, a[N + 10], ans[N + 10], sum[S + 10], cnt[N + 10], bid[N + 10];

struct Query {
    int l, r, a, b, id;
    bool operator<(const Query &rhs) const {
        if (bid[l] == bid[rhs.l]) return r < rhs.r;
        return bid[l] < bid[rhs.l];
    }
} Q[N + 10];

void add(int x) {
    cnt[x]++;
    if (cnt[x] == 1) sum[bid[x]]++;
}
void del(int x) {
    cnt[x]--;
    if (cnt[x] == 0) sum[bid[x]]--;
}

int query(int l, int r) {
    int ret = 0;
    if (bid[l] == bid[r]) {
        for (int i = l; i <= r; i++) ret += (cnt[i] ? 1 : 0);
        return ret;
    }
    for (int i = bid[l] + 1; i < bid[r]; i++) ret += sum[i];
    for (int i = l; bid[i] != bid[l] + 1; i++) ret += (cnt[i] ? 1 : 0);
    for (int i = r; bid[i] != bid[r] - 1; i--) ret += (cnt[i] ? 1 : 0);
    return ret;
}

int main() {
    fr(n), fr(m), blo = sqrt(n);
    for (int i = 1; i <= n; i++) bid[i] = (i - 1) / blo + 1, fr(a[i]), Max = max(Max, a[i]);
    for (int i = 1; i <= m; i++) fr(Q[i].l), fr(Q[i].r), fr(Q[i].a), fr(Q[i].b), Q[i].id = i;
    sort(Q + 1, Q + m + 1);
    for (int i = 1, l = 1, r = 0; i <= m; i++) {
        while (Q[i].l < l) add(a[--l]);
        while (Q[i].r > r) add(a[++r]);
        while (Q[i].l > l) del(a[l++]);
        while (Q[i].r < r) del(a[r--]);
        ans[Q[i].id] = query(Q[i].a, Q[i].b);
    }
```

```
    for (int i = 1; i <= m; i++) printf("%d\n", ans[i]);
    return 0;
}
```

带修莫队

给普通莫队加一维时间，这个不是难点。难点是高维莫队的复杂度分析：

设块长为 B，每一维值域大小为 n，询问数为 m，则 k 维莫队时间复杂度为 $\Theta\left(\frac{n^k}{B^{k-1}}+mB\right)$，由基本不等式得当 $B=\frac{n}{\sqrt[k]{m}}$ 时时间复杂度取最小值 $\Theta\left(n\sqrt[k]{m^{k-1}}\right)$。

```
const int N = 133333, M = 1e6;

char opt[10];
int n, q, bsz, Qtot, Ctot, col[N + 10], bid[N + 10], cnt[M + 10], ans[N + 10];

struct Query {
    int l, r, id, t;
    bool operator<(const Query &rhs) const {
        if (bid[l] == bid[rhs.l]) {
            if (bid[r] == bid[rhs.r]) return t < rhs.t;
            return r < rhs.r;
        }
        return l < rhs.l;
    }
} Q[N + 10];

struct Modify {
    int pos, val;
} C[N + 10];

void add(int x) { ans[0] += (++cnt[x] == 1); }
void del(int x) { ans[0] -= (--cnt[x] == 0); }
void modify(int p, int l, int r) {
    if (C[p].pos >= l && C[p].pos <= r) ans[0] -= (--cnt[col[C[p].pos]] == 0), ans[0] += (++cnt[C[p].val] == 1);
    swap(C[p].val, col[C[p].pos]);
}

signed main() {
    fr(n, q), bsz = pow(n, (double)2 / (double)3);
    for (int i = 1; i <= n; i++) fr(col[i]), bid[i] = (i - 1) / bsz + 1;
    for (int _ = 1, l, r; _ <= q; _++) {
        scanf("%s", opt + 1), fr(l, r);
        if (opt[1] == 'Q') Qtot++, Q[Qtot] = {l, r, Qtot, Ctot};
        else C[++Ctot] = {l, r};
    }
    sort(Q + 1, Q + Qtot + 1);
    for (int i = 1, l = 1, r = 0, t = 0; i <= Qtot; i++) {
        while (Q[i].l < l) add(col[--l]);
        while (Q[i].r > r) add(col[++r]);
        while (Q[i].l > l) del(col[l++]);
        while (Q[i].r < r) del(col[r--]);
        while (t < Q[i].t) modify(++t, Q[i].l, Q[i].r);
        while (t > Q[i].t) modify(t--, Q[i].l, Q[i].r);
        ans[Q[i].id] = ans[0];
    }
    for (int i = 1; i <= Qtot; i++) printf("%d\n", ans[i]);
    return 0;
}
```

括号序树上莫队

考场上把括号序画出来现推一下就会了吧：- (u,v) 存在祖先后辈关系，那么把两个点的左括号位置作为询问区间的两端点；- 反之，把一个点的左括号另一个点的右括号位置作为询问区间的两端点，但是这样会漏掉 LCA，记得把 LCA 补上。

```
const int N = 4e4, M = 1e5;

int n, q, bsz, ans[M + 10], bid[(N << 1) + 10], col[N + 10], arr[(N << 1) + 10], cnt[N + 10];
```

```cpp
vector<int> e[N + 10];
bool vis[N + 10];

void adde(int x, int y) { e[x].push_back(y); }

struct Queries {
    int l, r, id, LCA;
    bool operator < (const Queries &rhs) const {
        if (bid[l] == bid[rhs.l]) return r < rhs.r;
        return bid[l] < bid[rhs.l];
    }
} Q[M + 10];

namespace TreeLink {
    int timer, sz[N + 10], fa[N + 10], dep[N + 10], ltop[N + 10], ch[N + 10], indfn[N + 10], outdfn[N + 10];
    void dfs1(int u, int p) {
        sz[u] = 1, fa[u] = p, dep[u] = dep[p] + 1, arr[++timer] = u, indfn[u] = timer;
        for (auto v : e[u]) {
            if (v == p) continue;
            dfs1(v, u), sz[u] += sz[v];
            if (sz[v] > sz[ch[u]]) ch[u] = v;
        }
        arr[++timer] = u, outdfn[u] = timer;
    }
    void dfs2(int u, int p) {
        ltop[u] = p;
        if (ch[u] == 0) return;
        dfs2(ch[u], p);
        for (auto v : e[u]) {
            if (v == fa[u] || v == ch[u]) continue;
            dfs2(v, v);
        }
    }
    int lca(int u, int v) {
        while (ltop[u] != ltop[v]) {
            if (dep[ltop[u]] < dep[ltop[v]]) swap(u, v);
            u = fa[ltop[u]];
        }
        if (dep[u] > dep[v]) swap(u, v);
        return u;
    }
} // namespace TreeLink
using namespace TreeLink;

void modify(int x, int p) {
    if (vis[p]) ans[0] -= (--cnt[x] == 0), vis[p] = 0;
    else ans[0] += (++cnt[x] == 1), vis[p] = 1;
}

signed main() {
    fr(n, q), bsz = sqrt(n);
    for (int i = 1; i <= n; i++) fr(col[i]), arr[i] = col[i];
    for (int i = 1; i <= n * 2; i++) bid[i] = (i - 1) / bsz + 1;
    for (int i = 1, u, v; i < n; i++) fr(u, v), adde(u, v), adde(v, u);
    sort(arr + 1, arr + 1 + n), arr[0] = unique(arr + 1, arr + 1 + n) - arr - 1;
    for (int i = 1; i <= n; i++) col[i] = lower_bound(arr + 1, arr + 1 + arr[0], col[i]) - arr;
    dfs1(1, 0), dfs2(1, 1);
    for (int i = 1, u, v; i <= q; i++) {
        fr(u, v);
        if (indfn[u] > indfn[v]) swap(u, v);
        int LCA = lca(u, v);
        if (LCA == u) Q[i] = {indfn[u], indfn[v], i, 0};
        else Q[i] = {outdfn[u], indfn[v], i, LCA};
    }
    sort(Q + 1, Q + 1 + q);
    for (int i = 1, l = 1, r = 0; i <= q; i++) {
        while (Q[i].l < l) l--, modify(col[arr[l]], arr[l]);
        while (Q[i].r > r) r++, modify(col[arr[r]], arr[r]);
        while (Q[i].l > l) modify(col[arr[l]], arr[l]), l++;
        while (Q[i].r < r) modify(col[arr[r]], arr[r]), r--;
        if (Q[i].LCA) modify(col[Q[i].LCA], Q[i].LCA);
```

```
        ans[Q[i].id] = ans[0];
        if (Q[i].LCA) modify(col[Q[i].LCA], Q[i].LCA);
    }
    for (int i = 1; i <= q; i++) printf("%d\n", ans[i]);
    return 0;
}
```

可并堆

随机堆

将堆 y 合并到堆 x 上时，以 0.5 的概率交换 x 的左右儿子，然后将 x 的左儿子与 y 合并，可以证明这样子合并得到的堆树高是 $\log n$ 级别的。

记得找堆顶要另外记一个父亲然后用并查集维护。

```
const int N = 1e5;

mt19937 rnd((unsigned long long)(new char));
int n, m, fa[N + 10], ch[N + 10][2];
bool vis[N + 10];
pair<int, int> val[N + 10];

int find(int u) {
    if (fa[u] == u) return u;
    return fa[u] = find(fa[u]);
}
int merge(int x, int y) {
    if (x == 0 || y == 0) return x | y;
    if (val[x] > val[y]) swap(x, y);
    if (rnd() & 1) swap(ch[x][0], ch[x][1]);
    return ch[x][0] = merge(ch[x][0], y), x;
}

struct OI {
    int RP, score;
} FJOI2022;

signed main() {
    FJOI2022.RP++, FJOI2022.score++;
    fr(n, m);
    for (int i = 1; i <= n; i++) fr(val[i].first), val[i].second = i, fa[i] = i;
    for (int i = 1, op, x, y; i <= m; i++) {
        fr(op, x);
        if (op == 1) {
            fr(y);
            if (vis[x] || vis[y]) continue;
            x = find(x), y = find(y);
            if (x != y) fa[x] = fa[y] = merge(x, y);
        } else {
            if (vis[x]) { puts("-1"); continue; }
            x = find(x), fa[ch[x][0]] = fa[ch[x][1]] = fa[x] = merge(ch[x][0], ch[x][1]), vis[x] = 1, ch[x][0] =
 ↪  ch[x][1] = 0, printf("%d\n", val[x].first);
        }
    }
    return 0;
}
```

然后这东西有点类似于线段树合并，可以在堆顶上打标记对整个堆操作。

例题：洛谷 P3261 [JLOI2015] 城池攻占。

平衡树

FHQ Treap

维护序列的时候按大小把树分裂开提出区间打标记，维护集合的时候还需要按权值分裂的操作。

分裂的时候先考虑左子树是否完全被包括，是的话则将左子树加入前半边树里，往右子树递归分裂；如果未被完全包括，将左子树加入后半边树里，往左子树分裂递归。

```cpp
void splitsz(Node *u, int sz, Node *&x, Node *&y) {
    if (u == null) return x = y = null, void();
    u->pushdown();
    if (u->ch[0]->sz + 1 <= sz) x = u, splitsz(u->ch[1], sz - u->ch[0]->sz - 1, u->ch[1], y);
    else y = u, splitsz(u->ch[0], sz, x, u->ch[0]);
    u->pushup();
}
```

（不要忘记 pushdown/up）

合并就简单不少！我们待合并的两棵树值域是不交的，以下假定 x 树内的值均小于 y 树内的，那么一定只有两种情况：1. 将 x 合并到 y 的左子树；2. 将 y 合并到 x 的右子树。

那么到底选用那种情况呢？我们考虑 Treap 还有一个随机权值满足 heap 的性质，也就是说（以小根堆为例）：1. 如果点 x 的随机权值比 y 的随机权值小，那么将 y 合并到 x 的右子树；2. 否则，将 x 合并到 y 的左子树。

```cpp
Node *merge(Node *x, Node *y) {
    if (x == null || y == null) return (x == null) ? y : x;
    x->pushdown(), y->pushdown();
    if (x->rndval < y->rndval) return x->ch[1] = merge(x->ch[1], y), x->pushup(), x;
    return y->ch[0] = merge(x, y->ch[0]), y->pushup(), y;
}
```

简单的维护序列例题：洛谷 P3372 【模板】线段树 1。

```cpp
#define int long long

const int N = 1e5;

int n, m;

mt19937 rnd((unsigned long long)(new char));

struct Node {
    int val, rndval, sum, atag, sz;
    Node *ch[2];
    void add(int x) { val += x, sum += sz * x, atag += x; }
    void pushup() { sum = ch[0]->sum + ch[1]->sum + val, sz = ch[0]->sz + ch[1]->sz + 1; }
    void pushdown() { if (atag) ch[0]->add(atag), ch[1]->add(atag), atag = 0; }
} mem[N + 10], *atot = mem, *rt, *null;

void Init() { null = atot, null->ch[0] = null->ch[1] = null, rt = null; }
Node *newNode(int val) { return atot++, atot->ch[0] = atot->ch[1] = null, atot->sum = atot->val = val, atot->sz = 1,
 ↪  atot->rndval = rnd(), atot; }

void splitsz(Node *u, int sz, Node *&x, Node *&y) {
    if (u == null) return x = y = null, void();
    u->pushdown();
    if (u->ch[0]->sz + 1 <= sz) x = u, splitsz(u->ch[1], sz - u->ch[0]->sz - 1, u->ch[1], y);
    else y = u, splitsz(u->ch[0], sz, x, u->ch[0]);
    u->pushup();
}
Node *merge(Node *x, Node *y) {
    if (x == null || y == null) return (x == null) ? y : x;
    x->pushdown(), y->pushdown();
    if (x->rndval < y->rndval) return x->ch[1] = merge(x->ch[1], y), x->pushup(), x;
    return y->ch[0] = merge(x, y->ch[0]), y->pushup(), y;
}

signed main() {
    Init(), fr(n, m);
    for (int i = 1; i <= n; i++) rt = merge(rt, newNode(fr()));
    for (int _ = 1, op, l, r, v; _ <= m; _++) {
        fr(op, l, r);
        Node *x, *y, *z;
        splitsz(rt, l - 1, x, y), splitsz(y, r - l + 1, y, z);
        if (op == 1) fr(v), y->add(v);
        else printf("%lld\n", y->sum);
        rt = merge(x, merge(y, z));
    }
    return 0;
}
```

数据结构优化建图

前缀和优化建图

感觉没啥需要特别说的，就是利用前缀和的性质，把前缀缩成一个点然后建图，感性理解一下场上现推就是了。

例题：洛谷 P6378 [PA2010] Riddle。

线段树优化建图

useful trick。用于一些需要点向区间连边 / 区间向区间连边的题目。

怎么做？建两棵线段树，一棵树上父亲向儿子连边权为 0 的有向边（记为 T_1），一棵树上儿子向父亲连边权为 0 的有向边（记为 T_2），两棵树相同位置的叶子节点之间连边权为 0 的无向边。

- 点向点连边：T_1, T_2 对应叶子节点相连。
- 点向区间连边：T_2 的叶子向 T_1 的区间连边。
- 区间向点连边：T_2 的区间向 T_1 的叶子连边。
- 区间向区间连边：建一个虚点，T_1, T_2 的区间向虚点连边。

感觉考场上手画两棵线段树就能现推了啊 ☒

例题：CF786B Legacy。

手写 Bitset

```
struct Bitset {
    unsigned long long A[782];
    Bitset operator|(Bitset b) const {
        Bitset c;
        for (int i = 0; i <= len; i++) c.A[i] = A[i] | b.A[i];
        return c;
    }
    Bitset operator^(Bitset b) const {
        Bitset c;
        for (int i = 0; i <= len; i++) c.A[i] = A[i] ^ b.A[i];
        return c;
    }
    void reset() {
        for (int i = 0; i <= len; i++) A[i] = 0;
    }
    void set(int x) { A[x >> 6] |= 1ull << (x & 63); }
    void flip(int x) { A[x >> 6] ^= 1ull << (x & 63); }
    int count() {
        int ret = 0;
        for (int i = 0; i <= len; i++) ret += dt[A[i] >> 48] + dt[(A[i] >> 32) & S] + dt[(A[i] >> 16) & S] + dt[A[i] &
 ↪  S];
        return ret;
    }
}
```

图论

树上问题

dfs 序树上背包

那么如果每个点的体积不是 1 了，我们该怎么写复杂度优秀的树上背包呢？

我们考虑把树拍成一个 dfs 序（此处的 dfs 序与常规的 dfs 序不太一样，我们在 dfs 退栈的时候将节点加入 dfs 序列），然后在 dfs 序上做背包，记 $f_{i,j}$ 表示 dfs 序前 i 个组成的森林，体积为 j 的答案。转移分两种：1. i 不选，那么 i 的子树也都不能选，从 i 的前一个兄弟节点转移过来；2. i 选，那么 i 的子树可以选，从 i 的第一个子节点转移过来；

状态数 $\Theta(nm)$，转移 $\Theta(1)$，所以总时间复杂度 $\Theta(nm)$，很优秀！

对于不定根的树上背包，套一个点分治即可。时间复杂度多个 log。

```cpp
const int N = 1e3, M = 1e4;

int n, m, timer, ans, tot, rt, mx[N + 10], nfd[N + 10], v[N + 10], w[N + 10], sz[N + 10], f[N + 5][M + 5];
vector<int> e[N + 10];
bool vis[N + 10];

void adde(int x, int y) { e[x].push_back(y); }

void dfs(int u, int p) {
    sz[u] = 1;
    for (auto v : e[u]) if (v != p && vis[v] == 0) dfs(v, u), sz[u] += sz[v];
    nfd[++timer] = u;
}

void getrt(int u, int p) {
    sz[u] = 1, mx[u] = 0;
    for (auto v : e[u]) if (v != p && vis[v] == 0) getrt(v, u), sz[u] += sz[v], mx[u] = max(mx[u], sz[v]);
    mx[u] = max(mx[u], tot - sz[u]);
    if (mx[u] < mx[rt]) rt = u;
}

void solve(int u) {
    vis[u] = 1, timer = 0, dfs(u, 0);
    for (int i = 0; i <= timer; i++) for (int j = 0; j <= m; j++) f[i][j] = 0;
    for (int i = 1; i <= timer; i++)
        for (int j = 1; j <= m; j++) {
            f[i][j] = f[i - sz[nfd[i]]][j];
            if (j >= v[nfd[i]]) f[i][j] = max(f[i][j], f[i - 1][j - v[nfd[i]]] + w[nfd[i]]);
        }
    for (int i = 0; i <= m; i++) ans = max(ans, f[timer][i]);
    for (auto v : e[u]) if (vis[v] == 0) tot = sz[v], rt = 0, getrt(v, 0), solve(rt);
}

signed main() {
    fr(n, m);
    for (int i = 1; i <= n; i++) fr(v[i], w[i]);
    for (int i = 1, x, y; i < n; i++) fr(x, y), adde(x, y), adde(y, x);
    tot = n, mx[0] = n + 1, rt = 0, getrt(1, 0), solve(rt);
    return printf("%d\n", ans), 0;
}
```

树的直径与重心

直径小结论：若树上所有边边权均为正，则树的所有直径中点重合。

重心性质：1. 以树的重心为根时，所有子树的大小都不超过整棵树大小的一半。2. 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。3. 把两棵树通过一条边相连得到一棵新的树，那么新的树的重心在连接原来两棵树的重心的路径上。4. 在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离。

感觉这俩玩意很有可能拿出来出思维题。

重链剖分

随便记一点，这东西可千万不能忘。

维护重儿子和重链链顶，然后赋 dfs 序的时候记得先搜重儿子，做链操作的时候就两个点一直跳深度大的点的重链直到两个点位于一个重链上即可。

虚树

说个简单的虚树建法：- 把所有关键点扔到一个序列里，然后把序列按照 dfs 序从大到小排序；- 把排序后的相邻两个关键点的 lca 加入到序列里，然后把序列按照 dfs 序从大到小排序；- 给序列去重，然后给序列里相邻两个点 a_i, a_{i+1} 的 lca 向 a_{i+1} 连边。

时间复杂度 $\Theta(k \log k)$，k 为关键点数量。常数比栈做法大一点。

```cpp
int dfn[MAXN];
int h[MAXN], m, a[MAXN], len;  // 存储关键点

bool cmp(int x, int y) {
```

```
    return dfn[x] < dfn[y];  // 按照 dfs 序排序
}

void build_virtual_tree() {
    sort(h + 1, h + m + 1, cmp);  // 把关键点按照 dfs 序排序
    for (int i = 1; i < m; ++i) {
        a[++len] = h[i];
        a[++len] = lca(h[i], h[i + 1]);  // 插入 lca
    }
    a[++len] = h[m];
    sort(a + 1, a + len + 1, cmp);  // 把所有虚树上的点按照 dfs 序排序
    len = unique(a + 1, a + len + 1) - a - 1;  // 去重
    for (int i = 1, lc; i < len; ++i) {
        lc = lca(a[i], a[i + 1]);
        conn(lc, a[i + 1]);  // 连边，如有边权 就是 distance(lc,a[i+1])
    }
}
```

树上启发式合并（dsu on tree）

用于快速计算树上的一些与子树相关的贡献。一定程度上可与线段树合并相互替代。

算法过程：暴力 dfs 计算贡献，但是每次保留重儿子的贡献到父亲，可以证明这样做的时间复杂度为 $\Theta(n \log n)$。

```
void clr(int u, int p) {
    ... // 清除点 u 的贡献
    for (auto v : e[u]) if (v != p) clr(v, u);
}
void calc(int u, int p, int c = 0) {
    ... // 添加点 u 的贡献
    for (auto v : e[u]) if (v != p && v != c) calc(v, u);
}
void dfs(int u, int p, bool f) { // f = 1 表示保留贡献，反之为不保留贡献
    for (auto [v, w] : e[u]) if (v != p && v != ch[u]) dfs(v, u, 0);
    if (ch[u]) dfs(ch[u], u, 1);
    calc(u, p, ch[u]);
    ... // 计算 u 子树的贡献
    if (f == 0) clr(u, p);
}
```

例题：CF600E Lomsat gelral。

长链剖分

与重链剖分按子树大小分配重儿子类似，长链剖分按子树高度分配长儿子。

```
void dfs(int u, int p) {
    ht[u] = 1;
    for (auto v : e[u]) {
        if (v == p) continue;
        dfs(v, u), ht[u] = max(ht[u], ht[v] + 1);
        if (ht[v] > ht[ch[u]]) ch[u] = v;
    }
}
```

性质：1. 树上所有长链长度和为 n。证明显然。2. 一个节点跳跃长链到根节点，跳跃的次数最多为 $\Theta(\sqrt{n})$。证明：考虑到每次跳跃的时候长链的长度一定单调不减，因此最多跳 $\Theta(\sqrt{n})$ 次。

长链剖分优化 DP：这东西才是长剖的重点！感觉与 dsu on tree 有异曲同工之妙。

适用范围：用于合并一些与子树深度有关的信息。例如优化状态带有深度的树上 DP。

例如一个 DP：记 $f_{u,i}$ 为 u 子树内与 u 距离为 i 的节点数量，那么我们有：

$$f_{u,i} = \sum_{(u,v)} f_{v,i-1}$$

暴力转移的时间复杂度为 $\Theta(n^2)$。化用 dsu on tree 的思路我们可以做到 $\Theta(n\sqrt{n})$。

但是还可以更快，我们考虑一件事情：不管是重链还是长链，所有的链长度之和都等于 n。也就是说，如果我们不清除贡献，那么时间复杂度就是 $\Theta(n)$ 的。

而之所以 dsu on tree 需要清除贡献，是因为所有节点计算贡献的时候都在一个数组上计算，如果不消除就会重复计算贡献。但是在这题中，DP 数组的贡献是记在 n 个数组上的，也就是说不会出现重复计算贡献的情况！那么我们就可以不清除贡献，时间复杂度变为 $\Theta(n)$。

实现方法：可以注意到，如果需要做到线性复杂度，那么我们需要将 u 的 DP 数组从长儿子 v 的 DP 数组直接继承过来，那么该如何实现呢？

我们用指针给 DP 数组分配空间，一条长链上的内存连续，只是 v 的 DP 数组是 u 平移一位，这样在 v 的 DP 数组上修改也会直接作用到 u 的 DP 数组上，这样就达到了我们的要求。

```
int mem[N + 10], *atot = mem, *f[N + 10];

void dfs1(int u, int p) {
    ht[u] = 1;
    for (auto v : e[u]) {
        if (v == p) continue;
        dfs1(v, u), ht[u] = max(ht[u], ht[v] + 1);
        if (ht[ch[u]] < ht[v]) ch[u] = v;
    }
}

void dfs(int u, int p) {
    if (f[u] == 0) f[u] = atot, atot += ht[u];
    f[u][0] = 1;
    if (ch[u]) f[ch[u]] = f[u] + 1, dfs(ch[u], u);
    for (auto v : e[u]) {
        if (v == p || v == ch[u]) continue;
        dfs(v, u);
        for (int i = 1; i <= ht[v]; i++) f[u][i] += f[v][i - 1];
    }
}
```

例题：CF1009F Dominant Indices。

点分治 / 边分治

和分治类似，用于计算点对间的贡献。

分治过程也与普通分治类似：每次只计算过当前点/边的贡献。

现在问题就变为怎么均匀的分治问题使得复杂度不退化：- 对于点分治，我们每次取子树的重心进行分治。时间复杂度 $\Theta(n \log n)$。- 对于边分治，我们每次取一条边使得其尽量将劈开的两棵子树大小均匀。然而你发现好像……造个菊花图就寄了啊，怎么办捏？三度化！我们将树重构成一棵点度最多只有三的树即可，代码如下：

```
void remake(int u, int p) {
    int t = u;
    for (auto v : e[u]) {
        if (v == p) continue;
        adde(++ptot, v, 1, 1), adde(v, ptot, 1, 1), adde(t, ptot, 0, 1), adde(ptot, t, 0, 1), t = ptot, remake(v, u);
    }
}
```

然而即使三度化之后也不能保证一条边能完全把一棵树均匀劈开，只能做到 1:2 劈开，所以时间复杂度 $\Theta(n \log_{1.5} n)$，加上三度化之后点变多所以常数还要大一点。但是边分最大的优势就是它好写，因为它只需要考虑边两端合并信息，不需要像点分治那样对于一堆子树合并信息。

最短路

SPFA & Dijkstra

打标记的顺序：- SPFA 中的标记是用于判断某个点是否在队列里；- 而 Dijkstra 中的标记是用于判断某个点是否被松弛过了。

```
struct Node {
    long long val;
    int u;
    bool operator<(const Node &a) const { return val > a.val; }
};
```

```

namespace Heap {
    template <class T> struct heap {
        T h[M + 10];
        int sz;
        void push(T x) { h[++sz] = x, std::push_heap(h + 1, h + sz + 1); }
        void pop() { std::pop_heap(h + 1, h + sz + 1), sz--; }
        T top() { return h[1]; }
        int size() { return sz; }
        int empty() { return sz & 1; }
    };
} // namespace Heap
using namespace Heap;
heap<Node> q;

while (q.size()) {
    int u = q.top().u; q.pop();
    if (vis[u]) continue;
    vis[u] = 1;
    for (auto [v, w] : e[u]) if (dis[u] + w < dis[v]) q.push({dis[v] = dis[u] + w, v});
}
```

SPFA 判负环

每个点进队时记录一下进队次数，如果达到 $n+1$ 次则表示图中存在负环。

差分约束系统

差分约束系统：就是给你 n 个变量 x_i 然后有 m 个约束条件，每个约束条件是两个变量做差的关系 $x_i - x_j \le c_k$。求 n 个变量的值。

可以发现差分约束条件转换成 $x_i \le x_j + c_k$ 后长得很像最短路里的三角形不等式 $dis_v \le dis_u + w$。因此我们对于每个约束条件建 j 连向 i 边权为 c_k 的边，然后整一个超级源点 0 令 $dis_0 = 0$ 并对每个点连边权为 0 的边跑最短路。如果图上存在负环则该差分约束系统无解，原因显然。反之可以求出最小解（因为跑的是最短路，若要求最大解则建一个最长路模型即可）。

然后说一下 $x_i = x_j$ 条件的转换：转换为 $x_i - x_j \le 0$ 和 $x_j - x_i \le 0$ 即可。

最小生成树

Kruskal 重构树

在跑 Kruskal 的过程中每次合并两个点 (u, v) 的时候新建一个点 p 作为一棵新树上 u, v 的父亲，然后跑完整个 Kruskal 后得到的树就叫 Kruskal 重构树。

它有一个超牛逼的性质：- 原图中两个点之间的所有简单路径上最大边权的最小值 = 最小生成树上两个点之间的简单路径上的最大值 = Kruskal 重构树上两点之间的 LCA 的权值。

证明显然，你脑补一下 Kruskal 的过程就知道了。

然后根据这个性质我们还有：- 到点 x 的简单路径上最大边权的最小值 $\le v$ 的所有点 y 均在 Kruskal 重构树上的某一棵子树内，且恰好为该子树的所有叶子节点。- 那么我们在重构树上找到 x 到根的路径上最浅的节点 u 满足 $a_u \le v$，则 u 子树内所有叶子节点都是满足条件的点。

如果要求路径上边权最小值的最大值，那么做 Kruskal 过程的时候将边从大到小排序即可。

```
const int N = 14e4, M = 18, MOD = 1e9 + 7;

int n, m, _, tot, A, B, C, P, ans, timer, etot, hd[N + 10], ufa[N + 10], val[N + 10], sz[N + 10],
    dep[N + 10], dfn[N + 10], efn[M + 3][(N << 1) + 10], lg2[(N << 1) + 10];
struct Edge { int u, v, w; } E[N + 10]; // vector<int> e[N + 10];
struct EDGE { int to, nxt; } e[(N << 1) + 10];

// void adde(int x, int y) { e[x].push_back(y); }
void adde(int x, int y) { e[++etot] = {y, hd[x]}, hd[x] = etot; }

int find(int u) { return u == ufa[u] ? u : ufa[u] = find(ufa[u]); }

int rnd() { return A = (A * B + C) % P; }

```

```cpp
void dfs(int u, int p) {
    dep[u] = dep[p] + 1, dfn[u] = ++timer, efn[0][timer] = u;
    for (int i = hd[u]; i; i = e[i].nxt) {
        int v = e[i].to;
        if (v == p) continue;
        dfs(v, u), efn[0][++timer] = u;
    }
}
int cmin(int a, int b) { return dep[a] < dep[b] ? a : b; }
void InitST() {
    for (int i = 2; i <= timer; i++) lg2[i] = lg2[i >> 1] + 1;
    for (int j = 1; j <= lg2[timer]; j++)
       for (int i = 1; i + (1 << j) - 1 <= timer; i++) efn[j][i] = cmin(efn[j - 1][i], efn[j - 1][i + (1 << (j - 1))]);
}
int lca(int u, int v) {
    int l = dfn[u], r = dfn[v];
    if (l > r) swap(l, r);
    int s = lg2[r - l + 1];
    return cmin(efn[s][l], efn[s][r - (1 << s) + 1]);
}

void build() {
    sort(E + 1, E + 1 + m, [] (Edge &a, Edge &b) { return a.w < b.w; }), tot = n;
    for (int i = 1; i <= (n << 1); i++) ufa[i] = i;
    for (int i = 1, etot = 0; i <= m && etot < n; i++) {
        int u = find(E[i].u), v = find(E[i].v);
        if (u == v) continue;
        ufa[u] = ufa[v] = ++tot, val[tot] = E[i].w, adde(u, tot), adde(tot, u), adde(v, tot), adde(tot, v), etot++;
    }
}

signed main() {
    fr(n, m);
    for (int i = 1, u, v, w; i <= m; i++) fr(u, v, w), E[i] = {u, v, w};
    build(), dfs(tot, 0), InitST(), fr(_, A, B, C, P);
    for (int i = 1, a, b; i <= _; i++) a = rnd() % n + 1, b = rnd() % n + 1, ans = (ans + val[lca(a, b)]) % MOD;
    return printf("%d\n", ans), 0;
}
```

Boruvka

每次对于每个连通块都找到一个最小的与其他连通块的出边，最后把边全部连起来。可以发现一次拓展连通块就少一半，因此最多跑 $\Theta(\log n)$ 轮，时间复杂度 $\Theta(m \log n)$。

之所以说这东西是因为它在处理完全图最小生成树上有奇效。结合一些数据结构维护最小边是很容易的，所以有的时候用它来处理最小生成树很方便！

欧拉路径 / 欧拉回路

欧拉回路存在的必要条件：- 对于有向图：所有的点出度和入度相等。- 对于无向图：所有的点度都是偶数。

欧拉路径存在的必要条件：- 对于有向图：最多只有两个点的出入度不相等，其中出度多的那个点为起点，出度少的那个点为终点。- 对于无向图：只有 0 个或者 2 个点的度数为奇数。

怎么求欧拉路径 / 欧拉回路？搜索就行了，搜过的边打个标记，加个当前弧优化保证复杂度。时间复杂度 $\Theta(n+m)$。

注意：对于求欧拉回路的情况，我们需要找一个有出边的点作为起点开始搜索。以及要判断最后求出来的答案序列长度是否为 m。

欧拉回路代码（UOJ #117 欧拉回路）：

```cpp
const int N = 1e5, M = 2e5;

int n, m, t, etot = 1, deg[N + 10], hd[N + 10], ans[M + 10];
struct Edge { int to, nxt; bool vis; } e[(M << 1) + 10];

#define adde(x, y) (e[++etot] = {y, hd[x], 0}, hd[x] = etot)

void dfs(int u) {
    for (int &i = hd[u], v = e[i].to; i; v = e[i = e[i].nxt].to)
        if (e[i].vis == 0) {
```

```
            int tmp = i;
            e[i].vis = 1;
            if (t == 1) e[i ^ 1].vis = 1;
            dfs(v), ans[++ans[0]] = tmp;
        }
}

signed main() {
    fr(t, n, m);
    if (t == 2) {
        for (int i = 1, x, y; i <= m; i++) fr(x, y), adde(x, y), deg[x]++, deg[y]--;
        for (int i = 1; i <= n; i++) if (deg[i]) return puts("NO"), 0;
        for (int i = 1; i <= n && ans[0] == 0; i++) dfs(i);
        if (ans[0] != m) return puts("NO"), 0;
        puts("YES");
        for (int i = m; i; i--) printf("%d%c", ans[i] - 1, " \n"[i == 1]);
    } else {
        for (int i = 1, x, y; i <= m; i++) fr(x, y), adde(x, y), adde(y, x), deg[x]++, deg[y]++;
        for (int i = 1; i <= n; i++) if (deg[i] & 1) return puts("NO"), 0;
        for (int i = 1; i <= n && ans[0] == 0; i++) dfs(i);
        if (ans[0] != m) return puts("NO"), 0;
        puts("YES");
        for (int i = m; i; i--) printf("%d%c", ((ans[i] & 1) ? -1 : 1) * (ans[i] / 2), " \n"[i == 1]);
    }
    return 0;
}
```

欧拉路径代码（洛谷 P7771 【模板】欧拉路径）：

```
const int N = 1e5, M = 2e5;

int n, m, cnt, tot, s, t, ans[M + 10], ideg[N + 10], odeg[N + 10];
vector<pair<int, int>> e[N + 10];

void adde(int x, int y) { e[x].push_back({y, 0}), odeg[x]++, ideg[y]++; }

void dfs(int u) {
    while (e[u].size()) {
        auto &[v, vis] = e[u].back(); e[u].pop_back();
        if (vis == 0) vis = 1, dfs(v);
    }
    ans[++tot] = u;
}

signed main() {
    fr(n, m);
    for (int i = 1, x, y; i <= m; i++) fr(x, y), adde(x, y);
    for (int i = 1; i <= n; i++) sort(e[i].begin(), e[i].end()), reverse(e[i].begin(), e[i].end());
    for (int i = 1; i <= n; i++)
        if (ideg[i] != odeg[i]) {
            cnt++;
            if (ideg[i] + 1 == odeg[i]) s = i;
            if (ideg[i] == odeg[i] + 1) t = i;
        }
    if (cnt == 0) s = 1;
    else if (cnt != 2 || s == 0 || t == 0) puts("No"), exit(0);
    dfs(s);
    for (int i = tot; i; i--) printf("%d%c", ans[i], " \n"[i == 1]);
    return 0;
}
```

Tarjan & 图的连通分量相关性质 & 割点割边

还是三合一。

强连通：定义有向图强连通当且仅当图中任意两个结点连通。

点双连通：定义无向图点双连通当且仅当图中任意两个结点在去掉除它们俩之外的任意一个点之后还连通。

边双连通：定义无向图边双连通当且仅当图中任意两个结点在去掉任意一条边之后还连通。

__ **连通分量**：极大的 __ 连通子图。

关于 low：low 表示的是 dfs 栈帧中能够到达的点的 dfn 最小值!! 在栈里!!! 很重要!! 记住这一点 low 值的更新方式就不会忘!!

求所有的 SCC：当遇到 low = dfn 的情况时，把栈里当前点之下的东西全部弹出来，这就是一个 SCC。

```
void tarjan(int u) {
    dfn[u] = low[u] = ++timer, instk[u] = 1, stk.push(u);
    for (auto v : e[u])
        if (dfn[v] == 0) tarjan(v), low[u] = min(low[u], low[v]);
        else if (instk[v]) low[u] = min(low[u], dfn[v]);
    if (dfn[u] == low[u]) {
        scc.push_back({});
        while (1) {
            int cur = stk.top();
            stk.pop(), instk[cur] = 0, to[cur] = scc.size(), scc.back().push_back(cur);
            if (cur == u) break;
        }
    }
}
```

无向图的时候不需要判断是否在栈里，因为其实所有搜过的点都在栈里，原因显然。

另外就算不要求 SCC 也要记得退栈，不然求 low 的时候会寄，道理显然。

有向图缩 SCC 后得到的是 DAG。很重要。

求割点：$\text{low}_v \geq \text{dfn}_u$ 的点 u 就是割点。注意特判根节点是否有超过一个不连通的儿子，是的话则根节点也是割点。

求割边：$\text{low}_v > \text{dfn}_u$ 的边 (u, v) 就是割边。

2-SAT

解决如下一类问题模型：- 有 n 个布尔变量 x_i；- 有 m 个限制条件，均为「x_a 为 b 或 x_c 为 d」；- 求一组可行解。

构造图论模型，例如「x_1 为真或 x_2 为假」就连两条边：- $\neg x_1 \to \neg x_2$；- $x_2 \to x_1$。

即：- 若 x_1 为假，则 x_2 必假；- 若 x_2 为真，则 x_1 必真。

然后可以发现建出来的这个图上一个 SCC 内的变量取值必须都相同，这样我们就能够判断无解了：若 x_i 与 $\neg x_i$ 不在一个 SCC 内，则无解。

反之必有解，一种构造解的方案是以锁完点后的 SCC 拓扑序大小来构造解，说人话就是设 x_i 所在 SCC 编号为 p，$\neg x_i$ 所在 SCC 编号为 q，则 x_i 的取值为 $[p < q]$。

```
const int N = 2e6;

int n, m, timer, stot, dfn[N + 10], low[N + 10], ideg[N + 10], to[N + 10];
bool instk[N + 10];
stack<int> stk;
vector<int> e[N + 10];

void tarjan(int u) {
    dfn[u] = low[u] = ++timer, instk[u] = 1, stk.push(u);
    for (auto v : e[u])
        if (dfn[v] == 0) tarjan(v), low[u] = min(low[u], low[v]);
        else if (instk[v]) low[u] = min(low[u], low[v]);
    if (dfn[u] == low[u]) {
        stot++;
        while (1) {
            int cur = stk.top();
            stk.pop(), instk[cur] = 0, to[cur] = stot;
            if (cur == u) break;
        }
    }
}

signed main() {
    fr(n, m);
    for (int i = 1; i <= m; i++) {
        int a = fr(), va = fr(), b = fr(), vb = fr();
        e[a + n * va].push_back(b + n * (vb ^ 1)), e[b + n * vb].push_back(a + n * (va ^ 1));
```

```
    }
    for (int i = 1; i <= (n << 1); i++)
        if (dfn[i] == 0) tarjan(i);
    for (int i = 1; i <= n; i++)
        if (to[i] == to[i + n]) return puts("No"), 0;
    puts("Yes");
    for (int i = 1; i <= n; i++) printf("%d%c", to[i] < to[i + n], " \n"[i == n]);
    return 0;
}
```

二分图

二分图最大匹配

我知道网络流建模大家都会，所以我只讲一下匈牙利（KM）做法。

本质上就是不断找增广路的过程：- 对于一个点 u，如果它已经被搜过，就不往下搜；- 反之，我们找到它的所有后继节点 v，然后往下搜索，如果搜出一条增广路，则让 u 与 v 匹配。- 对每个点都进行一次增广，每增广成功一次最大匹配就增加一。

```
bool dfs(int u, int idx) {
    if (vis[u] == idx) return 0;
    vis[u] = idx;
    for (auto v : e[u]) if (mch[v] == 0 || dfs(mch[v], idx)) return mch[v] = u, 1;
    return 0;
}

int solve() {
    int ans = 0;
    for (int i = 1; i <= n; i++) if (dfs(i, i)) ans++;
    return ans;
}
```

一些二分图相关问题

- 二分图最小点覆盖 = 二分图最大匹配。
- 二分图最小边覆盖 = 二分图最大独立集 = 二分图点数 − 二分图最小点覆盖。

网络流

Dinic

不懂原理，只会背板。

不断地 bfs 残量网络建出分层图，然后在分层图上搜索：- 搜到汇点则返回当前的流量。- 反之遍历分层图上当前点每条有余流的出边，用当前点的余流和这条出边的余流取 min 作为供给下一个点的流量，将返回的流量贡献计算。记得给反向边退流。- 小优化：如果搜出去的出边返回的流量是 0，给这个出点打上标记之后不再给这个点流量。- 记得写当前弧优化。

```
#define int long long

const int N = 1e2, M = 5e3, INF = 0x7f7f7f7f7f7f7f7f;

int n, m, s, t, ans, d[N + 10], ecur[N + 10], que[N + 10], Hd, Tl;

namespace Edges {
    struct Edge { int to, nxt, flow; } e[(M << 1) + 10];
    int hd[N + 10], etot = 1;
    void add(int x, int y, int z) { e[++etot] = {y, hd[x], z}, hd[x] = etot; }
    void adde(int x, int y, int z) { add(x, y, z), add(y, x, 0); }
} // namespace Edges
using namespace Edges;

bool bfs() {
    memset(d, -1, sizeof(d)), Hd = 1, Tl = 0, que[++Tl] = s, d[s] = 0, ecur[s] = hd[s];
    while (Hd <= Tl) {
        int u = que[Hd++];
        for (int i = hd[u]; i; i = e[i].nxt) {
            int v = e[i].to;
            if (d[v] != -1 || e[i].flow == 0) continue;
            que[++Tl] = v, d[v] = d[u] + 1, ecur[v] = hd[v];
```

```
            if (v == t) return 1;
        }
    }
    return 0;
}

int dfs(int u, int limit) {
    if (u == t) return limit;
    int flow = 0;
    for (int i = ecur[u]; i && flow < limit; i = e[i].nxt) {
        int v = e[i].to; ecur[u] = i;
        if (e[i].flow == 0 || d[v] != d[u] + 1) continue;
        int f = dfs(v, min(e[i].flow, limit - flow));
        if (f == 0) d[v] = -1;
        e[i].flow -= f, e[i ^ 1].flow += f, flow += f;
    }
    return flow;
}

int dinic() {
    int ans = 0, flow = 0;
    while (bfs()) while (flow = dfs(s, INF)) ans += flow;
    return ans;
}
```

时间复杂度上界 $\Theta(n^2m)$。实际运行中远远达不到这个上界。

zkw 费用流

EK 拉爆了，来学多路增广的费用流！

其实和 Dinic 一样，只是把 bfs 换成 SPFA，然后记得搜索过程中计算一下费用。有一点不同，就是没有必要一直在同一个残量网络上面跑 dinic。另外反向边的费用记得也要取反。

```
const int N = 4e2, M = 1.5e4, INF = 0x3f3f3f3f;

int n, m, s, t, maxflow, mincost, d[N + 10], ecur[N + 10], que[M + 10], Hd, Tl;
bool vis[N + 10];

namespace Edges {
    struct Edge { int to, nxt, val, flow; } e[(M << 1) + 10];
    int hd[N + 10], etot = 1;
    #define adde(x, y, w, z) (e[++etot] = {y, hd[x], w, z}, hd[x] = etot)
} // namespace Edges
using namespace Edges;

bool bfs() {
    memset(d, 0x3f, (n + 1) * 4), Hd = 1, Tl = 0, que[++Tl] = s, d[s] = 0, ecur[s] = hd[s];
    while (Hd <= Tl) {
        int u = que[Hd++];
        vis[u] = 0;
        for (int i = hd[u]; i; i = e[i].nxt) {
            int v = e[i].to;
            if (e[i].flow == 0) continue;
            if (d[u] + e[i].val < d[v]) {
                d[v] = d[u] + e[i].val;
                if (vis[v] == 0) que[++Tl] = v, vis[v] = 1;
            }
        }
    }
    return d[t] != INF;
}

int dfs(int u, int lim) {
    if (u == t) return lim;
    int flow = 0; vis[u] = 1;
    for (int i = hd[u]; i && lim; i = e[i].nxt) {
        int v = e[i].to;
        if (vis[v] || e[i].flow == 0 || d[v] != d[u] + e[i].val) continue;
        int f = dfs(v, cmin(e[i].flow, lim));
        if (f == 0) d[v] = INF;
```

```
            e[i].flow -= f, e[i ^ 1].flow += f, flow += f, lim -= f, mincost += e[i].val * f;
        }
        return vis[u] = 0, flow;
    }

    void dinic() {
        int flow = 0;
        while (bfs()) maxflow += dfs(s, INF);
    }
```

复杂度上界是非多项式级别的，但是一般没人卡。我觉得卡了这个的出题人是啥比。

超神秘网络流优化

据说是我校传家宝。

将边反过来建，跑 bfs/SPFA 的时候从源点往汇点跑，dinic 从汇点往源点跑，有神秘的优化效果！

平面图最小割

平面图最小割 = 对偶图最短路。大家应该都懂这个结论吧。

有向图环覆盖

等价于每个点都有一个出度和一个入度。

把每个点拆成入点和出点。然后原图上的边 $a \to b$ 就变为 $a_{\text{out}} \to b_{\text{in}}$。

该二分图存在完美匹配则存在环覆盖，构造方案显然。

Dilworth 定理

最长反链 = 最小链覆盖。

DAG 最小不相交链覆盖

把每个点拆成入点和出点。然后原图上的边 $a \to b$ 就变为 $a_{\text{out}} \to b_{\text{in}}$。

最小不相交链覆盖即为 $n-$ 该二分图最大匹配。

DAG 最小相交链覆盖

Floyd 跑传递闭包，然后在把传递闭包看成一个新的 DAG，在这个 DAG 上求最小不相交链覆盖即可。

时间复杂度多个传递闭包的 $\Theta(\frac{n^3}{w})$。

```
const int N = 1e3, M = 1e6, INF = 0x7f7f7f7f;

int n, m, s, t, a[N + 10], d[N + 10], ecur[N + 10], que[N + 10], Hd, Tl;
bitset<505> f[505];

namespace Edges {
    struct Edge { int to, nxt, flow; } e[(M << 1) + 10];
    int hd[N + 10], etot = 1;
    void add(int x, int y, int z) { e[++etot] = {y, hd[x], z}, hd[x] = etot; }
    void adde(int x, int y, int z) { add(x, y, z), add(y, x, 0); }
} // namespace Edges
using namespace Edges;

bool bfs() {
    memset(d, -1, sizeof(d)), Hd = 1, Tl = 0, que[++Tl] = s, d[s] = 0, ecur[s] = hd[s];
    while (Hd <= Tl) {
        int u = que[Hd++];
        for (int i = hd[u]; i; i = e[i].nxt) {
            int v = e[i].to;
            if (d[v] != -1 || e[i].flow == 0) continue;
            que[++Tl] = v, d[v] = d[u] + 1, ecur[v] = hd[v];
            if (v == t) return 1;
```

```
        }
    }
    return 0;
}

int dfs(int u, int limit) {
    if (u == t) return limit;
    int flow = 0;
    for (int i = ecur[u]; i && flow < limit; i = e[i].nxt) {
        int v = e[i].to; ecur[u] = i;
        if (e[i].flow == 0 || d[v] != d[u] + 1) continue;
        int f = dfs(v, min(e[i].flow, limit - flow));
        if (f == 0) d[v] = -1;
        e[i].flow -= f, e[i ^ 1].flow += f, flow += f;
    }
    return flow;
}

int dinic() {
    int ans = 0, flow = 0;
    while (bfs()) while (flow = dfs(s, INF)) ans += flow;
    return ans;
}

signed main() {
    fr(n, m), s = 0, t = 2 * n + 1;
    for (int i = 1, u, v; i <= m; i++) fr(u, v), f[u][v] = 1;
    for (int i = 1; i <= n; i++) adde(s, i, 1), adde(i + n, t, 1);
    for (int i = 1; i <= n; i++) for (int j = 1; j <= n; j++) if (f[i][j]) f[i] |= f[j];
    for (int i = 1; i <= n; i++) for (int j = 1; j <= n; j++) if (f[i][j]) adde(i, j + n, 1);
    printf("%d\n", n - dinic());
    return 0;
}
```

最大权闭合子图

一个有向图中，每个点有点权 $w_i \in \mathbb{Z}$。称一个子图是闭合的，当且仅当满足：- 对于所有边 $u \to v$，点 u 在子图中则点 v 必在子图中；

求最大权值和的闭合子图。

经典模型。一定要记住：- 建超级源 S 和超级汇 T；- 对于所有 $w_i > 0$ 的点 i，连边 $S \to i$，流量为 w_i；- 对于所有 $w_i < 0$ 的点 i，连边 $i \to S$，流量为 $-w_i$；- 对于原图上的所有边 $u \to v$，连边 $u \to v$，流量为 $+\infty$。

新图跑完最小割后，与 S 相连的部分即为最大权值和的闭合子图。最大权值和为所有正点权之和 $-$ 最小割。

杂项

随机化

随机调整法

参考自《邓明扬一类调整算法在信息学竞赛中的应用》。

对一个初始解进行不断地调整从而获得较优解的方法。以 100% 的概率接受较优解，以一定的概率接受等优解。

```
const int N = 14;

int arr[(1 << N) + 10], a[(1 << N) + 10], p[(1 << N) + 10], g[(1 << N) + 10];

int tmp[(1 << N) + 10];

bool visval[(1 << N) + 10], vispos[(1 << N) + 10];

int main() {
    mt19937 rnd(chrono::system_clock::now().time_since_epoch().count());
    int n = fr(), xsum = 0, tot = 1 << n;
    for (int i = 0; i < (1 << n); i++) fr(a[i]), xsum ^= a[i], arr[i] = i;
    if (xsum)
        return puts("NO"), 0;
```

```
    for (int i = 0; i < (1 << n); i++) {
        int pos = rnd() % tot, arrpos = arr[pos];
        if (pos != tot - 1) arr[pos] = arr[tot - 1];
        tot--, tmp[0] = 0;
        bool flag = 0;
        for (int j = 0; j < (1 << n); j++)
            if (vispos[j] == 0 && visval[arrpos ^ a[j]] == 0) {
                flag = visval[arrpos ^ a[j]] = vispos[j] = 1;
                p[j] = arrpos;
                g[arrpos ^ a[j]] = j;
                break;
            } else if (vispos[j] == 0) tmp[++tmp[0]] = j;
        if (flag == 0) {
            int j = tmp[rand() % tmp[0] + 1];
            vispos[g[arrpos ^ a[j]]] = 0;
            arr[tot++] = p[g[arrpos ^ a[j]]];
            p[j] = arrpos;
            vispos[j] = 1;
            visval[p[j] ^ a[j]] = 1;
            g[p[j] ^ a[j]] = j;
            i--;
        }
    }
    for (int i = 0; i < (1 << n); i++) printf("%d ", p[i] ^ a[i]);
    puts("");
    for (int i = 0; i < (1 << n); i++) printf("%d ", p[i]);
    puts("");
    return 0;
}
```

模拟退火

随机调整法的进阶版本。整一个当前温度 T 和降温系数 α，不断降温至 $T < \epsilon$，每次降温都伴随着一次对解的调整更新。

以 100% 的概率接受较优解，以 $e^{-\frac{\Delta}{T}}$ 的概率接受较劣解。其中 Δ 为较劣解与当前解的差距。

注意调参。

```
const int N = 1e4;
const double alpha = 0.974, eps = 1e-7;

int n;
double ansx, ansy, ansd, x[N + 10], y[N + 10], w[N + 10];

double dist(double xa, double ya, double xb, double yb) { return sqrt((xa - xb) * (xa - xb) + (ya - yb) * (ya - yb)); }

double calc(double tx, double ty) {
    double res = 0;
    for (int i = 1; i <= n; i++) res += dist(x[i], y[i], tx, ty) * w[i];
    if (res < ansd) ansd = res, ansx = tx, ansy = ty;
    return res;
}

double rnd() { return (double)rand() / RAND_MAX; }

void SimulateAnneal() {
    double temp = 2e3, nowx = ansx, nowy = ansy;
    while (temp > eps) {
        double tx = nowx + temp * (rnd() * 2 - 1), ty = nowy + temp * (rnd() * 2 - 1);
        double delta = calc(tx, ty) - calc(nowx, nowy);
        if (delta < 0 || rnd() < exp(-delta / temp)) nowx = tx, nowy = ty;
        temp *= alpha;
    }
}

int main() {
    srand(999392699);
    fr(n);
    for (int i = 1; i <= n; i++) scanf("%lf%lf%lf", &x[i], &y[i], &w[i]), ansx += x[i], ansy += y[i];
    ansx /= n, ansy /= n, ansd = calc(ansx, ansy);
    for (int i = 1; i <= 5; i++) SimulateAnneal();
```

```
    for (int i = 1; i <= 100000; i++) {
        double tx = ansx + 0.001 * (rnd() * 2 - 1), ty = ansy + 0.001 * (rnd() * 2 - 1);
        calc(tx, ty);
    }
    return printf("%.3lf %.3lf\n", ansx, ansy), 0;
}
```

0/1 分数规划

问题：最大/小化下式：

$$\frac{\sum_{i=1}^{n} a_i \times w_i}{\sum_{i=1}^{n} b_i \times w_i}$$

其中 $w_i \in \{0, 1\}$。

显然答案满足单调性，二分一个答案 mid，check 其是否可行，以最大值为例：

$$\begin{aligned} &\frac{\sum a_i \times w_i}{\sum b_i \times w_i} > \text{mid} \\ \Longrightarrow &\sum a_i \times w_i - \text{mid} \times \sum b_i \cdot w_i > 0 \\ \Longrightarrow &\sum w_i \times (a_i - \text{mid} \times b_i) > 0 \end{aligned}$$

然后就是求 $\sum w_i \times (a_i - \text{mid} \times b_i)$，根据题意用你聪明的脑袋瓜子选用合适的算法解决即可。

$\Theta(k^2 \log n)$ 常系数齐次线性递推

快速记忆方法：一遍加法卷积卷 a, b，一遍减法卷积卷结果和递推式。最后把卷完 n 次的东西与初始值加权求和得到答案。

初始 $a_2 = w_1 = 1$。

```
void mul(const mint *a, const mint *b, mint *c) {
    static mint t[N + 10];
    for (int i = 1; i <= k * 2 - 1; i++) t[i] = 0;
    for (int i = 1; i <= k; i++) for (int j = 1; j <= k; j++) t[i + j - 1] += a[i] * b[j];
    for (int i = k * 2 - 1; i >= k + 1; i--) for (int j = 1; j <= k; j++) t[i - j] += t[i] * arr[j];
    for (int i = 1; i <= k; i++) c[i] = t[i];
}
mint calc(int n) {
    static mint a[N + 10], w[N + 10], ret; ret = 0, a[2] = w[1] = 1;
    for (; n; n >>= 1, mul(a, a, a)) if (n & 1) mul(a, w, w);
    for (int i = 1; i <= k; i++) ret += f[i] * w[i];
    return ret;
}
```

上述代码中 f 为数列的初始项，k 为递推阶数，arr 为递推系数，calc(n) 求的是数列第 n 项。

BM 算法

然而只有上面那个东西是没啥用的，配合 BM 才能打出超强力的效果!!!

BM 用来求一个数列的最短递推式。构造其实很简单，分三步走：1. 如果当前递推式已经契合，不需要更新；2. 如果不契合且当前递推式是第一个版本，直接给第二个版本扔进去 i 个 0；3. 如果不契合且不是第一个版本，令 f 为当前的 Δ 与上一个版本的 Δ 之商，fail_i 为版本 i 的失配位置。然后按如下构造当前递推式：- 往新版本递推式里扔 $i - \text{fail}_{c-1} - 1$ 个 0，再把 w 扔进去；- 把上一个版本的递推式全部乘上 $-w$ 然后扔进新版本递推式末尾；- 把当前版本递推式加到新版本递推式上；- 去除末尾的 0。

```
const int N = 1e4;

int m, k, fail[N + 10];
mint f[N + 10], d[N + 10], arr[N + 10];
vector<mint> g[N + 10];
```

```cpp

void solve() {
    int c = 0;
    for (int i = 1; i <= m; i++) {
        mint tmp = f[i];
        for (int j = 0; j < g[c].size(); j++) tmp -= g[c][j] * f[i - j - 1];
        d[i] = tmp;
        if (tmp == 0) continue;

        fail[c] = i;
        if (c == 0) { g[++c].resize(i); continue; }

        mint w = d[i] / d[fail[c - 1]];
        c++, g[c].resize(i - fail[c - 2] - 1), g[c].push_back(w);
        for (int j = 0; j < g[c - 2].size(); j++) g[c].push_back(-w * g[c - 2][j]);

        if (g[c].size() < g[c - 1].size()) g[c].resize(g[c - 1].size());
        for (int j = 0; j < g[c - 1].size(); j++) g[c][j] += g[c - 1][j];

        while (g[c].back() == 0) g[c].pop_back();
    }
    k = g[c].size();
    for (int i = 0; i < k; i++) arr[i + 1] = g[c][i], printf("%d%c", arr[i + 1].val(), " \n"[i + 1 == k]);
}
void mul(const mint *a, const mint *b, mint *c) {
    static mint t[N + 10];
    for (int i = 1; i <= k * 2 - 1; i++) t[i] = 0;
    for (int i = 1; i <= k; i++) for (int j = 1; j <= k; j++) t[i + j - 1] += a[i] * b[j];
    for (int i = k * 2 - 1; i >= k + 1; i--) for (int j = 1; j <= k; j++) t[i - j] += t[i] * arr[j];
    for (int i = 1; i <= k; i++) c[i] = t[i];
}
mint calc(int n) {
    static mint a[N + 10], w[N + 10], ret; ret = 0, a[2] = w[1] = 1;
    for (; n; n >>= 1, mul(a, a, a)) if (n & 1) mul(a, w, w);
    for (int i = 1; i <= k; i++) ret += f[i] * w[i];
    return ret;
}

signed main() {
    int n; fr(m, n);
    for (int i = 1; i <= m; i++) f[i] = fr();
    solve(), printf("%d\n", calc(n).val());
    return 0;
}
```

Tricks

1. 骨牌覆盖类问题优先考虑黑白染色。黑白染色完应当考虑一个强结论：黑白数量相同的图可以被完美覆盖。Source：CF1268B。
2. 满足 $\sum a \le k$ 的序列 $\langle a_1, a_2, \dots, a_t \rangle$ 的 $k + 1 - \sum a$ 之和就等价于：
   - 满足 $x + \sum a \le k + 1$ 的正整数序列 $\langle x, a_1, a_2, \dots, a_t \rangle$ 的数量，又等价于：
     - 满足 $x + y + \sum a = k + 2$ 的正整数序列 $\langle x, y, a_1, a_2, \dots, a_t \rangle$ 的数量。隔板法得到答案为 $\binom{k+1}{t+1}$。

   Source：ARC144D。
3. 图的最大匹配唯一等价于最大匹配就是完美匹配。Source：CF1032F。
4. 维护形如 $\sum_{i=l}^{r} i^k a_i$ 的式子时，可以用 k 次前缀和预处理，然后差分求出答案。Source：YZOJ 5592。
5. 当答案是指数级别的且指数与 n 相关时，可以对 n 使用欧拉定理，减小 n 的规模；当答案中还有非指数的与 n 相关的项，可以将 n 对 P 取模。二者兼有时需要将两个模数相乘。Source：YZOJ 5678。
6. $$\operatorname{lcm}(S) = \prod_{T \subseteq S} \gcd(T)^{(-1)^{|T|+1}}$$

证明用 min — max 反演。

7. bitset 优化字符串匹配。
8. 树上随机游走时，f_u 可以表示成 f_{fa_u} 的一次函数。Source：CF802L。
9. 要求选择的若干个值的平均值最大时，应全选最大值。Source：CF1088E。
10. 多手玩。多手玩。多手玩。多手玩。多手玩。多手玩。**手玩题意有利于理解题意，也有利于发现关键性质**。Source：CF1260E。
11. 扩展域并查集可以解决条件冲突类问题。Source：CF1713E。
12. 在博弈论中的两个状态 a, b，如果 a 能转移到 b，并且 a 能转移到所有 b 能转移到的状态，那么 a 是必胜态。证明分 b 是必胜态还是必败态讨论：

    - 若 b 是必胜态，则 b 一定能转移到一个必败态 c，而 a 能直接转移到 c；
    - 若 b 是必败态，则 a 可以转移到 b。

    因此 a 必然可以转移到一个必败态，则 a 是必胜态。Source：ARC137C。
13. $\binom{i}{j} \equiv 1 \pmod 2$ 当且仅当 $i \operatorname{and} j = j$，其中 and 为按位与运算。Source：CF1713F。
14. 对长度非 2^k 的序列做 FWT 时，可以不用将 n 补齐至 2^k，而是在 FWT 的循环条件中判断当前访问位置是否超界，可以证明这样做 FWT 仍然是正确的。

```
void OR(int *f) {
    for (int o = 2, k = 1; o <= m; o <<= 1, k <<= 1)
        for (int i = 0; i < m; i += o)
            for (int j = 0; j < k && i + j + k < n; j++) f[i + j + k] ^= f[i + j];
}
```

Source：CF1713F。