All-in at the River

Standard Code Library

Shanghai Jiao Tong University

Boren Tan
Zonghan Yang
Shangfei Yang

Regentropfen sind meine Tränen
Wind ist mein Atem und mein Erzählung
Zweige und Blätter sind meine Hände
denn mein Körper ist in Wurzeln gehüllt

wenn die Jahreszeit des Tauens kommt
werde ich wach und singe ein Lied
das Vergissmeinnicht, das du mir gegeben hast
ist hier

Contents

1. 数学

1.1 插值

1.1.1 牛顿插值

牛顿插值的原理是**二项式反演**.
二项式反演:

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

可以用e^x和e^{-x} 的麦克劳林展开式证明.
套用二项式反演的结论即可得到牛顿插值:

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

$$r_i=\sum_{j=0}^{i}(-1)^{i-j}\binom{i}{j}f(j)$$

其中k表示$f(n)$的最高次项系数.
实现时可以用k次差分替代右边的式子:

```
for (int i = 0; i <= k; i++)
    r[i] = f(i);
for (int j = 0; j < k; j++)
    for (int i = k; i > j; i--)
        r[i] -= r[i - 1];
```

注意到预处理r_i 的式子满足卷积形式,必要时可以用FFT优化至$O(k\log k)$ 预处理.

1.1.2 拉格朗日插值

$$f(x)=\sum_{i}f\left(x_i\right)\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

1.2 多项式

1.2.1 FFT

```
// 使用时一定要注意double的精度是否足够(极限大概是10 ^ 14)

const double pi = acos((double)-1.0);

// 手写复数类
// 支持加减乘三种运算
// += 运算符如果用的不多可以不重载
struct Complex {
    double a, b; // 由于long double精度和double几乎相同, 通常没有必要用long double

    Complex(double a = 0.0, double b = 0.0) : a(a), b(b) {}

    Complex operator + (const Complex &x) const {
        return Complex(a + x.a, b + x.b);
    }

    Complex operator - (const Complex &x) const {
        return Complex(a - x.a, b - x.b);
    }

    Complex operator * (const Complex &x) const {
        return Complex(a * x.a - b * x.b, a * x.b + b * x.a);
    }

    Complex &operator += (const Complex &x) {
        return *this = *this + x;
    }
} w[maxn], w_inv[maxn];

// FFT初始化 O(n)
// 需要调用sin, cos函数
void FFT_init(int n) {
    for (int i = 0; i < n; i++) // 根据单位根的旋转性质可以节省计算单位根逆元的时间
        w[i] = w_inv[n - i - 1] = Complex(cos(2 * pi / n * i), sin(2 * pi / n * i));
    // 当然不存单位根也可以, 只不过在FFT次数较多时很可能会增大常数
}

// FFT主过程 O(n\log n)
void FFT(Complex *A, int n, int tp) {
    for (int i = 1, j = 0, k; i < n - 1; i++) {
        k = n;
        do
            j ^= (k >>= 1);
        while (j < k);

        if (i < j)
            swap(A[i], A[j]);
    }

    for (int k = 2;k <= n; k *= 2)
        for (int i = 0; i < n; i += k)
            for (int j = 0; j < k * 2;j++) {
                Complex a = A[i + j], b = (tp > 0? w : w_inv)[n / k * j] * A[i + j + (k / 2)];

                A[i + j] = a + b;
                A[i + j + k / 2] = a - b;
            }

    if (tp < 0)
        for (int i = 0; i < n; i++)
            A[i].a /= n;
}
```

1.2.2 NTT

```
constexpr int p = 998244353, g = 3; // p为模数, g为p的任意一个原根

void NTT(int *A, int n, int tp) { // n为变换长度, tp为1或-1, 表示正/逆变换
    for (int i = 1, j = 0, k; i < n - 1; i++) { // O(n)旋转算法, 原理是模拟加1
        k = n;
        do
            j ^= (k >>= 1);
        while (j < k);

        if(i < j)
            swap(A[i], A[j]);
    }

    for (int k = 2; k <= n; k <<= 1) {
        int wn = qpow(g, (tp > 0 ? (p - 1) / k : (p - 1) / k * (long long)(p - 2) % (p - 1)));
        for (int i = 0; i < n; i += k) {
            int w = 1;
            for (int j = 0; j < (k >> 1); j++, w = (long long)w * wn % p){
```

```
                int a = A[i + j], b = (long long)w * A[i + j + (k >> 1)] % p;
                A[i + j] = (a + b) % p;
                A[i + j + (k >> 1)] = (a - b + p) % p;
            } // 更好的写法是预处理单位根的次幂
        }
    }

    if (tp < 0) {
        int inv = qpow(n, p - 2); // 如果能预处理逆元更好
        for (int i = 0; i < n; i++)
            A[i] = (long long)A[i] * inv % p;
    }
}
```

1.2.3 任意模数卷积

任意模数卷积有两种比较naive的做法，三模数NTT和拆系数FFT. 一般来说后者常数比前者小一些.
但卷积答案不超过10^{18}的时候可以改用双模数NTT，比FFT是要快的.

三模数NTT

原理是选取三个乘积大于结果的NTT模数，最后中国剩余定理合并.

```
//以下为三模数NTT，原理是选取三个乘积大于结果的NTT模数，最后中国剩余定理合并
//以对23333333(不是质数)取模为例
constexpr int maxn = 262200, Mod = 23333333, g = 3, m[] = {998244353, 1004535809, 1045430273}, m0_inv = 669690699, m1_inv = 332747959, M_inv = 942377029; // 这三个模数最小原根都是3
constexpr long long M = (long long)m[0] * m[1];

// 主函数(当然更多时候包装一下比较好)
// 用来卷积的是A和B
// 需要调用mul
int n, N = 1, A[maxn], B[maxn], C[maxn], D[maxn], ans[3][maxn];
int main() {
    scanf("%d", &n);

    while (N < n * 2)
        N *= 2;

    for (int i = 0; i < n; i++)
        scanf("%d", &A[i]);
    for (int i = 0; i < n; i++)
        scanf("%d", &B[i]);

    for (int i = 0; i < 3; i++)
        mul(m[i], ans[i]);

    for (int i = 0; i < n; i++)
        printf("%d ", China(ans[0][i], ans[1][i], ans[2][i]));

    return 0;
}

// mul O(n\log n)
// 包装了模NTT模数的卷积
// 需要调用NTT
void mul(int p, int *ans) {
    copy(A, A + N, C);
    copy(B, B + N, D);

    NTT(C, N, 1, p);
    NTT(D, N, 1, p);

    for (int i = 0; i < N; i++)
        ans[i] = (long long)C[i] * D[i] % p;

    NTT(ans, N, -1, p);
}

// 中国剩余定理 O(1)
// 由于直接合并会爆long long，采用神奇的方法合并
// 需要调用O(1)快速乘
inline int China(int a0, int a1, int a2) {
    long long A = (mul((long long)a0 * m1_inv, m[1], M) + mul((long long)a1 * m0_inv, m[0], M)) % M;
    int k = ((a2 - A) % m[2] + m[2]) % m[2] * M_inv % m[2];
    return (k % Mod * (M % Mod) % Mod + A % Mod) % Mod;
}
```

拆系数FFT

原理是选一个数M，把每一项改写成$aM + b$的形式再分别相乘.

```
constexpr int maxn = 262200, p = 23333333, M = 4830; // M取值要使得结果不超过10^14

// 需要开的数组
struct Complex {
    // 内容略
} w[maxn], w_inv[maxn], A[maxn], B[maxn], C[maxn], D[maxn], F[maxn], G[maxn], H[maxn];

// 主函数(当然更多时候包装一下比较好)
// 需要调用FFT初始化，FFT
int main() {
    scanf("%d", &n);

    int N = 1;
    while (N < n * 2)
        N *= 2;

    for (int i = 0, x; i < n; i++) {
        scanf("%d", &x);
        A[i] = x / M;
        B[i] = x % M;
    }

    for (int i = 0, x; i < n; i++) {
        scanf("%d", &x);
        C[i] = x / M;
        D[i] = x % M;
    }

    FFT_init(N);

    FFT(A, N, 1);
    FFT(B, N, 1);
    FFT(C, N, 1);
    FFT(D, N, 1);

    for (int i = 0; i < N; i++) {
        F[i] = A[i] * C[i];
        G[i] = A[i] * D[i] + B[i] * C[i];
        H[i] = B[i] * D[i];
    }

    FFT(F, N, -1);
    FFT(G, N, -1);
    FFT(H, N, -1);

    for (int i = 0; i < n; i++)
```

```
        printf("%d ", (int)((M * M * ((long long)(F[i].a + 0.5) % p) % p + M * ((long long)(G[i].a + 0.5) % p) % p + (long long)(H[i].a + 0.5) % p) % p));

    return 0;
}
```

1.2.4 多项式操作

```
// A为输入, C为输出, n为所需长度且必须是2^k
// 多项式求逆, 要求A常数项不为0
void get_inv(int *A, int *C, int n) {
    static int B[maxn];

    memset(C, 0, sizeof(int) * (n * 2));
    C[0] = qpow(A[0],p - 2); // 一般常数项都是1, 直接赋值为1就可以

    for (int k = 2; k <= n; k <<= 1) {
        memcpy(B, A, sizeof(int) * k);
        memset(B + k, 0, sizeof(int) * k);

        NTT(B, k * 2, 1);
        NTT(C,k * 2, 1);

        for (int i = 0; i < k * 2; i++) {
            C[i] = (2 - (long long)B[i] * C[i]) % p * C[i] % p;
            if (C[i] < 0)
                C[i] += p;
        }

        NTT(C, k * 2, -1);

        memset(C + k, 0, sizeof(int) * k);
    }
}

// 开根
void get_sqrt(int *A, int *C, int n) {
    static int B[maxn], D[maxn];

    memset(C, 0, sizeof(int) * (n * 2));
    C[0] = 1; // 如果不是1就要考虑二次剩余

    for (int k = 2; k <= n; k *= 2) {
        memcpy(B, A, sizeof(int) * k);
        memset(B + k, 0, sizeof(int) * k);

        get_inv(C, D, k);

        NTT(B, k * 2, 1);
        NTT(D, k * 2, 1);

        for (int i = 0; i < k * 2; i++)
            B[i] = (long long)B[i] * D[i]%p;

        NTT(B, k * 2, -1);

        for (int i = 0; i < k; i++)
            C[i] = (long long)(C[i] + B[i]) * inv_2 % p;//inv_2是2的逆元
    }
}

// 求导
void get_derivative(int *A, int *C, int n) {
    for (int i = 1; i < n; i++)
        C[i - 1] = (long long)A[i] * i % p;

    C[n - 1] = 0;
}

// 不定积分, 最好预处理逆元
void get_integrate(int *A, int *C, int n) {
    for (int i = 1; i < n; i++)
        C[i] = (long long)A[i - 1] * qpow(i, p - 2) % p;

    C[0] = 0; // 不定积分没有常数项
}

// 多项式ln, 要求A常数项不为0
void get_ln(int *A, int *C, int n) { // 通常情况下A常数项都是1
    static int B[maxn];

    get_derivative(A, B, n);
    memset(B + n, 0, sizeof(int) * n);

    get_inv(A, C, n);

    NTT(B, n * 2, 1);
    NTT(C, n * 2, 1);

    for (int i = 0;i < n * 2; i++)
        B[i] = (long long)B[i] * C[i] % p;

    NTT(B, n * 2, -1);

    get_integrate(B, C, n);

    memset(C+n,0,sizeof(int)*n);
}

// 多项式exp, 要求A没有常数项
// 常数很大且总代码较长, 一般来说最好替换为分治FFT
// 分治FFT依据: 设G(x) = exp F(x), 则有 g_i = \sum_{k=1}^{i-1} f_{i-k} * k * g_k
void get_exp(int *A, int *C, int n) {
    static int B[maxn];

    memset(C, 0, sizeof(int) * (n * 2));
    C[0] = 1;

    for (int k = 2; k <= n; k <<= 1) {
        get_ln(C, B, k);

        for (int i = 0; i < k; i++) {
            B[i] = A[i] - B[i];
            if (B[i] < 0)
                B[i] += p;
        }
        (++B[0]) %= p;

        NTT(B, k * 2, 1);
        NTT(C, k * 2, 1);

        for (int i = 0; i < k * 2; i++)
            C[i] = (long long)C[i] * B[i] % p;

        NTT(C, k * 2, -1);

        memset(C + k, 0, sizeof(int) * k);
    }
}

// 多项式k次幂, 在A常数项不为1时需要转化
```

```
// 常数较大且总代码较长，在时间要求不高时最好替换为暴力
  ↪ 快速幂
void get_pow(int *A, int *C, int n, int k) {
    static int B[maxn];

    get_ln(A, B, n);

    for (int i = 0; i < n; i++)
        B[i] = (long long)B[i] * k % p;

    get_exp(B, C, n);
}

// 多项式除法，A / B，结果输出在C
// A的次数为n，B的次数为m
void get_div(int *A, int *B, int *C, int n, int m) {
    static int f[maxn], g[maxn], gi[maxn];

    if (n < m) {
        memset(C, 0, sizeof(int) * m);
        return;
    }

    int N = 1;
    while (N < (n - m + 1))
        N <<= 1;

    memset(f, 0, sizeof(int) * N * 2);
    memset(g, 0, sizeof(int) * N * 2);
    // memset(gi, 0, sizeof(int) * N);

    for (int i = 0; i < n - m + 1; i++)
        f[i] = A[n - i - 1];
    for (int i = 0; i < m && i < n - m + 1; i++)
        g[i] = B[m - i - 1];

    get_inv(g, gi, N);

    for (int i = n - m + 1; i < N; i++)
        gi[i] = 0;

    NTT(f, N * 2, 1);
    NTT(gi, N * 2, 1);

    for (int i = 0; i < N * 2; i++)
        f[i] = (long long)f[i] * gi[i] % p;

    NTT(f, N * 2, -1);

    for (int i = 0; i < n - m + 1; i++)
        C[i] = f[n - m - i];
}

// 多项式取模，余数输出到C，商输出到D
void get_mod(int *A, int *B, int *C, int *D, int n, int
  ↪ m) {
    static int b[maxn], d[maxn];

    if (n < m) {
        memcpy(C, A, sizeof(int) * n);

        if (D)
            memset(D, 0, sizeof(int) * m);

        return;
    }

    get_div(A, B, d, n, m);

    if (D) { // D是商，可以选择不要
        for (int i = 0; i < n - m + 1; i++)
            D[i] = d[i];
    }

    int N = 1;
    while (N < n)
        N *= 2;

    memcpy(b, B, sizeof(int) * m);

    NTT(b, N, 1);
    NTT(d, N, 1);

    for (int i = 0; i < N; i++)
        b[i] = (long long)d[i] * b[i] % p;

    NTT(b, N, -1);

    for (int i = 0; i < m - 1; i++)
        C[i] = (A[i] - b[i] + p) % p;

    memset(b, 0, sizeof(int) * N);
    memset(d, 0, sizeof(int) * N);
}

// 多点求值要用的数组
int q[maxn], ans[maxn]; // q是要代入的各个系数，ans是求出
  ↪ 的值
int tg[25][maxn * 2], tf[25][maxn]; // 辅助数组，tg是预处
  ↪ 理乘积，
// tf是项数越来越少的f，tf[0]就是原来的函数

void pretreat(int l, int r, int k) { // 多点求值预处理
    static int A[maxn], B[maxn];

    int *g = tg[k] + l * 2;

    if (r - l + 1 <= 200) {
        g[0] = 1;

        for (int i = l; i <= r; i++) {
            for (int j = i - l + 1; j; j--) {
                g[j] = (g[j - 1] - (long long)g[j] *
                  ↪ q[i]) % p;
                if (g[j] < 0)
                    g[j] += p;
            }
            g[0] = (long long)g[0] * (p - q[i]) % p;
        }

        return;
    }

    int mid = (l + r) / 2;

    pretreat(l, mid, k + 1);
    pretreat(mid + 1, r, k + 1);

    if (!k)
        return;

    int N = 1;
    while (N <= r - l + 1)
        N *= 2;

    int *gl = tg[k + 1] + l * 2, *gr = tg[k + 1] + (mid +
      ↪ 1) * 2;

    memset(A, 0, sizeof(int) * N);
```

```
    memset(B, 0, sizeof(int) * N);

    memcpy(A, gl, sizeof(int) * (mid - l + 2));
    memcpy(B, gr, sizeof(int) * (r - mid + 1));

    NTT(A, N, 1);
    NTT(B, N, 1);

    for (int i = 0; i < N; i++)
        A[i] = (long long)A[i] * B[i] % p;

    NTT(A, N, -1);

    for (int i = 0; i <= r - l + 1; i++)
        g[i] = A[i];
}

void solve(int l, int r, int k) { // 多项式多点求值主过程
    int *f = tf[k];

    if (r - l + 1 <= 200) {
        for (int i = l; i <= r; i++) {
            int x = q[i];

            for (int j = r - l; ~j; j--)
                ans[i] = ((long long)ans[i] * x + f[j]) %
                    p;
        }

        return;
    }

    int mid = (l + r) / 2;
    int *ff = tf[k + 1], *gl = tg[k + 1] + l * 2, *gr =
        tg[k + 1] + (mid + 1) * 2;

    get_mod(f, gl, ff, NULL, r - l + 1, mid - l + 2);
    solve(l, mid, k + 1);

    memset(gl, 0, sizeof(int) * (mid - l + 2));
    memset(ff, 0, sizeof(int) * (mid - l + 1));

    get_mod(f, gr, ff, NULL, r - l + 1, r - mid + 1);
    solve(mid + 1, r, k + 1);

    memset(gr, 0, sizeof(int) * (r - mid + 1));
    memset(ff, 0, sizeof(int) * (r - mid));
}

// f < x^n, m个询问, 询问是0-based, 当然改成1-based也很简
    单
void get_value(int *f, int *x, int *a, int n, int m) {
    if (m <= n)
        m = n + 1;
    if (n < m - 1)
        n = m - 1; // 补零方便处理

    memcpy(tf[0], f, sizeof(int) * n);
    memcpy(q, x, sizeof(int) * m);

    pretreat(0, m - 1, 0);
    solve(0, m - 1, 0);

    if (a) // 如果a是NULL, 代表不复制答案, 直接用ans数组
        memcpy(a, ans, sizeof(int) * m);
}
```

1.2.5 更优秀的多项式多点求值

这个做法不需要写求逆和取模, 但是神乎其技, 完全搞不懂原理

清空和复制之类的地方容易抄错, 抄的时候要注意

```
int q[maxn], ans[maxn]; // q是要代入的各个系数, ans是求出
    的值
int tg[25][maxn * 2], tf[25][maxn]; // 辅助数组, tg是预处
    理乘积,
// tf是项数越来越少的f, tf[0]就是原来的函数

void pretreat(int l, int r, int k) { // 预处理
    static int A[maxn], B[maxn];

    int *g = tg[k] + l * 2;

    if (r - l + 1 <= 1) {
        g[0] = 1;

        for (int i = l; i <= r; i++) {
            for (int j = i - l + 1; j; j--) {
                g[j] = (g[j - 1] - (long long)g[j] *
                    q[i]) % p;
                if (g[j] < 0)
                    g[j] += p;
            }
            g[0] = (long long)g[0] * (p - q[i]) % p;
        }

        reverse(g, g + r - l + 2);

        return;
    }

    int mid = (l + r) / 2;

    pretreat(l, mid, k + 1);
    pretreat(mid + 1, r, k + 1);

    int N = 1;
    while (N <= r - l + 1)
        N *= 2;

    int *gl = tg[k + 1] + l * 2, *gr = tg[k + 1] + (mid +
        1) * 2;

    memset(A, 0, sizeof(int) * N);
    memset(B, 0, sizeof(int) * N);

    memcpy(A, gl, sizeof(int) * (mid - l + 2));
    memcpy(B, gr, sizeof(int) * (r - mid + 1));

    NTT(A, N, 1);
    NTT(B, N, 1);

    for (int i = 0; i < N; i++)
        A[i] = (long long)A[i] * B[i] % p;

    NTT(A, N, -1);

    for (int i = 0; i <= r - l + 1; i++)
        g[i] = A[i];
}

void solve(int l, int r, int k) { // 主过程
    static int a[maxn], b[maxn];

    int *f = tf[k];

    if (l == r) {
        ans[l] = f[0];
```

```cpp
        return;
    }

    int mid = (l + r) / 2;
    int *ff = tf[k + 1], *gl = tg[k + 1] + l * 2, *gr = tg[k + 1] + (mid + 1) * 2;

    int N = 1;
    while (N < r - l + 2)
        N *= 2;

    memcpy(a, f, sizeof(int) * (r - l + 2));
    memcpy(b, gr, sizeof(int) * (r - mid + 1));
    reverse(b, b + r - mid + 1);

    NTT(a, N, 1);
    NTT(b, N, 1);
    for (int i = 0; i < N; i++)
        b[i] = (long long)a[i] * b[i] % p;

    reverse(b + 1, b + N);
    NTT(b, N, 1);
    int n_inv = qpow(N, p - 2);
    for (int i = 0; i < N; i++)
        b[i] = (long long)b[i] * n_inv % p;

    for (int i = 0; i < mid - l + 2; i++)
        ff[i] = b[i + r - mid];

    memset(a, 0, sizeof(int) * N);
    memset(b, 0, sizeof(int) * N);

    solve(l, mid, k + 1);

    memset(ff, 0, sizeof(int) * (mid - l + 2));

    memcpy(a, f, sizeof(int) * (r - l + 2));
    memcpy(b, gl, sizeof(int) * (mid - l + 2));
    reverse(b, b + mid - l + 2);

    NTT(a, N, 1);
    NTT(b, N, 1);
    for (int i = 0; i < N; i++)
        b[i] = (long long)a[i] * b[i] % p;

    reverse(b + 1, b + N);
    NTT(b, N, 1);
    for (int i = 0; i < N; i++)
        b[i] = (long long)b[i] * n_inv % p;

    for (int i = 0; i < r - mid + 1; i++)
        ff[i] = b[i + mid - l + 1];

    memset(a, 0, sizeof(int) * N);
    memset(b, 0, sizeof(int) * N);

    solve(mid + 1, r, k + 1);

    memset(gl, 0, sizeof(int) * (mid - l + 2));
    memset(gr, 0, sizeof(int) * (r - mid + 1));
    memset(ff, 0, sizeof(int) * (r - mid + 1));
}

// f < x^n, m个询问, 0-based
void get_value(int *f, int *x, int *a, int n, int m) {
    static int c[maxn], d[maxn];

    if (m <= n)
        m = n + 1;
    if (n < m - 1)
        n = m - 1; // 补零

    memcpy(q, x, sizeof(int) * m);

    pretreat(0, m - 1, 0);

    int N = 1;
    while (N < m)
        N *= 2;

    get_inv(tg[0], c, N);

    fill(c + m, c + N, 0);
    reverse(c, c + m);

    memcpy(d, f, sizeof(int) * m);

    NTT(c, N * 2, 1);
    NTT(d, N * 2, 1);
    for (int i = 0; i < N * 2; i++)
        c[i] = (long long)c[i] * d[i] % p;
    NTT(c, N * 2, -1);

    for (int i = 0; i < m; i++)
        tf[0][i] = c[i + n];

    solve(0, m - 1, 0);

    if (a) // 如果a是NULL, 代表不复制答案, 直接用ans数组
        memcpy(a, ans, sizeof(int) * m);
}
```

1.2.6 拉格朗日反演

如果$f(x)$与$g(x)$互为复合逆 则有

$$[x^n]g(x) = \frac{1}{n}[x^{n-1}]\left(\frac{x}{f(x)}\right)^n$$

$$[x^n]h(g(x)) = \frac{1}{n}[x^{n-1}]h'(x)\left(\frac{x}{f(x)}\right)^n$$

1.2.7 半在线卷积

```cpp
void solve(int l, int r) {
    if (r <= m)
        return;

    if (r - l == 1) {
        if (l == m)
            f[l] = a[m];
        else
            f[l] = (long long)f[l] * inv[l - m] % p;

        for (int i = l, t = (long long)l * f[l] % p; i <= n; i += l)
            g[i] = (g[i] + t) % p;

        return;
    }

    int mid = (l + r) / 2;

    solve(l, mid);

    if (l == 0) {
        for (int i = 1; i < mid; i++) {
            A[i] = f[i];
            B[i] = (c[i] + g[i]) % p;
```

```cpp
        }
        NTT(A, r, 1);
        NTT(B, r, 1);
        for (int i = 0; i < r; i++)
            A[i] = (long long)A[i] * B[i] % p;
        NTT(A, r, -1);

        for (int i = mid; i < r; i++)
            f[i] = (f[i] + A[i]) % p;
    }
    else {
        for (int i = 0; i < r - l; i++)
            A[i] = f[i];
        for (int i = l; i < mid; i++)
            B[i - l] = (c[i] + g[i]) % p;
        NTT(A, r - l, 1);
        NTT(B, r - l, 1);
        for (int i = 0; i < r - l; i++)
            A[i] = (long long)A[i] * B[i] %p;
        NTT(A, r - l, -1);

        for (int i = mid; i < r; i++)
            f[i] = (f[i] + A[i - l]) % p;

        memset(A, 0, sizeof(int) * (r - l));
        memset(B, 0, sizeof(int) * (r - l));

        for (int i = l; i < mid; i++)
            A[i - l] = f[i];
        for (int i = 0; i < r - l; i++)
            B[i] = (c[i] + g[i]) % p;
        NTT(A, r - l, 1);
        NTT(B, r - l, 1);
        for (int i = 0; i < r - l; i++)
            A[i] = (long long)A[i] * B[i] % p;
        NTT(A, r - l, -1);

        for (int i = mid; i < r; i++)
            f[i] = (f[i] + A[i - l]) % p;
    }

    memset(A, 0, sizeof(int) * (r - l));
    memset(B, 0, sizeof(int) * (r - l));

    solve(mid, r);
}
```

1.2.8 常系数齐次线性递推$O(k \log k \log n)$

如果只有一次这个操作可以像代码里一样加上一个只求一次逆的优化，否则就乖乖每次做完整的除法和取模

```cpp
// 多项式取模，余数输出到C，商输出到D
void get_mod(int *A, int *B, int *C, int *D, int n, int
    ↪ m) {
    static int b[maxn], d[maxn];
    static bool flag = false;

    if (n < m) {
        memcpy(C, A, sizeof(int) * n);

        if (D)
            memset(D, 0, sizeof(int) * m);

        return;
    }

    get_div(A, B, d, n, m);

    if (D) { // D是商，可以选择不要
        for (int i = 0; i < n - m + 1; i++)
            D[i] = d[i];
    }

    int N = 1;
    while (N < n)
        N *= 2;

    if (!flag) {
        memcpy(b, B, sizeof(int) * m);
        NTT(b, N, 1);

        flag = true;
    }

    NTT(d, N, 1);

    for (int i = 0; i < N; i++)
        d[i] = (long long)d[i] * b[i] % p;

    NTT(d, N, -1);

    for (int i = 0; i < m - 1; i++)
        C[i] = (A[i] - d[i] + p) % p;

    // memset(b, 0, sizeof(int) * N);
    memset(d, 0, sizeof(int) * N);
}

// g < x^n，f是输出答案的数组
void pow_mod(long long k, int *g, int n, int *f) {
    static int a[maxn], t[maxn];

    memset(f, 0, sizeof(int) * (n * 2));

    f[0] = a[1] = 1;

    int N = 1;
    while (N < n * 2 - 1)
        N *= 2;

    while (k) {
        NTT(a, N, 1);

        if (k & 1) {
            memcpy(t, f, sizeof(int) * N);

            NTT(t, N, 1);
            for (int i = 0; i < N; i++)
                t[i] = (long long)t[i] * a[i] % p;
            NTT(t, N, -1);

            get_mod(t, g, f, NULL, n * 2 - 1, n);
        }

        for (int i = 0; i < N; i++)
            a[i] = (long long)a[i] * a[i] % p;
        NTT(a, N, -1);

        memcpy(t, a, sizeof(int) * (n * 2 - 1));
        get_mod(t, g, a, NULL, n * 2 - 1, n);
        fill(a + n - 1, a + N, 0);

        k >>= 1;
    }

    memset(a, 0, sizeof(int) * (n * 2));
}
```

```
// f_n = \sum_{i = 1} ^ m  f_{n - i} a_i
// f是0~m-1项的初值
int linear_recurrence(long long n, int m, int *f, int *a) {
    static int g[maxn], c[maxn];

    memset(g, 0, sizeof(int) * (m * 2 + 1));

    for (int i = 0; i < m; i++)
        g[i] = (p - a[m - i]) % p;
    g[m] = 1;

    pow_mod(n, g, m + 1, c);

    int ans = 0;
    for (int i = 0; i < m; i++)
        ans = (ans + (long long)c[i] * f[i]) % p;

    return ans;
}
```

1.3 FWT快速沃尔什变换

```
// 注意FWT常数比较小，这点与FFT/NTT不同
// 以下代码均以模质数情况为例，其中n为变换长度，tp表示正/逆变换

// 按位或版本
void FWT_or(int *A, int n, int tp) {
    for (int k = 2; k <= n; k *= 2)
        for (int i = 0; i < n; i += k)
            for (int j = 0; j < k / 2; j++) {
                if (tp > 0)
                    A[i + j + k / 2] = (A[i + j + k / 2] + A[i + j]) % p;
                else
                    A[i + j + k / 2] = (A[i + j + k / 2] - A[i + j] + p)%p;
            }
}

// 按位与版本
void FWT_and(int *A, int n, int tp) {
    for (int k = 2; k <= n; k *= 2)
        for (int i = 0; i < n; i += k)
            for (int j = 0; j < k / 2; j++) {
                if (tp > 0)
                    A[i + j] = (A[i + j] + A[i + j + k / 2]) % p;
                else
                    A[i + j] = (A[i + j] - A[i + j + k / 2] + p) % p;
            }
}

// 按位异或版本
void FWT_xor(int *A, int n, int tp) {
    for (int k = 2; k <= n; k *= 2)
        for (int i = 0; i < n; i += k)
            for (int j = 0; j < k / 2; j++) {
                int a = A[i + j], b = A[i + j + k / 2];
                A[i + j] = (a + b) % p;
                A[i + j + k / 2] = (a - b + p) % p;
            }

    if (tp < 0) {
        int inv = qpow(n % p, p - 2); // n的逆元，在不取模时需要用每层除以2代替
        for (int i = 0; i < n; i++)
            A[i] = A[i] * inv % p;
    }
}
```

1.4 单纯形

```
const double eps = 1e-10;

double A[maxn][maxn], x[maxn];
int n, m, t, id[maxn * 2];

// 方便起见，这里附上主函数
int main() {
    scanf("%d%d%d", &n, &m, &t);

    for (int i = 1; i <= n; i++) {
        scanf("%lf", &A[0][i]);
        id[i] = i;
    }

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++)
            scanf("%lf", &A[i][j]);

        scanf("%lf", &A[i][0]);
    }

    if (!initalize())
        printf("Infeasible"); // 无解
    else if (!simplex())
        printf("Unbounded"); // 最优解无限大

    else {
        printf("%.15lf\n", -A[0][0]);
        if (t) {
            for (int i = 1; i <= m; i++)
                x[id[i + n]] = A[i][0];
            for (int i = 1; i <= n; i++)
                printf("%.15lf ",x[i]);
        }
    }
    return 0;
}

//初始化
//对于初始解可行的问题，可以把初始化省略掉
bool initalize() {
    while (true) {
        double t = 0.0;
        int l = 0, e = 0;

        for (int i = 1; i <= m; i++)
            if (A[i][0] + eps < t) {
                t = A[i][0];
                l = i;
            }

        if (!l)
            return true;

        for (int i = 1; i <= n; i++)
            if (A[l][i] < -eps && (!e || id[i] < id[e]))
                e = i;

        if (!e)
            return false;

```

```
        pivot(l, e);
    }
}

//求解
bool simplex() {
    while (true) {
        int l = 0, e = 0;
        for (int i = 1; i <= n; i++)
            if (A[0][i] > eps && (!e || id[i] < id[e]))
                e = i;

        if (!e)
            return true;

        double t = 1e50;
        for (int i = 1; i <= m; i++)
            if (A[i][e] > eps && A[i][0] / A[i][e] < t) {
                l = i;
                t = A[i][0]/A[i][e];
            }

        if (!l)
            return false;

        pivot(l, e);
    }
}

//转轴操作,本质是在凸包上沿着一条棱移动
void pivot(int l, int e) {
    swap(id[e], id[n + l]);
    double t = A[l][e];
    A[l][e] = 1.0;

    for (int i = 0; i <= n; i++)
        A[l][i] /= t;

    for (int i = 0; i <= m; i++)
        if (i != l) {
            t = A[i][e];
            A[i][e] = 0.0;
            for (int j = 0; j <= n; j++)
                A[i][j] -= t * A[l][j];
        }
}
```

1.5 线性代数

1.5.1 行列式取模

```
int p;

int Gauss(int A[maxn][maxn], int n) {
    int det = 1;

    for (int i = 1; i <= n; i++) {
        for (int j = i + 1; j <= n; j++)
            while (A[j][i]) {
                int t = (p - A[i][i] / A[j][i]) % p;
                for (int k = i; k <= n; k++)
                    A[i][k] = (A[i][k] + (long long)A[j][k] * t) % p;

                swap(A[i], A[j]);
                det = (p - det) % p; // 交换一次之后行列式取负
            }

            if (!A[i][i])
                return 0;

            det = (long long)det * A[i][i] % p;
    }

    return det;
}
```

1.5.2 线性基

```
void add(unsigned long long x) {
    for (int i = 63; i >= 0; i--)
        if (x >> i & 1) {
            if (b[i])
                x ^= b[i];
            else {
                b[i] = x;

                for (int j = i - 1; j >= 0; j--)
                    if (b[j] && (b[i] >> j & 1))
                        b[i] ^= b[j];

                for (int j = i + 1; j < 64; j++)
                    if (b[j] >> i & 1)
                        b[j] ^= b[i];

                break;
            }
        }
}
```

1.5.3 线性代数知识

行列式:

$$\det A = \sum_{\sigma} \operatorname{sgn}(\sigma) \prod_{i} a_{i,\sigma_i}$$

逆矩阵:

$$B = A^{-1} \iff AB = 1$$

代数余子式:

$$M_{i,j} = (-1)^{(i+j)} detA - \{i,j\}$$

也就是A去掉一行一列之后的行列式

同时我们有

$$M = \frac{A^{-1}}{\det A}$$

1.5.4 矩阵树定理

1.6 常见数列

1.6.1 伯努利数

$$B(x) = \sum_{i \ge 0} \frac{B_i x^i}{i!} = \frac{x}{e^x - 1}$$

$$B_n = [n = 0] - \sum_{i=0}^{n-1} \binom{n}{i} \frac{B_i}{n - k + 1}$$

$$\sum_{i=0}^{n} \binom{n+1}{i} B_i = 0$$

$$S_n(m) = \sum_{i=0}^{m-1} i^n = \sum_{i=0}^{n} \binom{n}{i} B_{n-i} \frac{m^{i+1}}{i+1}$$

1.6.2 分拆数

```cpp
int b = sqrt(n);
ans[0] = tmp[0] = 1;

for (int i = 1; i <= b; ++i) {
    for (int rep = 0; rep < 2; ++rep)
        for (int j = i; j <= n - i * i; ++j)
            add(tmp[j], tmp[j - i]);

    for (int j = i * i; j <= n; ++j)
        add(ans[j], tmp[j - i * i]);
}
```

1.6.3 斯特林数

第一类斯特林数

$\begin{bmatrix} n \\ k \end{bmatrix}$表示$n$个元素划分成$k$个轮换的方案数.

求同一行：分治FFT $O(n\log^2 n)$

求同一列：用一个轮换的指数生成函数做k次幂

$$\sum_{n=0}^{\infty} \begin{bmatrix} n \\ k \end{bmatrix} \frac{x^n}{n!} = \frac{(\ln(1-x))^k}{k!}$$

第二类斯特林数

$\begin{Bmatrix} n \\ k \end{Bmatrix}$表示$n$个元素划分成$k$个子集的方案数.

求一个：容斥，狗都会做

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n$$

求同一行：FFT，狗都会做

求同一列：指数生成函数

$$\sum_{n=0}^{\infty} \begin{Bmatrix} n \\ k \end{Bmatrix} \frac{x^n}{n!} = \frac{(e^x-1)^k}{k!}$$

普通生成函数

$$\sum_{n=0}^{\infty} \begin{Bmatrix} n \\ k \end{Bmatrix} x^n = x^k \left(\prod_{i=1}^{k} (1-ix) \right)^{-1}$$

2. 数论

2.1 $O(n)$预处理逆元

```cpp
// 要求p为质数

inv[0] = inv[1] = 1;
for (int i = 2; i <= n; i++)
    inv[i] = (long long)(p - (p / i)) * inv[p % i] % p;
        // p为模数
// i ^ -1 = -(p / i) * (p % i) ^ -1
```

2.2 杜教筛

```cpp
// 用于求可以用狄利克雷卷积构造出好求和的东西的函数的前
    缀和(有点绕)
// 有些题只要求n <= 10 ^ 9, 这时就没必要开long long了, 但
    记得乘法时强转

//常量/全局变量/数组定义
const int maxn = 5000005, table_size = 5000000, p =
    1000000007, inv_2 = (p + 1) / 2;
bool notp[maxn];
int prime[maxn / 20], phi[maxn], tbl[100005];
// tbl用来顶替哈希表, 其实开到n ^ {1 / 3}就够了, 不过保
    险起见开成\sqrt n比较好
long long N;

// 主函数前面加上这么一句
memset(tbl, -1, sizeof(tbl));

// 线性筛预处理部分略去

// 杜教筛主过程 总计O(n ^ {2 / 3})
// 递归调用自身
// 递推式还需具体情况具体分析,这里以求欧拉函数前缀和(mod
    10 ^ 9 + 7)为例
int S(long long n) {
    if (n <= table_size)
        return phi[n];
    else if (~tbl[N / n])
        return tbl[N / n];
    // 原理: n除以所有可能的数的结果一定互不相同

    int ans = 0;
    for (long long i = 2, last; i <= n; i = last + 1) {
        last = n / (n / i);
        ans = (ans + (last - i + 1) % p * S(n / i)) % p;
            // 如果n是int范围的话记得强转
    }

    ans = (n % p * ((n + 1) % p) % p * inv_2 - ans + p) %
        p; // 同上
    return tbl[N / n] = ans;
}
```

2.3 线性筛

```cpp
// 此代码以计算约数之和函数\sigma_1(对10^9+7取模)为例
// 适用于任何f(p^k)便于计算的积性函数
constexpr int p = 1000000007;

int prime[maxn / 10], sigma_one[maxn], f[maxn], g[maxn];
// f: 除掉最小质因子后剩下的部分
// g: 最小质因子的幂次, 在f(p^k)比较复杂时很有用,
    但f(p^k)可以递推时就可以省略了
// 这里没有记录最小质因子, 但根据线性筛的性质, 每个合数
    只会被它最小的质因子筛掉
bool notp[maxn]; // 顾名思义

void get_table(int n) {
    sigma_one[1] = 1; // 积性函数必有f(1) = 1
    for (int i = 2; i <= n; i++) {
        if (!notp[i]) { // 质数情况
            prime[++prime[0]] = i;
            sigma_one[i] = i + 1;
            f[i] = g[i] = 1;
        }

        for (int j = 1; j <= prime[0] && i * prime[j] <=
            n; j++) {
            notp[i * prime[j]] = true;

            if (i % prime[j]) { // 加入一个新的质因子, 这
                种情况很简单
                sigma_one[i * prime[j]] = (long
                    long)sigma_one[i] * (prime[j] + 1) %
                    p;
                f[i * prime[j]] = i;
                g[i * prime[j]] = 1;
            }
```

```
            else { // 再加入一次最小质因子,需要再进行分类讨论
                f[i * prime[j]] = f[i];
                g[i * prime[j]] = g[i] + 1;
                // 对于f(p^k)可以直接递推的函数, 这里的判断可以改成
                // i / prime[j] % prime[j] != 0, 这样可以省下f[]的空间,
                // 但常数很可能会稍大一些

                if (f[i] == 1) // 质数的幂次, 这里\sigma_1可以递推
                    sigma_one[i * prime[j]] = (sigma_one[i] + i * prime[j]) % p;
                    // 对于更一般的情况,可以借助g[]计算f(p^k)
                else sigma_one[i * prime[j]] = // 否则直接利用积性, 两半乘起来
                    (long long)sigma_one[i * prime[j] / f[i]] * sigma_one[f[i]] % p;
                break;
            }
        }
    }
}
```

2.4 Miller-Rabin

```
// 复杂度可以认为是常数

// 封装好的函数体
// 需要调用check
bool Miller_Rabin(long long n) {
    if (n == 1)
        return false;
    if (n == 2)
        return true;
    if (n % 2 == 0)
        return false;

    for (int i : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (i > n)
            break;
        if (!check(n, i))
            return false;
    }

    return true;
}

// 用一个数检测
// 需要调用long long快速幂和O(1)快速乘
bool check(long long n, long long b) { // b: base
    long long a = n - 1;
    int k = 0;

    while (a % 2 == 0) {
        a /= 2;
        k++;
    }

    long long t = qpow(b, a, n); // 这里的快速幂函数需要写O(1)快速乘
    if (t == 1 || t == n - 1)
        return true;

    while (k--) {
        t = mul(t, t, n); // mul是O(1)快速乘函数
        if(t == n - 1)
            return true;
    }

    return false;
}
```

2.5 Pollard's Rho

```
// 注意,虽然Pollard's Rho的理论复杂度是O(n ^ {1 / 4})的,
// 但实际跑起来比较慢, 一般用于做long long范围内的质因数分解


// 封装好的函数体
// 需要调用solve
void factorize(long long n, vector<long long> &v) { // v用于存分解出来的质因子, 重复的会放多个
    for (int i : {2, 3, 5, 7, 11, 13, 17, 19})
        while (n % i == 0) {
            v.push_back(i);
            n /= i;
        }

    solve(n, v);
    sort(v.begin(), v.end()); // 从小到大排序后返回
}

// 递归过程
// 需要调用Pollard's Rho主过程, 同时递归调用自身
void solve(long long n, vector<long long> &v) {
    if (n == 1)
        return;

    long long p;
    do
        p = Pollards_Rho(n);
    while (!p); // p是任意一个非平凡因子

    if (p == n) {
        v.push_back(p); // 说明n本身就是质数
        return;
    }

    solve(p, v); // 递归分解两半
    solve(n / p, v);
}

// Pollard's Rho主过程
// 需要使用Miller-Rabin作为子算法
// 同时需要调用O(1)快速乘和gcd函数
long long Pollards_Rho(long long n) {
    // assert(n > 1);

    if (Miller_Rabin(n))
        return n;

    long long c = rand() % (n - 2) + 1, i = 1, k = 2, x = rand() % (n - 3) + 2, u = 2; // 注意这里rand函数需要重定义一下
    while (true) {
        i++;
        x = (mul(x, x, n) + c) % n; // mul是O(1)快速乘函数

        long long g = gcd((u - x + n) % n, n);
        if (g > 1 && g < n)
            return g;
```

```
        if (u == x)
            return 0; // 失败, 需要重新调用

        if (i == k) {
            u = x;
            k *= 2;
        }
    }
}
```

2.6 常用公式

2.6.1 莫比乌斯反演

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

$$f(d)=\sum_{d|k}g(k)\Leftrightarrow g(d)=\sum_{d|k}\mu\left(\frac{k}{d}\right)f(k)$$

2.6.2 其他常用公式

$$\mu*I=e\quad(e(n)=[n=1])$$

$$\varphi*I=id$$

$$\mu*id=\varphi$$

$$\sigma_0=I*I,\ sigma_1=id*I,\ sigma_k=id^{k-1}*I$$

$$\sum_{i=1}^{n}[(i,n)=1]\,i=n\frac{\varphi(n)+e(n)}{2}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{i}[(i,j)=d]=S_\varphi\left(\left\lfloor\frac{n}{d}\right\rfloor\right)$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m}[(i,j)=d]=\sum_{d|k}\mu\left(\frac{k}{d}\right)\left\lfloor\frac{n}{k}\right\rfloor\left\lfloor\frac{m}{k}\right\rfloor$$

3. 图论

3.1 最小生成树

3.1.1 Boruvka算法

思想: 每次选择连接每个连通块的最小边, 把连通块缩起来.
每次连通块个数至少减半, 所以迭代$O(\log n)$次即可得到最小生成树.
一种比较简单的实现方法: 每次迭代遍历所有边, 用并查集维护连通性和每个连通块的最小边权.
应用: 最小异或生成树

3.1.2 动态最小生成树

```
// 动态最小生成树的离线算法比较容易,而在线算法通常极为复杂
// 一个跑得比较快的离线做法是对时间分治,在每层分治时找出一定在/不在MST上的边,只带着不确定边继续递归
// 简单起见,找确定边的过程用Kruskal算法实现,过程中的两种重要操作如下:
// - Reduction:待修改边标为+INF,跑MST后把非树边删掉,减少无用边
// - Contraction:待修改边标为-INF,跑MST后缩除待修改边之外的所有MST边,计算必须边
// 每轮分治需要Reduction-Contraction,借此减少不确定边,从而保证复杂度
// 复杂度证明:假设当前区间有k条待修改边,n和m表示点数和边数,那么最坏情况下R-C的效果为(n, m) -> (n, n + k - 1) -> (k + 1, 2k)


// 全局结构体与数组定义
struct edge { //边的定义
    int u, v, w, id; // id表示边在原图中的编号
    bool vis; // 在Kruskal时用,记录这条边是否是树边
    bool operator < (const edge &e) const { return w < e.w; }
} e[20][maxn], t[maxn]; // 为了便于回滚,在每层分治存一个副本


// 用于存储修改的结构体,表示第id条边的权值从u修改为v
struct A {
    int id, u, v;
} a[maxn];


int id[20][maxn]; // 每条边在当前图中的编号
int p[maxn], size[maxn], stk[maxn], top; // p和size是并查集数组,stk是用来撤销的栈
int n, m, q; // 点数,边数,修改数


// 方便起见,附上可能需要用到的预处理代码
for (int i = 1; i <= n; i++) { // 并查集初始化
    p[i] = i;
    size[i] = 1;
}

for (int i = 1; i <= m; i++) { // 读入与预标号
    scanf("%d%d%d", &e[0][i].u, &e[0][i].v, &e[0][i].w);
    e[0][i].id = i;
    id[0][i] = i;
}

for (int i = 1; i <= q; i++) { // 预处理出调用数组
    scanf("%d%d", &a[i].id, &a[i].v);
    a[i].u = e[0][a[i].id].w;
    e[0][a[i].id].w = a[i].v;
}

for(int i = q; i; i--)
    e[0][a[i].id].w = a[i].u;

CDQ(1, q, 0, m, 0); // 这是调用方法


// 分治主过程 O(nlog^2n)
// 需要调用Reduction和Contraction
void CDQ(int l, int r, int d, int m, long long ans) { // CDQ分治
    if (l == r) { // 区间长度已减小到1,输出答案,退出
        e[d][id[d][a[l].id]].w = a[l].v;
        printf("%lld\n", ans + Kruskal(m, e[d]));
        e[d][id[d][a[l].id]].w=a[l].u;
        return;
    }

    int tmp = top;

    Reduction(l, r, d, m);
    ans += Contraction(l, r, d, m); // R-C

    int mid = (l + r) / 2;

    copy(e[d] + 1, e[d] + m + 1, e[d + 1] + 1);
    for (int i = 1; i <= m; i++)
```

```
        id[d + 1][e[d][i].id] = i; // 准备好下一层要用的
          ↪ 数组

    CDQ(l, mid, d + 1, m, ans);

    for (int i = l; i <= mid; i++)
        e[d][id[d][a[i].id]].w = a[i].v; // 进行左边的修
          ↪ 改

    copy(e[d] + 1, e[d] + m + 1, e[d + 1] + 1);
    for (int i = 1; i <= m; i++)
        id[d + 1][e[d][i].id] = i; // 重新准备下一层要用
          ↪ 的数组

    CDQ(mid + 1, r, d + 1, m, ans);

    for (int i = top; i > tmp; i--)
        cut(stk[i]);//撤销所有操作
    top = tmp;
}


// Reduction(减少无用边):待修改边标为+INF,跑MST后把非树
  ↪ 边删掉,减少无用边
// 需要调用Kruskal
void Reduction(int l, int r, int d, int &m) {
    for (int i = l; i <= r; i++)
        e[d][id[d][a[i].id]].w = INF;//待修改的边标为INF

    Kruskal(m, e[d]);

    copy(e[d] + 1, e[d] + m + 1, t + 1);

    int cnt = 0;
    for (int i = 1; i <= m; i++)
        if (t[i].w == INF || t[i].vis){ // 非树边扔掉
            id[d][t[i].id] = ++cnt; // 给边重新编号
            e[d][cnt] = t[i];
        }

    for (int i = r; i >= l; i--)
        e[d][id[d][a[i].id]].w = a[i].u; // 把待修改的边
          ↪ 改回去

    m=cnt;
}


// Contraction(缩必须边):待修改边标为-INF,跑MST后缩除待
  ↪ 修改边之外的所有树边
// 返回缩掉的边的总权值
// 需要调用Kruskal
long long Contraction(int l, int r, int d, int &m) {
    long long ans = 0;

    for (int i = l; i <= r; i++)
        e[d][id[d][a[i].id]].w = -INF; // 待修改边标
          ↪ 为-INF

    Kruskal(m, e[d]);
    copy(e[d] + 1, e[d] + m + 1, t + 1);

    int cnt = 0;
    for (int i = 1; i <= m ; i++) {

        if (t[i].w != -INF && t[i].vis) { // 必须边
            ans += t[i].w;
            mergeset(t[i].u, t[i].v);
        }
        else { // 不确定边
            id[d][t[i].id]=++cnt;
            e[d][cnt]=t[i];
        }
    }

    for (int i = r ; i >= l; i--) {
        e[d][id[d][a[i].id]].w = a[i].u; // 把待修改的边
          ↪ 改回去
        e[d][id[d][a[i].id]].vis = false;
    }

    m = cnt;

    return ans;
}


// Kruskal算法 O(mlogn)
// 方便起见,这里直接沿用进行过缩点的并查集,在过程结束后
  ↪ 撤销即可
long long Kruskal(int m, edge *e) {
    int tmp = top;
    long long ans = 0;

    sort(e + 1, e + m + 1); // 比较函数在结构体中定义过了

    for (int i = 1; i <= m; i++) {
        if (findroot(e[i].u) != findroot(e[i].v)) {
            e[i].vis = true;
            ans += e[i].w;
            mergeset(e[i].u, e[i].v);
        }
        else
            e[i].vis = false;
    }

    for(int i = top; i > tmp; i--)
        cut(stk[i]); // 撤销所有操作
    top = tmp;

    return ans;
}


// 以下是并查集相关函数
int findroot(int x) { // 因为需要撤销,不写路径压缩
    while (p[x] != x)
        x = p[x];

    return x;
}

void mergeset(int x, int y) { // 按size合并,如果想跑得更
  ↪ 快就写一个按秩合并
    x = findroot(x); // 但是按秩合并要再开一个栈记录合并
      ↪ 之前的秩
    y = findroot(y);

    if (x == y)
        return;

    if (size[x] > size[y])
        swap(x, y);

    p[x] = y;
    size[y] += size[x];
    stk[++top] = x;
}

void cut(int x) { // 并查集撤销
```

```
    int y = x;

    do
        size[y = p[y]] -= size[x];
    while (p[y]! = y);

    p[x] = x;
}
```

3.1.3 Steiner Tree 斯坦纳树

问题: 一张图上有k个关键点，求让关键点两两连通的最小生成树
做法: 状压DP，$f_{i,S}$表示以i号点为树根，i与S中的点连通的最小边权和
转移有两种:

1. 枚举子集:
$$f_{i,S} = \min_{T \subset S} \left\{ f_{i,T} + f_{i,S \setminus T} \right\}$$
2. 新加一条边:
$$f_{i,S} = \min_{(i,j) \in E} \left\{ f_{j,S} + w_{i,j} \right\}$$

第一种直接枚举子集DP就行了，第二种可以用SPFA或者Dijkstra松弛(显然负边一开始全选就行了，所以只需要处理非负边).
复杂度$O(n3^k + 2^k m \log n)$.

3.2 最短路

3.2.1 Dijkstra

见k短路(注意那边是求到t的最短路)

3.2.2 k短路

```
// 注意这是个多项式算法，在k比较大时很有优势，但k比较小时最好还是用A*
// DAG和有环的情况都可以，有重边或自环也无所谓，但不能有零环
// 以下代码以Dijkstra + 可持久化左偏树为例

constexpr int maxn = 1005, maxe = 10005, maxm = maxe * 30; //点数,边数,左偏树结点数

// 结构体定义
struct A { // 用来求最短路
    int x, d;

    A(int x, int d) : x(x), d(d) {}

    bool operator < (const A &a) const {
        return d > a.d;
    }
};

struct node { // 左偏树结点
    int w, i, d; // i: 最后一条边的编号 d: 左偏树附加信息
    node *lc, *rc;

    node() {}

    node(int w, int i) : w(w), i(i), d(0) {}

    void refresh(){
        d = rc -> d + 1;
    }
} null[maxm], *ptr = null, *root[maxn];

struct B { // 维护答案用
    int x, w; // x是结点编号，w表示之前已经产生的权值
    node *rt; // 这个答案对应的堆顶,注意可能不等于任何一个结点的堆

    B(int x, node *rt, int w) : x(x), w(w), rt(rt) {}

    bool operator < (const B &a) const {
        return w + rt -> w > a.w + a.rt -> w;
    }
};

// 全局变量和数组定义
vector<int> G[maxn], W[maxn], id[maxn]; // 最开始要存反向图，然后把G清空作为儿子列表
bool vis[maxn], used[maxe]; // used表示边是否在最短路树上
int u[maxe], v[maxe], w[maxe]; // 存下每条边,注意是有向边
int d[maxn], p[maxn]; // p表示最短路树上每个点的父边
int n, m, k, s, t; // s, t分别表示起点和终点

// 以下是主函数中较关键的部分
for (int i = 0 ; i <= n; i++)
    root[i] = null; // 一定要加上!!!

// (读入&建反向图)

Dijkstra();

// (清空G, W, id)

for (int i = 1; i <= n; i++)
    if (p[i]) {
        used[p[i]] = true; // 在最短路树上
        G[v[p[i]]].push_back(i);
    }

for (int i = 1; i <= m; i++) {
    w[i] -= d[u[i]] - d[v[i]]; // 现在的w[i]表示这条边能使路径长度增加多少
    if (!used[i])
        root[u[i]] = merge(root[u[i]], newnode(w[i], i));
}

dfs(t);

priority_queue<B> heap;
heap.push(B(s, root[s], 0)); // 初始状态是找贡献最小的边加进去

printf("%d\n",d[s]); // 第1短路需要特判
while (--k) { // 其余k - 1短路径用二叉堆维护
    if (heap.empty())
        printf("-1\n");
    else {
        int x = heap.top().x, w = heap.top().w;
        node *rt = heap.top().rt;
        heap.pop();

        printf("%d\n", d[s] + w + rt -> w);

        if (rt -> lc != null || rt -> rc != null)
            heap.push(B(x, merge(rt -> lc, rt -> rc), w)); // pop掉当前边，换成另一条贡献大一点的边
        if (root[v[rt -> i]] != null)
            heap.push(B(v[rt -> i], root[v[rt -> i]], w + rt -> w)); // 保留当前边，往后面再接上另一条边
    }
```

```
}
// 主函数到此结束


// Dijkstra预处理最短路 O(m\log n)
void Dijkstra() {
    memset(d, 63, sizeof(d));
    d[t] = 0;
    priority_queue<A> heap;
    heap.push(A(t, 0));

    while (!heap.empty()) {
        int x = heap.top().x;
        heap.pop();

        if(vis[x])
            continue;

        vis[x] = true;
        for (int i = 0; i < (int)G[x].size(); i++)
            if (!vis[G[x][i]] && d[G[x][i]] > d[x] + W[x]
             ↪ [i]) {
                d[G[x][i]] = d[x] + W[x][i];
                p[G[x][i]] = id[x][i];

                heap.push(A(G[x][i], d[G[x][i]]));
            }
    }
}

// dfs求出每个点的堆 总计O(m\log n)
// 需要调用merge, 同时递归调用自身
void dfs(int x) {
    root[x] = merge(root[x], root[v[p[x]]]);

    for (int i = 0; i < (int)G[x].size(); i++)
        dfs(G[x][i]);
}

// 包装过的new node() O(1)
node *newnode(int w, int i) {
    *++ptr = node(w, i);
    ptr -> lc = ptr -> rc = null;
    return ptr;
}

// 带可持久化的左偏树合并 总计O(\log n)
// 递归调用自身
node *merge(node *x, node *y) {
    if (x == null)
        return y;
    if (y == null)
        return x;

    if (x -> w > y -> w)
        swap(x, y);

    node *z = newnode(x -> w, x -> i);
    z -> lc = x -> lc;
    z -> rc = merge(x -> rc, y);

    if (z -> lc -> d > z -> rc -> d)
        swap(z -> lc, z -> rc);
    z -> refresh();

    return z;
}
```

3.3 Tarjan

3.3.1 强连通分量

3.3.2 割点 点双

3.4 桥 边双

3.5 仙人掌

3.5.1 仙人掌DP

```
struct edge{
    int to, w, prev;
}e[maxn * 2];

vector<pair<int, int> > v[maxn];

vector<long long> d[maxn];

stack<int> stk;

int p[maxn];

bool vis[maxn], vise[maxn * 2];

int last[maxn], cnte;

long long f[maxn], g[maxn], sum[maxn];

int n, m, cnt;

void addedge(int x, int y, int w) {
    v[x].push_back(make_pair(y, w));
}

void dfs(int x) {

    vis[x] = true;

    for (int i = last[x]; ~i; i = e[i].prev) {
        if (vise[i ^ 1])
            continue;

        int y = e[i].to, w = e[i].w;

        vise[i] = true;

        if (!vis[y]) {
            stk.push(i);
            p[y] = x;
            dfs(y);

            if (!stk.empty() && stk.top() == i) {
                stk.pop();
                addedge(x, y, w);
            }
        }

        else {
            cnt++;

            long long tmp = w;
            while (!stk.empty()) {
                int i = stk.top();
                stk.pop();

                int yy = e[i].to, ww = e[i].w;

                addedge(cnt, yy, 0);

                d[cnt].push_back(tmp);
```

```
                tmp += ww;

                if (e[i ^ 1].to == y)
                    break;
            }

            addedge(y, cnt, 0);

            sum[cnt] = tmp;
        }
    }
}

void dp(int x) {

    for (auto o : v[x]) {
        int y = o.first, w = o.second;
        dp(y);
    }

    if (x <= n) {
        for (auto o : v[x]) {
            int y = o.first, w = o.second;

            f[x] += 2 * w + f[y];
        }

        g[x] = f[x];

        for (auto o : v[x]) {
            int y = o.first, w = o.second;

            g[x] = min(g[x], f[x] - f[y] - 2 * w + g[y] +
              w);
        }
    }
    else {
        f[x] = sum[x];
        for (auto o : v[x]) {
            int y = o.first;

            f[x] += f[y];
        }

        g[x] = f[x];

        for (int i = 0; i < (int)v[x].size(); i++) {
            int y = v[x][i].first;

            g[x] = min(g[x], f[x] - f[y] + g[y] +
              min(d[x][i], sum[x] - d[x][i]));
        }
    }
}
```

3.6 二分图

3.6.1 KM二分图最大权匹配

```
const long long INF = 0x3f3f3f3f3f3f3f3f;

long long w[maxn][maxn], lx[maxn], ly[maxn], slack[maxn];
// 边权 顶标 slack
// 如果要求最大权完美匹配就把不存在的边设为-INF,否则所有
  边对0取max

bool visx[maxn], visy[maxn];

int boy[maxn], girl[maxn], p[maxn], q[maxn], head, tail;
  // p : pre

int n, m, N, e;

// 增广
bool check(int y) {
    visy[y] = true;

    if (boy[y]) {
        visx[boy[y]] = true;
        q[tail++] = boy[y];
        return false;
    }

    while (y) {
        boy[y] = p[y];
        swap(y, girl[p[y]]);
    }

    return true;
}

// bfs每个点
void bfs(int x) {
    memset(q, 0, sizeof(q));
    head = tail = 0;

    q[tail++] = x;
    visx[x] = true;

    while (true) {
        while (head != tail) {
            int x = q[head++];

            for (int y = 1; y <= N; y++)
                if (!visy[y]) {
                    long long d = lx[x] + ly[y] - w[x]
                      [y];

                    if (d < slack[y]) {
                        p[y] = x;
                        slack[y] = d;

                        if (!slack[y] && check(y))
                            return;
                    }
                }
        }

        long long d = INF;
        for (int i = 1; i <= N; i++)
            if (!visy[i])
                d = min(d, slack[i]);

        for (int i = 1; i <= N; i++) {
            if (visx[i])
                lx[i] -= d;

            if (visy[i])
                ly[i] += d;
            else
                slack[i] -= d;
        }

        for (int i = 1; i <= N; i++)
            if (!visy[i] && !slack[i] && check(i))
                return;
    }
```

```cpp
}

// 主过程
long long KM() {
    for (int i = 1; i <= N; i++) {
        // lx[i] = 0;
        ly[i] = -INF;
        // boy[i] = girl[i] = -1;

        for (int j = 1; j <= N; j++)
            ly[i] = max(ly[i], w[j][i]);
    }

    for (int i = 1; i <= N; i++) {
        memset(slack, 0x3f, sizeof(slack));
        memset(visx, 0, sizeof(visx));
        memset(visy, 0, sizeof(visy));
        bfs(i);
    }

    long long ans = 0;
    for (int i = 1; i <= N; i++)
        ans += w[i][girl[i]];
    return ans;
}

// 为了方便贴上主函数
int main() {

    scanf("%d%d%d", &n, &m, &e);
    N = max(n, m);

    while (e--) {
        int x, y, c;
        scanf("%d%d%d", &x, &y, &c);
        w[x][y] = max(c, 0);
    }

    printf("%lld\n", KM());

    for (int i = 1; i <= n; i++) {
        if (i > 1)
            printf(" ");
        printf("%d", w[i][girl[i]] > 0 ? girl[i] : 0);
    }
    printf("\n");

    return 0;
}
```

3.7 一般图匹配

3.7.1 高斯消元

```cpp
// 这个算法基于Tutte定理和高斯消元,思维难度相对小一些,也更方便进行可行边的判定
// 注意这个算法复杂度是满的,并且常数有点大,而带花树通常是跑不满的
// 以及,根据Tutte定理,如果求最大匹配的大小的话直接输出Tutte矩阵的秩/2即可
// 需要输出方案时才需要再写后面那些乱七八糟的东西


// 复杂度和常数所限,1s之内500已经是这个算法的极限了
const int maxn = 505, p = 1000000007; // p可以是任意10^9以内的质数

// 全局数组和变量定义
int A[maxn][maxn], B[maxn][maxn], t[maxn][maxn], id[maxn], a[maxn];
bool row[maxn] = {false}, col[maxn] = {false};
int n, m, girl[maxn]; // girl是匹配点,用来输出方案

// 为了方便使用,贴上主函数
// 需要调用高斯消元和eliminate
int main() {
    srand(19260817); // 膜蛤专用随机种子,换一个也无所谓

    scanf("%d%d", &n, &m); // 点数和边数
    while (m--) {
        int x, y;
        scanf("%d%d", &x, &y);
        A[x][y] = rand() % p;
        A[y][x] = -A[x][y]; // Tutte矩阵是反对称矩阵
    }

    for (int i = 1; i <= n; i++)
        id[i] = i; // 输出方案用的,因为高斯消元的时候会交换列
    memcpy(t, A, sizeof(t));
    Gauss(A, NULL, n);

    m = n;
    n = 0; // 这里变量复用纯属个人习惯......

    for (int i = 1; i <= m; i++)
        if (A[id[i]][id[i]])
            a[++n] = i; // 找出一个极大满秩子矩阵

    for (int i = 1;i <= n; i++)
        for (int j = 1; j <= n; j++)
            A[i][j]=t[a[i]][a[j]];

    Gauss(A,B,n);

    for (int i = 1; i <= n; i++)
        if (!girl[a[i]])
            for (int j = i + 1; j <= n; j++)
                if (!girl[a[j]] && t[a[i]][a[j]] && B[j][i]) {
                    // 注意上面那句if的写法,现在t是邻接矩阵的备份,
                    // 逆矩阵j行i列不为0当且仅当这条边可行
                    girl[a[i]] = a[j];
                    girl[a[j]] = a[i];
                    eliminate(i, j);
                    eliminate(j, i);
                    break;
                }

    printf("%d\n", n >> 1);
    for (int i = 1; i <= m; i++)
        printf("%d ", girl[i]);

    return 0;
}

// 高斯消元 O(n^3)
// 在传入B时表示计算逆矩阵,传入NULL则只需计算矩阵的秩
void Gauss(int A[][maxn], int B[][maxn], int n){
    if(B) {
        memset(B, 0, sizeof(t));
        for (int i = 1; i <= n; i++)
            B[i][i] = 1;
    }

    for (int i = 1; i <= n; i++) {
        if (!A[i][i]) {
```

```
            for (int j = i + 1; j <= n; j++)
                if (A[j][i]) {
                    swap(id[i], id[j]);
                    for (int k = i; k <= n; k++)
                        swap(A[i][k], A[j][k]);

                    if (B)
                        for (int k = 1; k <= n; k++)
                            swap(B[i][k], B[j][k]);
                    break;
                }

            if (!A[i][i])
                continue;
        }

        int inv = qpow(A[i][i], p - 2);

        for (int j = 1; j <= n; j++)
            if (i != j && A[j][i]){
                int t = (long long)A[j][i] * inv % p;

                for (int k = i; k <= n; k++)
                    if (A[i][k])
                        A[j][k] = (A[j][k] - (long long)t
                          * A[i][k]) % p;

                if (B)
                    for (int k = 1; k <= n; k++)
                        if (B[i][k])
                            B[j][k] = (B[j][k] - (long
                              long)t * B[i][k])%p;
            }
    }

    if (B)
        for (int i = 1; i <= n; i++) {
            int inv = qpow(A[i][i], p - 2);

            for (int j = 1; j <= n; j++)
                if (B[i][j])
                    B[i][j] = (long long)B[i][j] * inv %
                      p;
        }
}

// 消去一行一列 O(n^2)
void eliminate(int r, int c) {
    row[r] = col[c] = true; // 已经被消掉

    int inv = qpow(B[r][c], p - 2);

    for (int i = 1; i <= n; i++)
        if (!row[i] && B[i][c]) {
            int t = (long long)B[i][c] * inv % p;

            for (int j = 1; j <= n; j++)
                if (!col[j] && B[r][j])
                    B[i][j] = (B[i][j] - (long long)t *
                      B[r][j]) % p;
        }
}
```

3.7.2 带花树

```
// 带花树通常比高斯消元快很多,但在只需要求最大匹配大小的
  时候并没有高斯消元好写
// 当然输出方案要方便很多

// 全局数组与变量定义
vector<int> G[maxn];
int girl[maxn], f[maxn], t[maxn], p[maxn], vis[maxn],
  tim, q[maxn], head, tail;
int n, m;


// 封装好的主过程 O(nm)
int blossom() {
    int ans = 0;

    for (int i = 1; i <= n; i++)
        if (!girl[i])
            ans += bfs(i);

    return ans;
}


// bfs找增广路 O(m)
bool bfs(int s) {
    memset(t, 0, sizeof(t));
    memset(p, 0, sizeof(p));

    for (int i = 1; i <= n; i++)
        f[i] = i; // 并查集

    head = tail = 0;
    q[tail++] = s;
    t[s] = 1;

    while (head != tail){
        int x = q[head++];
        for (int y : G[x]){
            if (findroot(y) == findroot(x) || t[y] == 2)
                continue;

            if (!t[y]){
                t[y] = 2;
                p[y] = x;

                if (!girl[y]){
                    for (int u = y, t; u; u = t) {
                        t = girl[p[u]];
                        girl[p[u]] = u;
                        girl[u] = p[u];
                    }
                    return true;
                }
                t[girl[y]] = 1;
                q[tail++] = girl[y];
            }
            else if (t[y] == 1) {
                int z = LCA(x, y);
                shrink(x, y, z);
                shrink(y, x, z);
            }
        }
    }

    return false;
}

//缩奇环 O(n)
void shrink(int x, int y, int z) {
    while (findroot(x) != z){
        p[x] = y;
        y = girl[x];

```

```
        if (t[y] == 2) {
            t[y] = 1;
            q[tail++] = y;
        }

        if(findroot(x) == x)
            f[x] = z;
        if(findroot(y) == y)
            f[y] = z;

        x = p[y];
    }
}

//暴力找LCA O(n)
int LCA(int x, int y) {
    tim++;
    while (true) {
        if (x) {
            x = findroot(x);

            if (vis[x] == tim)
                return x;
            else {
                vis[x] = tim;
                x = p[girl[x]];
            }
        }
        swap(x, y);
    }
}

//并查集的查找 O(1)
int findroot(int x) {
    return x == f[x] ? x : (f[x] = findroot(f[x]));
}
```

3.7.3 带权带花树

(有一说一这玩意实在太难写了，抄之前建议先想想算法是不是假的或者有SB做法)

```
//maximum weight blossom,  change g[u][v].w to INF - g[u][v].w when minimum weight blossom is needed
//type of ans is long long
//replace all int to long long if weight of edge is long long

struct WeightGraph {
    static const int INF = INT_MAX;
    static const int MAXN = 400;
    struct edge{
        int u, v, w;
        edge() {}
        edge(int u, int v, int w): u(u), v(v), w(w) {}
    };
    int n, n_x;
    edge g[MAXN * 2 + 1][MAXN * 2 + 1];
    int lab[MAXN * 2 + 1];
    int match[MAXN * 2 + 1], slack[MAXN * 2 + 1], st[MAXN * 2 + 1], pa[MAXN * 2 + 1];
    int flower_from[MAXN * 2 + 1][MAXN+1], S[MAXN * 2 + 1], vis[MAXN * 2 + 1];
    vector<int> flower[MAXN * 2 + 1];
    queue<int> q;
    inline int e_delta(const edge &e){ // does not work inside blossoms
        return lab[e.u] + lab[e.v] - g[e.u][e.v].w * 2;
    }
    inline void update_slack(int u, int x){
        if(!slack[x] || e_delta(g[u][x]) < e_delta(g[slack[x]][x]))
            slack[x] = u;
    }
    inline void set_slack(int x){
        slack[x] = 0;
        for(int u = 1;u <= n; ++u)
            if(g[u][x].w > 0 && st[u] != x && S[st[u]] == 0)
                update_slack(u, x);
    }
    void q_push(int x){
        if(x <= n)q.push(x);
        else for(size_t i = 0;i < flower[x].size(); i++)
            q_push(flower[x][i]);
    }
    inline void set_st(int x, int b){
        st[x]=b;
        if(x > n) for(size_t i = 0;i < flower[x].size(); ++i)
                set_st(flower[x][i], b);
    }
    inline int get_pr(int b, int xr){
        int pr = find(flower[b].begin(), flower[b].end(), xr) - flower[b].begin();
        if(pr % 2 == 1){
            reverse(flower[b].begin() + 1, flower[b].end());
            return (int)flower[b].size() - pr;
        } else return pr;
    }
    inline void set_match(int u, int v){
        match[u]=g[u][v].v;
        if(u > n){
            edge e=g[u][v];
            int xr = flower_from[u][e.u], pr=get_pr(u, xr);
            for(int i = 0;i < pr; ++i)
                set_match(flower[u][i], flower[u][i ^ 1]);
            set_match(xr, v);
            rotate(flower[u].begin(), flower[u].begin()+pr, flower[u].end());
        }
    }
    inline void augment(int u, int v){
        for(; ; ){
            int xnv=st[match[u]];
            set_match(u, v);
            if(!xnv)return;
            set_match(xnv, st[pa[xnv]]);
            u=st[pa[xnv]], v=xnv;
        }
    }
    inline int get_lca(int u, int v){
        static int t=0;
        for(++t; u || v; swap(u, v)){
            if(u == 0)continue;
            if(vis[u] == t)return u;
            vis[u] = t;
            u = st[match[u]];
            if(u) u = st[pa[u]];
        }
        return 0;
    }
    inline void add_blossom(int u, int lca, int v){
        int b = n + 1;
        while(b <= n_x && st[b]) ++b;
```

```cpp
        if(b > n_x) ++n_x;
        lab[b] = 0, S[b] = 0;
        match[b] = match[lca];
        flower[b].clear();
        flower[b].push_back(lca);
        for(int x = u, y; x != lca; x = st[pa[y]]) {
            flower[b].push_back(x),
            flower[b].push_back(y = st[match[x]]),
            q_push(y);
        }
        reverse(flower[b].begin() + 1, flower[b].end());
        for(int x = v, y; x != lca; x = st[pa[y]]) {
            flower[b].push_back(x),
            flower[b].push_back(y = st[match[x]]),
            q_push(y);
        }
        set_st(b, b);
        for(int x = 1; x <= n_x; ++x) g[b][x].w = g[x][b].w = 0;
        for(int x = 1; x <= n; ++x) flower_from[b][x] = 0;
        for(size_t i = 0 ; i < flower[b].size(); ++i){
            int xs = flower[b][i];
            for(int x = 1; x <= n_x; ++x)
                if(g[b][x].w == 0 || e_delta(g[xs][x]) < e_delta(g[b][x]))
                    g[b][x] = g[xs][x], g[x][b] = g[x][xs];
            for(int x = 1;x <= n; ++x)
                if(flower_from[xs][x]) flower_from[b][x] = xs;
        }
        set_slack(b);
    }
    inline void expand_blossom(int b){ // S[b]  ==  1
        for(size_t i = 0; i < flower[b].size(); ++i)
            set_st(flower[b][i], flower[b][i]);
        int xr = flower_from[b][g[b][pa[b]].u], pr = get_pr(b, xr);
        for(int i = 0; i < pr; i += 2){
            int xs = flower[b][i], xns = flower[b][i + 1];
            pa[xs] = g[xns][xs].u;
            S[xs] = 1, S[xns] = 0;
            slack[xs] = 0, set_slack(xns);
            q_push(xns);
        }
        S[xr] = 1, pa[xr] = pa[b];
        for(size_t i = pr + 1;i < flower[b].size(); ++i){
            int xs = flower[b][i];
            S[xs] = -1, set_slack(xs);
        }
        st[b] = 0;
    }
    inline bool on_found_edge(const edge &e){
        int u = st[e.u], v = st[e.v];
        if(S[v] == -1){
            pa[v] = e.u, S[v] = 1;
            int nu = st[match[v]];
            slack[v] = slack[nu] = 0;
            S[nu] = 0, q_push(nu);
        }else if(S[v] == 0){
            int lca = get_lca(u, v);
            if(!lca) return augment(u, v), augment(v, u), true;
            else add_blossom(u, lca, v);
        }
        return false;
    }
    inline bool matching(){
        memset(S + 1, -1, sizeof(int) * n_x);
        memset(slack + 1, 0, sizeof(int) * n_x);
        q = queue<int>();
        for(int x = 1;x <= n_x; ++x)
            if(st[x] == x && !match[x]) pa[x]=0, S[x]=0, q_push(x);
        if(q.empty())return false;
        for(;;){
            while(q.size()){
                int u = q.front();q.pop();
                if(S[st[u]] == 1)continue;
                for(int v = 1;v <= n; ++v)
                    if(g[u][v].w > 0 && st[u] != st[v]){
                        if(e_delta(g[u][v]) == 0){
                            if(on_found_edge(g[u][v]))return true;
                        }else update_slack(u, st[v]);
                    }
            }
            int d = INF;
            for(int b = n + 1; b <= n_x;++b)
                if(st[b] == b && S[b] == 1)d = min(d, lab[b]/2);
            for(int x = 1; x <= n_x; ++x)
                if(st[x] == x && slack[x]){
                    if(S[x] == -1)d = min(d, e_delta(g[slack[x]][x]));
                    else if(S[x] == 0)d = min(d, e_delta(g[slack[x]][x])/2);
                }
            for(int u = 1; u <= n; ++u){
                if(S[st[u]] == 0){
                    if(lab[u] <= d)return 0;
                    lab[u] -= d;
                }else if(S[st[u]] == 1)lab[u] += d;
            }
            for(int b = n+1; b <= n_x; ++b)
                if(st[b] == b){
                    if(S[st[b]] == 0) lab[b] += d * 2;
                    else if(S[st[b]] == 1) lab[b] -= d * 2;
                }
            q=queue<int>();
            for(int x = 1; x <= n_x; ++x)
                if(st[x] == x && slack[x] && st[slack[x]] != x && e_delta(g[slack[x]][x]) == 0)
                    if(on_found_edge(g[slack[x]][x]))return true;
            for(int b = n + 1; b <= n_x; ++b)
                if(st[b] == b && S[b] == 1 && lab[b] == 0)expand_blossom(b);
        }
        return false;
    }
    inline pair<long long, int> solve(){
        memset(match + 1, 0, sizeof(int) * n);
        n_x = n;
        int n_matches = 0;
        long long tot_weight = 0;
        for(int u = 0; u <= n; ++u) st[u] = u, flower[u].clear();
        int w_max = 0;
        for(int u = 1; u <= n; ++u)
            for(int v = 1; v <= n; ++v){
                flower_from[u][v] = (u == v ? u : 0);
                w_max = max(w_max, g[u][v].w);
            }
```

```
        for(int u = 1; u <= n; ++u) lab[u] = w_max;
        while(matching()) ++n_matches;
        for(int u = 1; u <= n; ++u)
            if(match[u] && match[u] < u)
                tot_weight += g[u][match[u]].w;
        return make_pair(tot_weight, n_matches);
    }
    inline void init(){
        for(int u = 1; u <= n; ++u)
            for(int v = 1; v <= n; ++v)
                g[u][v]=edge(u, v, 0);
    }
};
```

3.8 最大流

3.8.1 Dinic

```
// 注意Dinic适用于二分图或分层图,对于一般稀疏图ISAP更
  ↪ 优,稠密图则HLPP更优

struct edge{
    int to, cap, prev;
} e[maxe * 2];

int last[maxn], len, d[maxn], cur[maxn], q[maxn];

memset(last, -1, sizeof(last));

void AddEdge(int x, int y, int z) {
    e[len].to = y;
    e[len].cap = z;
    e[len].prev = last[x];
    last[x] = len++;
}

int Dinic() {
    int flow = 0;
    while (bfs(), ~d[t]) {
        memcpy(cur, last, sizeof(int) * (t + 5));
        flow += dfs(s, inf);
    }
    return flow;
}

void bfs() {
    int head = 0, tail = 0;
    memset(d, -1, sizeof(int) * (t + 5));
    q[tail++] = s;
    d[s] = 0;

    while (head != tail){
        int x = q[head++];
        for (int i = last[x]; ~i; i = e[i].prev)
            if (e[i].cap > 0 && d[e[i].to] == -1) {
                d[e[i].to] = d[x] + 1;
                q[tail++] = e[i].to;
            }
    }
}

int dfs(int x, int a) {
    if (x == t || !a)
        return a;

    int flow = 0, f;
    for (int &i = cur[x]; ~i; i = e[i].prev)
        if (e[i].cap > 0 && d[e[i].to] == d[x] + 1 && (f
          ↪ = dfs(e[i].to, min(e[i].cap,a)))) {

                e[i].cap -= f;
                e[i^1].cap += f;
                flow += f;
                a -= f;

                if (!a)
                    break;
            }

    return flow;
}
```

3.8.2 ISAP

```
// 注意ISAP适用于一般稀疏图,对于二分图或分层图情
  ↪ 况Dinic比较优,稠密图则HLPP更优


// 边的定义
// 这里没有记录起点和反向边,因为反向边即为正向边xor 1,起
  ↪ 点即为反向边的终点
struct edge{
    int to, cap, prev;
} e[maxe * 2];


// 全局变量和数组定义
int last[maxn], cnte = 0, d[maxn], p[maxn], c[maxn],
  ↪ cur[maxn], q[maxn];
int n, m, s, t; // s, t一定要开成全局变量


// 重要!!!
// main函数最前面一定要加上如下初始化
memset(last, -1, sizeof(last));


// 加边函数 O(1)
// 包装了加反向边的过程,方便调用
// 需要调用AddEdge
void addedge(int x, int y, int z) {
    AddEdge(x, y, z);
    AddEdge(y, x, 0);
}


// 真·加边函数 O(1)
void AddEdge(int x, int y, int z) {
    e[cnte].to = y;
    e[cnte].cap = z;
    e[cnte].prev = last[x];
    last[x] = cnte++;
}


// 主过程 O(n^2 m)
// 返回最大流的流量
// 需要调用bfs,augment
// 注意这里的n是编号最大值,在这个值不为n的时候一定要开个
  ↪ 变量记录下来并修改代码
// 非递归
int ISAP() {
    bfs();

    memcpy(cur, last, sizeof(cur));

    int x = s, flow = 0;

    while (d[s] < n) {
```

```cpp
            if (x == t) {//如果走到了t就增广一次,并返回s重新
            ↪ 找增广路
                flow += augment();
                x = s;
            }

            bool ok = false;
            for (int &i = cur[x]; ~i; i = e[i].prev)
                if (e[i].cap && d[x] == d[e[i].to] + 1) {
                    p[e[i].to] = i;
                    x = e[i].to;

                    ok = true;
                    break;
                }

            if (!ok) { // 修改距离标号
                int tmp = n - 1;
                for (int i = last[x]; ~i; i = e[i].prev)
                    if (e[i].cap)
                        tmp = min(tmp, d[e[i].to] + 1);

                if (!--c[d[x]])
                    break; // gap优化,一定要加上

                c[d[x] = tmp]++;
                cur[x] = last[x];

                if(x != s)
                    x = e[p[x] ^ 1].to;
            }
        }
        return flow;
}

// bfs函数 O(n+m)
// 预处理到t的距离标号
// 在测试数据组数较少时可以省略,把所有距离标号初始化为0
void bfs() {
    memset(d, -1, sizeof(d));

    int head = 0, tail = 0;
    d[t] = 0;
    q[tail++] = t;

    while (head != tail) {
        int x = q[head++];
        c[d[x]]++;

        for (int i = last[x]; ~i; i = e[i].prev)
            if (e[i ^ 1].cap && d[e[i].to] == -1) {
                d[e[i].to] = d[x] + 1;
                q[tail++] = e[i].to;
            }
    }
}

// augment函数 O(n)
// 沿增广路增广一次,返回增广的流量
int augment() {
    int a = (~0u) >> 1; // INT_MAX

    for (int x = t; x != s; x = e[p[x] ^ 1].to)
        a = min(a, e[p[x]].cap);

    for (int x = t; x != s; x = e[p[x] ^ 1].to) {
        e[p[x]].cap -= a;
        e[p[x] ^ 1].cap += a;
    }

    return a;
}
```

3.8.3 HLPP最高标号预流推进

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

using namespace std;

constexpr int maxn = 1205, maxe = 120005, inf =
    ↪ 2147483647;

struct edge {
    int to, cap, prev;
} e[maxe * 2];

int n, m, s, t;
int last[maxn], cnte;
int h[maxn], ex[maxn], gap[maxn * 2];
bool inq[maxn];

struct cmp {
    bool operator() (int x, int y) const {
        return h[x] < h[y];
    }
};

priority_queue<int, vector<int>, cmp> heap;

void AddEdge(int x, int y, int z) {
    e[cnte].to = y;
    e[cnte].cap = z;
    e[cnte].prev = last[x];
    last[x] = cnte++;
}

void addedge(int x, int y, int z) {
    AddEdge(x, y, z);
    AddEdge(y, x, 0);
}

bool bfs() {
    static int q[maxn];

    fill(h, h + n + 1, 2 * n);
    int head = 0, tail = 0;
    q[tail++] = t;
    h[t] = 0;

    while (head < tail) {
        int x = q[head++];
        for (int i = last[x]; ~i; i = e[i].prev)
            if (e[i ^ 1].cap && h[e[i].to] > h[x] + 1) {
                h[e[i].to] = h[x] + 1;
                q[tail++] = e[i].to;
            }
    }

    return h[s] < 2 * n;
}

void push(int x) {
    for (int i = last[x]; ~i; i = e[i].prev)
        if (e[i].cap && h[x] == h[e[i].to] + 1) {
            int d = min(ex[x], e[i].cap);

            e[i].cap -= d;
            e[i ^ 1].cap += d;
            ex[x] -= d;
```

```cpp
            ex[e[i].to] += d;

            if (e[i].to != s && e[i].to != t && !inq[e[i].to]) {
                heap.push(e[i].to);
                inq[e[i].to] = true;
            }

            if (!ex[x])
                break;
        }
}

void relabel(int x) {
    h[x] = 2 * n;

    for (int i = last[x]; ~i; i = e[i].prev)
        if (e[i].cap)
            h[x] = min(h[x], h[e[i].to] + 1);
}

int hlpp() {
    if (!bfs())
        return 0;

    // memset(gap, 0, sizeof(int) * 2 * n);
    h[s] = n;

    for (int i = 1; i <= n; i++)
        gap[h[i]]++;

    for (int i = last[s]; ~i; i = e[i].prev)
        if (e[i].cap) {
            int d = e[i].cap;

            e[i].cap -= d;
            e[i ^ 1].cap += d;
            ex[s] -= d;
            ex[e[i].to] += d;

            if (e[i].to != s && e[i].to != t && !inq[e[i].to]) {
                heap.push(e[i].to);
                inq[e[i].to] = true;
            }
        }

    while (!heap.empty()) {
        int x = heap.top();
        heap.pop();
        inq[x] = false;

        push(x);
        if (ex[x]) {
            if (!--gap[h[x]]) { // gap
                for (int i = 1; i <= n; i++)
                    if (i != s && i != t && h[i] > h[x])
                        h[i] = n + 1;
            }

            relabel(x);
            ++gap[h[x]];
            heap.push(x);
            inq[x] = true;
        }
    }

    return ex[t];
}


int main() {

    memset(last, -1, sizeof(last));

    scanf("%d%d%d%d", &n, &m, &s, &t);

    while (m--) {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        addedge(x, y, z);
    }

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

    return 0;
}
```

3.9 费用流

3.9.1 SPFA费用流

```cpp
constexpr int maxn = 20005, maxm = 200005;

struct edge {
    int to, prev, cap, w;
} e[maxm * 2];

int last[maxn], cnte, d[maxn], p[maxn]; // 记得把last初始化成-1, 不然会死循环
bool inq[maxn];

void spfa(int s) {

    memset(d, -63, sizeof(d));
    memset(p, -1, sizeof(p));

    queue<int> q;

    q.push(s);
    d[s] = 0;

    while (!q.empty()) {
        int x = q.front();
        q.pop();
        inq[x] = false;

        for (int i = last[x]; ~i; i = e[i].prev)
            if (e[i].cap) {
                int y = e[i].to;

                if (d[x] + e[i].w > d[y]) {
                    p[y] = i;
                    d[y] = d[x] + e[i].w;
                    if (!inq[y]) {
                        q.push(y);
                        inq[y] = true;
                    }
                }
            }
    }
}

int mcmf(int s, int t) {
    int ans = 0;

    while (spfa(s), d[t] > 0) {
        int flow = 0x3f3f3f3f;
        for (int x = t; x != s; x = e[p[x] ^ 1].to)
            flow = min(flow, e[p[x]].cap);

```

```cpp
        ans += flow * d[t];

        for (int x = t; x != s; x = e[p[x] ^ 1].to) {
            e[p[x]].cap -= flow;
            e[p[x] ^ 1].cap += flow;
        }
    }

    return ans;
}

void add(int x, int y, int c, int w) {
    e[cnte].to = y;
    e[cnte].cap = c;
    e[cnte].w = w;

    e[cnte].prev = last[x];
    last[x] = cnte++;
}

void addedge(int x, int y, int c, int w) {
    add(x, y, c, w);
    add(y, x, 0, -w);
}
```

3.10 弦图相关

From NEW CODE!!

1. 团数 $\leq$ 色数，弦图团数 $=$ 色数
2. 设 $next(v)$ 表示 $N(v)$ 中最前的点．令 w* 表示所有满足 $A \in B$ 的 w 中最后的一个点，判断 $v \cup N(v)$ 是否为极大团，只需判断是否存在一个 w，满足 $Next(w) = v$ 且 $|N(v)| + 1 \leq |N(w)|$ 即可．
3. 最小染色：完美消除序列从后往前依次给每个点染色，给每个点染上可以染的最小的颜色
4. 最大独立集：完美消除序列从前往后能选就选
5. 弦图最大独立集数 $=$ 最小团覆盖数，最小团覆盖：设最大独立集为 $\{p_1, p_2, \ldots, p_t\}$，则 $\{p_1 \cup N(p_1), \ldots, p_t \cup N(p_t)\}$ 为最小团覆盖

4. 数据结构

4.1 线段树

4.1.1 非递归线段树

~~让fstqwq手撕~~

- 如果$M = 2^k$，则只能维护$[1, M-2]$范围
- 找叶子：i对应的叶子就是$i + M$
- 单点修改：找到叶子然后向上跳
- 区间查询：左右区间各扩展一位，转换成开区间查询

```cpp
int query(int l, int r) {
    l += M - 1;
    r += M + 1;

    int ans = 0;
    while (l ^ r != 1) {
        ans += sum[l ^ 1] + sum[r ^ 1];

        l >>= 1;
        r >>= 1;
    }

    return ans;
}
```

区间修改要标记永久化，并且求区间和和求最值的代码不太一样

区间加，区间求和

```cpp
void update(int l, int r, int d) {
    int len = 1, cntl = 0, cntr = 0; // cntl, cntr是左右两边分别实际修改的区间长度
    for (l += n - 1, r += n + 1; l ^ r ^ 1; l >>= 1, r >>= 1, len <<= 1) {
        tree[l] += cntl * d, tree[r] += cntr * d;
        if (~l & 1) tree[l ^ 1] += d * len, mark[l ^ 1] += d, cntl += len;
        if (r & 1) tree[r ^ 1] += d * len, mark[r ^ 1] += d, cntr += len;
    }

    for (; l; l >>= 1, r >>= 1)
        tree[l] += cntl * d, tree[r] += cntr * d;
}

int query(int l, int r) {
    int ans = 0, len = 1, cntl = 0, cntr = 0;
    for (l += n - 1, r += n + 1; l ^ r ^ 1; l >>= 1, r >>= 1, len <<= 1) {
        ans += cntl * mark[l] + cntr * mark[r];
        if (~l & 1) ans += tree[l ^ 1], cntl += len;
        if (r & 1) ans += tree[r ^ 1], cntr += len;
    }

    for (; l; l >>= 1, r >>= 1)
        ans += cntl * mark[l] + cntr * mark[r];

    return ans;
}
```

区间加，区间求最大值

```cpp
void update(int l, int r, int d) {
    for (l += N - 1, r += N + 1; l ^ r ^ 1; l >>= 1, r >>= 1) {
        if (l < N) {
            tree[l] = max(tree[l << 1], tree[l << 1 | 1]) + mark[l];
            tree[r] = max(tree[r << 1], tree[r << 1 | 1]) + mark[r];
        }

        if (~l & 1) {
            tree[l ^ 1] += d;
            mark[l ^ 1] += d;
        }
        if (r & 1) {
            tree[r ^ 1] += d;
            mark[r ^ 1] += d;
        }
    }

    for (; l; l >>= 1, r >>= 1)
        if (l < N) tree[l] = max(tree[l << 1], tree[l << 1 | 1]) + mark[l],
                   tree[r] = max(tree[r << 1], tree[r << 1 | 1]) + mark[r];
}
```

```cpp

void query(int l, int r) {
    int maxl = -INF, maxr = -INF;

    for (l += N - 1, r += N + 1; l ^ r ^ 1; l >>= 1, r >>= 1) {
        maxl += mark[l];
        maxr += mark[r];

        if (~l & 1)
            maxl = max(maxl, tree[l ^ 1]);
        if (r & 1)
            maxr = max(maxr, tree[r ^ 1]);
    }

    while (l) {
        maxl += mark[l];
        maxr += mark[r];

        l >>= 1;
        r >>= 1;
    }

    return max(maxl, maxr);
}
```

4.1.2 线段树维护矩形并

4.1.3 主席树

这种东西能不能手撕啊

4.2 陈丹琦分治

```cpp
// 四维偏序

void CDQ1(int l, int r) {
    if (l >= r)
        return;

    int mid = (l + r) / 2;

    CDQ1(l, mid);
    CDQ1(mid + 1, r);

    int i = l, j = mid + 1, k = l;

    while (i <= mid && j <= r) {
        if (a[i].x < a[j].x) {
            a[i].ins = true;
            b[k++] = a[i++];
        }
        else {
            a[j].ins = false;
            b[k++] = a[j++];
        }
    }

    while (i <= mid) {
        a[i].ins = true;
        b[k++] = a[i++];
    }

    while (j <= r) {
        a[j].ins = false;
        b[k++] = a[j++];
    }

    copy(b + l, b + r + 1, a + l); // 后面的分治会破坏排序，所以要复制一份

    CDQ2(l, r);
}

void CDQ2(int l, int r) {
    if (l >= r)
        return;

    int mid = (l + r) / 2;

    CDQ2(l, mid);
    CDQ2(mid + 1, r);

    int i = l, j = mid + 1, k = l;

    while (i <= mid && j <= r) {
        if (b[i].y < b[j].y) {
            if (b[i].ins)
                add(b[i].z, 1); // 树状数组

            t[k++] = b[i++];
        }
        else{
            if (!b[j].ins)
                ans += query(b[j].z - 1);

            t[k++] = b[j++];
        }
    }

    while (i <= mid) {
        if (b[i].ins)
            add(b[i].z, 1);

        t[k++] = b[i++];
    }

    while (j <= r) {
        if (!b[j].ins)
            ans += query(b[j].z - 1);

        t[k++] = b[j++];
    }

    for (i = l; i <= mid; i++)
        if (b[i].ins)
            add(b[i].z, -1);

    copy(t + l, t + r + 1, b + l);
}
```

4.3 Treap

```cpp
// 注意：相同键值可以共存

struct node { // 结点类定义
    int key, size, p; // 分别为键值，子树大小，优先度
    node *ch[2]; // 0表示左儿子，1表示右儿子

    node(int key = 0) : key(key), size(1), p(rand()) {}

    void refresh() {
        size = ch[0] -> size + ch[1] -> size + 1;
    } // 更新子树大小(和附加信息，如果有的话)
} null[maxn], *root = null, *ptr = null; // 数组名叫做null是为了方便开哨兵节点
```

```cpp
// 如果需要删除而空间不能直接开下所有结点，则需要再写一个垃圾回收
// 注意：数组里的元素一定不能delete，否则会导致RE

// 重要!在主函数最开始一定要加上以下预处理:
null -> ch[0] = null -> ch[1] = null;
null -> size = 0;

// 伪构造函数 O(1)
// 为了方便，在结点类外面再定义一个伪构造函数
node *newnode(int x) { // 键值为x
    *++ptr = node(x);
    ptr -> ch[0] = ptr -> ch[1] = null;
    return ptr;
}

// 插入键值 期望O(\log n)
// 需要调用旋转
void insert(int x, node *&rt) { // rt为当前结点，建议调用时传入root，下同
    if (rt == null) {
        rt = newnode(x);
        return;
    }

    int d = x > rt -> key;
    insert(x, rt -> ch[d]);
    rt -> refresh();

    if (rt -> ch[d] -> p < rt -> p)
        rot(rt, d ^ 1);
}

// 删除一个键值 期望O(\log n)
// 要求键值必须存在至少一个，否则会导致RE
// 需要调用旋转
void erase(int x, node *&rt) {
    if (x == rt -> key) {
        if (rt -> ch[0] != null && rt -> ch[1] != null) {
            int d = rt -> ch[0] -> p < rt -> ch[1] -> p;
            rot(rt, d);
            erase(x, rt -> ch[d]);
        }
        else
            rt = rt -> ch[rt -> ch[0] == null];
    }
    else
        erase(x, rt -> ch[x > rt -> key]);

    if (rt != null)
        rt -> refresh();
}

// 求元素的排名(严格小于键值的个数 + 1) 期望O(\log n)
// 非递归
int rank(int x, node *rt) {
    int ans = 1, d;
    while (rt != null) {
        if ((d = x > rt -> key))
            ans += rt -> ch[0] -> size + 1;

        rt = rt -> ch[d];
    }

    return ans;
}

// 返回排名第k(从1开始)的键值对应的指针 期望O(\log n)
// 非递归
node *kth(int x, node *rt) {
    int d;
    while (rt != null) {
        if (x == rt -> ch[0] -> size + 1)
            return rt;

        if ((d = x > rt -> ch[0] -> size))
            x -= rt -> ch[0] -> size + 1;

        rt = rt -> ch[d];
    }

    return rt;
}

// 返回前驱(最大的比给定键值小的键值)对应的指针 期望O(\log n)
// 非递归
node *pred(int x, node *rt) {
    node *y = null;
    int d;

    while (rt != null) {
        if ((d = x > rt -> key))
            y = rt;

        rt = rt -> ch[d];
    }

    return y;
}

// 返回后继（最小的比给定键值大的键值）对应的指针 期望O(\log n)
// 非递归
node *succ(int x, node *rt) {
    node *y = null;
    int d;

    while (rt != null) {
        if ((d = x < rt -> key))
            y = rt;

        rt = rt -> ch[d ^ 1];
    }

    return y;
}

// 旋转(Treap版本) O(1)
// 平衡树基础操作
// 要求对应儿子必须存在，否则会导致后续各种莫名其妙的问题
void rot(node *&x, int d) { // x为被转下去的结点，会被修改以维护树结构
    node *y = x -> ch[d ^ 1];

    x -> ch[d ^ 1] = y -> ch[d];
    y -> ch[d] = x;

    x -> refresh();
    (x = y) -> refresh();
}
```

4.4 Splay

(参见LCT,除了splay()需要传一个点表示最终它的父亲,其他写法都和LCT相同)

4.5 树分治

4.5.1 动态树分治

```
// 为了减小常数，这里采用bfs写法，实测预处理比dfs快将近
  ↪ 一半
// 以下以维护一个点到每个黑点的距离之和为例

// 全局数组定义
vector<int> G[maxn], W[maxn];
int size[maxn], son[maxn], q[maxn];
int p[maxn], depth[maxn], id[maxn][20], d[maxn][20]; //
  ↪ id是对应层所在子树的根
int a[maxn], ca[maxn], b[maxn][20], cb[maxn][20]; // 维护
  ↪ 距离和用的
bool vis[maxn], col[maxn];

// 建树 总计O(n\log n)
// 需要调用找重心和预处理距离，同时递归调用自身
void build(int x, int k, int s, int pr) { // 结点, 深度,
  ↪ 连通块大小, 点分树上的父亲
    x = getcenter(x, s);
    vis[x] = true;
    depth[x] = k;
    p[x] = pr;

    for (int i = 0; i < (int)G[x].size(); i++)
        if (!vis[G[x][i]]) {
            d[G[x][i]][k] = W[x][i];
            p[G[x][i]] = x;

            getdis(G[x][i],k,G[x][i]); // bfs每个子树, 预
              ↪ 处理距离
        }

    for (int i = 0; i < (int)G[x].size(); i++)
        if (!vis[G[x][i]])
            build(G[x][i], k + 1, size[G[x][i]], x); //
              ↪ 递归建树
}

// 找重心 O(n)
int getcenter(int x, int s) {
    int head = 0, tail = 0;
    q[tail++] = x;

    while (head != tail) {
        x = q[head++];
        size[x] = 1; // 这里不需要清空, 因为以后要用的话
          ↪ 一定会重新赋值
        son[x] = 0;

        for (int i = 0; i < (int)G[x].size(); i++)
            if (!vis[G[x][i]] && G[x][i] != p[x]) {
                p[G[x][i]] = x;
                q[tail++] = G[x][i];
            }
    }

    for (int i = tail - 1; i; i--) {
        x = q[i];
        size[p[x]] += size[x];

        if (size[x] > size[son[p[x]]])
            son[p[x]] = x;
    }

    x = q[0];
    while (son[x] && size[son[x]] * 2 >= s)
        x = son[x];

    return x;
}

// 预处理距离 O(n)
// 方便起见, 这里直接用了笨一点的方法, O(n\log n)全存下
  ↪ 来
void getdis(int x, int k, int rt) {
    int head = 0, tail = 0;
    q[tail++] = x;

    while (head != tail) {
        x = q[head++];
        size[x] = 1;
        id[x][k] = rt;

        for (int i = 0; i < (int)G[x].size(); i++)
            if (!vis[G[x][i]] && G[x][i] != p[x]) {
                p[G[x][i]] = x;
                d[G[x][i]][k] = d[x][k] + W[x][i];

                q[tail++] = G[x][i];
            }
    }

    for (int i = tail - 1; i; i--)
        size[p[q[i]]] += size[q[i]]; // 后面递归建树要用
          ↪ 到子问题大小
}

// 修改 O(\log n)
void modify(int x) {
    if (col[x])
        ca[x]--;
    else
        ca[x]++; // 记得先特判自己作为重心的那层

    for (int u = p[x], k = depth[x] - 1; u; u = p[u],
      ↪ k--) {
        if (col[x]) {
            a[u] -= d[x][k];
            ca[u]--;

            b[id[x][k]][k] -= d[x][k];
            cb[id[x][k]][k]--;
        }
        else {
            a[u] += d[x][k];
            ca[u]++;

            b[id[x][k]][k] += d[x][k];
            cb[id[x][k]][k]++;
        }
    }

    col[x] ^= true;
}

// 询问 O(\log n)
int query(int x) {
    int ans = a[x]; // 特判自己是重心的那层

    for (int u = p[x], k = depth[x] - 1; u; u = p[u],
      ↪ k--)
        ans += a[u] - b[id[x][k]][k] + d[x][k] * (ca[u] -
          ↪ cb[id[x][k]][k]);

    return ans;
}
```

4.5.2 紫荆花之恋

```
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<vector>
using namespace std;
const int maxn=100010;
const double alpha=0.7;
struct node{
    static int randint(){
        static int a=1213,b=97818217,p=998244353,x=751815431;
        x=a*x+b;x%=p;
        return x<0?(x+=p):x;
    }
    int data,size,p;
    node *ch[2];
    node(int d):data(d),size(1),p(randint()){}
    inline void refresh(){size=ch[0]->size+ch[1]->size+1;}
}*null=new node(0),*root[maxn],*root1[maxn][50];
void addnode(int,int);
void rebuild(int,int,int,int);
void dfs_getcenter(int,int,int&);
void dfs_getdis(int,int,int,int);
void dfs_destroy(int,int);
void insert(int,node*&);
int order(int,node*);
void destroy(node*&);
void rot(node*&,int);
vector<int>G[maxn],W[maxn];
int size[maxn]={0},siz[maxn][50]={0},son[maxn];
bool vis[maxn];
int depth[maxn],p[maxn],d[maxn][50],id[maxn][50];
int n,m,w[maxn],tmp;
long long ans=0;
int main(){
    freopen("flowera.in","r",stdin);
    freopen("flowera.out","w",stdout);
    null->size=0;
    null->ch[0]=null->ch[1]=null;
    scanf("%*d%d",&n);
    fill(vis,vis+n+1,true);
    fill(root,root+n+1,null);
    for(int i=0;i<=n;i++)fill(root1[i],root1[i]+50,null);
    scanf("%*d%*d%d",&w[1]);
    insert(-w[1],root[1]);
    size[1]=1;
    printf("0\n");
    for(int i=2;i<=n;i++){
        scanf("%d%d%d",&p[i],&tmp,&w[i]);
        p[i]^=(ans%(int)1e9);
        G[i].push_back(p[i]);
        W[i].push_back(tmp);
        G[p[i]].push_back(i);
        W[p[i]].push_back(tmp);
        addnode(i,tmp);
        printf("%lld\n",ans);
    }
    return 0;
}
void addnode(int x,int z){//wj-dj>=di-wi
    depth[x]=depth[p[x]]+1;
    size[x]=1;
    insert(-w[x],root[x]);
    int rt=0;
    for(int u=p[x],k=depth[p[x]];u;u=p[u],k--){
        if(u==p[x]){
            id[x][k]=x;
            d[x][k]=z;
        }
        else{
            id[x][k]=id[p[x]][k];
            d[x][k]=d[p[x]][k]+z;
        }
        ans+=order(w[x]-d[x][k],root[u])-order(w[x]-d[x][k],root1[id[x][k]][k]);
        insert(d[x][k]-w[x],root[u]);
        insert(d[x][k]-w[x],root1[id[x][k]][k]);
        size[u]++;
        siz[id[x][k]][k]++;
        if(siz[id[x][k]][k]>size[u]*alpha+5)rt=u;
    }
    id[x][depth[x]]=0;
    d[x][depth[x]]=0;
    if(rt){
        dfs_destroy(rt,depth[rt]);
        rebuild(rt,depth[rt],size[rt],p[rt]);
    }
}
void rebuild(int x,int k,int s,int pr){
    int u=0;
    dfs_getcenter(x,s,u);
    vis[x=u]=true;
    p[x]=pr;
    depth[x]=k;
    size[x]=s;
    d[x][k]=id[x][k]=0;
    destroy(root[x]);
    insert(-w[x],root[x]);
    if(s<=1)return;
    for(int i=0;i<(int)G[x].size();i++)if(!vis[G[x][i]]){
        p[G[x][i]]=0;
        d[G[x][i]][k]=W[x][i];
        siz[G[x][i]][k]=p[G[x][i]]=0;
        destroy(root1[G[x][i]][k]);
        dfs_getdis(G[x][i],x,G[x][i],k);
    }
    for(int i=0;i<(int)G[x].size();i++)if(!vis[G[x][i]])rebuild(G[x][i],k+1,size[G[x][i]],x);
}
void dfs_getcenter(int x,int s,int &u){
    size[x]=1;
    son[x]=0;
    for(int i=0;i<(int)G[x].size();i++)if(!vis[G[x][i]]&&G[x][i]!=p[x]){
        p[G[x][i]]=x;
        dfs_getcenter(G[x][i],s,u);
        size[x]+=size[G[x][i]];
        if(size[G[x][i]]>size[son[x]])son[x]=G[x][i];
    }
    if(!u||max(s-size[x],size[son[x]])<max(s-size[u],size[son[u]
}
void dfs_getdis(int x,int u,int rt,int k){
    insert(d[x][k]-w[x],root[u]);
    insert(d[x][k]-w[x],root1[rt][k]);
    id[x][k]=rt;
    siz[rt][k]++;
    size[x]=1;
    for(int i=0;i<(int)G[x].size();i++)if(!vis[G[x][i]]&&G[x][i]!=p[x]){
        p[G[x][i]]=x;
        d[G[x][i]][k]=d[x][k]+W[x][i];
        dfs_getdis(G[x][i],u,rt,k);
        size[x]+=size[G[x][i]];
    }
}
```

```
void dfs_destroy(int x,int k){
    vis[x]=false;
    for(int i=0;i<(int)G[x].size();i++)if(depth[G[x][i]]>=k&&G[x][i]!=p[x]){
        p[G[x][i]]=x;
        dfs_destroy(G[x][i],k);
    }
}
void insert(int x,node *&rt){
    if(rt==null){
        rt=new node(x);
        rt->ch[0]=rt->ch[1]=null;
        return;
    }
    int d=x>=rt->data;
    insert(x,rt->ch[d]);
    rt->refresh();
    if(rt->ch[d]->p<rt->p)rot(rt,d^1);
}
int order(int x,node *rt){
    int ans=0,d;
    x++;
    while(rt!=null){
        if((d=x>rt->data))ans+=rt->ch[0]->size+1;
        rt=rt->ch[d];
    }
    return ans;
}
void destroy(node *&x){
    if(x==null)return;
    destroy(x->ch[0]);
    destroy(x->ch[1]);
    delete x;
    x=null;
}
void rot(node *&x,int d){
    node *y=x->ch[d^1];
    x->ch[d^1]=y->ch[d];
    y->ch[d]=x;
    x->refresh();
    (x=y)->refresh();
}
```

4.6 LCT

4.6.1 不换根(弹飞绵羊)

```
#define isroot(x) ((x) != (x) -> p -> ch[0] && (x) != (x) -> p -> ch[1]) // 判断是不是Splay的根
#define dir(x) ((x) == (x) -> p -> ch[1]) // 判断它是它父亲的左 / 右儿子

struct node { // 结点类定义
    int size; // Splay的子树大小
    node *ch[2], *p;

    node() : size(1) {}
    void refresh() {
        size = ch[0] -> size + ch[1] -> size + 1;
    } // 附加信息维护
} null[maxn];

// 在主函数开头加上这句初始化
null -> size = 0;

// 初始化结点
void initalize(node *x) {
    x -> ch[0] = x -> ch[1] = x -> p = null;
}

// Access 均摊O(\log n)
// LCT核心操作，把结点到根的路径打通，顺便把与重儿子的连边变成轻边
// 需要调用splay
node *access(node *x) {
    node *y = null;

    while (x != null) {
        splay(x);

        x -> ch[1] = y;
        (y = x) -> refresh();

        x = x -> p;
    }

    return y;
}

// Link 均摊O(\log n)
// 把x的父亲设为y
// 要求x必须为所在树的根节点否则会导致后续各种莫名其妙的问题
// 需要调用splay
void link(node *x, node *y) {
    splay(x);
    x -> p = y;
}

// Cut 均摊O(\log n)
// 把x与其父亲的连边断掉
// x可以是所在树的根节点，这时此操作没有任何实质效果
// 需要调用access和splay
void cut(node *x) {
    access(x);
    splay(x);

    x -> ch[0] -> p = null;
    x -> ch[0] = null;

    x -> refresh();
}

// Splay 均摊O(\log n)
// 需要调用旋转
void splay(node *x) {
    while (!isroot(x)) {
        if (isroot(x -> p)) {
            rot(x -> p, dir(x) ^ 1);
            break;
        }

        if (dir(x) == dir(x -> p))
            rot(x -> p -> p, dir(x -> p) ^ 1);
        else
            rot(x -> p, dir(x) ^ 1);
        rot(x -> p, dir(x) ^ 1);
    }
}

// 旋转(LCT版本) O(1)
// 平衡树基本操作
// 要求对应儿子必须存在，否则会导致后续各种莫名其妙的问题
void rot(node *x, int d) {
    node *y = x -> ch[d ^ 1];

    y -> p = x -> p;
    if (!isroot(x))
```

```cpp
        x -> p -> ch[dir(x)] = y;

    if ((x -> ch[d ^ 1] = y -> ch[d]) != null)
        y -> ch[d] -> p = x;
    (y -> ch[d] = x) -> p = y;

    x -> refresh();
    y -> refresh();
}
```

4.6.2 换根/维护生成树

```cpp
#define isroot(x) ((x) -> p == null || ((x) -> p -> ch[0] != (x) && (x) -> p -> ch[1] != (x)))
#define dir(x) ((x) == (x) -> p -> ch[1])

using namespace std;

const int maxn = 200005;

struct node{
    int key, mx, pos;
    bool rev;
    node *ch[2], *p;

    node(int key = 0): key(key), mx(key), pos(-1), rev(false) {}

    void pushdown() {
        if (!rev)
            return;

        ch[0] -> rev ^= true;
        ch[1] -> rev ^= true;
        swap(ch[0], ch[1]);

        if (pos != -1)
            pos ^= 1;

        rev = false;
    }

    void refresh() {
        mx = key;
        pos = -1;
        if (ch[0] -> mx > mx) {
            mx = ch[0] -> mx;
            pos = 0;
        }
        if (ch[1] -> mx > mx) {
            mx = ch[1] -> mx;
            pos = 1;
        }
    }
} null[maxn * 2];

void init(node *x, int k) {
    x -> ch[0] = x -> ch[1] = x -> p = null;
    x -> key = x -> mx = k;
}

void rot(node *x, int d) {
    node *y = x -> ch[d ^ 1];
    if ((x -> ch[d ^ 1] = y -> ch[d]) != null)
        y -> ch[d] -> p = x;

    y -> p = x -> p;
    if (!isroot(x))
        x -> p -> ch[dir(x)] = y;

    (y -> ch[d] = x) -> p = y;

    x -> refresh();
    y -> refresh();
}

void splay(node *x) {
    x -> pushdown();

    while (!isroot(x)) {
        if (!isroot(x -> p))
            x -> p -> p -> pushdown();
        x -> p -> pushdown();
        x -> pushdown();

        if (isroot(x -> p)) {
            rot(x -> p, dir(x) ^ 1);
            break;
        }

        if (dir(x) == dir(x -> p))
            rot(x -> p -> p, dir(x -> p) ^ 1);
        else
            rot(x -> p, dir(x) ^ 1);

        rot(x -> p, dir(x) ^ 1);
    }
}

node *access(node *x) {
    node *y = null;

    while (x != null) {
        splay(x);

        x -> ch[1] = y;
        (y = x) -> refresh();

        x = x -> p;
    }

    return y;
}

void makeroot(node *x) {
    access(x);
    splay(x);
    x -> rev ^= true;
}

void link(node *x, node *y) {
    makeroot(x);
    x -> p = y;
}

void cut(node *x, node *y) {
    makeroot(x);
    access(y);
    splay(y);

    y -> ch[0] -> p = null;
    y -> ch[0] = null;
    y -> refresh();
}

node *getroot(node *x) {
    x = access(x);
    while (x -> pushdown(), x -> ch[0] != null)
```

```cpp
        x = x -> ch[0];
    splay(x);
    return x;
}

node *getmax(node *x, node *y) {
    makeroot(x);
    x = access(y);

    while (x -> pushdown(), x -> pos != -1)
        x = x -> ch[x -> pos];
    splay(x);

    return x;
}

// 以下为主函数示例
for (int i = 1; i <= m; i++) {
    init(null + n + i, w[i]);
    if (getroot(null + u[i]) != getroot(null + v[i])) {
        ans[q + 1] -= k;
        ans[q + 1] += w[i];

        link(null + u[i], null + n + i);
        link(null + v[i], null + n + i);
        vis[i] = true;
    }
    else {
        int ii = getmax(null + u[i], null + v[i]) - null - n;
        if (w[i] >= w[ii])
            continue;

        cut(null + u[ii], null + n + ii);
        cut(null + v[ii], null + n + ii);

        link(null + u[i], null + n + i);
        link(null + v[i], null + n + i);

        ans[q + 1] -= w[ii];
        ans[q + 1] += w[i];
    }
}
```

4.6.3 维护子树信息

```cpp
// 这个东西虽然只需要抄板子但还是极其难写，常数极其巨大，没必要的时候就不要用
// 如果维护子树最小值就需要套一个可删除的堆来维护，复杂度会变成O(n\log^2 n)
// 注意由于这道题与边权有关，需要边权拆点变点权

// 宏定义
#define isroot(x) ((x) -> p == null || ((x) != (x) -> p -> ch[0]&& (x) != (x) -> p -> ch[1]))
#define dir(x) ((x) == (x) -> p -> ch[1])

// 节点类定义
struct node { // 以维护子树中黑点到根距离和为例
    int w, chain_cnt, tree_cnt;
    long long sum, suml, sumr, tree_sum; // 由于换根需要子树反转，需要维护两个方向的信息
    bool rev, col;
    node *ch[2], *p;

    node() : w(0), chain_cnt(0), tree_cnt(0),sum(0),suml(0), sumr(0),
        tree_sum(0), rev(false), col(false) {}

    inline void pushdown() {
        if(!rev)
            return;

        ch[0]->rev ^= true;
        ch[1]->rev ^= true;
        swap(ch[0], ch[1]);
        swap(suml, sumr);

        rev = false;
    }

    inline void refresh() { // 如果不想这样特判就pushdown一下
        // pushdown();

        sum = ch[0] -> sum + ch[1] -> sum + w;
        suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] -> suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1] -> suml) + (tree_cnt + ch[1] -> chain_cnt) * (ch[0] -> sum + w) + tree_sum;
        sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] -> sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1] -> sumr) + (tree_cnt + ch[0] -> chain_cnt) * (ch[1] -> sum + w) + tree_sum;
        chain_cnt = ch[0] -> chain_cnt + ch[1] -> chain_cnt + tree_cnt;
    }
} null[maxn * 2]; // 如果没有边权变点权就不用乘2了

// 封装构造函数
node *newnode(int w) {
    node *x = nodes.front(); // 因为有删边加边，可以用一个队列维护可用结点
    nodes.pop();
    initalize(x);
    x -> w = w;
    x -> refresh();
    return x;
}

// 封装初始化函数
// 记得在进行操作之前对所有结点调用一遍
inline void initalize(node *x) {
    *x = node();
    x -> ch[0] = x -> ch[1] = x -> p = null;
}

// 注意一下在Access的同时更新子树信息的方法
node *access(node *x) {
    node *y = null;

    while (x != null) {
        splay(x);

        x -> tree_cnt += x -> ch[1] -> chain_cnt - y -> chain_cnt;
        x->tree_sum += (x -> ch[1] -> rev ? x -> ch[1] -> sumr : x -> ch[1] -> suml) - y -> suml;
        x -> ch[1] = y;

        (y = x) -> refresh();
        x = x -> p;
    }

    return y;
}

// 找到一个点所在连通块的根
// 对比原版没有变化
node *getroot(node *x) {
```

```cpp
    x = access(x);

    while (x -> pushdown(), x -> ch[0] != null)
        x = x -> ch[0];
    splay(x);

    return x;
}

// 换根，同样没有变化
void makeroot(node *x) {
    access(x);
    splay(x);
    x -> rev ^= true;
    x -> pushdown();
}

// 连接两个点
// !!! 注意这里必须把两者都变成根，因为只能修改根结点
void link(node *x, node *y) {
    makeroot(x);
    makeroot(y);

    x -> p = y;
    y -> tree_cnt += x -> chain_cnt;
    y -> tree_sum += x -> suml;
    y -> refresh();
}

// 删除一条边
// 对比原版没有变化
void cut(node *x, node *y) {
    makeroot(x);
    access(y);
    splay(y);

    y -> ch[0] -> p = null;
    y -> ch[0] = null;
    y -> refresh();
}

// 修改/询问一个点，这里以询问为例
// 如果是修改就在换根之后搞一些操作
long long query(node *x) {
    makeroot(x);
    return x -> suml;
}

// Splay函数
// 对比原版没有变化
void splay(node *x) {
    x -> pushdown();

    while (!isroot(x)) {
        if (!isroot(x -> p))
            x -> p -> p -> pushdown();
        x -> p -> pushdown();
        x -> pushdown();

        if (isroot(x -> p)) {
            rot(x -> p,dir(x) ^ 1);
            break;
        }

        if (dir(x) == dir(x -> p))
            rot(x -> p -> p, dir(x -> p) ^ 1);
        else
            rot(x -> p, dir(x) ^ 1);

        rot(x -> p, dir(x) ^ 1);
    }
}

// 旋转函数
// 对比原版没有变化
void rot(node *x, int d) {
    node *y = x -> ch[d ^ 1];

    if ((x -> ch[d^1] = y -> ch[d]) != null)
        y -> ch[d] -> p = x;

    y -> p = x -> p;
    if (!isroot(x))
        x -> p -> ch[dir(x)] = y;

    (y -> ch[d] = x) -> p = y;

    x -> refresh();
    y -> refresh();
}
```

4.6.4 模板题:动态QTREE4(询问树上相距最远点)

```cpp
#include<bits/stdc++.h>
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>
#include<ext/pb_ds/priority_queue.hpp>

#define isroot(x) ((x)->p==null||((x)!=(x)->p->ch[0]&&(x)!=(x)->p->ch[1]))
#define dir(x) ((x)==(x)->p->ch[1])

using namespace std;
using namespace __gnu_pbds;

const int maxn=100010;
const long long INF=100000000000000000ll;

struct binary_heap{
    __gnu_pbds::priority_queue<long long,less<long long>,binary_heap_tag>q1,q2;
    binary_heap(){}
    void push(long long x){if(x>(-INF)>>2)q1.push(x);}
    void erase(long long x){if(x>(-INF)>>2)q2.push(x);}
    long long top(){
        if(empty())return -INF;
        while(!q2.empty()&&q1.top()==q2.top()){
            q1.pop();
            q2.pop();
        }
        return q1.top();
    }
    long long top2(){
        if(size()<2)return -INF;
        long long a=top();
        erase(a);
        long long b=top();
        push(a);
        return a+b;
    }
    int size(){return q1.size()-q2.size();}
    bool empty(){return q1.size()==q2.size();}
}heap;//全局堆维护每条链的最大子段和
struct node{
    long long sum,maxsum,prefix,suffix;
    int key;
    binary_heap heap;//每个点的堆存的是它的子树中到它的最远距离，如果它是黑点的话还会包括自己
    node *ch[2],*p;
    bool rev;
```

```
    node(int k=0):sum(k),maxsum(-INF),prefix(-INF),
        suffix(-INF),key(k),rev(false){}
    inline void pushdown(){
        if(!rev)return;
        ch[0]->rev^=true;
        ch[1]->rev^=true;
        swap(ch[0],ch[1]);
        swap(prefix,suffix);
        rev=false;
    }
    inline void refresh(){
        pushdown();
        ch[0]->pushdown();
        ch[1]->pushdown();
        sum=ch[0]->sum+ch[1]->sum+key;
        prefix=max(ch[0]->prefix,
            ch[0]->sum+key+ch[1]->prefix);
        suffix=max(ch[1]->suffix,
            ch[1]->sum+key+ch[0]->suffix);
        maxsum=max(max(ch[0]->maxsum,ch[1]->maxsum),
            ch[0]->suffix+key+ch[1]->prefix);
        if(!heap.empty()){
            prefix=max(prefix,
                ch[0]->sum+key+heap.top());
            suffix=max(suffix,
                ch[1]->sum+key+heap.top());
            maxsum=max(maxsum,max(ch[0]->suffix,
                ch[1]->prefix)+key+heap.top());
            if(heap.size()>1){
                maxsum=max(maxsum,heap.top2()+key);
            }
        }
    }
}null[maxn<<1],*ptr=null;
void addedge(int,int,int);
void deledge(int,int);
void modify(int,int,int);
void modify_color(int);
node *newnode(int);
node *access(node*);
void makeroot(node*);
void link(node*,node*);
void cut(node*,node*);
void splay(node*);
void rot(node*,int);
queue<node*>freenodes;
tree<pair<int,int>,node*>mp;
bool col[maxn]={false};
char c;
int n,m,k,x,y,z;
int main(){
    null->ch[0]=null->ch[1]=null->p=null;
    scanf("%d%d%d",&n,&m,&k);
    for(int i=1;i<=n;i++){
        newnode(0);
    }
    heap.push(0);
    while(k--){
        scanf("%d",&x);
        col[x]=true;
        null[x].heap.push(0);
    }
    for(int i=1;i<n;i++){
        scanf("%d%d%d",&x,&y,&z);
        if(x>y)swap(x,y);
        addedge(x,y,z);
    }
    while(m--){
        scanf(" %c%d",&c,&x);
        if(c=='A'){
            scanf("%d",&y);
            if(x>y)swap(x,y);
            deledge(x,y);
        }
        else if(c=='B'){
            scanf("%d%d",&y,&z);
            if(x>y)swap(x,y);
            addedge(x,y,z);
        }
        else if(c=='C'){
            scanf("%d%d",&y,&z);
            if(x>y)swap(x,y);
            modify(x,y,z);
        }
        else modify_color(x);
        printf("%lld\n",(heap.top()>0?heap.top():-1));
    }
    return 0;
}
void addedge(int x,int y,int z){
    node *tmp;
    if(freenodes.empty())tmp=newnode(z);
    else{
        tmp=freenodes.front();
        freenodes.pop();
        *tmp=node(z);
    }
    tmp->ch[0]=tmp->ch[1]=tmp->p=null;
    heap.push(tmp->maxsum);
    link(tmp,null+x);
    link(tmp,null+y);
    mp[make_pair(x,y)]=tmp;
}
void deledge(int x,int y){
    node *tmp=mp[make_pair(x,y)];
    cut(tmp,null+x);
    cut(tmp,null+y);
    freenodes.push(tmp);
    heap.erase(tmp->maxsum);
    mp.erase(make_pair(x,y));
}
void modify(int x,int y,int z){
    node *tmp=mp[make_pair(x,y)];
    makeroot(tmp);
    tmp->pushdown();
    heap.erase(tmp->maxsum);
    tmp->key=z;
    tmp->refresh();
    heap.push(tmp->maxsum);
}
void modify_color(int x){
    makeroot(null+x);
    col[x]^=true;
    if(col[x])null[x].heap.push(0);
    else null[x].heap.erase(0);
    heap.erase(null[x].maxsum);
    null[x].refresh();
    heap.push(null[x].maxsum);
}
node *newnode(int k){
    *(++ptr)=node(k);
    ptr->ch[0]=ptr->ch[1]=ptr->p=null;
    return ptr;
}
node *access(node *x){
    splay(x);
    heap.erase(x->maxsum);
    x->refresh();
    if(x->ch[1]!=null){
        x->ch[1]->pushdown();
        x->heap.push(x->ch[1]->prefix);
        x->refresh();
        heap.push(x->ch[1]->maxsum);
    }
```

```cpp
    x->ch[1]=null;
    x->refresh();
    node *y=x;
    x=x->p;
    while(x!=null){
        splay(x);
        heap.erase(x->maxsum);
        if(x->ch[1]!=null){
            x->ch[1]->pushdown();
            x->heap.push(x->ch[1]->prefix);
            heap.push(x->ch[1]->maxsum);
        }
        x->heap.erase(y->prefix);
        x->ch[1]=y;
        (y=x)->refresh();
        x=x->p;
    }
    heap.push(y->maxsum);
    return y;
}
void makeroot(node *x){
    access(x);
    splay(x);
    x->rev^=true;
}
void link(node *x,node *y){//新添一条虚边⊠维护y对应的堆
    makeroot(x);
    makeroot(y);
    x->pushdown();
    x->p=y;
    heap.erase(y->maxsum);
    y->heap.push(x->prefix);
    y->refresh();
    heap.push(y->maxsum);
}
void cut(node *x,node *y){//断开一条实边⊠一条链变成两条链⊠需要维护全局堆
    makeroot(x);
    access(y);
    splay(y);
    heap.erase(y->maxsum);
    heap.push(y->ch[0]->maxsum);
    y->ch[0]->p=null;
    y->ch[0]=null;
    y->refresh();
    heap.push(y->maxsum);
}
void splay(node *x){
    x->pushdown();
    while(!isroot(x)){
        if(!isroot(x->p))
            x->p->p->pushdown();
        x->p->pushdown();
        x->pushdown();
        if(isroot(x->p)){
            rot(x->p,dir(x)^1);
            break;
        }
        if(dir(x)==dir(x->p))
            rot(x->p->p,dir(x->p)^1);
        else rot(x->p,dir(x)^1);
        rot(x->p,dir(x)^1);
    }
}
void rot(node *x,int d){
    node *y=x->ch[d^1];
    if((x->ch[d^1]=y->ch[d])!=null)
        y->ch[d]->p=x;
    y->p=x->p;
    if(!isroot(x))
        x->p->ch[dir(x)]=y;
    (y->ch[d]=x)->p=y;
    x->refresh();
    y->refresh();
}
```

4.7 虚树

```cpp
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<vector>
using namespace std;
const int maxn=1000005;
struct Tree{
    vector<int>G[maxn],W[maxn];
    int p[maxn],d[maxn],size[maxn],mn[maxn],mx[maxn];
    bool col[maxn];
    long long ans_sum;
    int ans_min,ans_max;
    void add(int x,int y,int z){
        G[x].push_back(y);
        W[x].push_back(z);
    }
    void dfs(int x){
        size[x]=col[x];
        mx[x]=(col[x]?d[x]:-0x3f3f3f3f);
        mn[x]=(col[x]?d[x]:0x3f3f3f3f);
        for(int i=0;i<(int)G[x].size();i++){
            d[G[x][i]]=d[x]+W[x][i];
            dfs(G[x][i]);
            ans_sum+=(long long)size[x]*size[G[x][i]]*d[x];
            ans_max=max(ans_max,mx[x]+mx[G[x][i]]-(d[x]<<1));
            ans_min=min(ans_min,mn[x]+mn[G[x][i]]-(d[x]<<1));
            size[x]+=size[G[x][i]];
            mx[x]=max(mx[x],mx[G[x][i]]);
            mn[x]=min(mn[x],mn[G[x][i]]);
        }
    }
    void clear(int x){
        G[x].clear();
        W[x].clear();
        col[x]=false;
    }
    void solve(int rt){
        ans_sum=0;
        ans_max=1<<31;
        ans_min=(~0u)>>1;
        dfs(rt);
        ans_sum<<=1;
    }
}virtree;
void dfs(int);
int LCA(int,int);
vector<int>G[maxn];
int f[maxn][20],d[maxn],dfn[maxn],tim=0;
bool cmp(int x,int y){return dfn[x]<dfn[y];}
int n,m,lgn=0,a[maxn],s[maxn],v[maxn];
int main(){
    scanf("%d",&n);
    for(int i=1,x,y;i<n;i++){
        scanf("%d%d",&x,&y);
        G[x].push_back(y);
        G[y].push_back(x);
    }
    G[n+1].push_back(1);
    dfs(n+1);
```

```cpp
    for(int i=1;i<=n+1;i++)G[i].clear();
    lgn--;
    for(int j=1;j<=lgn;j++)for(int i=1;i<=n;i++)f[i]
      [j]=f[f[i][j-1]][j-1];
    scanf("%d",&m);
    while(m--){
        int k;
        scanf("%d",&k);
        for(int i=1;i<=k;i++)scanf("%d",&a[i]);
        sort(a+1,a+k+1,cmp);
        int top=0,cnt=0;
        s[++top]=v[++cnt]=n+1;
        long long ans=0;
        for(int i=1;i<=k;i++){
            virtree.col[a[i]]=true;
            ans+=d[a[i]]-1;
            int u=LCA(a[i],s[top]);
            if(s[top]!=u){
                while(top>1&&d[s[top-1]]>=d[u]){
                    virtree.add(s[top-1],s[top],d[s[top]]-d[s[top-1]]);
                    top--;
                }
                if(s[top]!=u){
                    virtree.add(u,s[top],d[s[top]]-d[u]);
                    s[top]=v[++cnt]=u;
                }
            }
            s[++top]=a[i];
        }
        for(int
          i=top-1;i;i--)virtree.add(s[i],s[i+1],d[s[i+1]]-d[s[i]]);
        virtree.solve(n+1);
        ans*=k-1;
        printf("%lld %d
          %d\n",ans-virtree.ans_sum,virtree.ans_min,virtree.ans_max);
        for(int i=1;i<=k;i++)virtree.clear(a[i]);
        for(int i=1;i<=cnt;i++)virtree.clear(v[i]);

    }
    return 0;
}
void dfs(int x){
    dfn[x]=++tim;
    d[x]=d[f[x][0]]+1;
    while((1<<lgn)<d[x])lgn++;
    for(int i=0;i<(int)G[x].size();i++)if(G[x][i]!=f[x]
      [0]){
        f[G[x][i]][0]=x;
        dfs(G[x][i]);
    }
}
int LCA(int x,int y){
    if(d[x]!=d[y]){
        if(d[x]<d[y])swap(x,y);
        for(int
          i=lgn;i>=0;i--)if(((d[x]-d[y])>>i)&1)x=f[x]
          [i];
    }
    if(x==y)return x;
    for(int i=lgn;i>=0;i--)if(f[x][i]!=f[y][i]){
        x=f[x][i];
        y=f[y][i];
    }
    return f[x][0];
}
```

4.8 长链剖分

```cpp
// 顾名思义，长链剖分是取最深的儿子作为重儿子

// O(n)维护以深度为下标的子树信息
vector<int> G[maxn], v[maxn];
int n, p[maxn], h[maxn], son[maxn], ans[maxn];

// 原题题意：求每个点的子树中与它距离是几的点最多,相同的
  取最大深度
// 由于vector只能在后面加入元素，为了写代码方便，这里反
  过来存
void dfs(int x) {
    h[x] = 1;

    for (int y : G[x])
        if (y != p[x]){
            p[y] = x;
            dfs(y);

            if (h[y] > h[son[x]])
                son[x] = y;
        }

    if (!son[x]) {
        v[x].push_back(1);
        ans[x] = 0;
        return;
    }

    h[x] = h[son[x]] + 1;
    swap(v[x],v[son[x]]);

    if (v[x][ans[son[x]]] == 1)
        ans[x] = h[x] - 1;
    else
        ans[x] = ans[son[x]];

    v[x].push_back(1);

    int mx = v[x][ans[x]];
    for (int y : G[x])
        if (y != p[x] && y != son[x]) {
            for (int j = 1; j <= h[y]; j++) {
                v[x][h[x] - j - 1] += v[y][h[y] - j];

                int t = v[x][h[x] - j - 1];
                if (t > mx || (t == mx && h[x] - j - 1 >
                  ans[x])) {
                    mx = t;
                    ans[x] = h[x] - j - 1;
                }
            }

            v[y].clear();
        }
}
```

4.9 梯子剖分

```cpp
// 在线求一个点的第k祖先 O(n\log n)-O(1)
// 理论基础：任意一个点x的k级祖先y所在长链长度一定>=k

// 全局数组定义
vector<int> G[maxn], v[maxn];
int d[maxn], mxd[maxn], son[maxn], top[maxn], len[maxn];
int f[19][maxn], log_tbl[maxn];

// 在主函数中两遍dfs之后加上如下预处理
```

```cpp
log_tbl[0] = -1;
for (int i = 1; i <= n; i++)
    log_tbl[i] = log_tbl[i / 2] + 1;
for (int j = 1; (1 << j) < n; j++)
    for (int i = 1; i <= n; i++)
        f[j][i] = f[j - 1][f[j - 1][i]];

// 第一遍dfs, 用于计算深度和找出重儿子
void dfs1(int x) {
    mxd[x] = d[x];

    for (int y : G[x])
        if (y != f[0][x]){
            f[0][y] = x;
            d[y] = d[x] + 1;

            dfs1(y);

            mxd[x] = max(mxd[x], mxd[y]);
            if (mxd[y] > mxd[son[x]])
                son[x] = y;
        }
}

// 第二遍dfs, 用于进行剖分和预处理梯子剖分(每条链向上延伸一倍)数组
void dfs2(int x) {
    top[x] = (x == son[f[0][x]] ? top[f[0][x]] : x);

    for (int y : G[x])
        if (y != f[0][x])
            dfs2(y);

    if (top[x] == x) {
        int u = x;
        while (top[son[u]] == x)
            u = son[u];

        len[x] = d[u] - d[x];
        for (int i = 0; i < len[x]; i++, u = f[0][u])
            v[x].push_back(u);

        u = x;
        for (int i = 0; i < len[x] && u; i++, u = f[0][u])
            v[x].push_back(u);
    }
}

// 在线询问x的k级祖先 O(1)
// 不存在时返回0
int query(int x, int k) {
    if (!k)
        return x;
    if (k > d[x])
        return 0;

    x = f[log_tbl[k]][x];
    k ^= 1 << log_tbl[k];
    return v[top[x]][d[top[x]] + len[top[x]] - d[x] + k];
}
```

4.10 左偏树

(参见k短路)

4.11 常见根号思路

通用

- 出现次数大于$\sqrt{n}$的数不会超过$\sqrt{n}$个
- 对于带修改问题, 如果不方便分治或者二进制分组, 可以考虑对操作分块, 每次查询时暴力最后的$\sqrt{n}$个修改并更正答案
- **根号分治**: 如果分治时每个子问题需要$O(N)$(N是全局问题的大小)的时间, 而规模较小的子问题可以$O(n^2)$解决, 则可以使用根号分治
  - 规模大于$\sqrt{n}$的子问题用$O(N)$的方法解决, 规模小于$\sqrt{n}$的子问题用$O(n^2)$暴力
  - 规模大于$\sqrt{n}$的子问题最多只有$\sqrt{n}$个
  - 规模不大于$\sqrt{n}$的子问题大小的平方和也必定不会超过$n\sqrt{n}$
- 如果输入规模之和不大于n(例如给定多个小字符串与大字符串进行询问), 那么规模超过$\sqrt{n}$的问题最多只有$\sqrt{n}$个

序列

- 某些维护序列的问题可以用分块/块状链表维护
- 对于静态区间询问问题, 如果可以快速将左/右端点移动一位, 可以考虑莫队
  - 如果强制在线可以分块预处理, 但是一般空间需要$n\sqrt{n}$
    - 例题: 询问区间中有几种数出现次数恰好为k, 强制在线
  - 如果带修改可以试着想一想带修莫队, 但是复杂度高达$n^{\frac{5}{3}}$
- 线段树可以解决的问题也可以用分块来做到$O(1)$询问或是$O(1)$修改, 具体要看哪种操作更多

树

- 与序列类似, 树上也有树分块和树上莫队
  - 树上带修莫队很麻烦, 常数也大, 最好不要先考虑
  - 树分块不要想当然
- 树分治也可以套根号分治, 道理是一样的

字符串

- 循环节长度大于$\sqrt{n}$的子串最多只有$O(n)$个, 如果是极长子串则只有$O(\sqrt{n})$个

5. 字符串

5.1 KMP

```cpp
char s[maxn], t[maxn];
int fail[maxn];
int n, m;

void init() {
    // memset(fail, 0, sizeof(fail));

    for (int i = 1; i < m; i++) {
        int j = fail[i];
        while (j && t[i] != t[j])
            j = fail[j];

        if (t[i] == t[j])
```

```cpp
            fail[i + 1] = j + 1;
        else
            fail[i + 1] = 0;
    }
}

int KMP() {
    int cnt = 0, j = 0;

    for (int i = 0; i < n; i++) {
        while (j && s[i] != t[j])
            j = fail[j];

        if (s[i] == t[j])
            j++;
        if (j == m)
            cnt++;
    }

    return cnt;
}
```

5.1.1 ex-KMP

```cpp
//全局变量与数组定义
char s[maxn], t[maxn];
int n, m, a[maxn];

// 主过程 O(n + m)
// 把t的每个后缀与s的LCP输出到a中, s的后缀和自己的LCP存在nx中
// 0-based, s的长度是m, t的长度是n
void exKMP(const char *s, const char *t, int *a) {
    static int nx[maxn];

    memset(nx, 0, sizeof(nx));

    int j = 0;
    while (j + 1 < m && s[j] == s[j + 1])
        j++;
    nx[1] = j;

    for (int i = 2, k = 1;i < m; i++) {
        int pos = k + nx[k], len = nx[i - k];

        if (i + len < pos)
            nx[i] = len;
        else {
            j = max(pos - i, 0);
            while (i + j < m && s[j] == s[i + j])
                j++;

            nx[i] = j;
            k = i;
        }
    }

    j = 0;
    while (j < n && j < m && s[j] == t[j])
        j++;
    a[0] = j;

    for (int i = 1, k = 0; i < n; i++) {
        int pos = k + a[k], len = nx[i - k];
        if (i + len < pos)
            a[i] = len;
        else {
            j = max(pos - i, 0);
            while(j < m && i + j < n && s[j] == t[i + j])
                j++;

            a[i] = j;
            k = i;
        }
    }
}
```

5.2 AC自动机

```cpp
// Aho-Corasick Automata AC自动机
// By AntiLeaf
// 通过题目：bzoj3881 Divljak


// 全局变量与数组定义
int ch[maxm][26] = {{0}}, f[maxm][26] = {{0}}, q[maxm] = {0}, sum[maxm] = {0}, cnt = 0;


// 在字典树中插入一个字符串 O(n)
int insert(const char *c) {
    int x = 0;
    while (*c) {
        if (!ch[x][*c - 'a'])
            ch[x][*c - 'a'] = ++cnt;
        x = ch[x][*c++ - 'a'];
    }
    return x;
}


// 建AC自动机 O(n*sigma)
void getfail() {
    int x, head = 0, tail = 0;

    for (int c = 0; c < 26; c++)
        if (ch[0][c])
            q[tail++] = ch[0][c]; // 把根节点的儿子加入队列

    while (head != tail) {
        x = q[head++];

        G[f[x][0]].push_back(x);
        fill(f[x] + 1, f[x] + 26, cnt + 1);

        for (int c = 0; c < 26; c++) {
            if (ch[x][c]) {
                int y = f[x][0];

                while (y&&!ch[y][c])
                    y=f[y][0];

                f[ch[x][c]][0] = ch[y][c];
                q[tail++] = ch[x][c];
            }
            else
                ch[x][c] = ch[f[x][0]][c];
        }
    }
    fill(f[0], f[0] + 26, cnt + 1);
}
```

5.3 后缀数组

5.3.1 SA-IS

```cpp
// 注意求完的SA有效位只有1~n，但它是0-based，如果其他部分是1-based记得+1再用

constexpr int maxn = 100005, l_type = 0, s_type = 1;

// 判断一个字符是否为LMS字符
bool is_lms(int *tp, int x) {
    return x > 0 && tp[x] == s_type && tp[x - 1] == l_type;
}

// 判断两个LMS子串是否相同
bool equal_substr(int *s, int x, int y, int *tp) {
    do {
        if (s[x] != s[y])
            return false;
        x++;
        y++;
    } while (!is_lms(tp, x) && !is_lms(tp, y));

    return s[x] == s[y];
}

// 诱导排序(从*型诱导到L型,从L型诱导到S型)
// 调用之前应将*型按要求放入SA中
void induced_sort(int *s, int *sa, int *tp, int *buc, int *lbuc, int *sbuc, int n, int m) {
    for (int i = 0; i <= n; i++)
        if (sa[i] > 0 && tp[sa[i] - 1] == l_type)
            sa[lbuc[s[sa[i] - 1]]++] = sa[i] - 1;

    for (int i = 1; i <= m; i++)
        sbuc[i] = buc[i] - 1;

    for (int i = n; ~i; i--)
        if (sa[i] > 0 && tp[sa[i] - 1] == s_type)
            sa[sbuc[s[sa[i] - 1]]--] = sa[i] - 1;
}

// s是输入字符串，n是字符串的长度，m是字符集的大小
int *sais(int *s, int len, int m) {
    int n = len - 1;

    int *tp = new int[n + 1];
    int *pos = new int[n + 1];
    int *name = new int[n + 1];
    int *sa = new int[n + 1];
    int *buc = new int[m + 1];
    int *lbuc = new int[m + 1];
    int *sbuc = new int[m + 1];

    memset(buc, 0, sizeof(int) * (m + 1));

    for (int i = 0; i <= n; i++)
        buc[s[i]]++;

    for (int i = 1; i <= m; i++) {
        buc[i] += buc[i - 1];

        lbuc[i] = buc[i - 1];
        sbuc[i] = buc[i] - 1;
    }

    tp[n] = s_type;
    for (int i = n - 1; ~i; i--) {
        if (s[i] < s[i + 1])
            tp[i] = s_type;
        else if (s[i] > s[i + 1])
            tp[i] = l_type;
        else
            tp[i] = tp[i + 1];
    }

    int cnt = 0;
    for (int i = 1; i <= n; i++)
        if (tp[i] == s_type && tp[i - 1] == l_type)
            pos[cnt++] = i;

    memset(sa, -1, sizeof(int) * (n + 1));
    for (int i = 0; i < cnt; i++)
        sa[sbuc[s[pos[i]]]--] = pos[i];
    induced_sort(s, sa, tp, buc, lbuc, sbuc, n, m);

    memset(name, -1, sizeof(int) * (n + 1));
    int lastx = -1, namecnt = 1;
    bool flag = false;

    for (int i = 1; i <= n; i++) {
        int x = sa[i];

        if (is_lms(tp, x)) {
            if (lastx >= 0 && !equal_substr(s, x, lastx, tp))
                namecnt++;

            if (lastx >= 0 && namecnt == name[lastx])
                flag = true;

            name[x] = namecnt;
            lastx = x;
        }
    }
    name[n] = 0;

    int *t = new int[cnt];
    int p = 0;
    for (int i = 0; i <= n; i++)
        if (name[i] >= 0)
            t[p++] = name[i];

    int *tsa;
    if (!flag) {
        tsa = new int[cnt];

        for (int i = 0; i < cnt; i++)
            tsa[t[i]] = i;
    }
    else
        tsa = sais(t, cnt, namecnt);

    lbuc[0] = sbuc[0] = 0;
    for (int i = 1; i <= m; i++) {
        lbuc[i] = buc[i - 1];
        sbuc[i] = buc[i] - 1;
    }

    memset(sa, -1, sizeof(int) * (n + 1));
    for (int i = cnt - 1; ~i; i--)
        sa[sbuc[s[pos[tsa[i]]]]--] = pos[tsa[i]];
    induced_sort(s, sa, tp, buc, lbuc, sbuc, n, m);

    return sa;
}

// O(n)求height数组，注意是sa[i]与sa[i - 1]的LCP
```

```cpp
void get_height(int *s, int *sa, int *rnk, int *height, int n) {
    for (int i = 0; i <= n; i++)
        rnk[sa[i]] = i;

    int k = 0;
    for (int i = 0; i <= n; i++) {
        if (!rnk[i])
            continue;

        if (k)
            k--;

        while (s[sa[rnk[i]] + k] == s[sa[rnk[i] - 1] + k])
            k++;

        height[rnk[i]] = k;
    }
}

char str[maxn];
int n, s[maxn], sa[maxn], rnk[maxn], height[maxn];

// 方便起见附上主函数
int main() {
    scanf("%s", str);
    n = strlen(str);
    str[n] = '$';

    for (int i = 0; i <= n; i++)
        s[i] = str[i];

    memcpy(sa, sais(s, n + 1, 256), sizeof(int) * (n + 1));

    get_height(s, sa, rnk, height, n);

    return 0;
}
```

5.3.2 SAMSA

```cpp
bool vis[maxn * 2];
char s[maxn];
int n, id[maxn * 2], ch[maxn * 2][26], height[maxn], tim = 0;

void dfs(int x) {
    if (id[x]) {
        height[tim++] = val[last];
        sa[tim] = id[x];

        last = x;
    }

    for (int c = 0; c < 26; c++)
        if (ch[x][c])
            dfs(ch[x][c]);

    last = par[x];
}

int main() {
    last = ++cnt;

    scanf("%s", s + 1);
    n = strlen(s + 1);

    for (int i = n; i; i--) {
        expand(s[i] - 'a');
        id[last] = i;
    }

    vis[1] = true;
    for (int i = 1; i <= cnt; i++)
        if (id[i])
            for (int x = i, pos = n; x && !vis[x]; x = par[x]) {
                vis[x] = true;
                pos -= val[x] - val[par[x]];
                ch[par[x]][s[pos + 1] - 'a'] = x;
            }

    dfs(1);

    for (int i = 1; i <= n; i++) {
        if (i > 1)
            printf(" ");
        printf("%d", sa[i]); // 1-based
    }
    printf("\n");

    for (int i = 1; i < n; i++) {
        if (i > 1)
            printf(" ");
        printf("%d", height[i]);
    }
    printf("\n");

    return 0;
}
```

5.4 后缀自动机

(广义后缀自动机复杂度就是$O(n|\Sigma|)$，也没法做到更低了)

```cpp
// 在字符集比较小的时候可以直接开go数组，否则需要用map或者哈希表替换
// 注意!!!结点数要开成串长的两倍

// 全局变量与数组定义
int last, val[maxn], par[maxn], go[maxn][26], cnt;
int c[maxn], q[maxn]; // 用来桶排序

// 在主函数开头加上这句初始化
last = cnt = 1;

// 以下是按val进行桶排序的代码
for (int i = 1; i <= cnt; i++)
    c[val[i] + 1]++;
for (int i = 1; i <= n; i++)
    c[i] += c[i - 1]; // 这里n是串长
for (int i = 1; i <= cnt; i++)
    q[++c[val[i]]] = i;

//加入一个字符 均摊O(1)
void extend(int c) {
    int p = last, np = ++cnt;
    val[np] = val[p] + 1;

    while (p && !go[p][c]) {
        go[p][c] = np;
        p = par[p];
    }

    if (!p)
        par[np] = 1;
```

```
    else {
        int q = go[p][c];

        if (val[q] == val[p] + 1)
            par[np] = q;
        else {
            int nq = ++cnt;
            val[nq] = val[p] + 1;
            memcpy(go[nq], go[q], sizeof(go[q]));

            par[nq] = par[q];
            par[np] = par[q] = nq;

            while (p && go[p][c] == q){
                go[p][c] = nq;
                p = par[p];
            }
        }
    }

    last = np;
}
```

5.5 回文树

```
// 定理: 一个字符串本质不同的回文子串个数是O(n)的
// 注意回文树只需要开一倍结点, 另外结点编号也是一个可用的bfs序

// 全局数组定义
int val[maxn], par[maxn], go[maxn][26], last, cnt;
char s[maxn];

// 重要!在主函数最前面一定要加上以下初始化
par[0] = cnt = 1;
val[1] = -1;
// 这个初始化和广义回文树不一样, 写普通题可以用, 广义回文树就不要乱搞了

// extend函数 均摊O(1)
// 向后扩展一个字符
// 传入对应下标
void extend(int n) {
    int p = last, c = s[n] - 'a';
    while (s[n - val[p] - 1] != s[n])
        p = par[p];

    if (!go[p][c]) {
        int q = ++cnt, now = p;
        val[q] = val[p] + 2;

        do
            p=par[p];
        while (s[n - val[p] - 1] != s[n]);

        par[q] = go[p][c];
        last = go[now][c] = q;
    }
    else
        last = go[p][c];

    // a[last]++;
}
```

5.5.1 广义回文树

(代码是梯子剖分的版本,压力不大的题目换成直接倍增就好了,常数只差不到一倍)

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

using namespace std;

constexpr int maxn = 1000005, mod = 1000000007;

int val[maxn], par[maxn], go[maxn][26], fail[maxn][26], pam_last[maxn], pam_cnt;
int weight[maxn], pow_26[maxn];

int trie[maxn][26], trie_cnt, d[maxn], mxd[maxn], son[maxn], top[maxn], len[maxn], sum[maxn];
char chr[maxn];
int f[25][maxn], log_tbl[maxn];
vector<int> v[maxn];

vector<int> queries[maxn];

char str[maxn];
int n, m, ans[maxn];

int add(int x, int c) {
    if (!trie[x][c]) {
        trie[x][c] = ++trie_cnt;
        f[0][trie[x][c]] = x;
        chr[trie[x][c]] = c + 'a';
    }

    return trie[x][c];
}

int del(int x) {
    return f[0][x];
}

void dfs1(int x) {
    mxd[x] = d[x] = d[f[0][x]] + 1;

    for (int i = 0; i < 26; i++)
        if (trie[x][i]) {
            int y = trie[x][i];

            dfs1(y);

            mxd[x] = max(mxd[x], mxd[y]);
            if (mxd[y] > mxd[son[x]])
                son[x] = y;
        }
}

void dfs2(int x) {
    if (x == son[f[0][x]])
        top[x] = top[f[0][x]];
    else
        top[x] = x;

    for (int i = 0; i < 26; i++)
        if (trie[x][i]) {
            int y = trie[x][i];
            dfs2(y);
        }

    if (top[x] == x) {
        int u = x;
        while (top[son[u]] == x)
            u = son[u];

        len[x] = d[u] - d[x];

        for (int i = 0; i < len[x]; i++) {
            v[x].push_back(u);
```

```
            u = f[0][u];
        }

        u = x;
        for (int i = 0; i < len[x]; i++) { // 梯子剖分,要延长一倍
            v[x].push_back(u);
            u = f[0][u];
        }
    }
}

int get_anc(int x, int k) {
    if (!k)
        return x;
    if (k > d[x])
        return 0;

    x = f[log_tbl[k]][x];
    k ^= 1 << log_tbl[k];

    return v[top[x]][d[top[x]] + len[top[x]] - d[x] + k];
}

char get_char(int x, int k) { // 查询x前面k个的字符是哪个
    return chr[get_anc(x, k)];
}

int getfail(int x, int p) {
    if (get_char(x, val[p] + 1) == chr[x])
        return p;
    return fail[p][chr[x] - 'a'];
}

int extend(int x) {

    int p = pam_last[f[0][x]], c = chr[x] - 'a';

    p = getfail(x, p);

    int new_last;

    if (!go[p][c]) {
        int q = ++pam_cnt, now = p;
        val[q] = val[p] + 2;

        p = getfail(x, par[p]);

        par[q] = go[p][c];
        new_last = go[now][c] = q;

        for (int i = 0; i < 26; i++)
            fail[q][i] = fail[par[q]][i];

        if (get_char(x, val[par[q]]) >= 'a')
            fail[q][get_char(x, val[par[q]]) - 'a'] = par[q];

        if (val[q] <= n)
            weight[q] = (weight[par[q]] + (long long)(n - val[q] + 1) * pow_26[n - val[q]]) % mod;
        else
            weight[q] = weight[par[q]];
    }
    else
        new_last = go[p][c];

    pam_last[x] = new_last;

    return weight[pam_last[x]];
}

void bfs() {

    queue<int> q;

    q.push(1);

    while (!q.empty()) {
        int x = q.front();
        q.pop();

        sum[x] = sum[f[0][x]];
        if (x > 1)
            sum[x] = (sum[x] + extend(x)) % mod;

        for (int i : queries[x])
            ans[i] = sum[x];

        for (int i = 0; i < 26; i++)
            if (trie[x][i])
                q.push(trie[x][i]);
    }

}

int main() {

    pow_26[0] = 1;
    log_tbl[0] = -1;

    for (int i = 1; i <= 1000000; i++) {
        pow_26[i] = 26ll * pow_26[i - 1] % mod;
        log_tbl[i] = log_tbl[i / 2] + 1;
    }

    int T;
    scanf("%d", &T);

    while (T--) {
        scanf("%d%d%s", &n, &m, str);

        trie_cnt = 1;
        chr[1] = '#';

        int last = 1;
        for (char *c = str; *c; c++)
            last = add(last, *c - 'a');

        queries[last].push_back(0);

        for (int i = 1; i <= m; i++) {
            int op;
            scanf("%d", &op);

            if (op == 1) {
                char c;
                scanf(" %c", &c);

                last = add(last, c - 'a');
            }
            else
                last = del(last);

            queries[last].push_back(i);
        }

        dfs1(1);
        dfs2(1);

        for (int j = 1; j <= log_tbl[trie_cnt]; j++)
            for (int i = 1; i <= trie_cnt; i++)
                f[j][i] = f[j - 1][f[j - 1][i]];

```

```
        par[0] = pam_cnt = 1;


        for (int i = 0; i < 26; i++)
            fail[0][i] = fail[1][i] = 1;

        val[1] = -1;
        pam_last[1] = 1;

        bfs();

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

        for (int j = 0; j <= log_tbl[trie_cnt]; j++)
            memset(f[j], 0, sizeof(f[j]));

        for (int i = 1; i <= trie_cnt; i++) {
            chr[i] = 0;
            d[i] = mxd[i] = son[i] = top[i] = len[i] = pam_last[i] = sum[i] = 0;
            v[i].clear();
            queries[i].clear();

            memset(trie[i], 0, sizeof(trie[i]));
        }
        trie_cnt = 0;

        for (int i = 0; i <= pam_cnt; i++) {
            val[i] = par[i] = weight[i];

            memset(go[i], 0, sizeof(go[i]));
            memset(fail[i], 0, sizeof(fail[i]));
        }
        pam_cnt = 0;

    }

    return 0;
}
```

5.6 Manacher马拉车

```
//n为串长,回文半径输出到p数组中
//数组要开串长的两倍
void manacher(const char *t, int n) {
    static char s[maxn * 2];

    for (int i = n; i; i--)
        s[i * 2] = t[i];
    for (int i = 0; i <= n; i++)
        s[i * 2 + 1]='#';

    s[0] = '$';
    s[(n + 1) * 2] = '\0';
    n = n * 2 + 1;

    int mx = 0, j = 0;

    for (int i = 1; i <= n; i++) {
        p[i] = (mx > i ? min(p[j * 2 - i], mx - i) : 1);
        while (s[i - p[i]] == s[i + p[i]])
            p[i]++;

        if (i + p[i] > mx) {
            mx = i + p[i];
            j = i;
        }
    }
}
```

5.7 字符串原理

KMP和AC自动机的fail指针存储的都是它在串或者字典树上的最长后缀，因此要判断两个前缀是否互为后缀时可以直接用fail指针判断．当然它不能做子串问题，也不能做最长公共后缀．

后缀数组利用的主要是LCP长度可以按照字典序做RMQ的性质，与某个串的LCP长度≥某个值的后缀形成一个区间．另外一个比较好用的性质是本质不同的子串个数 = 所有子串数 - 字典序相邻的串的height．

后缀自动机实际上可以接受的是所有后缀，如果把中间状态也算上的话就是所有子串．它的fail指针代表的也是当前串的后缀，不过注意每个状态可以代表很多状态，只要右端点在right集合中且长度处在$(val_{par_p}, val_p]$中的串都被它代表．

后缀自动机的fail树也就是**反串**的后缀树．每个结点代表的串和后缀自动机同理，两个串的LCP长度也就是他们在后缀树上的LCA．

6. 动态规划

6.1 决策单调性$O(n \log n)$

```
int a[maxn], q[maxn], p[maxn], g[maxn]; // 存左端点,右端点就是下一个左端点 - 1

long long f[maxn], s[maxn];

int n, m;

long long calc(int l, int r) {
    if (r < l)
        return 0;

    int mid = (l + r) / 2;
    if ((r - l + 1) % 2 == 0)
        return (s[r] - s[mid]) - (s[mid] - s[l - 1]);
    else
        return (s[r] - s[mid]) - (s[mid - 1] - s[l - 1]);
}

int solve(long long tmp) {
    memset(f, 63, sizeof(f));
    f[0] = 0;

    int head = 1, tail = 0;

    for (int i = 1; i <= n; i++) {
        f[i] = calc(1, i);
        g[i] = 1;

        while (head < tail && p[head + 1] <= i)
            head++;
        if (head <= tail) {
            if (f[q[head]] + calc(q[head] + 1, i) < f[i]) {
                f[i] = f[q[head]] + calc(q[head] + 1, i);
                g[i] = g[q[head]] + 1;
            }
            while (head < tail && p[head + 1] <= i + 1)
                head++;
            if (head <= tail)
                p[head] = i + 1;
        }
        f[i] += tmp;

        int r = n;

        while(head <= tail) {
```

```cpp
            if (f[q[tail]] + calc(q[tail] + 1, p[tail]) > f[i] + calc(i + 1, p[tail])) {
                r = p[tail] - 1;
                tail--;
            }
            else if (f[q[tail]] + calc(q[tail] + 1, r) <= f[i] + calc(i + 1, r)) {
                if (r < n) {
                    q[++tail] = i;
                    p[tail] = r + 1;
                }
                break;
            }
            else {
                int L = p[tail], R = r;
                while (L < R) {
                    int M = (L + R) / 2;

                    if (f[q[tail]] + calc(q[tail] + 1, M) <= f[i] + calc(i + 1, M))
                        L = M + 1;
                    else
                        R = M;
                }

                q[++tail] = i;
                p[tail] = L;

                break;
            }
        }
        if (head > tail) {
            q[++tail] = i;
            p[tail] = i + 1;
        }
    }

    return g[n];
}
```

6.2 例题

7. Miscellaneous

7.1 $O(1)$快速乘

```cpp
// long double 快速乘
// 在两数直接相乘会爆long long时才有必要使用
// 常数比直接long long乘法 + 取模大很多，非必要时不建议使用
long long mul(long long a, long long b, long long p) {
    a %= p;
    b %= p;
    return ((a * b - p * (long long)((long double)a / p * b + 0.5)) % p + p) % p;
}

// 指令集快速乘
// 试机记得测试能不能过编译
inline long long mul(const long long a, const long long b, const long long p) {
    long long ans;
    __asm__ __volatile__ ("\tmulq %%rbx\n\tdivq %%rcx\n" : "=d"(ans) : "a"(a), "b"(b), "c"(p));
    return ans;
}
```

7.2 $O(n^2)$高精度

```cpp
// 注意如果只需要正数运算的话
// 可以只抄英文名的运算函数
// 按需自取
// 乘法O(n ^ 2)，除法O(10 * n ^ 2)

const int maxn = 1005;

struct big_decimal {
    int a[maxn];
    bool negative;

    big_decimal() {
        memset(a, 0, sizeof(a));
        negative = false;
    }

    big_decimal(long long x) {
        memset(a, 0, sizeof(a));
        negative = false;

        if (x < 0) {
            negative = true;
            x = -x;
        }

        while (x) {
            a[++a[0]] = x % 10;
            x /= 10;
        }
    }

    big_decimal(string s) {
        memset(a, 0, sizeof(a));
        negative = false;

        if (s == "")
            return;

        if (s[0] == '-') {
            negative = true;
            s = s.substr(1);
        }
        a[0] = s.size();
        for (int i = 1; i <= a[0]; i++)
            a[i] = s[a[0] - i] - '0';

        while (a[0] && !a[a[0]])
            a[0]--;
    }

    void input() {
        string s;
        cin >> s;
        *this = s;
    }

    string str() const {
        if (!a[0])
            return "0";

        string s;
        if (negative)
            s = "-";

        for (int i = a[0]; i; i--)
            s.push_back('0' + a[i]);

```

```cpp
        return s;
    }

    operator string () const {
        return str();
    }

    big_decimal operator - () const {
        big_decimal o = *this;
        if (a[0])
            o.negative ^= true;

        return o;
    }

    friend big_decimal abs(const big_decimal &u) {
        big_decimal o = u;
        o.negative = false;
        return o;
    }

    big_decimal &operator <<= (int k) {
        a[0] += k;

        for (int i = a[0]; i > k; i--)
            a[i] = a[i - k];

        for(int i = k; i; i--)
            a[i] = 0;

        return *this;
    }

    friend big_decimal operator << (const big_decimal &u, int k) {
        big_decimal o = u;
        return o <<= k;
    }

    big_decimal &operator >>= (int k) {
        if (a[0] < k)
            return *this = big_decimal(0);

        a[0] -= k;
        for (int i = 1; i <= a[0]; i++)
            a[i] = a[i + k];

        for (int i = a[0] + 1; i <= a[0] + k; i++)
            a[i] = 0;

        return *this;
    }

    friend big_decimal operator >> (const big_decimal &u, int k) {
        big_decimal o = u;
        return o >>= k;
    }

    friend int cmp(const big_decimal &u, const big_decimal &v) {
        if (u.negative || v.negative) {
            if (u.negative && v.negative)
                return -cmp(-u, -v);

            if (u.negative)
                return -1;

            if (v.negative)
                return 1;
        }

        if (u.a[0] != v.a[0])
            return u.a[0] < v.a[0] ? -1 : 1;

        for (int i = u.a[0]; i; i--)
            if (u.a[i] != v.a[i])
                return u.a[i] < v.a[i] ? -1 : 1;

        return 0;
    }

    friend bool operator < (const big_decimal &u, const big_decimal &v) {
        return cmp(u, v) == -1;
    }

    friend bool operator > (const big_decimal &u, const big_decimal &v) {
        return cmp(u, v) == 1;
    }

    friend bool operator == (const big_decimal &u, const big_decimal &v) {
        return cmp(u, v) == 0;
    }

    friend bool operator <= (const big_decimal &u, const big_decimal &v) {
        return cmp(u, v) <= 0;
    }

    friend bool operator >= (const big_decimal &u, const big_decimal &v) {
        return cmp(u, v) >= 0;
    }

    friend big_decimal decimal_plus(const big_decimal &u, const big_decimal &v) { // 保证u, v均为正数的话可以直接调用
        big_decimal o;

        o.a[0] = max(u.a[0], v.a[0]);

        for (int i = 1; i <= u.a[0] || i <= v.a[0]; i++) {
            o.a[i] += u.a[i] + v.a[i];

            if (o.a[i] >= 10) {
                o.a[i + 1]++;
                o.a[i] -= 10;
            }
        }

        if (o.a[o.a[0] + 1])
            o.a[0]++;

        return o;
    }

    friend big_decimal decimal_minus(const big_decimal &u, const big_decimal &v) { // 保证u, v均为正数的话可以直接调用
        int k = cmp(u, v);

        if (k == -1)
            return -decimal_minus(v, u);
        else if (k == 0)
            return big_decimal(0);
```

```
        big_decimal o;

        o.a[0] = u.a[0];

        for (int i = 1; i <= u.a[0]; i++) {
            o.a[i] += u.a[i] - v.a[i];

            if (o.a[i] < 0) {
                o.a[i] += 10;
                o.a[i + 1]--;
            }
        }

        while (o.a[0] && !o.a[o.a[0]])
            o.a[0]--;

        return o;
    }

    friend big_decimal decimal_multi(const big_decimal
      &u, const big_decimal &v) {
        big_decimal o;

        o.a[0] = u.a[0] + v.a[0] - 1;

        for (int i = 1; i <= u.a[0]; i++)
            for (int j = 1; j <= v.a[0]; j++)
                o.a[i + j - 1] += u.a[i] * v.a[j];

        for (int i = 1; i <= o.a[0]; i++)
            if (o.a[i] >= 10) {
                o.a[i + 1] += o.a[i] / 10;
                o.a[i] %= 10;
            }

        if (o.a[o.a[0] + 1])
            o.a[0]++;

        return o;
    }

    friend pair<big_decimal, big_decimal>
      decimal_divide(big_decimal u, big_decimal v) { //
      整除
        if (v > u)
            return make_pair(big_decimal(0), u);

        big_decimal o;
        o.a[0] = u.a[0] - v.a[0] + 1;

        int m = v.a[0];
        v <<= u.a[0] - m;

        for (int i = u.a[0]; i >= m; i--) {
            while (u >= v) {
                u = u - v;
                o.a[i - m + 1]++;
            }

            v >>= 1;
        }

        while (o.a[0] && !o.a[o.a[0]])
            o.a[0]--;

        return make_pair(o, u);
    }

    friend big_decimal operator + (const big_decimal &u,
      const big_decimal &v) {
        if (u.negative || v.negative) {
            if (u.negative && v.negative)
                return -decimal_plus(-u, -v);

            if (u.negative)
                return v - (-u);

            if (v.negative)
                return u - (-v);
        }

        return decimal_plus(u, v);
    }

    friend big_decimal operator - (const big_decimal &u,
      const big_decimal &v) {
        if (u.negative || v.negative) {
            if (u.negative && v.negative)
                return -decimal_minus(-u, -v);

            if (u.negative)
                return -decimal_plus(-u, v);

            if (v.negative)
                return decimal_plus(u, -v);
        }

        return decimal_minus(u, v);
    }

    friend big_decimal operator * (const big_decimal &u,
      const big_decimal &v) {
        if (u.negative || v.negative) {
            big_decimal o = decimal_multi(abs(u),
              abs(v));

            if (u.negative ^ v.negative)
                return -o;
            return o;
        }

        return decimal_multi(u, v);
    }

    big_decimal operator * (long long x) const {
        if (x >= 10)
            return *this * big_decimal(x);

        if (negative)
            return -(*this * x);

        big_decimal o;

        o.a[0] = a[0];

        for (int i = 1; i <= a[0]; i++) {
            o.a[i] += a[i] * x;

            if (o.a[i] >= 10) {
                o.a[i + 1] += o.a[i] / 10;
                o.a[i] %= 10;
            }
        }

        if (o.a[a[0] + 1])
            o.a[0]++;
```

```cpp
            return o;
        }

    friend pair<big_decimal, big_decimal> decimal_div(const big_decimal &u, const big_decimal &v) {
        if (u.negative || v.negative) {
            pair<big_decimal, big_decimal> o = decimal_div(abs(u), abs(v));

            if (u.negative ^ v.negative)
                return make_pair(-o.first, -o.second);
            return o;
        }

        return decimal_divide(u, v);
    }

    friend big_decimal operator / (const big_decimal &u, const big_decimal &v) { // v不能是0
        if (u.negative || v.negative) {
            big_decimal o = abs(u) / abs(v);

            if (u.negative ^ v.negative)
                return -o;
            return o;
        }

        return decimal_divide(u, v).first;
    }

    friend big_decimal operator % (const big_decimal &u, const big_decimal &v) {
        if (u.negative || v.negative) {
            big_decimal o = abs(u) % abs(v);

            if (u.negative ^ v.negative)
                return -o;
            return o;
        }

        return decimal_divide(u, v).second;
    }
};
```

7.3 笛卡尔树

```cpp
int s[maxn], root, lc[maxn], rc[maxn];

int top = 0;
s[++top] = root = 1;
for (int i = 2; i <= n; i++) {
    s[top + 1] = 0;
    while (a[i] < a[s[top]]) // 小根笛卡尔树
        top--;

    if (top)
        rc[s[top]] = i;
    else
        root = i;

    lc[i] = s[top + 1];
    s[++top] = i;
}
```

7.4 常用NTT素数及原根

$p = r \times 2^k + 1$	r	k	最小原根
104857601	25	22	3
167772161	5	25	3
469762049	7	26	3
985661441	235	22	3
998244353	119	23	3
1004535809	479	21	3
1005060097*	1917	19	**5**
2013265921	15	27	**31**
2281701377	17	27	3
31525197391593473	7	52	3
180143985094819841	5	55	**6**
1945555039024054273	27	56	**5**
4179340454199820289	29	57	3

*注：1005060097有点危险，在变化长度大于$524288 = 2^{19}$时不可用.

7.5 xorshift

```cpp
ull k1, k2;
const int mod = 10000000;
ull xorShift128Plus() {
    ull k3 = k1, k4 = k2;
    k1 = k4;
    k3 ^= (k3 << 23);
    k2 = k3 ^ k4 ^ (k3 >> 17) ^ (k4 >> 26);
    return k2 + k4;
}
void gen(ull _k1, ull _k2) {
    k1 = _k1, k2 = _k2;
    int x = xorShift128Plus() % threshold + 1;
    // do sth
}


uint32_t xor128(void) {
    static uint32_t x = 123456789;
    static uint32_t y = 362436069;
    static uint32_t z = 521288629;
    static uint32_t w = 88675123;
    uint32_t t;

    t = x ^ (x << 11);
    x = y; y = z; z = w;
    return w = w ^ (w >> 19) ^ (t ^ (t >> 8));
}
```

7.6 枚举子集

(注意这是$t \neq 0$的写法，如果可以等于0需要在循环里手动`break`)

```cpp
for (int t = s; t; (--t) &= s) {
    // do something
}
```

7.7 STL

7.7.1 vector

- `vector(int nSize)`：创建一个vector，元素个数为nSize
- `vector(int nSize, const T &value)`：创建一个vector，元素个数为nSize，且值均为value
- `vector(begin, end)`：复制[begin, end)区间内另一个数组的元素到vector中

- `void assign(int n, const T &x)`：设置向量中前n个元素的值为x
- `void assign(const_iterator first, const_iterator last)`：向量中[first, last)中元素设置成当前向量元素

7.7.2 list

- `assign()` 给list赋值
- `back()` 返回最后一个元素
- `begin()` 返回指向第一个元素的迭代器
- `clear()` 删除所有元素
- `empty()` 如果list是空的则返回true
- `end()` 返回末尾的迭代器
- `erase()` 删除一个元素
- `front()` 返回第一个元素
- `insert()` 插入一个元素到list中
- `max_size()` 返回list能容纳的最大元素数量
- `merge()` 合并两个list
- `pop_back()` 删除最后一个元素
- `pop_front()` 删除第一个元素
- `push_back()` 在list的末尾添加一个元素
- `push_front()` 在list的头部添加一个元素
- `rbegin()` 返回指向第一个元素的逆向迭代器
- `remove()` 从list删除元素
- `remove_if()` 按指定条件删除元素
- `rend()` 指向list末尾的逆向迭代器
- `resize()` 改变list的大小
- `reverse()` 把list的元素倒转
- `size()` 返回list中的元素个数
- `sort()` 给list排序
- `splice()` 合并两个list
- `swap()` 交换两个list
- `unique()` 删除list中重复的元

7.8 pb_ds

7.8.1 哈希表

```
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/hash_policy.hpp>
using namespace __gnu_pbds;

cc_hash_table<string, int> mp1; // 拉链法
gp_hash_table<string, int> mp2; // 查探法(快一些)
```

7.8.2 堆

默认也是大根堆，和`std::priority_queue`保持一致.

```
#include<ext/pb_ds/priority_queue.hpp>
using namespace __gnu_pbds;

__gnu_pbds::priority_queue<int> q;
__gnu_pbds::priority_queue<int, greater<int>,
  ↪ pairing_heap_tag> pq;
```

效率参考:

* 共有五种操作：push、pop、modify、erase、join

* pairing_heap_tag：push和join为$O(1)$，其余为均摊$\Theta(\log n)$

* binary_heap_tag：只支持push和pop，均为均摊$\Theta(\log n)$

* binomial_heap_tag：push为均摊$O(1)$，其余为$\Theta(\log n)$

* rc_binomial_heap_tag：push为$O(1)$，其余为$\Theta(\log n)$

* thin_heap_tag：push为$O(1)$，不支持join，其余为$\Theta(\log n)$；果只有increase_key，那么modify为均摊$O(1)$

* “不支持”不是不能用，而是用起来很慢

常用操作:

- `push()`：向堆中压入一个元素，返回迭代器
- `pop()`：将堆顶元素弹出
- `top()`：返回堆顶元素
- `size()`：返回元素个数
- `empty()`：返回是否非空
- `modify(point_iterator, const key)`：把迭代器位置的 key 修改为传入的 key
- `erase(point_iterator)`：把迭代器位置的键值从堆中删除
- `join(__gnu_pbds::priority_queue &other)`：把 other 合并到 `*this`，并把 other 清空

7.8.3 平衡树

```
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

tree<int, null_type, less<int>, rb_tree_tag,
  ↪ tree_order_statistics_node_update> t;

// rb_tree_tag 红黑树(还有splay_tree_tag和ov_tree_tag，后
  ↪ 者不知道是什么)
```

注意第五个参数要填`tree_order_statistics_node_update`才能使用排名操作.

- `insert(x)`：向树中插入一个元素x，返回`pair<point_iterator, bool>`
- `erase(x)`：从树中删除一个元素/迭代器x，返回一个 `bool` 表明是否删除成功
- `order_of_key(x)`：返回x的排名，0-based
- `find_by_order(x)`：返回排名(0-based)所对应元素的迭代器
- `lower_bound(x) / upper_bound(x)`：返回第一个≥或者>x的元素的迭代器

- `join(x)`：将x树并入当前树，前提是两棵树的类型一样，并且二者值域不能重叠，x树会被删除
- `split(x,b)`：分裂成两部分，小于等于x的属于当前树，其余的属于b树
- `empty()`：返回是否为空
- `size()`：返回大小

(注意平衡树不支持多重值，如果需要多重值，可以再开一个`unordered_map`来记录值出现的次数，将`x<<32`后加上出现的次数后插入. 注意此时应该为long long类型.)

7.9 rope

```cpp
#include <ext/rope>
using namespace __gnu_cxx;

push_back(x); // 在末尾添加x
insert(pos, x); // 在pos插入x，自然支持整个char数组的一次
  ↪ 插入
erase(pos, x); // 从pos开始删除x个
copy(pos, len, x); // 从pos开始到pos + len为止的部分，赋
  ↪ 值给x
replace(pos, x); // 从pos开始换成x
substr(pos, x); // 提取pos开始x个
at(x) / [x]; // 访问第x个元素
```

7.10 编译选项

- `-O2 -g -std=c++11`：狗都知道
- `-Wall -Wextra -Wconversion`：更多警告
- `-fsanitize=(address/undefined)`：检查有符号整数溢出(算ub)/数组越界

  注意无符号类型溢出不算ub

8. 注意事项

8.1 常见下毒手法

- 高精度高低位搞反了吗
- 线性筛抄对了吗
- sort比较函数是不是比了个寂寞
- 该取模的地方都取模了吗
- 边界情况(+1-1之类的)有没有想清楚
- **特判是否有必要,确定写对了吗**

8.2 场外相关

- 安顿好之后查一下附近的咖啡店,打印店,便利店之类的位置,以备不时之需
- 热身赛记得检查一下编译注意事项中的代码能否过编译,还有熟悉比赛场地,清楚洗手间在哪儿,测试打印机(如果可以)
- 比赛前至少要翻一遍板子,尤其要看原理与例题
- 比赛前一两天不要摸鱼,要早睡,有条件最好洗个澡;比赛当天不要起太晚,维持好的状态
- 赛前记得买咖啡,最好直接安排三人份,记得要咖啡因比较足的;如果主办方允许,就带些巧克力之类的高热量零食
- 入场之后记得检查机器,尤其要逐个检查键盘按键有没有坏的;如果可以的话,调一下gedit设置
- 开赛之前调整好心态,比赛而已,不必心急.

8.3 做题策略与心态调节

- 拿到题后立刻按照商量好的顺序读题，前半小时最好跳过题意太复杂的题(除非被过穿了)
- 签到题写完不要激动，稍微检查一下最可能的下毒点再交，避免无谓的罚时

  一两行的那种傻逼题就算了
- 读完题及时输出题意，一方面避免重复读题，一方面也可以让队友有一个初步印象，方便之后决定开题顺序
- 如果不能确定题意就不要贸然输出甚至上机，尤其是签到题，因为样例一般都很弱
- 一个题如果卡了很久又有其他题可以写，那不妨先放掉写更容易的题，不要在一棵树上吊死

  不要被一两道题搞得心态爆炸，一方面急也没有意义，一方面你很可能真的离AC就差一步
- 榜是不会骗人的，一个题如果被不少人过了就说明这个题很可能并没有那么难;如果不是有十足的把握就不要轻易开没什么人交的题;另外不要忘记最后一小时会封榜
- 想不出题/找不出毒自然容易犯困，一定不要放任自己昏昏欲睡，最好去洗手间冷静一下，没有条件就站起来踱步
- 思考的时候不要挂机，一定要在草稿纸上画一画，最好说出声来最不容易断掉思路
- 出完算法一定要check一下样例和一些trivial的情况，不然容易写了半天发现写了个假算法
- 上机前有时间就提前给需要思考怎么写的地方打草稿，不要浪费机时
- 查毒时如果最难的地方反复check也没有问题，就从头到脚仔仔细细查一遍，不要放过任何细节，即使是并查集和sort这种东西也不能想当然
- 后半场如果时间不充裕就不要冒险开难题，除非真的无事可做

  如果是没写过的东西也不要轻举妄动，在有其他好写的题的时候就等一会再说
- 大多数时候都要听队长安排，虽然不一定最正确但可以保持组织性
- 任何时候都不要着急，着急不能解决问题，不要当詰国王
- 输了游戏，还有人生;赢了游戏，还有人生.