Algorithm For Competitive Programming

Xiaobin Ren @ BUPT

December 5, 2020

Contents

1 基础算法

1.1 排序

```
//快排
const int N = 100010;
int a[N];
void quick_sort(int a[], int l, int r) {
    if (l >= r) return;
    int i  = l - 1, j = r + 1, x = a[l + r >> 1];
    while (i < j) {
        do i++; while (a[i] < x);
        do j--; while (a[j] > x);
        if (i < j) swap(a[i], a[j]);
    }
    quick_sort(a, l, j); quick_sort(a, j + 1, r);
}
//归并排序
void merge_sort(int q[], int l, int r) {
    if (l >= r) return;
    int mid = l + r >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);
    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];
    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];
    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
int main() {
    int n; cin >> n;
    for (int i = 0; i < n; i++) cin >> a[i];
    quick_sort(a, 0, n - 1);  //模板的下标右边可以取到
    // merge_sort(a, 0, n - 1);
    for (int i = 0; i < n; i++) cout << a[i] << ' ';
    cout << endl;
    return 0;
}
```

```cpp
/*
    例题：归并排序求逆序对数量 (后补充树状数组)
    给定一个长度为 n 的整数数列，请你计算数列中的逆序对的数量。
    第一行包含整数 n，表示数列的长度。
    第二行包含 n 个整数，表示整个数列。
*/
typedef long long ll;
const int N = 100010;
int tmp[N]; ll res = 0;
int a[N];
ll merge_sort(int l, int r) {
    if (l >= r) return 0;
    int mid = l + ((r - l) >> 1);
    res = merge_sort(l, mid) + merge_sort(mid + 1, r);
    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r) {
        if (a[i] <= a[j]) tmp[k++] = a[i++];
        else {
            tmp[k++] = a[j++];
            res += mid - i + 1;
        }
    }
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= r) tmp[k++] = a[j++];
    for (int i = l, j = 0; i <= r; i++, j++) a[i] = tmp[j];
    return res;
}
int main() {
    int n; scanf("%d", &n);
    for (int i = 0; i < n; i++) scanf("%d", &a[i]);
    cout << merge_sort(0, n - 1) << endl;
    return 0;
}
```

1.2 二分

```cpp
//整数二分
bool check(int x) {/* ... */} // 检查 x 是否满足某种性质

// 区间 [l, r] 被划分成 [l, mid] 和 [mid + 1, r] 时使用：
int bsearch_1(int l, int r) {
```

```
    while (l < r) {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;
        else l = mid + 1;
    }
    return l;
}
// 区间 [l, r] 被划分成 [l, mid - 1] 和 [mid, r] 时使用：
int bsearch_2(int l, int r) {
    while (l < r) {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
/*
    例题：给定一个按照升序排列的长度为 n 的整数数组，q 个查询。
    对于每个查询，返回一个元素 k 的起始位置和终止位置（位置从 0
↪   开始计数）
    如果数组中不存在该元素，则返回“-1 -1”。
*/
const int N = 100010;
int a[N];
int main() {
    int n, q; cin >> n >> q;
    for (int i = 0; i < n; i++) cin >> a[i];
    while (q--) {
        int x; cin >> x;
        int l = 0, r = n - 1;
        while (l < r) {
            int mid = l + r >> 1;
            if (a[mid] >= x) r = mid; //注意此处相等也要继续
            else l = mid + 1;
        }
        if (a[l] != x) cout << "-1 -1" << endl;
        else {
            cout << l << ' ';
            l = 0, r = n - 1;
            while (l < r) {
```

```
                int mid = l + r + 1 >> 1;
                if (a[mid] <= x) l = mid;
                else r = mid - 1;
            }
            cout << r << endl;
        }
    }
    return 0;
}
//浮点数二分
double bsearch_3(double l, double r) {
    const double eps = 1e-6;
    // eps 表示精度，取决于题目对精度的要求
    while (r - l > eps) {
        double mid = (l + r) / 2;
        if (check(mid)) r = mid;
        else l = mid;
    }
    return l;
}
//例题：给定一个浮点数 n，求它的三次方根，结果保留 6 位小数。
//数据范围 -10000 <= n <= 10000
int main() {
    double x; scanf("%lf", &x);
    double l = -10000; double r = 10000;
    double eps = 1e-8; //精度一般取比题目 要求多 2 位即可
    while (r - l > eps) {
        double mid = (r + l) / 2;
        if (mid * mid * mid <= x) l = mid;
        else r = mid;
    }
    printf("%f", l);
}
```

1.3 高精度

```
//高精度加法
// C = A + B, A >= 0, B >= 0
vector<int> add(vector<int> &A, vector<int> &B)
//例题：给定两个正整数，计算它们的和。
//1 <= 整数长度 <= 100000
```

```cpp
const int N = 1e6 + 10;
vector<int> add(vector<int> &a, vector<int> &b) {
    int t = 0; vector<int> c;
    for (int i = 0; i < a.size() || i < b.size(); i++) {
        if (i < a.size()) t += a[i];
        if (i < b.size()) t += b[i];
        c.push_back(t % 10);
        t /= 10;
    }
    if (t) c.push_back(1);
    return c;
}
int main() {
    string a, b;
    cin >> a >> b;
    vector<int> A, B;
    //初始逆序存储 返回的结果也是逆序存储的 vector
    for (int i = a.size() - 1; i >= 0; i--)
        A.push_back(a[i] - '0');
    for (int i = b.size() - 1; i >= 0; i--)
        B.push_back(b[i] - '0');
    auto c = add(A, B);
    for (int i = c.size() - 1; i >= 0; i--) printf("%d",
    ↪  c[i]);
}
//数组实现解法:
const int N = 1e6 + 10;
int a[N], b[N], sum[N];
void add(int c[], int a[], int b[]) {
    for (int i = 0, t  = 0 ; i < N; i++) {
        t += a[i] + b[i];
        c[i] = t % 10;
        t /= 10;
    }
}
int main() {
    string sa, sb;
    cin >> sa >> sb;
    //逆序存储
    for (int i = sa.size() - 1, j = 0; i >= 0; i--)
```

```
        a[j++] = sa[i] - '0';
    for (int i = sb.size() - 1, j = 0; i >= 0; i--)
        b[j++] = sb[i] - '0';
    add(sum, a, b);
    int i = N - 1;
    while (!sum[i]) i --;//去除前导零
    while (i >= 0) printf("%d", sum[i--]);
    return 0;
}
//高精度减法
// C = A - B, 满足 A >= B, A >= 0, B >= 0
vector<int> sub(vector<int> &A, vector<int> &B)
//例题：给定两个正整数，计算它们的差，计算结果可能为负数。
//1 <= 整数长度 <= 1e5
typedef vector<int> VI;
bool cmp(VI &a, VI &b) { //减法需要大数减去小数 分类讨论
    if (a.size() != b.size()) return a.size() > b.size();
    for (int i = a.size() - 1; i >= 0; i--) {
        if (a[i] != b[i]) return a[i] > b[i];
    }
    return true;
}
VI sub(VI &a, VI &b) {
    int t = 0;  VI c;
    for (int i = 0; i < a.size(); i++) {
        t = a[i] - t;
        if (i < b.size()) t -= b[i];
        c.push_back((t + 10) % 10);
        if (t < 0) t = 1; else t = 0;
    }
    while (c.size() > 1 && c.back() == 0) c.pop_back();
    return c;
}
int main() {
    string a, b;
    cin >> a >> b;
    vector<int> A, B;
    //逆序读入
    for (int i = a.size() - 1; i >= 0; i--)
        A.push_back(a[i] - '0');
```

```
    for (int i = b.size() - 1; i >= 0; i--)
        B.push_back(b[i] - '0');
    if (cmp(A, B)) {
        auto c = sub(A, B);
        for (int i = c.size() - 1; i >= 0; i--)
            printf("%d" , c[i]);
    }
    else {
        auto c = sub(B, A);
        printf("-");  //结果是负数  先输出负号 逆序输出
        for (int i = c.size() - 1; i >= 0; i--)
            printf("%d" , c[i]);
    }
    return 0;
}
//高精度乘低精度
// C = A * b, A >= 0, b > 0
vector<int> mul(vector<int> &A, int b)
//例题：给定两个正整数 A 和 B, 请你计算 A * B 的值。
//1<= A 的长度 <= 100000, 0<= B <=10000
typedef vector<int> VI;
VI mul(VI &a, int b) {
    int t = 0;
    VI c;  //答案
    for (int i = 0; i < a.size() || t; i++) {
        t += a[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }
    return c;
}
int main() {
    string a; int b; VI A;
    cin >> a >> b;
    if (b == 0 || a[0] == 0) { //特判 0 的情况
        cout << 0 << endl;
        return 0;
    }
    for (int i = a.size() - 1; i >= 0; i--)
        A.push_back(a[i] - '0');
```

```
    auto c = mul(A, b);
    for (int i = c.size() - 1; i >= 0; i--)
        printf("%d", c[i]);
    return 0;
}
//高精度除以低精度
// A / b = C ... r, A >= 0, b > 0
vector<int> div(vector<int> &A, int b, int &r)
//例题：给定两个非负整数 A, B, 请你计算 A / B 的商和余数。
//1 <= A 的长度 <= 100000 ,1 <= B <= 10000, B 一定不为 0
typedef vector<int> VI;
VI div(VI &a, int b, int &r) { //r 是余数
    r = 0;
    VI c;  //答案
    for (int i = a.size() - 1; i >= 0; i--) {
        r = r * 10 + a[i];
        c.push_back(r / b);
        r %= b;
    }
    reverse(c.begin(), c.end());
    while (c.size() > 1 && c.back() == 0) c.pop_back();
    return c;
}
int main() {
    string a; int b;
    cin >> a >> b;
    int r = 0; VI A;
    //逆序读入
    for (int i  = a.size() - 1; i >= 0; i--)
        A.push_back(a[i] - '0');
    auto c = div(A, b, r);
    for (int i = c.size() - 1; i >= 0; i--)
        printf("%d", c[i]);
    printf("\n");
    printf("%d\n", r); //余数
}
```

1.4 前缀和与差分

```
/*
一维前缀和：
```

```
    当 a[] 数组只初始化一次的时候
    可以直接用 a[] 读入的时候计算前缀和
    S[i] = a[1] + a[2] + ... a[i]
    a[l] + ... + a[r] = S[r] - S[l - 1]
二维前缀和：
    S[i, j] = 第 i 行 j 列格子左上部分所有元素的和
    以 (x1, y1) 为左上角，(x2, y2) 为右下角的子矩阵的和为：
    S[x2, y2]-S[x1-1,y2]-S[x2, y1-1]+S[x1-1, y1-1]
一维差分：
    给区间 [l, r] 中的每个数加上 c：B[l] += c, B[r + 1] -= c
二维差分：
    给以 (x1, y1) 为左上角，(x2, y2) 为右下角的子矩阵中的所有元
↪   素加上 c：
    S[x1, y1] += c, S[x2 + 1, y1] -= c,
    S[x1, y2 + 1] -= c, S[x2 + 1, y2 + 1] += c
*/
/*
    例题：子矩阵求和
    输入一个 n 行 m 列的整数矩阵，再输入 q 个询问，每个询问包含
↪   四个整数 x1, y1, x2, y2,
    表示一个子矩阵的左上角坐标和右下角坐标。对于每个询问输出子矩
↪   阵中所有数的和。
*/
int n, m, q;
const int N = 1010;
int s[N][N], a[N][N];
int main() {
    scanf("%d%d%d" , &n, &m, &q);
    //前缀和问题下标都从 1 开始处理
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            scanf("%d", &a[i][j]);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            s[i][j] = s[i][j - 1] + s[i - 1][j] - s[i - 1][j -
            ↪  1] + a[i][j];
    while (q--) {
        int x1, y1, x2, y2;
        scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
```

```
            printf("%d\n", s[x2][y2] - s[x1 - 1][y2] - s[x2][y1 -
            ↪  1] + s[x1 - 1][y1 - 1]);
        }
        return 0;
}
/*
    例题：一维差分
    输入一个长度为 n 的整数序列。
    接下来输入 m 个操作，每个操作包含三个整数 l，r，c，表示将序
↪   列中 [l，r] 之间的每个数加上 c。
    请你输出进行完所有操作后的序列。
*/
const int N = 100010;
int a[N], b[N];
int n , m;
void insert(int l, int r, int c) {
    b[l] += c;
    b[r + 1] -= c;
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        insert(i, i, a[i]); //构造差分序列
    }
    while (m--) {
        int l, r, c;
        cin >> l >> r >> c;
        insert(l, r, c);
    }
    //所有操作结束后 求前缀和 就是最终的修改结果
    for (int i = 1; i <= n; i++) b[i] += b[i - 1];
    for (int i = 1; i <= n; i++) cout << b[i] << ' ';
    cout << endl;
    return 0;
}

/*
    例题：二维差分
```

```
    输入一个 n 行 m 列的整数矩阵，再输入 q 个操作，每个操作包含五个整数 x1, y1, x2, y2, c，
    其中 (x1, y1) 和 (x2, y2) 表示一个子矩阵的左上角坐标和右下角坐标。
    每个操作都要将选中的子矩阵中的每个元素的值加上 c。
    请你将进行完所有操作后的矩阵输出。
*/
const int N = 1010;
int a[N][N], b[N][N];
int n, m, q;
//差分的核心，输入时候初始化可直接使用 insert()
void insert(int x1, int y1, int x2, int y2, int c) {
    b[x1][y1] += c;
    b[x2 + 1][y1] -= c;
    b[x1][y2 + 1] -= c;
    b[x2 + 1][y2 + 1] += c;

}
int main() {
    scanf("%d%d%d", &n, &m, &q);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            scanf("%d", &a[i][j]);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            insert(i, j, i, j, a[i][j]);
    while (q--) {
        int x1, y1, x2, y2, c;
        scanf("%d%d%d%d%d", &x1, &y1, &x2, &y2, &c);
        insert(x1, y1, x2, y2, c);
    }
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            b[i][j] += b[i - 1][j] + b[i][j - 1] - b[i - 1][j - 1]; //差分完求一次二维前缀和就是修改后的数组了
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (j == m) printf("%d\n", b[i][j]);
            else printf("%d ", b[i][j]);
    return 0;
```

```
}
```

1.5 双指针算法

```
/*
    双指针算法：
    常见问题分类：
    (1) 对于一个序列，用两个指针维护一段区间
    (2) 对于两个序列，维护某种次序，比如归并排序中合并两个有序序
    ↪ 列的操作
*/
for (int i = 0, j = 0; i < n; i ++ ) {
    while (j < i && check(i, j)) j ++ ;
    // 具体问题的逻辑
}
/*
    例题：最长连续不重复子序列
    给定一个长度为 n 的整数序列，请找出最长的不包含重复的数的连
    ↪ 续区间，输出它的长度。
*/
const int N = 100010;
int a[N], s[N];
int n;
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) scanf("%d", &a[i]);
    int res = 0;
    for (int i = 0, j = 0; i < n; i++) {
        s[a[i]]++;
        while (s[a[i]] > 1) {
            s[a[j]] -- ;
            j++;   // j 跳到下一个元素，直到当前区间没有重复元素
        }
        res = max(res, i - j + 1);
    }
    printf("%d", res);
    return 0;
}
/*
    例题：数组元素的目标和
```

```
    给定两个升序排序的有序数组 A 和 B，以及一个目标值 x。数组下
↪   标从 0 开始。
    请你求出满足 A[i] + B[j] = x 的数对 (i, j)。
    数据保证有唯一解。第一行包含三个整数 n, m, x, 分别表示 A 的长
↪   度，B 的长度以及目标值 x。
    第二行包含 n 个整数，表示数组 A。第三行包含 m 个整数，表示数
↪   组 B。
*/
const int N = 100010;
int a[N], b[N];
int n, m, x;
int main() {
    cin >> n >> m >> x;
    for (int i = 0; i < n; i++) cin >> a[i];
    for (int i = 0; i < m; i++) cin >> b[i];
    for (int i = 0, j = m - 1; i < n; i++) {
        while (j >= 0 && a[i] + b[j] > x) j--;
        if (j >= 0 && a[i] + b[j] == x) cout << i << ' ' << j
        ↪  << endl;
    }
    return 0;
}
```

1.6 位运算

```
/*
    求 n 的第 k 位数字: n >> k & 1
    返回 n 的最后一位 1：lowbit(n) = n & -n
*/
/*
    例题：二进制中 1 的个数
    给定一个长度为 n 的数列，请你求出数列中每个数的二进制表示中
↪   1 的个数。
    1 <= n <= 100000, 0 <= 数列中元素的值 <= 1e9
*/
const int N  = 100010;
int a[N];
int main() {
    int n; scanf("%d", &n);
    for (int i = 0; i < n; i++) scanf("%d", &a[i]);
```

```
    for (int i = 0; i < n; i++) {
        int cnt = 0;
        while (a[i]) {
            a[i] -= a[i] & -1 * a[i];
            cnt++;
        }
        printf("%d ", cnt);
    }
}
```

1.7 离散化

```
//离散化模板
vector<int> alls; // 存储所有待离散化的值
sort(alls.begin(), alls.end()); // 将所有值排序
// 去掉重复元素
alls.erase(unique(alls.begin(), alls.end()), alls.end());
// 二分求出 x 对应的离散化的值
int find(int x) { // 找到第一个大于等于 x 的位置
    int l = 0, r = alls.size() - 1;
    while (l < r) {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1; // 映射到 1, 2, ...n
}
/*
    例题：区间和
    假定有一个无限长的数轴，数轴上每个坐标上的数都是 0。
    现在，首先进行 n 次操作，每次操作将某一位置 x 上的数加 c
    接下来，进行 m 次询问，每个询问包含两个整数 l 和 r，你需要求
↪   出在区间 [l, r] 之间的所有数的和。
    -1e9 <= x <= 1e9, 1 <= n, m<= 1e5
    -1e9<=l<=r<=1e9, -10000<=c<=10000
*/
typedef pair<int, int> pii;
const int N = 300010; //区间 lr 都加入后数据范围三倍 开 30w
vector<pii> add, query;
vector<int> alls;
```

```cpp
int n, m;
int a[N], s[N];
int find(int x) {
    int l = 0, r = alls.size() - 1;
    while (l < r) {
        int mid = l + r >> 1;
        if ( alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1;
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        int x, c;
        cin >> x >> c;
        add.push_back({x, c});
        alls.push_back(x); //保存操作的下标
    }
    while (m--) {
        int l, r;
        cin >> l >> r;
        query.push_back({l , r});
        alls.push_back(l); alls.push_back(r);
    }
    sort(alls.begin(), alls.end());
    alls.erase(unique(alls.begin(), alls.end()), alls.end());
    for (auto item : add) {
        int x = find(item.first);
        a[x] += item.second;
    }
    for (int i = 1; i <= alls.size(); i++)
        s[i] = s[i - 1] + a[i];
    for (auto item : query) {
        int l = find(item.first), r = find(item.second);
        int res = s[r] - s[l - 1];
        cout << res << endl;
    }
    return 0;
}
```

1.8 区间合并

```cpp
/*
    例题：给定 n 个区间 [li,ri]，要求合并所有有交集的区间。
    注意如果在端点处相交，也算有交集。
    输出合并完成后的区间个数。
    例如：[1,3] 和 [2,6] 可以合并为一个区间 [1,6]。
    1 <= n <= 100000, -1e9 <= li <= ri <= 1e9
*/
typedef pair<int, int> PII;
const int N = 100010;
int n;
vector<PII> segs;
void merge(vector<PII> &segs) {
    vector<PII> res;
    sort(segs.begin(), segs.end());
    int st  = -2e9, ed = -2e9; //左右边界从无限远处来开始维护
    //注意维护的逻辑是先用上一个区间和新的区间比较 然后压入上一
    ↪ 个区间 新的区间又和下一个比较 最后要把最后一个压进去
    for (auto seg : segs) {
        if (ed < seg.first) {
            if (st != -2e9) res.push_back({st, ed});  //第一个
            ↪ 区间维护 不要加进去初始的最小的那个
            st = seg.first, ed = seg.second; //第一次循环就是从
            ↪ 第一个区间开始维护
        }
        else ed = max(ed, seg.second);
    }
    if (st != -2e9) res.push_back({st, ed}); //最后一个区间，同
    ↪ 时判断是否为空区间
    segs = res;
}
int main(int argc, char const *argv[]) {
    cin >> n;
    for (int i = 0; i < n; i++) {
        int l, r;
        scanf("%d%d", &l, &r);
        segs.push_back({l, r});
    }
    merge(segs); printf("%d", segs.size());
```

```
    return 0;
}
```

1.9 RMQ 一维/二维

```
//一维 预处理 O(n*logn) 查询 O(1)
const int N = 2e5 + 10;
const int M = 18; //区间长度 log2 取对数
//如果求区间最小 那么 init 和 query 函数的 max 改为 min 即可
int n, m; int w[N];
int f[N][M]; //f[i][j] 从 i 开始长度为 2^j 的区间最值
void init() { //预处理
    for (int j = 0; j < M; j++)
        for (int i = 1; i + (1 << j) - 1 <= n; i++) {
            if (!j) f[i][j] = w[i]; //只有一个数 最大就是本身
            else f[i][j] = max(f[i][j - 1], f[i + (1 << j -
            ↪  1)][j - 1]);
        }
}
int query(int l, int r) {
    int len = r - l + 1;
    int k = log(len) / log(2);  //下取整 <math.h>
    return max(f[l][k], f[r - (1 << k) + 1][k]);
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &w[i]);

    init(); //预处理 ST
    scanf("%d", &m); //m 组询问
    while (m --) {
        int l, r;
        scanf("%d%d", &l, &r);
        printf("%d\n", query(l, r));
    }
    return 0;
}
//二维：预处理 n*m*log*(n)*log(m) 查询 O(1)
int val[310][310];//矩阵
int dp[310][310][9][9];//最大值
```

```
int mm[310];//二进制位数减一，使用前初始化
void initRMQ(int n, int m) {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            dp[i][j][0][0] = val[i][j];
    for (int ii = 0; ii <= mm[n]; ii++)
        for (int jj = 0; jj <= mm[m]; jj++)
            if (ii + jj)
                for (int i = 1; i + (1 << ii) - 1 <= n; i++)
                    for (int j = 1; j + (1 << jj) - 1 <= m;
                     ↪  j++) {
                        if (ii)dp[i][j][ii][jj] =
                         ↪  max(dp[i][j][ii][jj], dp[i + (1 <<
                         ↪  (ii - 1))][j][ii - 1][jj]);
                        else dp[i][j][ii][jj] =
                         ↪  max(dp[i][j][ii][jj], dp[i][j + (1
                         ↪  << (jj - 1))][ii][jj - 1]);
                    }
}
//查询矩形内的最大值 (x1<=x2,y1<=y2)
int rmq(int x1, int y1, int x2, int y2) {
    int k1 = mm[x2 - x1 + 1];
    int k2 = mm[y2 - y1 + 1];
    x2 = x2 - (1 << k1) + 1;
    y2 = y2 - (1 << k2) + 1;
    return max(max(dp[x1][y1][k1][k2], dp[x1][y2][k1][k2]),
     ↪  max(dp[x2][y1][k1][k2], dp[x2][y2][k1][k2]));
}
int main() {
//在外面对 mm 数组进行初始化
    mm[0] = -1;
    for (int i = 1; i <= 305; i++)
        mm[i] = ((i & (i - 1)) == 0) ? mm[i - 1] + 1 : mm[i -
         ↪  1];
    int n, m;
    int Q;
    int r1, c1, r2, c2;
    while (scanf("%d%d", &n, &m) == 2) {
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= m; j++)
```

```
                scanf("%d", &val[i][j]);
        initRMQ(n, m);
        scanf("%d", &Q);
        while (Q--) { //Q 组查询
            scanf("%d%d%d%d", &r1, &c1, &r2, &c2);
            if (r1 > r2)swap(r1, r2);
            if (c1 > c2)swap(c1, c2);
            int tmp = rmq(r1, c1, r2, c2);
            printf("%d\n", tmp);
        }
    }
    return 0;
}
```

1.10 C++ STL

```
vector, 变长数组, 倍增的思想
    size()  返回元素个数
    empty()  返回是否为空
    clear()  清空
    front()/back()
    push_back()/pop_back()
    begin()/end()
    []
    支持比较运算, 按字典序

pair<int, int>
    first, 第一个元素
    second, 第二个元素
    支持比较运算, 以 first 为第一关键字, 以 second 为第二关键字
    ↪ (字典序)

string, 字符串
    size()/length()  返回字符串长度
    empty()
    clear()
    substr(起始下标, (子串长度))  返回子串
    c_str()  返回字符串所在字符数组的起始地址

queue, 队列
```

```
    size()
    empty()
    push()  向队尾插入一个元素
    front()  返回队头元素
    back()  返回队尾元素
    pop()  弹出队头元素

priority_queue, 优先队列, 默认是大根堆
    size()
    empty()
    push()  插入一个元素
    top()  返回堆顶元素
    pop()  弹出堆顶元素
    定义成小根堆的方式：priority_queue<int, vector<int>,
    ↪  greater<int>> q;

stack, 栈
    size()
    empty()
    push()  向栈顶插入一个元素
    top()  返回栈顶元素
    pop()  弹出栈顶元素

deque, 双端队列
    size()
    empty()
    clear()
    front()/back()
    push_back()/pop_back()
    push_front()/pop_front()
    begin()/end()
    []

set, map, multiset, multimap, 基于平衡二叉树（红黑树), 动态维护
↪  有序序列
    size()
    empty()
    clear()
    begin()/end()
    ++, -- 返回前驱和后继, 时间复杂度 O(logn)
```

```
    set/multiset
        insert()  插入一个数
        find()  查找一个数
        count()  返回某一个数的个数
        erase()
            (1) 输入是一个数 x, 删除所有 x, O(k + logn)
            (2) 输入一个迭代器, 删除这个迭代器
        lower_bound()/upper_bound()
            lower_bound(x)  返回大于等于 x 的最小的数的迭代器
            upper_bound(x)  返回大于 x 的最小的数的迭代器
    map/multimap
        insert()  插入的数是一个 pair
        erase()  输入的参数是 pair 或者迭代器
        find()
        []  注意 multimap 不支持此操作。时间复杂度是 O(logn)
        lower_bound()/upper_bound()

unordered_set, unordered_map, unordered_multiset,
↪  unordered_multimap, 哈希表
    和上面类似, 增删改查的时间复杂度是 O(1)
    不支持 lower_bound()/upper_bound(), 迭代器的 ++, --

bitset, 压位
    bitset<10000> s;
    ~, &, |, ^
    >>, <<
    ==, !=
    []

    count()  返回有多少个 1
    any()  判断是否至少有一个 1
    none()  判断是否全为 0
    set()  把所有位置成 1
    set(k, v)  将第 k 位变成 v
    reset()  把所有位变成 0
    flip()  等价于 ~
    flip(k) 把第 k 位取反
```

2 数据结构

2.1 链表

```
//单链表
// head 存储链表头，e[] 存储节点的值，ne[] 存储节点的 next 指针，
//idx 表示当前用到了哪个节点
int head, e[N], ne[N], idx;
void init() { // 初始化
    head = -1;
    idx = 0;
}
// 在链表头插入一个数 a
void insert(int a) {
    e[idx] = a, ne[idx] = head, head = idx ++ ;
}
// 将头结点删除，需要保证头结点存在
void remove() {
    head = ne[head];
}
/*
    例题：实现一个单链表，链表初始为空，支持三种操作：
    (1) 向链表头插入一个数；
    (2) 删除第 k 个插入的数后面的数；
    (3) 在第 k 个插入的数后插入一个数
    现在要对该链表进行 M 次操作，进行完所有操作后，从头到尾输出整
↪   个链表。
    注意：题目中第 k 个插入的数并不是指当前链表的第 k 个数。例如
↪   操作过程中一共插入了 n 个数，
    则按照插入的时间顺序，这 n 个数依次为：第 1 个插入的数，第 2
↪   个插入的数，…第 n 个插入的数。
    输入格式：第一行包含整数 M，表示操作次数。
    接下来 M 行，每行包含一个操作命令，操作命令可能为以下几种：
    (1) “H x”，表示向链表头插入一个数 x。
    (2) “D k”，表示删除第 k 个输入的数后面的数（当 k 为 0 时，
↪   表示删除头结点）
    (3) “I k x”，表示在第 k 个输入的数后面插入一个数 x（此操作
↪   中 k 均大于 0）
*/
const int N = 100010;
```

```cpp
int head, e[N], ne[N], idx;
//初始化
void init() {
    head = -1;
    idx = 0;
}
//将 x 插到头节点
void add_to_head(int x) {
    e[idx] = x, ne[idx] = head, head = idx++;
}
//将 x 插到下标是 k 的点后边
void add(int k, int x) {
    e[idx] = x;
    ne[idx] = ne[k];
    ne[k] = idx++;
}
//将下标是 k 后边的节点删除
void remove(int k) {
    ne[k] = ne[ne[k]];
}
int main() {
    int m; cin >> m;
    init();
    while (m--) {
        int k,  x;  char op; cin >> op;
        if (op == 'H') { cin >> x; add_to_head(x); }
        else if (op == 'D') {
            cin >> k;
            if (!k) head = ne[head];
            remove(k - 1);
        }
        else {cin >> k >> x; add(k - 1, x);}
    }
    for (int i = head; i != -1; i = ne[i]) printf("%d ",
    ↪  e[i]);
    printf("\n");
    return 0;
}
```

```
//双链表
// e[] 表示节点的值，l[] 表示节点的左指针，r[] 表示节点的右指针，
↪  idx 表示当前用到了哪个节点
int e[N], l[N], r[N], idx;
void init() { // 初始化
    //0 是左端点，1 是右端点
    r[0] = 1, l[1] = 0;
    idx = 2;
}
void insert(int a, int x) { // 在节点 a 的右边插入一个数 x
    e[idx] = x;
    l[idx] = a, r[idx] = r[a];
    l[r[a]] = idx, r[a] = idx ++ ;
}
void remove(int a) { // 删除节点 a
    l[r[a]] = l[a];
    r[l[a]] = r[a];
}
/*
实现一个双链表，双链表初始为空，支持 5 种操作：
(1) 在最左侧插入一个数；
(2) 在最右侧插入一个数；
(3) 将第 k 个插入的数删除；
(4) 在第 k 个插入的数左侧插入一个数；
(5) 在第 k 个插入的数右侧插入一个数
现在要对该链表进行 M 次操作，进行完所有操作，从左到右输出整个链
↪  表
注意：题目中第 k 个插入的数并不是指当前链表的第 k 个数。
例如操作过程中一共插入了 n 个数，则按照插入的时间顺序，这 n 个数
↪  依次为：
第 1 个插入的数，第 2 个插入的数，…第 n 个插入的数。

第一行包含整数 M，表示操作次数。
接下来 M 行，每行包含一个操作命令，操作命令可能为以下几种：
(1) “L x”，表示在链表的最左端插入数 x。
(2) “R x”，表示在链表的最右端插入数 x。
(3) “D k”，表示将第 k 个插入的数删除。
(4) “IL k x”，表示在第 k 个插入的数左侧插入一个数。
(5) “IR k x”，表示在第 k 个插入的数右侧插入一个数。
*/
```

```
const int N = 100010;
int l[N], r[N], e[N], idx;
void init() {
    r[0] = 1, l[1] = 0; //双链表的收尾都是 u 虚拟节点 方便处理
    ↪ 边界
    idx = 2;
}
void insert(int a, int x) {
// 在第 k (a) 个插入的节点后面插入 一个值为 x 的数 对于左边插入
↪ 可以直接用这个
//比如在 k 左边插入 就调用 insert(l[k], x);
    e[idx] = x;
    l[idx] = a, r[idx] = r[a];
    l[r[a]] = idx, r[a] = idx ++;
}
void remove(int a) {
    l[r[a]] = l[a];
    r[l[a]] = r[a];
}
int main() {
    int n;
    cin >> n;
    init();
    while (n --) {
        string op;
        int k, x;
        cin >> op;
        if (op == "L") {
            cin >> x;
            insert(0, x);
        } else if (op == "R") {
            cin >> x;
            insert(l[1], x);
        } else if (op == "D") {
            cin >> k;
            remove(k + 1); //idx 从 2 开始的
        } else if (op == "IL") {
            cin >> k >> x;
            insert(l[k + 1], x);
        } else {
```

```
            cin >> k >> x;
            insert(k + 1, x);
        }
    }
    for (int i = r[0]; i != 1; i = r[i]) cout << e[i] << ' ';
    cout << endl;
    return 0;
}
```

2.2 栈/单调栈

```
//数组模拟栈
// tt 表示栈顶
int stk[N], tt = 0;
// 向栈顶插入一个数
stk[ ++ tt] = x;
// 从栈顶弹出一个数
tt -- ;
// 栈顶的值
stk[tt];
// 判断栈是否为空
if (tt > 0) {
    /*code*/
}

/*
单调栈：
给定一个长度为 N 的整数数列，输出每个数左边第一个比它小的数，如果
↪ 不存在则输出-1。
*/
stack<int> s;
int main() {
    int n; cin >> n;
    while (n --) {
        int x ; cin >> x;
        while (s.size() && s.top() >= x) s.pop();
        if (!s.size()) cout << -1 << ' ';
        else cout << s.top() << ' '; //第一个数左边没有小的 肯
        ↪ 定是-1 所以不用考虑边界
        s.push(x);
```

```
    }
    return 0;
}
```

2.3 队列/单调队列

```
//普通队列
// hh 表示队头，tt 表示队尾
int q[N], hh = 0, tt = -1;
// 向队尾插入一个数
q[ ++ tt] = x;
// 从队头弹出一个数
hh ++ ;
// 队头的值
q[hh];
// 判断队列是否为空
if (hh <= tt) {
    //
}

//循环队列
// hh 表示队头，tt 表示队尾的后一个位置
int q[N], hh = 0, tt = 0;
// 向队尾插入一个数
q[tt ++ ] = x;
if (tt == N) tt = 0;
// 从队头弹出一个数
hh ++ ;
if (hh == N) hh = 0;
// 队头的值
q[hh];
// 判断队列是否为空
if (hh != tt) {
    //
}

//单调队列
/*
例题：滑动窗口：给定一个大小为 n <= 1e6 的数组。
有一个大小为 k 的滑动窗口，它从数组的最左边移动到最右边。
```

```
您只能在窗口中看到 k 个数字。
每次滑动窗口向右移动一个位置。
您的任务是确定滑动窗口位于每个位置时，窗口中的最大值和最小值。
*/
const int N = 1000010;
int n, k;
int a[N], q[N];  //q 存储单调队列的下标
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0); cout.tie(0);
    cin >> n >> k;
    for (int i = 0; i < n; i++) cin >> a[i];
    int hh = 0, tt = 0;
    for (int i = 0; i < n; i++) {
        if (hh <= tt && i - q[hh] + 1 > k) hh++;
        while (hh <= tt && a[q[tt]] >= a[i]) tt--;
        q[++tt] = i; // 需要先加入新的 因为新的可能是最小的
        //窗口完全滑进队列之后才开始判断
        if (i >= k - 1) cout << a[q[hh]] << ' ';
    }
    cout << endl;
    //最大值 修改上边代码即可 大于号改成小于号即可
    hh = 0, tt = -1;
    for (int i = 0; i < n; i++) {
        if (hh <= tt && i - k + 1 > q[hh]) hh++;
        while (hh <= tt && a[q[tt]] <= a[i]) tt--;
        q[++tt] = i; // 需要先加入新的 因为新的可能是最小的
        //窗口完全滑进队列之后才开始判断
        if (i >= k - 1) cout << a[q[hh]] << ' ';
    }
}
```

2.4 Trie

```
//Tire 模板
int son[N][26], cnt[N], idx;
// 0 号点既是根节点，又是空节点
// son[][] 存储树中每个节点的子节点
// cnt[] 存储以每个节点结尾的单词数量
```

```
// 插入一个字符串
void insert(char *str) {
    int p = 0;
    for (int i = 0; str[i]; i ++ ) {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
    }
    cnt[p] ++ ;
}

// 查询字符串出现的次数
int query(char *str) {
    int p = 0;
    for (int i = 0; str[i]; i ++ ) {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}

/*
例题：最大异或对
在给定的 N 个整数 A1，A2……AN 中选出两个进行 xor（异或）运算，得
↪ 到的结果最大是多少？
第一行输入一个整数 N，第二行输入 N 个整数 A1～AN。
1 <= N <= 1e5，0 <= Ai < 2^31
*/
const int N = 100010, M  = 3000010;
int n;
//每个数最多二进制表示 31 位长 N 个数
int son[M][2], idx; //注意对于特殊 son 数组 第一维要开总共的节
↪ 点个数
int a[N];
//此题略有贪心思路：从最高位开始异或 原因是低位对于数大小的贡献
//远小于高位 数大小由高位决定 复杂度 n*log(n)
void insert(int x) {
    int p = 0;
    for (int i = 30; ~i ; i --) { //从高位到低位存 所以逆着来
```

```
            int &s = son[p][x >> i & 1];  // >> i 是第 i + 1 位
            ↪ 所以循环最大 30
            if (!s) s = ++idx;
            p = s;
        }
    }
    int query(int x) {
        int res = 0, p = 0;
        for (int i = 30; ~i; i--) {
            int s = x >> i & 1;
            if (son[p][!s]) {
                res += 1 << i;  //该位对于答案的贡献
                p = son[p][!s];
            } else p = son[p][s]; //不存在不同的值 就只能选择相同的
            ↪ 二进制位
        }
        return res;
    }
    int main() {
        cin >> n;
        for (int i = 0; i < n; i++) {
            cin >> a[i];
            insert(a[i]);
        }
        int res = 0;
        for (int i = 0; i < n ; i++) res = max(res, query(a[i]));
        cout << res << endl;
        return 0;
    }
```

2.5 并查集

```
//(1) 朴素并查集：
int p[N]; //存储每个点的祖宗节点
// 返回 x 的祖宗节点
int find(int x) {
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}
// 初始化，假定节点编号是 1~n
```

```
for (int i = 1; i <= n; i ++ ) p[i] = i;
// 合并 a 和 b 所在的两个集合：
p[find(a)] = find(b);

//(2) 维护 size 的并查集：
int p[N], size[N];
//p[] 存储每个点的祖宗节点, size[] 只有祖宗节点的有意义, 表示祖
↪  宗节点所在集合中的点的数量
// 返回 x 的祖宗节点
int find(int x) {
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}
// 初始化, 假定节点编号是 1~n
for (int i = 1; i <= n; i ++ ) {
    p[i] = i;
    size[i] = 1;
}
// 合并 a 和 b 所在的两个集合：
size[find(b)] += size[find(a)];
p[find(a)] = find(b);

//(3) 维护到祖宗节点距离的并查集：
int p[N], d[N];
//p[] 存储每个点的祖宗节点, d[x] 存储 x 到 p[x] 的距离
// 返回 x 的祖宗节点
int find(int x) {
    if (p[x] != x) {
        int u = find(p[x]);
        d[x] += d[p[x]];
        p[x] = u;
    }
    return p[x];
}
// 初始化, 假定节点编号是 1~n
for (int i = 1; i <= n; i ++ ) {
    p[i] = i;
    d[i] = 0;
}
// 合并 a 和 b 所在的两个集合：
```

```
p[find(a)] = find(b);
d[find(a)] = distance; // 根据具体问题，初始化 find(a) 的偏移量

/*
例题：连通块中点的数量
给定一个包含 n 个点（编号为 1~n）的无向图，初始时图中没有边。
现在要进行 m 个操作，操作共有三种：
“C a b”，在点 a 和点 b 之间连一条边，a 和 b 可能相等
“Q1 a b”，询问 a 和 b 是否在同一个连通块中，a 和 b 可能相等
“Q2 a”，询问点 a 所在连通块中点的数量
第一行输入整数 n 和 m，1 <= n, m <= 1e5
接下来 m 行，每行包含一个操作指令，指令为“C a b”，“Q1 a b”或
↪ “Q2 a”中的一种。
输出格式
对于每个询问指令”Q1 a b”，如果 a 和 b 在同一个连通块中，则输出
↪ “Yes”，否则输出“No”
对于每个询问指令“Q2 a”，输出一个数表示点 a 所在连通块中点的数量
*/
const int N = 100010;
int p[N], size[N];
int find(int x) {
    if (x != p[x]) p[x] = find(p[x]);
    return p[x];
}
int main() {
    int n, m, a, b;
    scanf("%d%d", &n, &m);
    char s[2];
    for (int i = 1; i <= n; i++) {
        p[i] = i;
        size[i] = 1;
    }
    while (m--) {
        scanf("%s", s);
        if (s[0] == 'C') {
            scanf("%d%d", &a, &b);
            if ( find(a) == find(b)) continue;
            //注意需要先更新 size 再合并 如果先合并 size 就错了
            size[find(b)] += size[find(a)];
            p[find(a)] = find(b);
```

```
        } else if (s[1] == '1') {
            scanf("%d%d", &a, &b);
            if (find(a) == find(b)) puts("Yes");
            else puts("No");
        } else {
            scanf("%d", &a);
            printf("%d\n", size[find(a)]);
        }
    }
    return 0;
}


/*
例题：假设 x1,x2,x3,…代表程序中出现的变量，给定 n 个形如 xi=xj 或 xi xj 的变量相等/不等的约束条件，请判定是否可以分别为每一个变量赋予恰当的值，使得上述所有约束条件同时被满足。
例如，一个问题中的约束条件为：x1=x2，x2=x3，x3=x4，x1 x4，这些约束条件显然是不可能同时被满足的，因此应判定为不可被满足。
现在给出一些约束满足问题，请分别对它们进行判定。
输入文件的第 1 行包含 1 个正整数 t，表示需要判定的问题个数，注意这些问题之间是相互独立的。
对于每个问题，包含若干行：
第 1 行包含 1 个正整数 n，表示该问题中需要被满足的约束条件个数。
接下来 n 行，每行包括 3 个整数 i,j,e，描述 1 个相等/不等的约束条件，相邻整数之间用单个空格隔开。若 e=1，则该约束条件为 xi=xj；若 e=0，则该约束条件为 xi xj。
*/
//离散化 +DSU
//最多只有 2e6 个点 但是点下标最大 1e9 所以要离散化
//保序离散化是排序 + 判重 + 二分 不需要保序的直接 map/哈希表
//此题的离散化不需要保序 直接 map 即可
const int N = 2000010;
int n, m;
int p[N];
unordered_map<int, int> S;
struct Query {
    int x, y, e;
} query[N]; //存放离散化之后的询问的下标
int get(int x) {
```

```
    if (S.count(x) == 0) S[x] = ++n;
    return S[x];
}
int find(int x) {
    if (x != p[x]) p[x] = find(p[x]);
    return p[x];
}
int main() {
    int t;
    scanf("%d", &t);
    while (t --) {
        n = 0;
        S.clear();
        scanf("%d", &m);
        for (int i = 0; i < m; i++) {
            int x, y, e;
            scanf("%d%d%d", &x, &y, &e);
            query[i] = {get(x), get(y), e}; //加入的时候离散化
        }
        //n 就是离散化之后所有的点的个数
        for (int i = 1; i <= n; i++) p[i] = i; //初始化 DSU
        //合并所有相等的约束条件
        for (int i = 0; i < m; i++) {
            if (query[i].e == 1) {
                int pa = find(query[i].x), pb =
                ↪ find(query[i].y);
                p[pa] = pb;
            }
        }
        //检查所有不等条件
        bool has_conflict = false;
        for (int i = 0; i < m; i++)
            if (query[i].e == 0) {
                int pa = find(query[i].x), pb =
                ↪ find(query[i].y);
                if (pa == pb) {
                    has_conflict = true;
                    break;
                }
            }
```

```
        if (has_conflict) puts("NO");
        else puts("YES");
    }
    return 0;
}
```

2.6 树状数组

```
//树状数组 区间查询/单点修改
/*
给定长度为 N 的数列 A，然后输入 M 行操作指令。
第一类指令形如“C l r d”，表示把数列中第 l~r 个数都加 d。
第二类指令形如“Q X”，表示询问数列中第 x 个数的值。
对于每个询问，输出一个整数表示答案。
*/
const int N = 1e5 + 10;
int n, m;
int a[N];
ll tr[N];
int lowbit(int x) {
    return x & -x;
}
void add(int x , int c) {
    for (int i = x; i <= n; i += lowbit(i)) tr[i] += c;
}
ll sum(int x) {
    ll res = 0;
    for (int i = x; i; i -= lowbit(i)) res += tr[i];
    return res;
}
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1 ; i <= n; i++) scanf("%d", &a[i]);
    //初始化差分数组 暴力方法建树
    //查询某个数就是直接求和即可
    //所需的操作就是差分数组的单点修改 区间查询
    for (int i = 1; i <= n; i++) add(i, a[i] - a[i - 1]);
    while (m --) {
        char op[2];
        int l, r, d;
        scanf("%s%d", op, &l);
```

```
        if (*op == 'C') {
            scanf("%d%d", &r, &d);
            add(l, d); add(r + 1, -d);
        }
        else {
            printf("%lld\n", sum(l));
        }
    }
    return 0;
}
/*
例题：如果三个点 (i,yi),(j,yj),(k,yk) 满足 1<=i<j<k<=n 且
↪ yi>yj,yj<yk，则称这三个点构成 V 图腾；
如果三个点 (i,yi),(j,yj),(k,yk) 满足 1<=i<j<k<=n 且
↪ yi<yj,yj>yk，则称这三个点构成 ^ 图腾；
第一行一个数 n。第二行是 n 个数，分别代表 y1，y2,⋯,yn。
输出：两个数，中间用空格隔开，依次为 V 的个数和 ^ 的个数。
*/
//依次枚举每个点为最低点或者最高点
//从然后划分为左边和右边 计算左边有多少个 y 比当前点的 y 高或者
↪ 低 左右乘起来就是当前集合的总数、
//求出所有点代表的集合 然后求和即可
//在这个过程中 扫描到一个 y 比当前 y 大的点 那么当前 y 增加 1 还
↪ 需要统计区间和 2 个功能就可以用树状数组实现了
const int N = 200010;
int n;
int a[N];
int tr[N]; //树状数组
int gr[N], lo[N]; //每个位置前面的位置 有多少位置的高度比当前位
↪ 置的高度高 / 低
int lowbit(int x) {
    return x & -x;
}
void add(int x, int c) { //单点增加 修改全部父结点
    for (int i = x; i <= n; i += lowbit(i)) tr[i] += c;
}
int sum(int x) {
    int res = 0;
    for (int i = x; i; i -= lowbit(i)) res += tr[i];
    return res;
```

```
}
/*
从左向右依次遍历每个数 a[i]，使用树状数组统计在 i 位置之前所有比 a[i] 大的数的个数、以及比 a[i] 小的数的个数。
统计完成后，将 a[i] 加入到树状数组。
原理就是把 tr[] 下标当作高度，要把位置 i 看成横坐标的话，那么高度就得看成纵坐标，从左往右扫的时候只会求出这个点左边比自己小的和大的
第二次从右往左扫之前要先清空树状数组，然后重新扫 不然会重复 但是左边的结果已经存在了 gr 和 lo 数组中.
*/
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
    for (int i = 1; i <= n; i++) {
        int y = a[i]; //当前点的高度
        gr[i] = sum(n) - sum(y);
        lo[i] = sum(y - 1);
        add(y, 1);
    }
    //清空 然后倒着扫一次
    memset(tr, 0, sizeof tr);
    ll res1 = 0, res2 = 0; //v 以及 ^
    for (int i = n; i; i--) {
        int y = a[i];
        res1 += gr[i] * 1ll * (sum(n) - sum(y));
        res2 += lo[i] * 1ll * (sum(y - 1));
        add(y, 1);
    }
    printf("%lld %lld\n", res1, res2);
    return 0;
}
/*
例题：有 n 头奶牛，已知它们的身高为 1~n 且各不相同，但不知道每头奶牛的具体身高。
现在这 n 头奶牛站成一列，已知第 i 头牛前面有 Ai 头牛比它低，求每头奶牛的身高。
输入 n，第 2..n 行：每行输入一个整数 Ai，第 i 行表示第 i 头牛前面有 Ai 头牛比它低。
（注意：因为第 1 头牛前面没有牛，所以并没有将它列出）
```

```
*/
//从后往前扫 i 位置的排名就是剩下的牛里面 a[i] + 1 剩下是指前面没扫的包括当前位置的牛
//本质问题就是从剩余的数中找出第 k 小的数 以及删除某一个数的操作
//用 tr[] 数组的下标表示高度 所有元素置为 1 表示还没被删，从后往前扫
//对于每个 a[i] 也就是第 k 高 当 tr[j] 的前缀和等于 a[i] 的时候，也就是这时候的下标 j 就是答案
const int N = 100010;
int n; int a[N];
int ans[N];
int tr[N];
int lowbit(int x) {
    return x & -x;
}
void add(int x, int c) {
    for (int i = x; i <= n; i += lowbit(i)) tr[i] += c;
}
int sum(int x) {
    int res = 0;
    for (int i = x; i; i -= lowbit(i)) res += tr[i];
    return res;
}
int main() {
    scanf("%d", &n);
    //这里第一个 0 不读入默认 0
    for (int i = 2; i <= n; i++) scanf("%d", &a[i]);
    for (int i = 1; i <= n; i++) add(i, 1); //初始化树状数组
    for (int i = n; i; i--) {
        int k = a[i] + 1;
        //二分
        int l = 1, r = n;
        while (l < r) {
            int mid = l + r >> 1;
            if (sum(mid) >= k) r = mid;
            else l = mid + 1;
        }
        ans[i] = r;
        add(r, -1);
    }
```

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

2.7 线段树

```
/*
支持的操作：
单点修改/查询区间最大元素/区间 GCD
区间求和/区间最大连续子段和/区间修改
*/
/*
区间修改，区间查询模板
1、“C l r d”，表示把 A[l],A[l+1],⋯,A[r] 都加上 d。
2、“Q l r”，表示询问 数列中第 l~r 个数的和。
*/
const int N = 100010;
int n, m;
int w[N];
struct Node {
    int l, r;
    ll sum, add;
} tr[N * 4];
void pushup(int u) {
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
}
void pushdown(int u) {
    auto &root = tr[u], &left = tr[u << 1], &right = tr[u << 1
    ↪  | 1];
    if (root.add) {
        left.add += root.add, left.sum += 1ll * (left.r -
        ↪  left.l + 1) * root.add;
        right.add += root.add, right.sum += 1ll * (right.r -
        ↪  right.l + 1) * root.add;
        root.add = 0;
    }
}
void modify(int u, int l, int r, int d) {
    if (tr[u].l >= l && tr[u].r <= r) {
        tr[u].sum += 1ll * (tr[u].r - tr[u].l + 1) * d;
        tr[u].add += d;
```

```
    }
    else {
        pushdown(u); //需要分裂 然后递归给子节点打 tag
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u << 1, l, r, d);
        if (r > mid) modify(u << 1 | 1, l, r, d);
        pushup(u);
    }
}
ll query(int u, int l, int r) {
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].sum;
    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    ll sum = 0;
    if (l <= mid) sum = query(u << 1, l, r);
    if (r > mid) sum += query(u << 1 | 1, l, r);
    return sum;
}
void build(int u, int l, int r) {
    if (l == r) tr[u] = {l, r, w[r], 0};
    else {
        tr[u] = {l, r};
        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) scanf("%d", &w[i]);
    build(1, 1, n);
    char op[2];
    int l, r, d;
    while (m --) {
        scanf("%s%d%d", op, &l, &r);
        if (*op == 'C') {
            scanf("%d", &d);
            modify(1, l, r, d);
        }
        else printf("%lld\n", query(1, l, r));
```

```
    }
    return 0;
}
/*
区间最大连续子段和/单点修改
1、“1 x y”，查询区间 [x,y] 中的最大连续子段和
2、“2 x y”，把 A[x] 改成 y。
*/
//记录左子节点的最大后缀和 和 右边子节点的最大前缀和
//那么父节点的最大子段和就是 max{左子节点最大子段和，右子节点最大子段和 ，左子节点最大后缀 + 右子节点最大前缀
//这个过程是递归的 回溯的时候 子节点只会 pushup 自己段的最大前后缀给父节点
const int N = 5e5 + 10;
int n, m;
int w[N]; //序列
struct Node {
    int l, r;
    int sum; //当前段的区间和
    int lmax, rmax; //子节点最大前后缀
    int tmax; //当前段的最大子段和
} tr[N * 4];
void pushup(Node &u, Node &l, Node & r) {
    u.sum = l.sum + r.sum;
    u.lmax = max(l.lmax, l.sum + r.lmax); //维护
    u.rmax = max(r.rmax, r.sum + l.rmax);
    u.tmax = max(max(l.tmax, r.tmax), l.rmax + r.lmax);
}
void pushup(int u) {
    pushup(tr[u], tr[u << 1], tr[u << 1 | 1]);
}
void build(int u, int l, int r) {
    tr[u].l = l, tr[u].r = r;
    if (l == r) tr[u] = {l, r, w[r], w[r], w[r], w[r]};
    else {
        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}
```

```
int modify(int u, int x, int v) {
    if (tr[u].l == x && tr[u].r == x) tr[u] = {x, x, v, v, v,
    ↪  v};
    else {
        int mid = tr[u].l + tr[u].r >> 1;
        if (x <= mid) modify(u << 1, x, v);
        else modify(u << 1 | 1, x, v);
        pushup(u);
    }
}
Node query(int u, int l, int r) {
    if (tr[u].l >= l && tr[u].r <= r) return tr[u];
    else {
        int mid = tr[u].l + tr[u].r >> 1;
        if (r <= mid) return query(u << 1, l, r);
        else if (l > mid) return query(u << 1 | 1, l , r);
        else {
            auto left = query(u << 1, l , r);
            auto right = query(u << 1 | 1, l, r);
            Node res;
            pushup(res, left, right);
            return res;
        }
    }
}
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) scanf("%d", &w[i]);
    build(1, 1, n);
    int k, x, y;
    while (m --) {
        scanf("%d%d%d", &k, &x, &y);
        if (k == 1) {
            if (x > y) swap(x, y);
            printf("%d\n", query(1, x, y).tmax);
        }
        else modify(1, x, y);
    }
    return 0;
}
/*
```

```
单点修改/区间最大数
1 添加操作：向序列后添加一个数，序列长度变成 n+1；
2 询问操作：询问这个序列中最后 L 个数中最大的数是多少。
*/
const int N = 200010;
int m, p;
struct Node {
    int l, r ;
    int v; //区间 [l,r] 的最大值
} tr[N * 4];
//子节点更新父节点
void pushup(int u) {
    tr[u].v = max(tr[u << 1].v, tr[u << 1 | 1].v);
}
int query(int u, int l, int r) {
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].v; //查询的
    ↪ 区间完全包含在树中的某个段区间内 段的最大值就是候选值
    int mid = tr[u].l + tr[u].r >> 1;
    int v = 0;
    if (l <= mid) v = query(u << 1, l, r); // 左边 有交集 递归
    ↪ 向下查左边 注意 l <= mid
    if (r > mid) v = max(v, query(u << 1 | 1, l , r)); //右边有
    ↪ 交集 递归查右边
    return v;
}
void modify(int u, int x, int v) { //将 x 位置的点的值修改为 v
    if (tr[u].l == x && tr[u].r == x) tr[u].v = v; //叶节点找到
    ↪ 了直接修改
    else {
        int mid = tr[u].l + tr[u].r >> 1;
        if (x <= mid) modify(u << 1, x , v);
        else modify(u << 1 | 1, x, v);
        pushup(u); //修改之后要更新最大值 回溯的时候
    }
}
void build(int u, int l, int r) {
    tr[u] = {l, r};
    if (l == r) return ;
    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
```

```
    pushup(u);
}
int main() {
    int n = 0, last = 0; //上一次查询的结果
    scanf("%d%d", &m, &p);
    build(1, 1, m);
    char op[2];
    int x;
    int l;
    while (m --) {
        scanf("%s%d", op, &x);
        //如果是 A t，则表示向序列后面加一个数，加入的数是 (t+a)
        ↪ mod p。其中，t 是输入的参数，
        //a 是在这个添加操作之前最后一个询问操作的答案（如果之前
        ↪ 没有询问操作，则 a=0)。
        if (*op == 'Q') {
            last = query(1, n - x + 1, n);
            printf("%d\n", last);
        }
        else { //修改操作 由题意而定
            modify(1, n + 1, (last + x) % p);
            n ++;
        }
    }
    return 0;
}
/*
例题：维护序列
把数列中的一段数全部乘一个值；
把数列中的一段数全部加一个值；
询问数列中的一段数的和，结果模 p。
操作 1：1 t g c，表示把所有满足 t i g 的 ai 改为 ai×c；
操作 2：2 t g c，表示把所有满足 t i g 的 ai 改为 ai+c；
操作 3：3 t g，询问所有满足 t i g 的 ai 的和 模 P 的值。
*/
//全部使用先乘再加的操作顺序 这样可以维护计算的表达式一致性
//x * a + b) * c +  d) * e + f
//开 2 个 lazy tag 记录每次乘或者加即可
//加法 lazytag：与正常的线段树一样
//乘法 lazytag：将原先的加法 glazytag 乘上需要乘的数，
```

```
//再将乘法 lazytag 乘上需要乘的数。下发 lazytag 时先下发乘法标记
const int N = 100010;
int n, p, m;
int w[N];
struct Node {
    int l, r;
    int sum, add, mul;
} tr[N * 4];
void pushup(int u) {
    tr[u].sum = (tr[u << 1].sum + tr[u << 1 | 1].sum) % p;
}
void eval(Node &t, int add, int mul) {
    t.sum = ((ll)t.sum * mul + (ll)(t.r - t.l + 1) * add) % p;
    t.mul = (ll)t.mul * mul % p;
    t.add = ((ll)t.add * mul + add) % p;
}
void pushdown(int u) {
    eval(tr[u << 1], tr[u].add, tr[u].mul);
    eval(tr[u << 1 | 1], tr[u].add, tr[u].mul);
    tr[u].add = 0,  tr[u].mul = 1;
}
void modify(int u, int l, int r, int add, int mul) {
    if (tr[u].l >= l && tr[u].r <= r) eval(tr[u], add, mul);
    else {
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u << 1, l, r, add, mul);
        if (r > mid) modify(u << 1 | 1, l, r, add , mul);
        pushup(u);
    }
}
ll query(int u, int l, int r) {
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].sum;
    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    ll sum = 0;
    if (l <= mid) sum = query(u << 1, l, r) % p;
    if (r > mid) sum = (sum + query(u << 1 | 1, l , r)) % p;
    return sum;
}
void build(int u, int l, int r) {
```

```
    if (l == r) tr[u] = {l, r, w[r], 0, 1}; //加 0 乘 1
    else {
        tr[u] = {l, r, 0, 0, 1};
        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}
int main() {
    scanf("%d%d", &n, &p);
    for (int i = 1; i <= n; i++) scanf("%d", &w[i]);
    build(1, 1, n);
    scanf("%d", &m);
    while (m --) {
        int t, l, r, d;
        scanf("%d%d%d", &t, &l, &r);
        if (t == 1) {
            scanf("%d", &d);
            modify(1, l, r, 0, d);
        }
        else if (t == 2) {
            scanf("%d", &d);
            modify(1, l, r, d, 1);
        }
        else printf("%d\n", query(1, l, r));
    }
    return 0;
}
/*
区间修改/区间最大 GCD
1、“C l r d”, 表示把 A[l],A[l+1],…,A[r] 都加上 d。
2、“Q l r”, 表示询问 A[l],A[l+1],…,A[r] 的最大公约数 (GCD)
*/
//gcd(a,b) = gcd(a, b-a) 差分
const int N = 500010;
int n, m;
ll w[N];
struct Node {
    int l, r;
    ll sum, d; //差分数组的区间和 以及 gcd
```

```
} tr[N * 4];
void pushup(Node &u, Node &l, Node &r) {
    u.sum = l.sum + r.sum;
    u.d = gcd(l.d, r.d);
}
void pushup(int u) {
    pushup(tr[u], tr[u << 1], tr[u << 1 | 1]);
}
void build(int u, int l, int r) {
    ll b = w[r] - w[r - 1]; //建立差分数组
    if (l == r) tr[u] = {l, r, b, b};
    else {
        tr[u].l = l, tr[u].r = r;
        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}
void modify(int u, int x, ll v) {
    if (tr[u].l == x && tr[u].r == x) {
        ll b = tr[u].sum + v;
        tr[u] = {x, x, b, b};
    }
    else {
        int mid = tr[u].l + tr[u].r >>  1;
        if (x <= mid) modify(u << 1, x, v);
        else modify(u << 1 | 1, x, v);
        pushup(u);
    }
}
Node query(int u, int l, int r) {
    if (l > r) return {0};
    if (tr[u].l >= l && tr[u].r <= r) return tr[u];
    else {
        int mid = tr[u].l + tr[u].r >> 1;
        if (r <= mid) return query(u << 1, l, r);
        else if (l > mid) return query(u << 1 | 1, l, r);
        else {
            auto left = query(u << 1, l,  r);
            auto right = query(u << 1 | 1, l , r);
```

```
            Node res;
            pushup(res, left, right);
            return res;
        }
    }
}
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) scanf("%lld", &w[i]);
    build(1, 1, n);
    int l, r;
    ll d;
    char op[2];
    while (m --) {
        scanf("%s%d%d", op, &l, &r);
        if (*op == 'Q') {
            auto left = query(1, 1, l);
            Node right({0, 0, 0, 0});
            if (l + 1 <= r) right = query(1, l + 1, r);
            printf("%lld\n", abs(gcd(left.sum, right.d)));
        }
        else {
            scanf("%lld", &d);
            modify(1, l , d);
            if (r + 1 <= n)  modify(1, r + 1, -d);
        }
    }
    return 0;
}
```

2.8 可持久化数据结构

2.8.1 可持久化 Trie

```
/*
例题：给定一个非负整数序列 a，初始长度为 N
有 M 个操作，有以下两种操作类型：
1、"A x"：添加操作，表示在序列末尾加一个数 x，序列的长度 N 增大 1
2、"Q l r x"：询问操作，你需要找到一个位置 p，满足 l p r，使得：
↪  a[p] xor a[p+1] xor ··· xor a[N] xor x 最大，输出这个最大值。
*/
```

```cpp
//可持久化 01trie + 最大异或对
//可持久化 trie 记录 每个子树的元素在序列中的最大位置 如果位置 <l-1 那就不递归这个子树
const int N = 600010; //初始长度是 N 然后有 M 个操作 每次都可能 加 最多加 2 倍
const int M = N * 25; //二进制表示的节点总数 1e7 < 2 ^ 25
int n, m;
int tr[M][2]; //01 二进制 trie
int max_id[M];//每个子树最大的点在原序列中的下标是多大
int root[N];
int s[N];
int idx;
//a[p] xor a[p + 1] xor ··· xor a[N] xor x 相当于 s[N] ^ x ^ s[p - 1]
//S[N] ^ x 每次可以看成一个固定值 v 提前算出来,
//则相当于 求 l-1 <= p-1 <= r-1 使得 v 与 s[p-1] 异或最大
//i 是前缀和 s 下标 k 当前处理的二进制位 p 上一个版本 q 最新版本
void insert(int i, int k, int p, int q) {
    if (k < 0) {
        max_id[q] = i;
        return;
    }
    int v = s[i] >> k & 1;
    if (p) tr[q][v ^ 1] = tr[p][v ^ 1]; //上一个版本的另外一位
    tr[q][v]  = ++idx;
    insert(i, k - 1, tr[p][v], tr[q][v]);
    max_id[q] = max(max_id[tr[q][0]], max_id[tr[q][1]]);
}
int query(int root, int c, int l) { //当前待匹配的查的值是 c 左边界限制是 l
    int p = root;
    for (int i = 23; i >= 0; i--) {
        int v = c >> i & 1;
        //如果二进制位相反的节点存在而且 maxid 满足条件
        if (max_id[tr[p][v ^ 1]] >= l) p = tr[p][v ^ 1];
        else p = tr[p][v];
    }
    return c ^ s[max_id[p]];
}
int main() {
```

```
    scanf("%d%d", &n , &m);
    max_id[0] = -1;
    root[0] = ++idx;
    insert(0, 23, 0, root[0]);
    for (int i = 1; i <= n; i++) {
        int x;
        scanf("%d", &x);
        s[i] = s[i - 1] ^ x;
        root[i] = ++idx;
        insert(i, 23, root[i - 1], root[i]);
    }
    char op[2];
    int l, r, x;
    while (m --) {
        scanf("%s", op);
        if (*op == 'A') {
            scanf("%d", &x);
            ++n;
            s[n] = s[n - 1] ^ x;
            root[n] = ++idx;
            //从最高位开始插
            insert(n, 23, root[n - 1], root[n]);
        }
        else {
            scanf("%d%d%d",  &l, &r, &x);
            printf("%d\n", query(root[r - 1], s[n] ^ x, l -
            ↪  1));
        }
    }
    return 0;
}
```

2.8.2 可持久化线段树

```
/*
例题：区间第 k 小 O((N+M)logN)M 次询问
给定长度为 N 的整数序列 A, 下标为 1 N。
现在要执行 M 次操作，其中第 i 次操作为给出三个整数 li,ri,ki, 求
↪  A[li],A[li+1],⋯,A[ri](即 A 的下标区间 [li,ri]) 中第 ki 小
↪  的数是多少。
*/
```

```cpp
#define all(x) (x).begin(), (x).end()
const int N = 100010;
int n, m;
int a[N];
vector<int> nums;
struct Node {
    int l, r; //左子节点 右子节点 不是区间
    int cnt; //有多少个点落在 离散化之后的下标范围内
} tr[N * 4 + N * 17]; //额外多 N* logN 空间 没有修改 只有询问
int root[N]; //每个版本的根节点
int idx;
int find(int x) {
    return lower_bound(all(nums), x) - nums.begin();
}
int build(int l, int r) {
    int p = ++idx;
    if (l == r) return p;
    int mid = l + r >> 1;
    tr[p].l = build(l, mid), tr[p].r = build(mid + 1, r);
    return p;
}
//上一个版本的 root 数组的左边界下标 右边界下标 a[i] 离散化之后
↪ 的下标值
int insert(int p, int l, int r, int x) {
    int q = ++idx;
    tr[q] = tr[p]; //复制
    if (l == r) {
        tr[q].cnt ++;
        return q;
    }
    int mid = l + r >> 1;
    if (x <= mid) tr[q].l = insert(tr[p].l, l, mid, x);
    else tr[q].r = insert(tr[p].r, mid + 1, r, x);
    tr[q].cnt = tr[tr[q].l].cnt + tr[tr[q].r].cnt;
    return q;
}
//后边的版本 前面的版本 左右查询区间 第 k 大
int query(int q, int p, int l, int r, int k) {
    if (l == r) return r;
    int cnt  = tr[tr[q].l].cnt - tr[tr[p].l].cnt;
```

```
    int mid = l + r >> 1;
    if (k <= cnt) return query(tr[q].l, tr[p].l, l, mid, k);
    else return query(tr[q].r, tr[p].r, mid + 1, r, k - cnt);
}
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        nums.push_back(a[i]);
    }
    sort(all(nums));
    nums.erase(unique(all(nums)), nums.end()); //离散化
    root[0] = build(0, nums.size() - 1);
    for (int i = 1; i <= n; i++) root[i] = insert(root[i - 1],
    ↪  0, nums.size() - 1, find(a[i]));
    while (m --) {
        int l, r, k;
        scanf("%d%d%d", &l, &r, &k);
        printf("%d\n", nums[query(root[r], root[l - 1], 0,
        ↪  nums.size() - 1, k)]);
    }
    return 0;
}
```

2.9 平衡树

2.9.1 Treap

```
/*
1 插入数值 x
2 删除数值 x(若有多个相同的数，应只删除一个)。
3 查询数值 x 的排名 (若有多个相同的数，应输出最小的排名)。
4 查询排名为 x 的数值。
5 求数值 x 的前驱 (前驱定义为小于 x 的最大的数)。
6 求数值 x 的后继 (后继定义为大于 x 的最小的数)。
注意：数据保证查询的结果一定存在。
*/
const int N = 100010;
const int INF = 1e8; //边界点
struct Node {
    int l, r;
```

```
    int key, val;
    int cnt, sz; //该数出现的次数 每个以当前根的子树的大小
} tr[N];
int n; //题目操作数
int root, idx;
void pushup(int p) {
    tr[p].sz = tr[tr[p].l].sz + tr[tr[p].r].sz + tr[p].cnt;
}
int get_node(int key) { //创建节点
    tr[++idx].key = key;
    tr[idx].val = rand();
    tr[idx].cnt = 1;
    tr[idx].sz = 1;
    return idx;
}
void build() {
    get_node(-INF), get_node(INF);
    root = 1, tr[1].r = 2;
    pushup(root);
}
void zig(int &p) { //右旋转 注意传入的是引用
    int q = tr[p].l;
    tr[p].l = tr[q].r;
    tr[q].r = p;
    p = q; //更改指针的指向
    //刷新改变的位置以及在原树中的节点的位置
    pushup(tr[p].r), pushup(p);
}
void zag(int &p) {
    int q = tr[p].r;
    tr[p].r = tr[q].l;
    tr[q].l = p;
    p = q;
    pushup(tr[p].l), pushup(p);
}
void insert(int &p, int key) {
    if (!p) p = get_node(key);
    else if (tr[p].key == key) tr[p].cnt ++;
    else if (tr[p].key > key) {
        insert(tr[p].l, key);
```

```
        if (tr[tr[p].l].val > tr[p].val) zig(p); //不满足堆的性
        ↪ 质 就右旋
    }
    else {
        insert(tr[p].r, key);
        if (tr[tr[p].r].val > tr[p].val) zag(p); //左旋
    }
    pushup(p);
}
void del(int &p, int key) {
    if (!p) return ;
    if (tr[p].key == key) {
        if (tr[p].cnt > 1) tr[p].cnt --;
        else if (tr[p].l || tr[p].r) { //不是叶节点
            // 如果右子树是空 或者左子树 val 大于右子树 那么就向
            ↪ 下右旋
            if (!tr[p].r || tr[tr[p].l].val > tr[tr[p].r].val)
            ↪ {
                zig(p);
                //p 在树上是个固定位置的指针
                //zig 之后原来的 p 会跑到右边子节点 但是新的 p
                ↪ 依然固定的位置
                del(tr[p].r, key);
            } else {
                zag(p);
                del(tr[p].l, key);
            }
        }
        else p = 0; //是子节点 1
    }
    else if (tr[p].key > key) del(tr[p].l, key);
    else  del(tr[p].r, key);
    pushup(p);
}
int get_rank_by_key(int p, int key) { //通过数值找排名
    if (!p) return 0; //无解的情况
    if (tr[p].key == key) return tr[tr[p].l].sz + 1;
    //要查找的 key 比当前节点的 key 小 那么就去左树查排名
    if (tr[p].key > key) return get_rank_by_key(tr[p].l, key);
    //去右子树查在右子树的排名 所以要加上左子树的 size
```

```
    return tr[tr[p].l].sz + tr[p].cnt +
    ↪ get_rank_by_key(tr[p].r, key);
}
int get_key_by_rank(int p, int rank) { //通过排名找数值
    if (!p) return INF; //本题不会发生 比如只有 99 点 问第 1000
    ↪ 个点的排名
    //当前节点子树比 rank 还大 那就去左子树找
    if (tr[tr[p].l].sz >= rank) return
    ↪ get_key_by_rank(tr[p].l, rank);
    //左边的太小 但是加上现在节点 cnt 又太大 那么当前节点的 key
    ↪ 必然就是答案
    if (tr[tr[p].l].sz + tr[p].cnt >= rank) return tr[p].key;
    //在右子树的排名要先减掉左子树和当前节点的 cnt
    return get_key_by_rank(tr[p].r, rank - tr[tr[p].l].sz -
    ↪ tr[p].cnt);
}
//找到严格小于 key 的最大数
//注：找到小于等于 key 的最小数 板子里面的符号改成大于就行了
int get_prev(int p, int key) {
    if (!p) return -INF;
    if (tr[p].key >= key) return get_prev(tr[p].l, key);
    //如果比当前节点的 key 大 那么当前的 key 可能是答案 去右子树
    ↪ 找的时候取 max 即可
    return max(tr[p].key, get_prev(tr[p].r, key));
}
//找到大于 key 的最小数
//找到大于等于 key 的最小数 板子里面的符号改成小于就行了
int get_next(int p, int key) {
    if (!p) return INF;
    if (tr[p].key <= key) return get_next(tr[p].r, key);
    return min(tr[p].key, get_next(tr[p].l, key));
}
int main() {
    build();
    scanf("%d", &n);
    while (n --) {
        int opt, x;
        scanf("%d%d", &opt, &x);
        if (opt == 1) insert(root, x);
        else if (opt == 2) del(root, x);
```

```
        //由于 build 时候加入了-INF 节点 查 rank 和 key 时候 注意加一减一
        else if (opt == 3) printf("%d\n", get_rank_by_key(root, x) - 1);
        else if (opt == 4) printf("%d\n", get_key_by_rank(root, x + 1));
        else if (opt == 5) printf("%d\n", get_prev(root, x));
        else if (opt == 6) printf("%d\n", get_next(root, x));
    }
    return 0;
}
```

2.10 AC 自动机

```
/*
例题：给定 n 个长度不超过 50 的由小写英文字母组成的单词，以及一篇长为 m 的文章。请问，有多少个单词在文章中出现了。
第一行包含整数 T，表示共有 T 组测试数据。对于每组数据，第一行一个整数 n，接下去 n 行表示 n 个单词，最后一行输入一个字符串，表示文章。
1 <= n <= 1e4, 1 <= m <= 1e6
*/
//AC 自动机处理 next 的逻辑就是用当前节点的 next 来同时更新子节点的 next 同时延长前后缀匹配
const int N = 10010, S = 55;
const int M = 1000010;
int n, tr[N * S][26]; //trie
int cnt[N * S]; //每个节点结尾的单词的数量
char str[M];
int q[N * S]; //bfs 宽搜队列
int ne[N * S];
int idx; //trie idx
void insert() {
    int p = 0; //根节点
    for (int i = 0; str[i]; i ++) {
        int t = str[i] - 'a';
        if (!tr[p][t]) tr[p][t] = ++idx;
        p = tr[p][t];
    }
    cnt[p] ++;
```

```
}
//bfs 构建 AC 自动机
void build() {
    int hh = 0, tt = -1;
    for (int i = 0 ; i < 26; i++) {
        //将根节点的子节点全部入队 就是第一层
        if (tr[0][i]) q[++tt] = tr[0][i];
    }
    while (hh <= tt) {
        int t = q[hh++];
        for (int i = 0; i < 26; i++) {
            int &c = tr[t][i];
            //trie 图优化：
            //如果 j 的子节点 c 不存在 那么就让 c 直接跳到 j 的
            ↪ next 节点的相同字母的子节点
            if (!c)  tr[t][i] = tr[ne[t]][i];
            else {
                //如果子节点 c 存在 那就和朴素的写法一样 更新子
                ↪ 节点的 next
                ne[c] = tr[ne[t]][i];
                q[++tt] = c;
            }
        }
    }
}
int main() {
    int T;
    cin >> T;
    while (T --) {
        memset(tr, 0, sizeof tr);
        idx = 0;
        memset(cnt, 0, sizeof cnt);
        memset(ne, 0, sizeof ne);
        scanf("%d", &n);
        for (int i = 0; i < n; i++) {
            scanf("%s", str);
            insert();
        }
        build();
        scanf("%s", str); //待匹配的长串
```

```
        int res = 0;
        for (int i = 0, j = 0; str[i]; i++) {
            int t = str[i] - 'a';
            //trie 图优化:
            j = tr[j][t];  //不用再 while 循环了
            int p = j;
            while (p) { //会破坏复杂度
                res += cnt[p];
                cnt[p] = 0;
                p = ne[p];
            }
        }
        printf("%d\n", res);
    }
    return 0;
}
/*
例题: 每个单词分别在论文中出现多少次。第一行一个整数 N, 表示有多
↪  少个单词。接下来 N 行每行一个单词, 单词中只包含小写字母。
输入:
3
a
aa
aaa
输出:
6
3
1
*/
//题意是每个单词在所有单词出现的次数 不是一个长串进行匹配
/*
    一个单词如果不是其他单词的字串 那么就是一次
    如果是其他串的子串, 那就统计后缀
    总共的次数是所有的“前缀的后缀就是原串”的后缀总数 + 原来串
    计算方法: 逆向来算, 对于每个后缀 求出与之对应的前缀 也就是
↪  ne 数组处理
    ne[i], ne[ne[i]], ne[ne[ne[i]]].... 这样就会处理一个串的所
↪  有 与后缀匹配的前缀
    假设 f[i] 表示 i 结尾的单词的出现的次数 那直接把 f[i] 加到
↪  f[ne[i]] 上去
```

```
    因为单词都是在 trie 的树根连接的 也就是 trie 的前缀，一个后缀出现了 n 次，与之对应的前缀就出现了 n 次
    与后缀匹配的前缀在 ne 数组已经预处理好了，由 fail 指针的定义，i 向 ne[i] 连边，图中不会出现环
    直接按照拓扑序从下往上树根处递推，最后枚举一下所有单词，也就是和树根连接的这些，求和即可
*/
const int N = 1000010;
int n;
int tr[N][26], idx;
int f[N]; //每个单词出现的次数
int q[N], ne[N]; //q[] 是 bfs 队列
char str[N];
int id[210]; //每个单词对应的节点下标
void insert(int x) {
    int p = 0;
    for (int i = 0; str[i] ; i++) {
        int t = str[i] - 'a';
        if (!tr[p][t]) tr[p][t] = ++idx;
        p = tr[p][t];
        f[p] ++; //这里统计的是前缀而不是每个单词的结尾，因为单词会出现 其他单词中 每个字母都算
    }
    id[x] = p; //这里 id 记录的是每个单词结尾的节点号
}
void build() {
    int hh = 0, tt = -1;
    for (int i = 0; i < 26; i++)
        if (tr[0][i]) q[++tt] = tr[0][i];
    while (hh <= tt) {
        int t = q[hh++];
        for (int i = 0; i < 26; i++) {
            int &p = tr[t][i];
            if (!p) p = tr[ne[t]][i];
            else {
                ne[p] = tr[ne[t]][i];
                q[++tt] = p;
            }
        }
    }
```

```
}
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%s", str);
        insert(i);
    }
    build();
    //从树底部网上拓扑序遍历 那么 bfs 的倒序就是拓扑序
    for (int i = idx - 1; i >= 0; i--) f[ne[q[i]]] += f[q[i]];
    for (int i = 0; i < n; i++) printf("%d\n", f[id[i]]);
    return 0;
}
```

3 搜索

3.1 BFS

3.1.1 Flood Fill

```
/*
第一行包含两个整数 N 和 M。
接下来 N 行，每行包含 M 个字符，字符为”W”或”.”，用以表示矩形土
↪ 地的积水状况，字符之间没有空格。
*/
#define x first
#define y second
const int N = 1010;
typedef pair<int, int> pii;
int n, m;
char g[N][N];
const int M = N * N;
bool st[N][N];
pii q[M];
void bfs(int sx, int sy) {
    int hh = 0, tt = 0;  //初始化
    q[0] = {sx, sy};
    st[sx][sy] = true;
    while (hh <= tt) {
        pii t = q[hh++];
        for (int i = t.x - 1; i <= t.x + 1; i++) //八连通遍历
            for (int j = t.y - 1; j <= t.y + 1; j++) {
                // 是中心点自己 跳过 已经标记过了 在函数开头
                if (i == t.x && j == t.y) continue;
                if (i < 0 || i >= n || j < 0 || j >= m)
                    continue; // 越界
                //是否已经标记过 以及是否不是水洼
                if (g[i][j] == '.' || st[i][j]) continue;
                q[++tt] = {i, j};
                st[i][j] = true;
            }
    }
}
int main() {
    scanf("%d%d", &n, &m);
```

```
    for (int i = 0; i < n; i++)
        scanf("%s", g[i]);
    int cnt = 0;
    for (int i = 0;  i < n; i++) //逐行 bfs
        for (int j = 0 ; j < m; j++)
            if (g[i][j] == 'W' && !st[i][j]) {
                bfs(i, j);
                cnt ++;
            }
    printf("%d\n", cnt);
    return 0;
}

/*
山峰山谷问题：第一行包含一个正整数 n，表示地图的大小
接下来一个 n×n 的矩阵，表示地图上每个格子的高度 w
共一行，包含两个整数，表示山峰和山谷的数量。
输入样例：
5
5 7 8 3 1
5 5 7 6 6
6 6 6 2 8
5 7 2 5 8
7 1 0 1 7
输出样例：
3 3
*/
#define x first
#define y second
const int N = 1010, M = N * N;
typedef pair<int, int> pii;
int n;
int h[N][N];
pii q[M];
bool st[N][N];
//注意是高度相同 的连通块 再进行周围的判断 一个连通块的元素肯定
↪ 都是一样的一个连通块周围所有元素 都比连通块高 那就是山谷 反
↪ 之就是山峰
//bfs 函数调用一次就会遍历一整个连通块
```

```
void bfs(int sx, int sy, bool & has_heigher, bool & has_lower)
↪  {
    int hh = 0, tt = 0;
    q[0] = {sx, sy};
    st[sx][sy] = true;
    while (hh <= tt) {
        pii t = q[hh++];
        for (int i = t.x - 1; i <= t.x + 1; i++)
            for (int j = t.y - 1; j <= t.y + 1; j++) {
                //这个判断可有可无 如果不加 后边会在 st 判断时候
                ↪  不会入队
                if (i == t.x && j == t.y) continue;
                if (i < 0 || i >= n || j < 0 || j >= n)
                ↪  continue;
                //只要周围的点高度和中心点不一样，那就不在同一个
                ↪  连通块里
                //高度相同的 就直接不会进入下面的 if 判断
                if (h[i][j] != h[t.x][t.y]) {
                    if (h[i][j] > h[t.x][t.y]) has_heigher = 1;
                    else has_lower = 1;
                }
                else if (!st[i][j]) {
                    q[++tt] = {i, j};
                    st[i][j] = 1;
                }
            }
    }
}
int main() {
    scanf("%d", &n);
    for (int i = 0 ; i < n; i++)
        for (int j = 0 ; j  < n; j++)
            scanf("%d" , &h[i][j]);
    int peak = 0, valley = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0 ; j < n; j++)
            if (!st[i][j]) {
                bool has_heigher = false, has_lower = false;
                bfs(i , j, has_heigher, has_lower);
```

```
                //遍历整个连通块之后 周围没有比当前连通块高的就
                ↪ 是山峰
                //反之是山谷 或者是一整个连通块 那就都算 都 +1
                if (!has_heigher) peak++;
                if (!has_lower) valley ++;
            }
    printf("%d %d\n", peak, valley);
    return 0;
}
```

3.1.2 数码问题

```
/*
BFS 八数码求方案数：
输入占一行，将 3×3 的初始网格描绘出来空位用 x
输出占一行，包含一个整数，表示最少交换次数
*/
int bfs(string start) {
    string end = "12345678x";  //最终状态
    queue<string> q;
    unordered_map<string, int> d;
    q.push(start);
    d[start] = 0; //起点到起点的距离为 0
    int dx[4] = { -1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
    while (q.size()) {
        auto t = q.front();
        q.pop();
        int dist  = d[t];
        if (t == end) return dist;
        //状态转移
        int k = t.find('x');
        int x = k / 3, y = k % 3;  //  x 在矩阵的坐标
        for (int i = 0; i < 4; i++) {
            int a = x + dx[i], b = y + dy[i]; //枚举上下左右的
            ↪ 坐标
            if (a >= 0 && a < 3 & b >= 0 && b < 3) {
                swap(t[k], t[a * 3 + b]); //这里的交换是引用交
                ↪ 换 会改变 t 数组的值 不是浅拷贝
                if (!d.count(t)) { //新的状态
                    d[t] = dist + 1;
```

```
                    q.push(t);
                }
                swap(t[k], t[a * 3 + b]); //恢复状态
            }
        }
    }
    return -1;  //不存在方案
}
int main() {
    char str[2];
    string start;  //初始状态
    for (int i = 0; i < 9; i++) { //cin string 遇到空格就会忽略
    ↪ 所以要用 字符串拼接的方式读入
        char c;
        cin >> c;
        start += c;
    }
    cout << bfs(start) << endl;
    return 0;
}
```

3.1.3 BFS 最短路

```
/*
表示墙壁，0 表示可以走的路，只能横着走或竖着走，不能斜着走，要求
↪ 编程序找出从左上角到右下角的最短路线。
0, 1, 0, 0, 0,
0, 1, 0, 1, 0,
0, 0, 0, 0, 0,
0, 1, 1, 1, 0,
0, 0, 0, 1, 0,
第一行包含整数 n。接下来 n 行，每行包含 n 个整数 0 或 1，表示迷
↪ 宫。输出从左上角到右下角的最短路线，如果答案不唯一，输出任意
↪ 一条路径均可。
按顺序，每行输出一个路径中经过的单元格的坐标，左上角坐标为 (0,0),
↪ 右下角坐标为 (n-1,n-1)。
*/
typedef pair<int , int > pii;
#define x first
#define y second
```

```cpp
const int N = 1010, M = N * N;
int g[N][N];
int n;
pii pre[N][N];  //储存路径 当前节点 是由哪一个点转移过来的
pii q[M];
//输出路径的问题，一般从终点往起点搜，然后记录前驱
void bfs(int sx, int sy) {
    int dx[4] = { -1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
    int hh = 0, tt = 0;
    q[0] = {sx, sy};
    memset(pre, -1, sizeof pre);
    while (hh <= tt) {
        pii t = q[hh++];
        for (int i = 0 ; i < 4 ; i++) {
            int a = t.x + dx[i], b = t.y + dy[i];
            if (a < 0 || a >= n || b < 0 || b >= n) continue;
            if (g[a][b]) continue; // 1 不能走
            if (pre[a][b].x != -1) continue;
            q[++tt] = {a, b};
            pre[a][b] = t; //(a,b) 是从 t 转移过来的
        }
    }
}
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &g[i][j]);
    bfs(n - 1, n - 1); //bfs 一次就可以了 反向搜 然后从头输出
    pii end(0, 0);
    while (1) {
        printf("%d %d\n", end.x, end.y);
        if (end.x == n - 1 && end.y == n - 1) break;
        end = pre[end.x][end.y];
    }
    return 0;
}
/*
马走日问题：位置用’K’来标记，障碍的位置用’*’来标记，草的位置
↪ 用’H’来标记。输出一个整数，表示跳跃的最小次数。
```

```
输入样例:10 11
..........
....*.....
..........
...*.*....
.......*..
..*..*...H
*.........
...*...*..
.K........
...*.....*
..*....*..
输出样例：
5
*/
typedef pair<int, int> pii;
#define x first
#define y second
const int N = 155, M = N * N;
int n, m;
char g[N][N];
pii q[M];
bool st[N][N];
int dist[N][N]; //记录次数
int bfs() {
    //日字形的周围的点 共有 8 个 顺时针对应 xy 即可
    int dx[8] = { -2, -1, 1, 2, 2, 1, -1, -2};
    int dy[8] = {1, 2, 2, 1, -1, -2, -2, -1};
    int sx, sy;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            if (g[i][j] == 'K')
                sx = i, sy = j;
    int hh = 0, tt = 0;
    q[0] = {sx, sy};
    memset(dist, -1 , sizeof dist);
    dist[sx][sy] = 0;
    while (hh <= tt) {
        auto t = q[hh++];
        for (int i = 0 ; i < 8; i++) {
```

```
            int a = t.x + dx[i], b = t.y + dy[i];
            if (a < 0 || a >= n || b < 0 || b >= m) continue;
            if (g[a][b] == '*') continue;
            if (dist[a][b] != -1) continue; //被遍历过了
            if (g[a][b] == 'H') return dist[t.x][t.y] + 1;
            ↪ //已经找到
            dist[a][b] = dist[t.x][t.y] + 1;
            q[++tt] = {a, b};
        }
    }
    return -1;  //题目保证有解 所以写不写无所谓 前边已经返回了
}
int main() {
    cin >> m >> n;
    for (int i = 0 ; i < n; i++) cin >> g[i];
    cout << bfs() << endl;
    return 0;
}

/*
农夫和牛都位于数轴上，农夫起始位于点 N，牛位于点 K。农夫有两种移
↪ 动方式：
从 X 移动到 X-1 或 X+1，每次移动花费一分钟
从 X 移动到 2*X，每次移动花费一分钟
假设牛没有意识到农夫的行动，站在原地不动。农夫最少要花多少时间才
↪ 能抓住牛？输入：共一行，包含两个整数 N 和 K。
*/
const int N = 2e5 + 10;
int n, k;
int q[N];
int dist[N];
//注意 n 和 k 的大小关系是不确定的
int bfs() {
    memset(dist, -1 , sizeof dist);
    dist[n] = 0;
    q[0] = n;
    //相当于每次转移一层，每层都进行三种走法的尝试
    //最先到达 k 的走法 则是最树的最短路
    int hh = 0, tt = 0;
    while (hh <= tt) {
```

```
        int t = q[hh++];
        if (t == k) return dist[k];
        if (t + 1 < N && dist[t + 1] == -1) {
            dist[t + 1] = dist[t] + 1;
            q[++tt] = t + 1;
        }
        if (t - 1 >= 0 && dist[t - 1] == -1) {
            dist[t - 1] = dist[t] + 1;
            q[++tt] = t - 1;
        }
        if (t * 2 < N && dist[t * 2] == -1) {
            dist[t * 2] = dist[t] + 1;
            q[++tt] = t * 2;
        }
    }
    return -1;
}
int main() {
    cin >> n >> k;
    cout << bfs() << endl;
    return 0;
}
```

3.1.4 最少步数模型

```
/* 例题
1 2 3 4  //基本状态
8 7 6 5
A：交换上下两行；
B：将最右边的一列插入到最左边；
C：魔板中央对的 4 个数作顺时针旋转。
输入仅一行，包括 8 个整数，用空格分开，表示目标状态。
输出文件的第一行包括一个整数，表示最短操作序列的长度。
如果操作序列的长度大于 0，则在第二行输出字典序最小的操作序列。
*/
//最小步数模型的 BFS 里面，序列的状态使用哈希来存 包括八数码模型
//此题就用 stl 的 unordered_map 即可
//转移路径 就和迷宫那道题一样 记录一下转移的 pre 数组即可
//字典序的处理：每次都按照 A B C 的转移方式依次搜索
char g[2][4];
unordered_map<string, int> dist; //状态的步数
```

```
//从哪个状态转移过来的，操作用 char 存储
unordered_map<string, pair<char, string> > pre;
queue<string> q;
void set(string state) { //将字符串 顺时针放到矩阵中
    for (int i = 0; i < 4; i++) g[0][i] = state[i];
    for (int i = 3, j = 4; i >= 0; i--, j++) g[1][i] =
    ↪ state[j];
}
string get() { //以顺时针方向取出字符串
    string res;
    for (int i = 0 ; i < 4; i++) res += g[0][i];
    for (int i = 3;  i >= 0; i--) res += g[1][i];
    return res;
}
string move0(string state) {
    set(state);
    for (int i = 0 ; i < 4; i++) swap(g[0][i], g[1][i]);
    return get();
}
string move1(string state) {
    set(state);
    //将最后一列存下来，然后前面的往后移动一位，然后再把最后一列
    ↪ 插到第一列
    char v0 = g[0][3], v1 = g[1][3]; //存下最后一列
    for (int i = 3; i > 0; i--)
        for (int j = 0; j < 2; j++)
            g[j][i] = g[j][i - 1];
    g[0][0] = v0, g[1][0] = v1;  //最后一列插到第一列
    return get();
}
string move2(string state) {
    set(state);
    //每一个都顺时针移动，覆盖 存下左上角的元素
    char t = g[0][1];
    g[0][1] = g[1][1];
    g[1][1] = g[1][2];
    g[1][2] = g[0][2];
    g[0][2] = t;
    return get();
}
```

```
void bfs(string start, string end) {
    if (start == end) return ; //特判一下不需要操作的样例
    q.push(start);
    dist[start] = 0;
    while (q.size()) {
        auto t = q.front();
        q.pop();
        string m[3];  //三种操作 操作之后可以得到的状态序列 存到
        ↪ m 中
        m[0] = move0(t);
        m[1] = move1(t);
        m[2] = move2(t);
        for (int i = 0; i < 3; i++) { //三种操作 遍历
            string str = m[i];
            if (dist.count(str) == 0) {
                dist[str] = dist[t] + 1;
                //是由 t 状态转移过来的
                pre[str] = {char(i + 'A'), t};
                if (str == end) break;
                q.push(str);
            }
        }
    }
}
int main() {
    int x;
    string start, end; //start 是 12345678 逆序搜索
    for (int i = 0 ; i < 8; i++) {
        cin >> x;
        end += char(x + '0'); //将数字转为 char 拼接
    }
    for (int i = 0 ; i < 8; i++) start += char(i + '1');
    bfs(start, end);
    cout << dist[end] << endl;
    string res;
    while (end != start) {
        res += pre[end].first;
        end = pre[end].second;
    }
    //由于是反向搜索的操作顺序，所以需要逆序一下输出
```

```
    reverse(res.begin(), res.end());
    if (res.size()) cout << res << endl;
    return 0;
}
```

3.1.5 多源 BFS

```
/*
给定一个 N 行 M 列的 01 矩阵 A, A[i][j] 与 A[k][l] 之间的曼哈顿
↪ 距离定义为：
dist(A[i][j],A[k][l])=|i-k|+|j-l|
输出一个 N 行 M 列的整数矩阵 B, 其中：
B[i][j]=min{1 <= x <= N,1 <= y <=
↪ M,A[x][y]=1[dist(A[i][j],A[x][y])]}
输入:
第一行两个整数 n,m。接下来一个 N 行 M 列的 01 矩阵, 数字之间没有
↪ 空格。
输出格式: 一个 N 行 M 列的矩阵 B, 相邻两个整数之间用一个空格隔开
↪ 1 <= N,M <= 1000
输入样例:
3 4
0001
0011
0110
输出样例：
3 2 1 0
2 1 0 0
1 0 0 1
*/
typedef pair<int, int> pii;
#define x first
#define y second
const int N = 1010, M = N * N;
char g[N][N]; //读入没有空格 注意
int n, m;
pii q[M];
int dist[N][N];
//题意就是每个点到最近的 1 的距离, 如果为 1 自己距离那本身就是 0
//多源 BFS 求出离每个点最近的起点的距离
//先把所有 1 入队, 然后根据这些点开始向外扩张, 最终 bfs 全图
void bfs() {
```

```
    memset(dist, -1, sizeof dist);
    int hh = 0, tt = -1; //开始队内没有元素 不能写成 tt = 0
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            if (g[i][j] == '1') {
                dist[i][j] = 0;  //先把所有是 1 的点入队 保持距
                ↪ 离的单调性
                q[++tt] = {i, j};
            }
    int dx[4] = { -1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
    while (hh <= tt) {
        auto t = q[hh++];
        for (int i = 0 ; i < 4; i++) {
            int a = t.x + dx[i], b = t.y + dy[i];
            if (a < 0 || a >= n || b < 0 || b >= m) continue;
            if (dist[a][b] != -1) continue;
            dist[a][b] = dist[t.x][t.y] + 1;
            q[++tt] = {a, b};
        }
    }
}
int main() {
    scanf("%d%d", &n , &m);
    for (int i = 0 ; i < n; i++) scanf("%s", g[i]);
    bfs();
    for (int i = 0; i < n; i++) {
        for (int j = 0 ; j < m; j++)
            printf("%d ", dist[i][j]);
        printf("\n");
    }
    return 0;
}
```

3.1.6 双端队列 BFS

```
/* 例题：电路维修
第一行包含一个整数 T，表示测试数据的数目。
对于每组测试数据，第一行包含正整数 R 和 C，表示行数和列数。
之后 R 行，每行 C 个字符，字符是"/" 和"\" 中的一个，表示方向
输入样例：
1
```

```
3 5
\\/\\
\\///
/\\\\
输出样例：
1
*/
typedef pair<int, int> pii;
//一个点的横纵坐标和是奇数 就不能到达 偶数就可以到达
//这个图的边权只有 0 和 1, 能走就是 1, 不能走就是 0
//将 0 插入队头, 1 插入队尾, 这样才能保住队列的单调性
const int N = 510, M = N * N;
int n, m;
char g[N][N];
int dist[N][N];
bool st[N][N];  //判重
int bfs() {
    deque<pii> q;
    //每个点会入队多次
    int hh = 0, tt = 0;
    memset(st, 0, sizeof st);
    memset(dist, 0x3f, sizeof dist);
    char cs[] = "\\/\\/"; //每个点顺时针旋转四个方向的边
    //每个点对角线可以走到的 4 个点, 点-> 点
    int dx[4] = { -1, -1, 1, 1}, dy[4] = { -1, 1, 1, -1};
    //每个点 周围的 4 个斜线的坐标转换 点-> 线
    int ix[4] = { -1, -1, 0, 0}, iy[4] = { -1, 0, 0, -1};
    dist[0][0] = 0;
    q.push_back({0, 0});
    while (q.size()) {
        auto t = q.front();
        q.pop_front();
        int x = t.x, y = t.y;
        if (x == n && y == m) return dist[x][y];
        if (st[x][y]) continue;
        st[x][y] = true;
        for (int i = 0 ; i < 4; i++) {
            int a = x + dx[i], b = y + dy[i]; //周围的四个点
            if (a < 0 || a > n || b < 0 || b > m) continue;
            ↪ //格点是边数 +1 注意边界
```

```
                int ga = x + ix[i], gb = y + iy[i]; //周围的四条边
                int w = ( g[ga][gb] != cs[i] ); //能直达就是 0 需旋
                ↪ 转就是 1
                int d = dist[x][y] + w;
                if (d <= dist[a][b]) {
                    dist[a][b] = d;
                    if (!w) q.push_front({a, b});//边权是 0 入队头
                    else q.push_back({a, b}); //边权是 1 就加入队尾
                }
            }
        }
        return -1;  //不会被执行到
}
int main() {
    int t; scanf("%d", &t);
    while (t -- ) {
        scanf("%d%d", &n, &m);
        for (int i = 0; i < n; i ++ ) scanf("%s", g[i]);
        int t = bfs();
        if (n + m & 1 || t == 0x3f3f3f3f || t == -1) puts("NO
        ↪ SOLUTION");
        else printf("%d\n", bfs());
    }
    return 0;
}
```

3.1.7 双向 BFS

```
/* 例题：字串变换
已知有两个字串 A, B 及一组字串变换的规则（至多 6 个规则）：
A1 -> B1
A2 -> B2
…
规则的含义为：在 A 中的子串 A1 可以变换为 B1、A2 可以变换为 B2
↪ …。例如：A =’ abcd’ B =’ xyz’
变换规则为：
‘abc’ -> ‘xu’ ‘ud’ -> ‘y’ ‘y’ -> ‘yz’
则此时，A 可以经过一系列的变换变为 B，其变换的过程为：
‘abcd’ -> ‘xud’ -> ‘xy’ -> ‘xyz’
共进行了三次变换，使得 A 变换为 B 若在 10 步（包含 10 步）以内能
↪ 将 A 变换为 B ，则输出最少的变换步数；否则输出”NO ANSWER!”
```

```
输入格式如下：
A B
A1 B1
A2 B2 |-> 变换规则
…… /
所有字符串长度的上限为 20。
*/
//双向 BFS 一般都是在最少步数模型的题中，floodfill 和最短路不用
//双向 BFS 的过程中，每次选择规模较小的一端进行扩展，来平衡
const int N = 6;
string a[N], b[N];
int n; //规则的数量
int extend(queue<string> &q, unordered_map<string, int> &da,
 ↪  unordered_map<string, int> &db, string a[], string b[]) {
    string t = q.front();
    q.pop();
    for (int i = 0; i < t.size(); i++) // 逐个位置字符
        for (int j = 0; j < n; j++) //枚举规则
            if (t.substr(i, a[j].size()) == a[j]) { //匹配规则
             ↪  a
                string state = t.substr(0, i) + b[j] +
                 ↪  t.substr(i + a[j].size()); //替换匹配上的中
                 ↪  间一段，然后拼接起来为新的字符串
                if (db.count(state)) return da[t] + 1 +
                 ↪  db[state]; //a 可以扩展到 t，t 也可以扩展到
                 ↪  state，b 也可以扩展到 state 那么 ab 连通
                if (da.count(state)) continue ; //只有第一次扩
                 ↪  展才需要加进来
                da[state] = da[t] + 1;
                q.push(state);
            }
    return 11;  //没有和另外一端连通
}
int bfs(string A, string B) { //起点和终点
    queue<string> qa, qb;
    unordered_map<string , int> da, db; //两边到中点的距
    qa.push(A), da[A] = 0;  //起点开始搜
    qb.push(B), db[B] = 0; //终点开始搜
    while (qa.size() && qb.size()) { //一旦某一端没有元素了 说
     ↪  明 ab 没有连通
```

```
        //从较小的一端开始扩展 把 a 变成 b 或者 b 变成 a
        int t; //判断相遇需要的步数
        if (qa.size() <= qb.size()) t = extend(qa, da, db, a,
        ↪  b);
        else t =  extend(qb, db, da, b, a);
        if (t <= 10) return t;
    }
    return 11;
}
int main() {
    string A, B;
    cin >> A >> B;
    while (cin >> a[n] >> b[n]) n++; //规则的数量不一定，所以要
    ↪  统计
    int step = bfs(A, B);
    if (step > 10) puts("NO ANSWER!");
    else cout << step << endl;
    return 0;
}
```

3.1.8 A*

```
/*
例题：第 k 短路
给定一张 N 个点（编号 1,2…N），M 条边的有向图，求从起点 S 到终点
↪  T 的第 K 短路的长度，路径允许重复经过点或边。
输入格式：第一行包含两个整数 N 和 M。
接下来 M 行，每行包含三个整数 A,B 和 L，表示点 A 与点 B 之间存在
↪  有向边，且边长为 L。
最后一行包含三个整数 S,T 和 K 分别表示起点 S 终点 T 和第 K 短路
输出格式
输出占一行，包含一个整数，表示第 K 短路的长度，如果第 K 短路不存
↪  在，则输出“-1”。
*/
#define x first
#define y second
using namespace std;
typedef pair<int, int> pii;
typedef pair<int, pii> piii;
/*
```

```
    将队列换成优先队列 小根堆
    A-star 可以处理任意边权不论正负，但是不能有负权回路，不需要
↪   BFS 强制的边权为 1
    如果无解，那就不如朴素的 BFS，因为 A-star 每一次都是堆操作，
↪   log，但是朴素 BFS 是线性
    A* 算法只能保证终点出队是最小值，中间的点出队不一定是最优的
    但是 dijkstra 每个点出队 就已经是最优的了
    每个点不一定只被扩展一次 可能会被多次遍历 找最优解
*/
const int N = 1010, M = 200010; //要建反图，求出最短距离 就是估
↪   价函数
int n, m, S, T, K;
int h[N], rh[N], e[M], w[M], ne[M], idx;
int dist[N];
bool st[N];  //dijkstra 标记哪个点没有被用过
int cnt[N]; //每个点被搜过的次数
void add(int h[], int  a, int b, int c) {
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx ++;
}
void dijkstra() { //预处理估价函数 dist 数组
    //反向求一遍最短路 处理估价函数 注意 这里只处理该点到终点的
    ↪   最短路，
    //当前点到起点的距离 可能依然不是最优的
    priority_queue<pii, vector<pii>, greater<pii> > heap;
    //先把终点加进去
    heap.push({0, T});
    memset(dist, 0x3f, sizeof dist);
    dist[T] = 0;
    while (heap.size()) {
        auto t = heap.top();
        heap.pop();
        int ver = t.y;
        if (st[ver]) continue;
        st[ver] = true;
        for (int i = rh[ver]; ~i; i = ne[i]) {
            int j = e[i];
            if (dist[j] > dist[ver] + w[i]) {
```

```
                dist[j] = dist[ver] + w[i];
                heap.push({dist[j], j});
            }
        }
    }
}
int astar() {
    priority_queue<piii, vector<piii>, greater<piii> > heap;
    heap.push({dist[S], {0, S}}); //起点的估价值真实值和点编号
    // int cnt = 0; //终点被遍历的次数
    while (heap.size()) {
        auto t = heap.top();
        heap.pop();
        int ver = t.y.y, distance = t.y.x; //distance 真实距离
        ↪ 就是起点到该点的距离
        cnt[ver] ++;
        if (cnt[T] == K) return distance;
        for (int i = h[ver]; ~i; i = ne[i]) {
            int j = e[i]; //邻接的点 j 的估价值 加上
            if (cnt[j] < K) heap.push({distance + w[i] +
            ↪ dist[j], {distance + w[i], j}});
        }
    }
    return -1;
}
int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    memset(rh, -1, sizeof rh);
    for (int i = 0; i < m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(h, a, b, c); //正边
        add(rh, b, a, c); //反向边
    }
    scanf("%d%d%d", &S, &T, &K);
    if (S == T) K ++; //最短路至少有一条边 如果是环的情况, 那就
    ↪ 把 0 这种情况除掉
    dijkstra();
    printf("%d\n", astar());
```

```
    return 0;
}
```

3.2 DFS

3.2.1 DFS 连通性

```
/*
迷宫可以看成是由 n*n 的格点组成，每个格点只有 2 种状态，. 和 #，
↪ 前者表示可以通行后者表示不能通行。
第 1 行是测试数据的组数 k，后面跟着 k 组输入。
每组测试数据的第 1 行是一个正整数 n，表示迷宫的规模是 n*n 的。
接下来是一个 n*n 的矩阵，矩阵中的元素为 . 或者 #。
再接下来一行是 4 个整数 ha,la,hb,lb，描述 A 处在第 ha 行，第 la
↪ 列，B 处在第 hb 行，第 lb 列。
注意到 ha,la,hb,lb 全部是从 0 开始计数的
输入样例：
2
3
.##
..#
#..
0 0 2 2
5
.....
###.#
..#..
###..
...#.
0 0 4 0
输出样例:
YES
NO
*/
//dfs 只能保证连通性 不能求出以及保证最短路 优点是代码短 时间复
↪ 杂度都一样 线性搜一次
const int N = 110;
char g[N][N];
bool st[N][N];
int n;
int xa, ya, xb, yb;
```

```
int dx[4] = { -1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
bool dfs(int x, int  y) {
    if (g[x][y] == '#') return 0; //终点可能是障碍物 这一步要先
    ↪ 于下一行判断
    if (x == xb && y == yb) return 1;
    st[x][y] = 1;
    for (int i = 0; i < 4; i++) {
        int a = x + dx[i], b = y + dy[i];
        if (a < 0 || a >= n || b < 0 || b >= n) continue;
        if (st[a][b]) continue;
        // if((g[a][b]) == '#') continue;
        if (dfs(a, b)) return 1;
    }
    return false;
}
int main() {
    int t;
    cin >> t;
    while (t --) {
        cin >> n;
        for (int i = 0 ; i < n; i++) cin >> g[i];
        memset(st, 0, sizeof st);
        cin >> xa >> ya >> xb >> yb;
        if (dfs(xa, ya)) puts("YES"); //xa ya 能否走到目标点
        else puts("NO");
    }
    return 0;
}
```

3.2.2 迭代加深搜索

```
/*
满足如下条件的序列 X（序列中元素被标号为 1、2、3…m）被称为“加成
↪ 序列”：
1、X[1]=1
2、X[m]=n
3、X[1]<X[2]<…<X[m-1]<X[m]
4、对于每个 k（2 k m）都存在两个整数 i 和 j （1 i,j k-1，i 和 j 可
↪ 相等），使得 X[k]=X[i]+X[j]。
给定一个整数 n，找出符合上述条件的长度 m 最小的“加成序列”。
```

```
如果有多个满足要求的答案，只需要找出任意一个可行解。
输入包含多组测试用例。每组测试用例占据一行，包含一个整数 n。当输
↪ 入为单行的 0 时，表示输入结束。
输出格式
对于每个测试用例，输出一个满足需求的整数序列，数字之间用空格隔开。
↪ 每个输出占一行。
输入样例：
5
7
12
15
77
0
输出样例：
1 2 4 5
1 2 4 6 7
1 2 4 8 12
1 2 4 5 10 15
1 2 4 8 9 17 34 68 77
*/
const int N = 110;
int path[N];
int n;
//从大到小枚举下一个数 可以减少长度
//排除冗余：如果已经有 2 个数等于某个数 就不用尝试其它的数
bool dfs(int u, int max_depth) { //当前层数 最大层数
    if (u > max_depth)return false;
    //u-1 是因为必须搜索完上一层，这一层开始才能判断是否符合
    if (path[u - 1] == n) return true;
    //将 st 数组开在 dfs 里面，原因：因为 path 是全局，每次 dfs
    ↪ 更新的都是当前位置
    //的元素，双重循环只需要判重在此之前的元素的组合之后的和
    //所以每次都会初始化 st 数组 重新判定
    bool st[N] = {0};
    for (int i = u - 1; i >= 0; i--) //枚举 2 个数的和
        for (int j = i; j >= 0; j--) {
            int s = path[i] + path[j];
            // s <= path[u-1]  因为要保持序列的严格递增
            if (s > n || s <= path[u - 1] || st[s]) continue;
            st[s] = true;
```

```
            path[u] = s;
            if (dfs(u + 1, max_depth)) return true;
        }
    return false;
}
int main() {
    while (~scanf("%d", &n) && n) {
        path[0] = 1;
        int depth = 1;
        while (!dfs(1, depth)) depth++;
        for (int i = 0; i < depth; i++) cout << path[i] << '
        ↪  ';
        cout << endl;
    }
    return 0;
}
```

3.2.3 双向 DFS

```
/*
N 个物品，背包体积 W，每个物品体积 G[i]，求最多能装的体积
数据范围 1<= N <=46,1 <= W,G[i] <= 2^31-1
第一行两个整数，W 和 N。以后 N 行，每行一个正整数表示 G[i]
输出格式
仅一个整数，表示能装的最多的物品体积和
*/
/*
    可以看成 01 背包，但是背包复杂度 N*V 用 dp 做会超时
    N 比较小 所以考虑搜索 在所有方案里面找最大值
    预处理前 K 个物品，打表 排序 判重 改为双向 DFS
    注意预处理不一定要均分，可以不均分
    后 N - K 直接二分查前面的表 降为 log 复杂度查找 以及排序
    降序排序，优先搜索大数 贪心策略
    剪枝：不能超过上限 W
*/
const int N = 46;
int n, m, k;
int w[N];
//  由于什么都不选 就是全 0 也是合法 weights[0] 就是 0 cnt 从 1
↪  开始
int weights[1 << 25], cnt = 1; //weights 所有能凑出来的重量
```

```cpp
int ans; //答案
void dfs1(int u, int s) { //当前枚举到的数 当前的和
    if (u == k) {
        weights[cnt++] = s;
        return ;
    }
    dfs1(u + 1, s); //不选
    if (1ll * s + w[u] <= m) dfs1(u + 1, s + w[u]); //选
}
void dfs2(int u, int s) {
    if (u == n) {
        int l = 0, r = cnt - 1;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (weights[mid] <= m - s) l = mid;
            else r = mid - 1;
        }
        ans = max(ans, weights[l] + s);
        return ;
    }
    dfs2(u + 1, s);
    if (1ll * s + w[u] <= m) dfs2(u + 1, s + w[u]);
}
int main() {
    cin >> m >> n;
    for (int i = 0; i < n; i++) cin >> w[i];
    sort(w, w + n);
    reverse(w, w + n);
    k = n / 2 + 2; //前半段 25 后半段 21
    dfs1(0, 0);
    sort(weights, weights + cnt);
    // cnt = unique(weights, weights + cnt) - weights; //去重
    ↪ 之后前一部分能组合出多少不同的重量
    dfs2(k, 0);
    cout << ans << endl;
    return 0;
}
```

3.2.4 IDA*

```
/*
给定 n 本书，编号为 1-n。在初始状态下，书是任意排列的。在每一次操
↪ 作中，可以抽取其中连续的一段，再把这段插入到其他某个位置。把
↪ 书按照 1-n 的顺序依次排列。求最少操作。
输入格式
第一行包含整数 T，表示共有 T 组测试数据。每组数据包含两行，第一行
↪ 为整数 n，表示书的数量。
第二行为 n 个整数，表示 1-n 的一种任意排列。同行数之间用空格隔开。
输出格式
每组数据输出一个最少操作次数。
如果最少操作次数大于或等于 5 次，则输出”5 or more”。
每个结果占一行。
数据范围
1 <= n <= 15
输入样例：
3
6
1 3 4 6 2 5
5
5 4 3 2 1
10
6 8 5 3 4 7 2 9 1 10
输出样例：
2
3
5 or more
*/
//后继关系:n 和 n + 1 互为后继
//将任意一段改变位置，整个当前的序列后继关系 将被改变 3 个
//假设所有的错误的后继关系有 tot 个 估价函数就是 tot / 3 上取整
↪ 也就是 (tot + 2) / 3 下取整
//一个升序的序列 tot 就是 0 那么估价函数的值就是 0
const int N = 15;
int n;
int q[N];
int w[5][N];
int f() {
    int tot = 0;
```

```
    for (int i = 0; i + 1 < n; i++)
        if (q[i + 1] != q[i] + 1)
            tot ++;
    return (tot + 2) / 3;
}
bool dfs(int depth, int max_depth) {
    if (depth + f() >  max_depth) return false;
    if (f() == 0) return true;
    for (int len = 1; len <= n; len++) //区间长度
        for (int l = 0; l + len - 1 < n; l++) { //枚举区间起点
            int r = l + len - 1;
            for (int k = r + 1; k < n; k++) { //将当前区间放到
            ↪ k 位置的后面, 枚举 k 从 r + 1 开始
                //备份状态
                memcpy(w[depth], q, sizeof q);
                int y = l;
                for (int x = r + 1; x <= k; x ++, y ++) q[y] =
                ↪  w[depth][x];
                for (int x = l; x <= r; x ++, y ++) q[y] =
                ↪  w[depth][x];
                if (dfs(depth + 1, max_depth)) return true;
                //还原备份
                memcpy(q, w[depth], sizeof q);
            }
        }
    return false;
}
int main() {
    int t; cin >> t;
    while (t --) {
        cin >> n;
        for (int i = 0; i < n; i++) cin >> q[i];
        int depth = 0;
        while (depth < 5 && !dfs(0, depth)) depth ++;
        if (depth >= 5) cout << "5 or more" << endl;
        else cout << depth << endl;
    }
    return 0;
}
```

4 图论

4.1 图的 BFS/DFS

4.1.1 树的重心

```
/*
树的重心：DFS
给定一颗树，树中包含 n 个结点（编号 1~n）和 n-1 条无向边。找到树
↪ 的重心，并输出将重心删除后，剩余各个连通块中点数的最大值。
重心定义：重心是指树中的一个结点，如果将这个点删除后，剩余各个连
↪ 通块中点数的最大值最小，那么这个节点被称为树的重心。
第一行包含整数 n，表示树的结点数。接下来 n-1 行，每行包含两个整数
↪ a 和 b，表示点 a 和点 b 之间存在一条边。
输出一个整数 m，表示将重心删除后，剩余各个连通块中点数的最大值。
数据范围 1 <= n <= 1e5
*/
int ans = 0x3f3f3f3f;   //最终答案
const int N = 100010;
int e[2 * N], ne[2 * N], h[N], idx;//无向图边开 2 倍
int v[N]; //记忆化
int pos;  //重心位置
int sz[N];  //每个节点为根的子树的大小
int n;
//注意 一棵树的重心不唯一
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
void dfs(int x) {
    v[x] = 1; sz[x] = 1;
    int maxpart = 0;  //删掉 x 后分成的最大子树的大小
    for (int i = h[x]; i != -1; i = ne[i]) {
        int y = e[i];
        if (v[y]) continue;
        dfs(y);
        sz[x] += sz[y];
        maxpart = max(maxpart, sz[y]);  //要把 x 删掉了 所以看
        ↪ 一下和 x 相接的节点的子树权重
    }
```

```
    maxpart = max(maxpart, n - sz[x]);
    if (maxpart < ans) {
        ans = maxpart;
        pos = x; //重心位置
    }
}
int main() {
    int a, b; cin >> n;
    memset(h, -1 , sizeof h); //切记初始化要在加边之前
    for (int i = 0; i < n - 1; i++) {
        cin >> a >> b;
        add(a, b); add(b, a);
    }
    dfs(1); //从任意起点开始都可以  搜索顺序不重要
    cout << ans << endl;
    return 0;
}
```

4.1.2 BFS 最短路/点的层次

```
/*BFS 最短路
给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环。所有边的
↪ 长度都是 1，点的编号为 1~n。请你求出 1 号点到 n 号点的最短距
↪ 离，如果从 1 号点无法走到 n 号点，输出-1。
输入格式
第一行包含两个整数 n 和 m。
接下来 m 行，每行包含两个整数 a 和 b，表示存在一条从 a 走到 b 的
↪ 长度为 1 的边。
输出格式：输出一个整数，表示 1 号点到 n 号点的最短距离。
1 <= n,m <= 1e5
*/
const int N = 100010;
int e[N], ne[N], h[N], idx;
int n, m;
int d[N], q[N];
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
int bfs() {
```

```
    int hh = 0, tt = 0;
    q[0] = 1; //初始化队头
    memset(d, -1, sizeof d);  //-1 表示没有被遍历过
    d[1] = 0;
    while (hh <= tt) {
        int t = q[hh++];
        for (int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            if (d[j] == -1) {
                d[j] = d[t] + 1;
                q[++tt] = j;
            }
        }
    }
    return d[n];
}
int main() {
    memset(h, -1, sizeof h);
    cin >> n >> m;
    int a, b;
    while (m --) {
        cin >> a >> b;
        add(a, b);
    }
    cout << bfs() << endl;
    return 0;
}
```

4.1.3 BFS 最短路计数

```
/*
给出一个 N 个顶点 M 条边的无向无权图，顶点编号为 1 到 N。
问从顶点 1 开始，到其他每个点的最短路有几条。
输入格式
第一行包含 2 个正整数 N,M，为图的顶点数与边数。
接下来 M 行，每行两个正整数 x,y，表示有一条顶点 x 连向顶点 y 的
↪ 边，请注意可能有自环与重边。
输出格式
输出 N 行，每行一个非负整数，第 i 行输出从顶点 1 到顶点 i 有多少
↪ 条不同的最短路，由于答案有可能会很大，你只需要输出对 100003
↪ 取模后的结果即可。
```

```
如果无法到达顶点 i 则输出 0。
*/
//先求出到每个点的最短路的值 然后分别求出有多少路径
//最短路树 也就是拓扑图 记录每个点的 前驱
//最短路的求解过程 需要满足拓扑序 只有 dijkstra/bfs 满足入队出队
↪  一次就是最优 拓扑序的遍历
//spfa 实际上也可以，如果有负权的话

const int N = 100010, M = 400010;
const int mod = 100003;
int n, m;
int h[N], e[M], ne[M], idx;
int dist[N], cnt[N];
int q[N];
void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++;
}
void bfs() {
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    cnt[1] = 1; //原点到自己的初始最短路就是 1
    int hh = 0, tt = 0;
    q[0] = 1;
    while (hh <= tt) {
        int t = q[hh++];
        for (int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            if (dist[j] > dist[t] + 1) { //更新最短路 同时更新
            ↪  cnt
                dist[j] = dist[t] + 1;
                cnt[j] =  cnt[t]; //j 第一次被 t 更新
                q[++tt] =  j;
            }
            //j 已经被其他点更新过了，已经最优了，直接加上
            else if (dist[j] == dist[t] + 1) {
                cnt[j] = (cnt[j] + cnt[t]) % mod;
            }
        }
    }
}
```

```
int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m --) {
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b); add(b, a);
    }
    bfs();
    for (int i = 1; i <= n; i++) printf("%d\n", cnt[i]);
    return 0;
}
```

4.2 最短路

4.2.1 SPFA

```
/*
spfa: 平均情况下 O(m), 最坏情况下 O(nm), n 点数, m 边数
给定一个 n 个点 m 条边的有向图, 图中可能存在重边和自环,
边权可能为负数。求出 1 号点到 n 号点的最短距离
数据保证不存在负权回路。
*/
const int N = 100010, M = 100010;
int h[N], ne[M], idx, e[N], w[N], d[N];
bool v[N];
int n, m;
queue<int> q;
void spfa() {
    d[1] = 0; v[1] = 1;  //在队列中就是 1 不在就是 0
    q.push(1);
    while (q.size()) {
        int x = q.front(); q.pop();
        v[x] = 0;
        for (int i = h[x]; i != -1; i = ne[i]) {
            int y = e[i]; int u = w[i];
            if (d[x] + u < d[y]) {
                d[y] = d[x] + u;
                if (!v[y]) q.push(y), v[y] = 1;
            }
        }
```

```
    }
}
int main() {
    memset(h, -1, sizeof h);
    memset(v, 0, sizeof v);
    memset(d, 0x3f, sizeof d);
    cin >> n >> m;
    int a, b;
    int x;
    while (m --) {
        cin >> a >> b >> x;
        add(a, b, x);
    }
    spfa();
    if (d[n] == 0x3f3f3f3f) cout << "impossible"  << endl;
    else cout << d[n] << endl;
    return 0;
}

//spfa 判负环
const int N = 2010, M = 10010;
int h[N], ne[M], idx, d[N], w[N], e[M];
bool v[N];
queue<int> q;
int cnt[N]; //cnt 记录某个点到 n 号点的路径个数 不是路径长度
int n, m;
bool spfa() {
    //由于求整个图 而不是只求 1 号开始到其他点的路径否有负环, 所
    ↪ 以初始把所有点都加入队列
    for (int i = 1; i <= n; i++) q.push(i), v[i] = 1;
    while (q.size()) {
        int x = q.front(); q.pop();
        v[x] = 0;
        for (int i = h[x]; i != -1; i = ne[i]) {
            int y = e[i];
            if (d[y] > d[x] + w[i]) {
                d[y] = d[x] + w[i];
                cnt[y] = cnt[x] + 1; //更新路径之后 三角形不等
                ↪ 式, 然后边数加一
```

```
                if (cnt[y] >= n) return true; // n 个点 无环路
                ↪ 径最多 n - 1 条边 n 条边更新了说明存在环, 而
                ↪ 且是负环 因为正环不可能更新
                if (!v[y]) q.push(y), v[y] = 1;
            }
        }
    }
    return false;
}
int main() {
    memset(h, -1, sizeof h);
    memset(v, 0, sizeof v);
    memset(d, 0x3f, sizeof d);
    int a, b , x;
    cin >> n >> m;
    while (m --) {
        cin >> a >> b >> x;
        add(a, b, x);
    }
    spfa();
    if (spfa()) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

4.2.2 Bellman-Ford

```
/*
Bellman-Ford 求边数限制最短路
边可以为负 也可以有负权回路 (边数限制问题)
时间复杂度 O(nm), n 表示点数, m 表示边数
*/
/*
从 1 号点到 n 号点的最多经过 k 条边的最短距离,
如果无法从 1 号点走到 n 号点, 输出 impossible
*/
const int N = 510, M = 10010;
int dist[N];
int n, m, k;
int backup[N];
```

```cpp
//有边数限制 只能用 bellman ford 算法，同时由于边数限制 负环也就无意义了
//bellman ford 算法迭代 k 次 求出的 d 数组 代表 k 条边能到达的最短路
struct Edge { //bellman 可直接用 struct 来存边
    int a, b , w;
} edges[M];
int bellmanford() {
    dist[1] = 0;
    for (int i = 0; i < k; i++) { //迭代 k 次
        //记录上一次迭代的结果 防止迭代时发生边的串连
        memcpy(backup, dist, sizeof dist);
        for (int j = 0; j < m; j++) {
            int a = edges[j].a, b = edges[j].b, u = edges[j].w;
            dist[b] = min(dist[b], backup[a] + u);  //利用 backup
        }
    }
    if (dist[n] > 0x3f3f3f3f / 2) return -1; //不能写成 dist[n] == 0x3f3f3f3f 即使不存在但是会被更新，不等于无穷了
    else return dist[n];
}
int main() {
    memset(dist, 0x3f, sizeof dist);
    cin >> n >> m >> k; //k 是边数限制
    int x;
    for (int i = 0; i < m; i++) {
        int a, b, w;
        cin >> a >> b >> w;
        edges[i] = {a, b, w};
    }
    int t = bellmanford();
    if (t == -1) puts("impossible");
    else cout << t << endl;
    return 0;
}
```

4.2.3 Dijkstra

```
/*
朴素 dijkstra 算法
O(n^2+m), n 表示点数 m 表示边数
*/
const int N = 510, M = 100010;
int h[N], e[M], ne[M], w[M], d[N], idx;
bool v[N];  //是否被访问过 该节点
int n, m;
void dijkstra() {
    d[1] = 0;
    for (int i = 1; i < n; i++) {
        int x = 0;
        for (int j = 1; j <= n ; j++)
            if (!v[j] && (x == 0 || d[x] > d[j])) x = j;
        v[x] = 1;
        if (x == n) break; //对于只求第 n 个点的 此处可以优化,
        ↪ 但是求所有点的单源最短路 就不需要了。
        for (int i = h[x]; ~i; i = ne[i]) {
            int y = e[i], u = w[i];
            if (d[y] > d[x] + u) d[y] = d[x] + u;
        }
    }
}
int main() {
    memset(h, -1 , sizeof h);
    memset(d, 0x3f , sizeof d);
    memset(v, 0, sizeof v);
    cin >> n >> m;
    int a, b, x;
    while (m--) {
        cin >> a >> b >> x;
        add(a, b, x);
    }
    dijkstra();
    if (d[n] == 0x3f3f3f3f) cout << -1 << endl;//路径不存在
    else cout << d[n] << endl;
    return 0;
}
/*
```

```
堆优化版 dijkstra 时间复杂度 O(mlogn), n 表示点数, m 表示边数
*/
typedef pair<int, int> pii;
const int N = 150010, M = 150010;
int h[N], e[M], ne[M], w[M], d[N], idx;
bool v[N];  //是否被访问过 该节点
int n, m;
//第一维度是 dist  第二维是节点编号 pair 默认按照第一维排序
priority_queue<pii , vector<pii>, greater<pii> > q;
void dijkstra() {
    d[1] = 0;
    q.push({0, 1});  //初始化第一个节点 距离为 0
    while (q.size()) {
        int x = q.top().second; q.pop();
        if (v[x]) continue;
        v[x] = 1;
        for (int i = h[x]; i != -1; i = ne[i]) {
            int y = e[i], u = w[i];
            if (d[y] > d[x] + u) {
                d[y] = d[x] + u;
                q.push({d[y], y});
            }
        }
    }
}
/*
求单源次短路及其条数
*/
//先求最短路及路径数量 再求次短路 最后判断如果长度多 1 就加上
//d[i][0] 表示 1 到 i 的最短路径 cnt 同理
//d[i][1] 表示 1 到 i 的次短路径
//每个点的集合找到最短路和次短路 并且统计次数
const int N = 1010, M = 10010;
struct Ver {
    int ver, type, dist;//type 记录是最短还是次短路
    bool operator> (const Ver &W) const {
        return dist > W.dist; // 大根堆 重载大于号
    }
};
int n, m, S, T;
```

```
int h[N], e[M], w[M], ne[M], idx;
int dist[N][2];
int cnt[N][2];
bool st[N][2];
void add(int a, int b, int c) {
    e[idx] = b;
    ne[idx] = h[a];
    w[idx] = c;
    h[a] = idx ++;
}
int dijkstra() {
    memset(dist, 0x3f, sizeof dist);
    memset(st, 0, sizeof st);
    memset(cnt, 0, sizeof cnt);
    dist[S][0] = 0; cnt[S][0] = 1;
    priority_queue<Ver, vector<Ver>, greater<Ver> > heap;
    heap.push({S, 0, 0});
    while (heap.size()) {
        Ver t = heap.top();
        heap.pop();
        int ver = t.ver, type = t.type, distance = t.dist,
        ↪  count = cnt[ver][type];
        if (st[ver][type]) continue;
        st[ver][type] = true;
        for (int i = h[ver]; i != -1; i = ne[i]) {
            int j  = e[i];
            if (dist[j][0] > distance + w[i]) {
                dist[j][1] = dist[j][0], cnt[j][1] = cnt[j][0];
                ↪  //最小值沦为次小值
                heap.push({j, 1, dist[j][1]}); //次小值加入堆
                dist[j][0] = distance + w[i], cnt[j][0] =
                ↪  count;
                heap.push({j, 0, dist[j][0]}); //最小值加入堆
            }
            else if (dist[j][0] == distance + w[i]) cnt[j][0]
            ↪  += count;
            else if (dist[j][1] > distance + w[i]) {
                //比最小值小 比次小值大 更新次小值
                dist[j][1] = distance + w[i], cnt[j][1] =
                ↪  count;
```

```
                heap.push({j, 1, dist[j][1]});
            }
            else if (dist[j][1] == distance + w[i]) cnt[j][1]
             ↪  += count; //等于次小值
        }
    }
    int res = cnt[T][0];
    if (dist[T][0] + 1 == dist[T][1]) res += cnt[T][1];
    return res;
}
int main() {
    int t; cin >> t;
    while (t --) {
        scanf("%d%d", &n , &m);
        memset(h, -1 , sizeof h);
        idx = 0;
        while (m --) {
            int a, b, c;
            scanf("%d%d%d", &a, &b, &c);
            add(a, b, c);
        }
        scanf("%d%d", &S, &T);
        printf("%d\n", dijkstra());
    }
    return 0;
}
```

4.2.4 Floyd

```
//时间复杂度是 O(n^3), n 表示点数
//图中要求不存在负权回路
const int  N = 210, INF = 1e9;
int n, m , k;
int d[N][N];
void floyd() {
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
int main() {
```

```
    cin >> n >> m >> k;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;
    int a , b , x;
    while (m --) {
        cin >> a >> b >> x;
        d[a][b] = min(d[a][b], x); //判断重边
    }
    floyd();
    while (k--) {
        cin >> a >> b;
        if (d[a][b] > INF / 2)  cout << "impossible" << endl;
        ↪ //即使没有路径 也可能会把无穷给更新 不能判断 ==INF
        else cout << d[a][b] << endl;
    }
    return 0;
}
/*
传递闭包
给定 n 个变量和 m 个不等式。其中 n 小于等于 26，变量分别用前 n
↪ 的大写英文字母表示。
不等式之间具有传递性，即若 A>B 且 B>C ，则 A>C。
请从前往后遍历每对关系，每次遍历时判断：
如果能够确定全部关系且无矛盾，则结束循环，输出确定的次序；
如果发生矛盾，则结束循环，输出有矛盾；
如果循环结束时没有发生上述两种情况，则输出无定解。
输入格式
输入包含多组测试数据。
每组测试数据，第一行包含两个整数 n 和 m。
接下来 m 行，每行包含一个不等式，不等式全部为小于关系。
当输入一行 0 0 时，表示输入终止。
数据范围
2 <= n <= 26，变量只可能为大写字母 A~Z。
*/
//提前 break 题意：如果前 i 个已经推出所有关系 即使第 i + 1 个
↪ 矛盾 也当作成立
//矛盾判断：d[i][i] == 1 因为 A > B, B > A ->  A > A
//唯一确定：d[i][j], d[j][i] 必有一个是 1 一个是 0
```

```
//不能确定：d[i][j] == 0 && d[j][i] == 0
//排序：一个没有被其他数标记过的数 就是最小数
const int N = 26;
int n, m ;
bool g[N][N], d[N][N];
bool st[N];
void floyd() {
    memcpy(d, g, sizeof d);

    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                d[i][j] |= d[i][k] && d[k][j];
}
int check() {
    for (int i = 0; i < n; i++)
        if (d[i][i]) return 2;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < i; j++) //这里是枚举点对 i,j 所以 j
        ↪  不是 1 到 n
            //如果运行到这里 说明所有 d[i][i] 都是 0 如果 i==j
            //只要遇到一个 d[i][i] 就直接返回 0 了 不会再判断其
            ↪  他点对了所以这个判断的前提是 i!=j
            //或者 j 如果循环到 n 那么可以写为 if(!d[i][j] &&
            ↪  !d[j][i] && i != j)
            if (!d[i][j] && !d[j][i]) //都是 0  也就是不能确定
                return 0;
    return 1; //唯一确定
}
char get_min() {
    for (int i = 0; i < n; i++)
        if (!st[i]) {
            bool flag = true;
            for (int j = 0; j < n; j++) //找有没有比 i 小的 并
            ↪  且没有标记的 这里找到的一定是最小的 因为 i 是从
            ↪  0 开始枚举的
                if (!st[j] && d[j][i]) {
                    flag = false;
                    break;
                }
```

```
            if (flag) {
                st[i] = true;
                return 'A' + i;
            }
        }
}
int main() {
    while (cin >> n >> m , n || m) {
        memset(g, 0, sizeof g);
        int type = 0, t;  //type 0 不能确定 1 唯一 2 矛盾 t 轮
        ↪ 数
        for (int i = 1; i <= m; i++) { //迭代 m 次 就是 m 个不
        ↪ 等式关系
            char str[5];
            cin >> str;
            int a = str[0] - 'A', b = str[2] - 'A';
            if (!type) { //当所有状态都确定就不会进入这个判断
                //floyd 每加一个大小关系就判断一下
                //目的就是最早 break 由题意
                g[a][b] = 1;
                floyd();
                type = check();
                if (type) t = i;
            }
        }
        if (!type) puts("Sorted sequence cannot be
        ↪ determined.");
        else if (type == 2) printf("Inconsistency found after
        ↪ %d relations.\n", t);
        else { //唯一确定
            memset(st, 0, sizeof st);
            printf("Sorted sequence determined after %d
            ↪ relations: ", t);
            for (int i = 0; i < n; i++) printf("%c",
            ↪ get_min());
            printf(".\n");
        }
    }
    return 0;
}
```

4.3 最小生成树

4.3.1 Kruskal

```
/*
Kruskal 时间复杂度 O(mlogm), n 点数, m 边数
*/
const int N = 100010, M = 2 * N;
//kruskal 直接结构体就可以了
int n, m;
int p[N];  //并查集父亲节点
struct Edge {
    int a, b, w;
    bool operator< (const Edge &u) const {
        return w < u.w;
    }
} edges[M];
int find(int x) {
    if (x != p[x]) p[x] = find(p[x]);
    return p[x];
}
int main() {
    cin >> n >> m;
    int a, b, x;
    for (int i = 0; i < m; i++) {
        cin >> a >> b >> x;
        edges[i] = {a, b , x};
    }
    sort(edges, edges + m);
    for (int i = 1; i <= n; i++) p[i] = i; //初始化并查集
    int res = 0, cnt = 0;  //res 答案 cnt 记录当前加入的边数 最
    ↪ 小生成树一共 n - 1 条边
    for (int i = 0; i < m; i++) { //枚举所有的边
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;
        a = find(a), b = find(b);
        if (a != b) { //不连通 合并更新
            res += w;
            p[a] = b;
            cnt ++;
        }
    }
```

```
    if (cnt < n - 1) cout << "impossible" << endl;
    else cout << res << endl;
    return 0;
}
/*
例题：给定一棵 N 个节点的树，要求增加若干条边，把这棵树扩充为完全图，并满足图的唯一最小生成树仍然是这棵树。
求增加的边的权值总和最小是多少。
注意：树中的所有边权均为整数，且新加的所有边权也必须为整数。
*/
//边权从小到大排序  然后 kruskal 合并集合
//每次合并 2 个连通块时候 需要连接 2 个连通块内部所有的边 原来的树肯定至少有一条边已经连通这 2 个块
//假设原来树连通这 2 块的最小边权是 w 那么新加的边权就是 w + 1
//由于计算连通块内的点数 需要维护并查集的 SIZE
const int N = 6010;  //有 N-1 条边
int n;
struct Edge {
    int a, b, w;
    bool operator< (const Edge & t) {
        return w < t.w;
    }
} e[N];
int p[N], sz[N];
int find(int x) {
    if (x != p[x]) p[x] = find(p[x]);
    return p[x];
}
int main() {
    int t;
    cin >> t;
    while (t --) {
        cin >> n;
        memset(p, 0, sizeof p);
        memset(sz, 0, sizeof sz);
        memset(e, 0, sizeof e);
        for (int i = 0; i < n - 1; i++) { //注意是输入 n - 1 条边
            int a, b, w;
            cin >> a >> b >> w;
```

```
            e[i] = {a, b, w};
        }
        sort(e, e + n);
        for (int i = 0; i < n; i++) p[i] = i, sz[i] = 1;
        //因为是按照原来的存在的树的边排序的, 所以每次遍历到的 w
        ↪ 就是原来树的边权值
        int res = 0;
        for (int i = 0; i < n; i++) {
            int a = find(e[i].a), b = find(e[i].b), w = e[i].w
            ↪ ;
            if (a != b) {
                res += (sz[a] * sz[b] - 1) * (w + 1); //减去一
                ↪ 个已经存在的原来的树的边
                sz[b] += sz[a];
                p[a] = b;
            }
        }
        cout <<  res << endl;
    }
    return 0;
}
```

4.3.2 Prim

```
/*
朴素版 prim 时间复杂度 O(n^2+m), n 点数, m 边数
*/
const int N = 510;
const int INF = 0x3f3f3f3f;
int a[N][N], d[N], n, m, ans;
bool v[N];
void prim() {
    d[1] = 0;
    for (int i = 0; i < n; i++) { //这里是点数 每次处理一个点
        int x = 0;
        for (int j = 1; j <= n; j++)
            if (!v[j] && (x == 0 || d[j] < d[x]))  x = j; //找
            ↪ 到集合之外边权最小的点
        v[x] = 1;
        for (int y = 1; y <= n; y++) //更新 点 x 所有出边,
```

```
            //也就是更新另外一个集合到该集合的所有边权，直接扫一
            ↪ 遍没有访问过的边
            if (!v[y]) d[y] = min(d[y], a[x][y]);
    }
}
int main() {
    memset(d, 0x3f, sizeof d);
    memset(v, 0, sizeof v);
    memset(a, 0x3f, sizeof a);
    cin >> n >> m;
    int x, y, z;
    for (int i = 1; i <= n; i++) a[i][i] = 0;
    while (m--) {
        cin >> x >> y >> z;
        a[y][x] = a[x][y] = min(a[y][x], z);
    }
    prim();
    for (int i = 1; i <= n; i++) ans += d[i]; //n-1 条边和
    if (ans > INF / 2) cout << "impossible" << endl; //图不联
    ↪ 通，但是有负边加上减小了无穷的值
    else cout << ans << endl;
    return 0;
}
```

4.4 二分图

4.4.1 染色法

```
/*
染色法判别二分图
时间复杂度是 O(n+m)，n 表示点数，m 表示边数
给定一个 n 个点 m 条边的无向图，图中可能存在重边和自环。
请你判断这个图是否是二分图。第一行包含两个整数 n 和 m。
接下来 m 行，每行包含两个整数 u 和 v，表示点 u 和点 v 之间存在一
↪ 条边
*/
/*
一个图是二分图 当且仅当图中没有奇数环 如果没有奇数环 那么反之也
↪ 一定是二分图
证明直接用奇数环 假设起点属于一个集合 最后导致起点矛盾 QED.
染色过程 只要有一个点颜色确定 整个图的染色就已经确定了
```

```
所以总而言之 用 2 种颜色能染色成功 那就是二分图
*/
const int N = 100010, M = 2 * N;
int n, m;
int h[N], e[M], ne[M], idx;
int color[N];
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
//当前点是 u 颜色是 c 颜色用 1 和 2 表示不同的两种颜色
bool dfs(int u, int c) {
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (!color[j]) { //还没有染色
            dfs(j, 3 - c);  //3 - c 将 1 变为 2,  2 变为 1
        } else if (color[j] == c) return false; //已经染色 但是
        ↪  一条边 2 个顶点颜色一样 染色失败
    }
    return true;
}
int main() {
    memset(h, -1, sizeof h);
    cin >> n >> m;
    int a, b;
    while (m--) {
        cin >> a >> b;
        add(a, b);
        add(b, a);
    }
    bool flag = true;
    for (int i = 1; i <= n; i++) //染色每一个点
        if (!color[i]) {
            if (!dfs(i, 1)) { //染色矛盾
                flag = false;
                break;
            }
        }
```

```
    if (flag) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

4.4.2 匈牙利算法

```
/*
匈牙利算法复杂度 O(nm), n 点数, m 边数
求最大匹配数:
二分图, 其中左半部包含 n1 个点 (编号 1~n1), 右半部包含 n2 个点 (编号 1~n2), 二分图共包含 m 条边。
数据保证任意一条边的两个端点都不可能在同一部分中。
求出二分图的最大匹配数。
*/
const int N = 510, M = 100010;
int n1, n2, m;
int h[N], ne[M], e[M], idx;
int match[N];  //右边的点 对应左边的匹配点是哪个
bool v[N];
//枚举左边的点 边的存储方向为左指向右边
//由于是二分图 所以左边部分点之间不可能有边, 直接左右节点编号一样 混用无所谓 直接存就好
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
bool dfs(int x) { //只对左边的一个点进行的 dfs 操作
    for (int i = h[x]; ~i; i = ne[i]) {
        int j = e[i];
        if (!v[j]) { // 该右边的点还没有被左边该点匹配过
            v[j] = true;
            if (!match[j] || dfs(match[j])) { //j 已经被占用
                //v[j] = true, 则回溯过去的左边的点重选的时候不能选了
                //所以回溯过去原来的左边的点的只能在邻接表里面遍历下一个点
                match[j] = x;
                return true;
```

```
            }
        }
    }
    return false;
}
int main() {
    memset(h , -1 , sizeof h);
    cin >> n1 >> n2 >> m;
    int a, b;
    while (m --) {
        cin >> a >> b;
        add(a, b);
    }
    int res = 0;  //最终答案 最大匹配边数
    for (int i = 1; i <= n1; i++) { //枚举左边所有点
        memset(v, 0 , sizeof v); //每次枚举都清空右边的点来匹配
        if (dfs(i)) res++;
    }
    cout << res << endl;
    return 0;
}
/*
例题：关押罪犯
第一行为两个正整数 N 和 M 分别表示罪犯的数目以及存在仇恨的罪犯对
↪ 数。
接下来的 M 行每行为三个正整数 aj, bj, cj, 表示 aj 号和 bj 号罪犯
↪ 之间存在仇恨，其怨气值为 cj。
数据保证 1 aj<bj<N,0<cj 1000000000 且每对罪犯组合只出现一次。
*/
//二分图是 无向图的问题 但是有些有向图问题 本质还是 无向图
//二分边权 从冲突最大的开始 最终答案的取值范围为 [0~1e9] 因为如
↪ 果是二分图 那么就是 0 如果是一个奇环 三个边都是 1e9 那么怎么
↪ 分都是 1e9
//所以每次二分 mid 如果当前 mid 可以满足边权大于 mid 的点对都可
↪ 以分开，也就是染色判定成功，那么就缩小 mid
const int N = 20010, M = 200010; //无向边
int h[N], e[M], ne[M], w[M], idx;
int n, m;
int color[N]; //0 没有染色 1 白色 2 黑色
void add(int a, int b, int c) {
```

```
    e[idx] = b;
    ne[idx] = h[a];
    w[idx] = c;
    h[a] = idx ++;
}
bool dfs(int u, int c, int mid) {
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (w[i] <= mid) continue;
        if (color[j]) {
            if (color[j] == c) return false;
        }
        else if (!dfs(j, 3 - c, mid)) return false;
    }
    return true;
}
bool check(int mid) {
    memset(color, 0, sizeof color);
    for (int i = 1; i <= n; i++)
        if (!color[i])
            if (!dfs(i, 1, mid)) return false;
    return true;
}
int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m --) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c); add(b, a, c);
    }
    //二分
    int l = 0, r = 1e9;
    while (l < r) {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;
        else l = mid + 1;
    }
    printf("%d\n", r);
    return 0;
```

```
}
/*
例题：骑士放置 最大独立集 = 点数-匹配数-禁止放置的点
给定一个 N*M 的棋盘，有一些格子禁止放棋子。
问棋盘上最多能放多少个不能互相攻击的骑士（国际象棋的“骑士”，类似于中国象棋的“马”，按照“日”字攻击，但没有中国象棋“别马腿”的规则）。
输入格式
第一行包含三个整数 N,M,T，其中 T 表示禁止放置的格子的数量。
接下来 T 行每行包含两个整数 x 和 y，表示位于第 x 行第 y 列的格子禁止放置，行列数从 1 开始。
输出格式
输出一个整数表示结果。
数据范围 1 <= N,M <= 100
*/
//如果 2 个点能互相 攻击到 那么就连一条边 那么最大独立集就是答案
//对于一般图 求最大独立集是 NPC 问题，但是这个图可以二染色，就是二分图，可以匈牙利多项式时间解出
//这里所说的二染色，是指每个点“日”字连接的点颜色是不一样的 而且横纵坐标之和奇偶性不同
#define x first
#define y second
const int N = 110;
int n, m, k;
pii match[N][N];
bool g[N][N], st[N][N];
int dx[8] = { -2, -1, 1, 2, 2, 1, -1, -2};
int dy[8] = {1, 2, 2, 1, -1, -2, -2, -1};
bool dfs(int x, int y) {
    for (int i = 0; i < 8; i++) {
        int a = x + dx[i], b = y + dy[i];
        if (a < 1 || a > n || b < 1 || b > m) continue;
        if (g[a][b] || st[a][b]) continue;
        st[a][b] = true;
        pii t = match[a][b];
        if (t.x == 0 || dfs(t.x, t.y)) {
            match[a][b] = {x, y};
            return true;
        }
    }
```

```
    return false;
}
int main() {
    cin >> n >> m >> k;
    for (int i = 0; i < k; i++) {
        int x, y;
        cin >> x >> y;
        g[x][y] = true; //不能放的格子
    }
    int res  = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) {
            if ((i + j) & 1 || g[i][j]) continue; //只需要单向
            ↪ 匹配即可 就是枚举男的匹配女的
            memset(st, 0, sizeof st);
            if (dfs(i, j)) res ++;
        }
    cout << n * m - res - k  << endl;
    return 0;
}
```

4.5 差分约束

```
/*
给定 n 个区间 [ai,bi] 和 n 个整数 ci。
你需要构造一个整数集合 Z，使得 i [1,n],Z 中满足 ai x bi 的整数 x
↪ 不少于 ci 个。
求这样的整数集合 Z 最少包含多少个数。
第一行包含整数 n。
接下来 n 行，每行包含三个整数 ai,bi,ci。
*/

//题意有点绕：从 0~50000 中选出尽量少的数，使得每个区间 [ai,bi]
↪ 都至少 ci 个数被选，求选出数的个数
//求最小值那么就是最长路问题了 题目保证有解 就不用判断正环了
//设 s[i] 为 1 到 i 中被出的数的个数 也就是前 i 个数 dist[i] 同
↪ 理 最后答案是 dist[50001]
//需要满足: s[i] >= s[i - 1], s[i] - s[i - 1] <= 1, s[b] -
↪ s[a-1] >= c
const int N = 50010, M = 150010;
```

```
int n;
int h[N], e[M], w[M], ne[M], idx;
int dist[N];
int q[N];
bool st[N];
void add(int a, int b, int c) {
    e[idx] = b;
    ne[idx] = h[a];
    w[idx] = c;
    h[a] = idx ++;
}
void spfa() {
    memset(dist, 0xcf, sizeof dist);
    int hh = 0, tt = 1;
    q[0] = 0;
    dist[0] = 0;
    st[0] = true;
    while (hh != tt) {
        int t = q[hh++];
        if (hh == N) hh = 0;
        st[t] = false;
        for (int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            if (dist[j] < dist[t] + w[i]) {
                dist[j] =  dist[t] + w[i];
                if (!st[j]) {
                    q[tt ++] = j;
                    if (tt == N) tt = 0;
                    st[j] = true;
                }
            }
        }
    }
}
int main() {
    scanf("%d", &n);
    memset(h, -1, sizeof h);
    for (int i = 1; i <= 50001; i++) {
        add(i - 1, i , 0);
        add(i, i - 1, -1);
    }
```

```
    while (n --) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        a ++; b++;
        add(a - 1, b, c);
    }
    spfa();
    printf("%d\n", dist[50001]);
    return 0;
}
/*
例题：排队布局
第一行包含三个整数 N,ML,MD。
接下来 ML 行，每行包含三个正整数 A,B,L，表示奶牛 A 和奶牛 B 至多
↪ 相隔 L 的距离。
再接下来 MD 行，每行包含三个正整数 A,B,D，表示奶牛 A 和奶牛 B 至
↪ 少相隔 D 的距离。
输出一个整数，如果不存在满足要求的方案，输出-1；如果 1 号奶牛和 N
↪ 号奶牛间的距离可以任意大，输出-2；否则，输出在满足所有要求的
↪ 情况下，1 号奶牛和 N 号奶牛间可能的最大距离。
*/
//第一个就是判负环 全部加入队列跑一次 SPFA 即可
//所有奶牛的距离都是相对距离，所以需要虚拟原点，假设所有牛都在数
↪ 轴的正半轴 所有点都在 x0 的左边 也就是说 x0 是最右边的
//对于第二个要求 直接把 1 号点固定在 0 也就是 x1 = 0 因为都是相
↪ 对距离 不影响距离 最后看一下 xn 是不是可以无限大 这也等价于
↪ 1 号点到其他点的最短距离
//求一下 xn 的最短路 如果无穷大 就是无穷大 否则就是最大值
//约束条件：xi <= x[i+1] + 0, xb <= xa + l, xa <= xb - d
const int INF = 0x3f3f3f3f;
const int N = 1010, M = 21010;  //边数判断就是满足条件的类 每类
↪ 需要多少点 全部加起来
//因为加边 的时候 会满足所有条件 也就是说每个条件 都要单独加边
int q[N], h[N], e[M], ne[M], idx, w[M], cnt[N], dist[N];
bool st[N];
int n, m1, m2;
void add(int a, int b, int c) {
    e[idx] = b;
    ne[idx] = h[a];
    w[idx] = c;
```

```
    h[a] = idx ++;
}
bool spfa(int sz) {
    memset(dist, 0x3f, sizeof dist);
    memset(st, 0, sizeof st);
    memset(cnt, 0, sizeof cnt);
    int hh = 0, tt = 1;
    for (int i = 1; i <= sz; i++) { //加入队列 求负环
        dist[i] = 0;
        q[tt++] = i;
        st[i] = true;
    }
    while (hh != tt) {
        int t = q[hh++];
        if (hh == N) hh = 0;
        st[t] = false;
        for (int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            if (dist[j] > dist[t] + w[i]) {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if (cnt[j] >= n)  return false;
                if (!st[j]) {
                    q[tt++] = j;
                    if (tt == N) tt = 0;
                    st[j] = true;
                }
            }
        }
    }
    return true;
}
int main() {
    memset(h, -1, sizeof h);
    scanf("%d%d%d", &n, &m1, &m2);
    for (int i = 1; i < n; i++) add(i + 1, i, 0);
    while (m1 --) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        if (b < a ) swap(a, b);
```

```
        add(a, b, c);
    }
    while (m2 --) {
        int a, b , c;
        scanf("%d%d%d", &a, &b, &c);
        if (b < a) swap(a, b);
        add(b, a, -c);
    }
    if (!spfa(n)) puts("-1"); //前 n 个点的图有负环
    else {
        spfa(1); //将第一个点放进去求
        if (dist[n] == INF) puts("-2");
        else printf("%d\n", dist[n]);
    }
    return 0;
}
```

4.6 最近公共祖先

```
/*
倍增求 LCA
给定一棵包含 n 个节点无向树，节点编号互不相同，但不一定是 1~n。
有 m 个询问，每个询问给出了一对节点的编号 x 和 y，询问 x 与 y 的
↪ 祖孙关系。
*/
//规定 d[0] = 0 第 0 层的深度是 0 ，以及跳出根节点后 f[x][k] =
↪ 0 这样跳出后就不会出界 而是继续判断
//若 a 是 b 的祖先 那么 lca(a, b) 就是 a
const int N = 40010, M = 2 * N;
int n, m;
int h[N], e[M], ne[M], idx;
int depth[N], fa[N][16]; // k 最大是 log2(N)
int q[N]; //BFS 队列
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
void bfs(int root) {
    memset(depth, 0x3f, sizeof depth);
```

```
    depth[0] = 0, depth[root] = 1; //根节点所在的层数定义为 1
    int hh = 0, tt = 0;
    q[0] = root;
    while (hh <= tt) {
        int t = q[hh++];
        for (int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            if (depth[j] > depth[t] + 1) {
                depth[j] = depth[t] + 1;
                q[++tt] = j;
                fa[j][0] = t; //跳一步
                for (int k = 1; k <= 15; k++) //处理 j 的所有祖
                ↪ 先节点
                    fa[j][k] = fa[fa[j][k - 1]][k - 1];
            }
        }
    }
}
int lca(int a, int b) {
    if (depth[a] < depth[b]) swap(a, b); //从低的往高的跳
    //跳到同一层
    for (int k = 15; k >= 0; k--)
        if (depth[fa[a][k]] >= depth[b])
            a = fa[a][k];
    if (a == b) return a; //return b 这种情况是 b 是 a 的祖先
    //同时往上跳
    for (int k = 15; k >= 0; k--)
        if (fa[a][k] != fa[b][k]) {
            a = fa[a][k];
            b = fa[b][k];
        }
    return fa[a][0]; //或者 fa[b][0]
}
int main() {
    scanf("%d", &n);
    int root = 0;
    memset(h, -1, sizeof h);
    for (int i = 0; i < n; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
```

```
        if (b == -1) root = a;
        else add(a, b), add(b, a);
    }
    bfs(root); //预处理 depth 数组和倍增数组
    scanf("%d", &m);
    while (m --) {
        int a, b ;
        scanf("%d%d", &a, &b);
        int p = lca(a, b);
        if (p == a) puts("1");
        else if (p == b) puts("2");
        else puts("0");
    }
    return 0;
}
/* 次小生成树
给定一张 N 个点 M 条边的无向图，求无向图的严格次小生成树。
设最小生成树的边权之和为 sum，严格次小生成树就是指边权之和大于
↪ sum 的生成树中最小的一个。
*/
//倍增 LCA 优化寻找最大和次大边权的过程 暴力 dfs 改为倍增求最大
↪ 次大
//其余步骤不变 先 kruskal 然后枚举所有非树边
//d1[i][j] 表示从 i 开始向上跳 2^j 步 过程当中，路上的最大边权
//d2[i][j] 表示从 i 开始向上跳 2^j 步 路径上的次大边权
const int N = 100010, M = 300010;
const int INF = 0x3f3f3f3f;
int n, m;
int h[N], e[M], w[M], ne[M], idx;
struct Edge {
    int a, b, w;
    bool used; //是否是树边
    bool operator< (const Edge &t) {
        return w < t.w;
    }
} edge[M];
int p[N]; //kruskal 的并查集
int depth[N], fa[N][17], d1[N][17], d2[N][17]; //最大边和次大边
int q[N]; //BFS 求节点的树深度
void add(int a, int b, int c) {
```

```
    e[idx] = b;
    ne[idx] = h[a];
    w[idx] = c;
    h[a] = idx ++;
}
int find(int x) {
    if (x != p[x]) p[x] = find(p[x]);
    return p[x];
}
ll kruskal() {
    for (int i = 1; i <= n; i++) p[i] = i;
    sort(edge, edge + m);
    ll res = 0;
    for (int i = 0; i < m; i++) {
        int a = find(edge[i].a), b = find(edge[i].b), w =
        ↪  edge[i].w;
        if (a != b) {
            p[a] = b;
            res += w;
            edge[i].used = true;
        }
    }
    return res;
}
void build() {
    memset(h, -1, sizeof h);
    for (int i = 0; i < m; i++)
        if (edge[i].used) {
            int a = edge[i].a, b = edge[i].b, w = edge[i].w;
            add(a, b, w); add(b, a, w);
        }
}
void bfs() { //预处理节点深度 以及每个点跳 2^k 后路径上的最大边
↪  和次大边
    memset(depth, 0x3f, sizeof depth);
    depth[0] = 0, depth[1] = 1;
    int hh = 0, tt = 0;
    q[0] = 1;
    while (hh <= tt) {
        int t = q[hh ++];
```

```
        for (int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            if (depth[j] > depth[t] + 1) {
                depth[j] = depth[t] + 1;
                q[++tt] = j;
                fa[j][0] = t;
                d1[j][0] = w[i], d2[j][0] = -INF;
                for (int k = 1; k <= 16; k++) {
                    int anc =  fa[j][k - 1];
                    fa[j][k] = fa[anc][k - 1];
                    //每次更新一段 都是用 2 小端来更新, 先预处理
                    ↪ 2 次都跳 2^(k-1) 步然后遍历一次求出
                    int distance[4] = {d1[j][k - 1], d2[j][k -
                    ↪ 1], d1[anc][k - 1], d2[anc][k - 1]};
                    d1[j][k] = d2[j][k] = -INF;
                    for (int u = 0; u < 4; u++) {
                        int d = distance[u];
                        if (d > d1[j][k]) d2[j][k] = d1[j][k],
                        ↪ d1[j][k] = d;
                        else if (d != d1[j][k] && d >
                        ↪ d2[j][k]) d2[j][k] = d;
                    }
                }
            }
        }
    }
}
int lca(int a, int b, int w) {
    static int distance[N * 2];  //缓存每一次跳的一段路上的最大
    ↪ 和次大 最后取 max
    int cnt = 0;
    if (depth[a] < depth[b]) swap(a, b);
    for (int k = 16; k >= 0; k--)
        if (depth[fa[a][k]] >= depth[b]) {
            distance[cnt++] = d1[a][k];
            distance[cnt++] = d2[a][k];
            a = fa[a][k];
        }
    if (a != b) {
        for (int k = 16; k >= 0; k--)
```

```
            if (fa[a][k] != fa[b][k]) { //将
                distance[cnt++] = d1[a][k];
                distance[cnt++] = d2[a][k];
                distance[cnt++] = d1[b][k];
                distance[cnt++] = d2[b][k];
                a = fa[a][k], b = fa[b][k];
            }
        distance[cnt++] = d1[a][0];
        distance[cnt++] = d1[b][0]; //最后加上到 lca 的一步
    }
    int dist1 = -INF, dist2 = -INF;  //最大值和次大值
    for (int i = 0; i < cnt; i++) {
        int d = distance[i];
        if (d > dist1) dist2 = dist1, dist1 = d;
        else if (d != dist1 && d > dist2) dist2 = d;
    }
    if (w > dist1) return w - dist1;
    if (w > dist2) return w - dist2;
    return INF;
}
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 0; i < m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        edge[i] = {a, b, c};
    }
    ll sum = kruskal(); //最小生成树的权值和
    build(); //建图
    bfs(); //预处理
    ll res = 1e18;
    //枚举非树边
    for (int i = 0; i < m; i++)
        if (!edge[i].used) {
            int a = edge[i].a, b = edge[i].b, w = edge[i].w;
            res = min(res, sum + lca(a, b, w)); //lca 返回的是
            ↪ 加上最大的边 然年减去原来的树边的值
        }
    printf("%lld\n", res);
    return 0;
```

```
}
/*
给出 n 个点的一棵树，多次询问两点之间的最短距离。
边是无向的。
所有节点的编号是 1,2,⋯,n。
输入格式
第一行为两个整数 n 和 m。n 表示点数，m 表示询问次数；
下来 n-1 行，每行三个整数 x,y,k，表示点 x 和点 y 之间存在一条边长度为 k；
再接下来 m 行，每行两个整数 x,y，表示询问点 x 到点 y 的最短距离。
树中结点编号从 1 到 n。
*/
//在线做法 倍增 离线做法 Tarjan 算法 复杂度 O(N + M) N 是节点数量 M 是询问数量 比倍增更快
//树上 2 点的最短距离 就是 2 点的路径长度 求出 2 个点到 root 节点的距离求和 然后减去 lca 到 root 距离的 2 倍
const int N = 20010, M = 2 * N; //无向边 开 2 倍
int n, m;
int h[N], e[M], w[M], ne[M], idx;
int p[N]; //并查集父节点
int res[N]; //询问的结果
int st[N]; //tarjan 过程中的标记数组
int dist[N]; //每个点和 root 的距离
vector<pii> query[N]; //询问 first 查询的另外一个点 second 是查询编号
void add(int a, int b, int c) {
    e[idx] = b;
    ne[idx] = h[a];
    w[idx] = c;
    h[a] = idx ++;
}
int find(int x) {
    if (x != p[x]) p[x] = find(p[x]);
    return p[x];
}
void dfs(int u, int fa) { //预处理到 root 的距离
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (j == fa) continue; // 防止反向搜
        dist[j] = dist[u] + w[i];
```

```
        dfs(j, u);
    }
}
//正在搜的点标记为 1 搜过的点也就是回溯过的点 标记为 2 还没有搜过的点标记为 0
void tarjan(int u) {
    st[u] = 1; //正在搜的点
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (!st[j]) {
            tarjan(j);
            p[j] = u; //遍历 u 的所有相邻点 然后合并进并查集
        }
    }
    for (auto item : query[u]) { // 遍历所有和 u 相关的查询
        int y = item.first, id = item.second;
        if (st[y] == 2) { //如果要查询的点 以及被合并过了
            int anc = find(y);
            res[id] = dist[u] + dist[y] - dist[anc] * 2;
        }
    }
    st[u] = 2; //被遍历过了
}
int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    for (int i = 0; i < n - 1; i ++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c); add(b, a, c);
    }
    for (int i = 0; i < m; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        if (a != b) { //如果 2 节点相同 那就是 0, res 是全局数组 默认就是 0
            query[a].pb({b, i}); //和 a 查询相关的点是 b 查询编号是 i
            query[b].pb({a, i});
        }
```

```
    }
    for (int i = 1; i <= n; i++) p[i] = i; //初始化并查集
    dfs(1, -1); ////随便拿一个点当 root 即可 从 1 号点开始
    tarjan(1); //假设 1 号就是 root 了
    for (int i = 0; i < m; i++) printf("%d\n", res[i]);
    return 0;
}
```

4.7 Tarjan 算法

4.7.1 有向图强连通分量

```
/*
现在有 N 头牛，编号从 1 到 N，给你 M 对整数 (A,B)，表示牛 A 认为牛 B 受欢迎。
这种关系是具有传递性的，如果 A 认为 B 受欢迎，B 认为 C 受欢迎，那么牛 A 也认为牛 C 受欢迎。
求出有多少头牛被除自己之外的所有牛认为是受欢迎的。
输入格式
第一行两个数 N,M；
接下来 M 行，每行两个数 A,B，意思是 A 认为 B 是受欢迎的（给出的信息有可能重复，即有可能出现多个 A,B）。
*/
//如果暴力 可以在反图上遍历每个点是否能到其他所有点 复杂度 O(N * (N + M)) 会超时
//Tarjan 强连通分量 + 缩点 + 拓扑图 拓扑图至少存在一个出度为 0 的点
//A 认为 B 受欢迎，那么就从 A 向 B 连一条边
//对于缩点之后的拓扑图 如果存在 2 个及以上的出度为 0 的点 那么答案就是 0 因为其中一个出度为 0 的点 肯定不会被另外一个出度为 0 的点欢迎
//如果只有一个出度为 0 的缩点 那么就是答案 这个缩点代表的强连通分量里面的缩点之前的点的个数就是答案
//没必要建立缩点之后的拓扑图 只需要遍历一下强连通分量 然后统计出边即可
const int N = 10010, M = 50010;
int n, m;
int h[N], e[M], ne[M], idx;
int dfn[N], low[N], timestamp; //时间戳 low 数组
int stk[N], top;  //栈和栈顶
```

```
bool in_stk[N]; //每个点是否在栈中
int id[N], scc_cnt, sz[N];  //每个点所属的强连通分量编号 强连通分量的总数 每个点所在强连通分量的点的数量
int dout[N]; //记录每个强连通分量的出度
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
void tarjan(int u) {
    dfn[u] = low[u] = ++timestamp; //初始化时间戳 和 low 数组 然后后期更新 low 数组
    stk[++top] = u, in_stk[u] = true; //入栈
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (!dfn[j]) {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
        else if (in_stk[j]) low[u] = min(low[u], dfn[j]); //如果已经在栈中了 就直接拿来更新 low
    }
    if (dfn[u] == low[u]) { //找到强连通分量
        ++ scc_cnt; //数量 +1
        int y;
        do {
            y = stk[top--];
            in_stk[y] = false;
            id[y] = scc_cnt;
            sz[scc_cnt]++;
        } while (u != y); //弹出栈顶到 u 的所有元素 这些元素就组成一个强连通分量
    }
}
int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m --) {
        int a, b ;
        scanf("%d%d", &a, &b);
```

```
        add(a, b);
    }
    for (int i = 1; i <= n; i++)
        if (!dfn[i])
            tarjan(i);
    //统计所有点的出度 同是处理缩边后的图的连通分量的出度
    for (int i = 1; i <= n; i++)
        for (int j = h[i]; j != -1; j = ne[j]) {
            int k = e[j];
            int a = id[i], b = id[k];
            if (a != b) dout[a]++; //a b 是两个不同强连通分量的
            ↪ 编号 那么就在这个缩点图里面 a 的出度加一
        }
    int zeros = 0, sum = 0; //出度为 0 的分量个数 以及所有这样
    ↪ 分量内小的点的总数量
    for (int i = 1; i <= scc_cnt; i++)
        if (!dout[i]) {
            zeros ++;
            sum += sz[i];
            if (zeros > 1) { //没有符合条件的牛
                sum = 0;
                break;
            }
        }
    printf("%d\n", sum);
    return 0;
}
/*
恒星的亮度最暗是 1
现在对于 N 颗我们关注的恒星，有 M 对亮度之间的相对关系已经判明。
你的任务就是求出这 N 颗恒星的亮度值总和至少有多大。
输入格式
第一行给出两个整数 N 和 M。
之后 M 行，每行三个整数 T，A，B，表示一对恒星 (A，B) 之间的亮度
↪ 关系。恒星的编号从 1 开始。
如果 T = 1，说明 A 和 B 亮度相等。
如果 T = 2，说明 A 的亮度小于 B 的亮度。
如果 T = 3，说明 A 的亮度不小于 B 的亮度。
如果 T = 4，说明 A 的亮度大于 B 的亮度。
如果 T = 5，说明 A 的亮度不大于 B 的亮度。
```

```
*/
//此题也可以 spfa 差分约束跑负环 但是需要栈优化 而且可能会被卡掉
//用强连通分量可以线性复杂度稳过掉
//因为所有的环必然在强连通分量中 此问题有解的前提是图中没有正环 不然不存在最长路
//也就是说会正环出现 a >= b, b >=c, c >= a 这种矛盾关系
//同理 如果一个强连通分量中有一个正边权的边 那么必然存在正环 直接意味着无解
//如果有解 那么一个强连通分量内的边权 全部都是 0 意思是每个强连通分量内的点到起点的距离都一样的
//那么就进行缩点 然后在拓扑图上直接求和即可
//注意还要一个虚拟原点 又每个边权都 >=1 设虚拟原点为 x[0] 那么 x[i] >= x[0] + 1 每个点向虚拟远点连边权为 1 的边
const int N = 100010, M = 600010;  //每个点对的关系 最坏是都相等, 然后加上虚拟原点的边 一共开 3 倍
//然后要建缩点后的图 所以需要 6 倍的边数
int n, m;
int h[N], e[M], ne[M], w[M], idx;
int hs[N]; //缩点图的表头
int dfn[N], low[N], timestamp;
int stk[N], top;
bool in_stk[N];
int id[N], scc_cnt, sz[N]; //答案加上每个连通分量内点的个数 * 距离 所以需要维护 size
int dist[N]; //最长路
void add(int h[], int a, int b, int c) {
    e[idx] = b;
    ne[idx] = h[a];
    w[idx] = c;
    h[a] = idx ++;
}
void tarjan(int u) {
    dfn[u] = low[u] = ++timestamp;
    stk[++top] = u, in_stk[u] = true;
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (!dfn[j]) {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
```

```
        else if (in_stk[j])  low[u] = min(low[u], dfn[j]);
    }
    if (dfn[u] == low[u]) {
        ++scc_cnt;
        int y;
        do {
            y = stk[top--];
            in_stk[y] = false;
            id[y] = scc_cnt;
            sz[scc_cnt]++;
        } while (y != u);
    }
}
int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    memset(hs, -1, sizeof hs);
    //虚拟原点连边
    for (int i = 1; i <= n; i++) add(h, 0, i, 1);
    while (m --) {
        int t, a, b;
        scanf("%d%d%d", &t, &a, &b);
        if (t == 1) add(h, b, a, 0), add(h, a, b, 0);
        else if (t == 2) add(h, a, b, 1);
        else if (t == 3) add(h, b, a, 0);
        else if (t == 4) add(h, b, a, 1);
        else add(h, a, b, 0);
    }
    tarjan(0); //0 号点可以到其他所有点
    bool success = true;
    //建新的缩点图
    for (int i = 0; i <= n; i++) {
        for (int j = h[i]; j != -1; j = ne[j]) {
            int k = e[j];
            int a = id[i], b = id[k];
            if (a == b) {
                if (w[j] > 0) { //存在正边 无解
                    success = false;
                    break;
                }
```

```
            }
            else add(hs, a, b, w[j]);
        }
        if (!success) break; //如果一个分量不成立那么直接退出即
        ↪ 可 无解
    }
    if (!success) puts("-1");
    else {
        for (int i = scc_cnt; i; i --) //反向就是拓扑序
            for (int j = hs[i]; j != -1; j = ne[j]) {
                int k = e[j];
                dist[k] = max(dist[k], dist[i] + w[j]);
            }
        ll res  = 0;
        for (int i = 1; i <= scc_cnt; i++) res += 1ll *
        ↪ dist[i] * sz[i];
        printf("%lld\n", res);
    }
    return 0;
}
```

4.7.2 无向图边双连通分量

```
/*
例题：冗余路径
*/
/*
    一些连通分量的性质：
    一个孤立的点是一个点连通分量
    每个割点至少在 2 个点连通分量中
    两个割点之间的边不一定是桥 一个桥的两个端点也不一定是割点
    点连通分量和边连通分量没有直接关系 也就是点连不一定是边连
    一个树的每个边都是桥 但每个点不一定是割点，比如叶节点
    树中任意一条边的 2 个端点构成一个点联通
*/
/*
    题意为给定一个无向连通图 求最少需要加多少条边可以变为 e-DCC
    路径相互分离的定义是 两条路径所有的边都不一样不能有相同的边
    所以最终的图的状态是任意一条边都至少在一个环中 等价于 e-DCC
    解法：先 tarjan 缩点 然后缩点后的边 全部都是桥而且是拓扑的
    对于所有度数为 1 的点，度是出度 + 入度 都肯定需要连边
```

```
    而且在树中 度为 1 的点 就是所有的叶子节点 那么最少的连法 就是尽量多的" 配对"
    假设度为 1 的点是 cnt 个 那么答案最少就是 (cnt + 1) / 2
*/
const int N = 5010, M = 20010;
int n, m;
int dcc_cnt;
int h[N], e[M], ne[M], idx;
int dfn[N], low[N], timestamp;
int stk[N], id[N], top;
// bool in_stk[N];
bool is_bridge[M];  //判断是不是桥
int d[N]; //每个点的度数
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
void tarjan(int u, int from) {
    dfn[u] = low[u] = ++timestamp;
    stk[++top] = u;
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (!dfn[j]) {
            tarjan(j, i);
            low[u] = min(low[u], low[j]); //是搜索树的边 用 low 更新
            if (dfn[u] < low[j]) //如果成立那就是桥
                is_bridge[i] = is_bridge[i ^ 1] = true;
        }
        else if (i != (from ^ 1)) //i 不是搜索树的边 用 dfn[j] 来更新
            low[u] = min(low[u], dfn[j]);
    }
    if (dfn[u] == low[u]) {
        ++ dcc_cnt;
        int y;
        do {
            y = stk[top --];
            // in_stk[y] = false;
```

```
            id[y] = dcc_cnt;
        } while (y != u);
    }
}
int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m --) {
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b); add(b, a);
    }
    tarjan(1, -1); //连通图 从一个点开始 dfs 即可 由于是无向边
    ↪ 要记录当前的边是哪个边过来的
    for (int i = 0; i < idx; i ++) //枚举每个边即可
        if (is_bridge[i])
            d[id[e[i]]] ++;//给 i 的出边所在点连通分量的度数加
            ↪ 一由于是无向边到出边的点又会反过来给 i 加上
    int cnt = 0;
    for (int i = 1; i <= dcc_cnt; i++)
        if (d[i] == 1) cnt ++;
    printf("%d\n", (cnt + 1) / 2);
    return 0;
}
```

4.7.3 无向图点双连通分量

```
/*
例题: Tarjan 判断割点
给定一个由 n 个点 m 条边构成的无向图，请你求出该图删除一个点之后，
↪ 连通块最多有多少。输入包含多组数据。
每组数据第一行包含两个整数 n,m。
接下来 m 行，每行包含两个整数 a,b，表示 a,b 两点之间有边连接。
数据保证无重边。
点的编号从 0 到 n-1。
读入以一行 0 0 结束。
*/
//注意题目给的并以一定是连通图 所以 要先求一次所有连通块 然后分
↪ 别处理每个连通块
```

```
//一共有 cnt 个连通块，那么处理完一个连通块假设该连通块可以分为
↪  s 个最多，那么处理完该连通块的结果就是 s + cnt - 1
//依次同样方法处理所有连通块即可
//求一个连通块的 s: 判断割点
const int N = 10010, M = 30010;
int n, m;
int h[N], e[M], ne[M], idx;
int dfn[N], low[N], timestamp;
int root, ans;
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
void tarjan(int u) {
    dfn[u] = low[u] = ++timestamp;
    int cnt = 0; //当前连通块中的 cnt
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (!dfn[j]) {
            tarjan(j);
            low[u] = min(low[u], low[j]);
            if (low[j] >= dfn[u])
                cnt ++;
        }
        else low[u] = min(low[u], dfn[j]);
    }
    if (u != root && cnt) cnt ++; //u 节点有 cnt 个子树 去掉 u
    ↪  后 u 的父亲连通块就会多出来 以及 cnt 个单独的子树
    ans = max(ans, cnt);
}
int main() {
    while (scanf("%d%d", &n, &m) , n || m) {
        memset(dfn, 0, sizeof dfn);
        memset(h, -1, sizeof h);
        idx = timestamp = 0;
        while (m --) {
            int a, b;
            scanf("%d%d", &a, &b);
            add(a, b); add(b, a);
```

```
        }
        int cnt = 0;
        ans = 0; //每次清空答案
        for (root = 0; root < n; root++) {
            if (!dfn[root]) {
                cnt ++;  //非连通图 tarjan 还可以顺便求出连通块
                ↪ 数量 因为 tarjan 的本质就是 dfs
                tarjan(root);
            }
        }
        printf("%d\n", ans + cnt - 1);
    }
    return 0;
}
//例题: 一个孤立的点也是一个 v-DCC
/*
矿场搭建: 某些挖煤点设立救援出口, 使得无论哪一个挖煤点坍塌之后,
↪ 其他挖煤点的工人都有一条道路通向救援出口。
请写一个程序, 用来计算至少需要设置几个救援出口, 以及不同最少救援
↪ 出口的设置方案总数。
*/
/*
    一个图中至少要有 2 个救援出口 因为如果有 1 个 那么这个出口坏
    ↪ 掉 就不能出去
    注意题意暗示 只会有一个挖煤点出事 而且不是连通图 有多个连通
    ↪ 块 处理每个连通块的方案数 总的方案数 就是乘起来即可
    1) 如果一个连通块中没有割点 那么随便选 2 个即可 假设连通块点
    ↪ 的数量是 cnt 那么就是 cnt 选 2
    2) 如果有割点 就要缩点 每个割点都至少属于 2 个 v-DCC 假设缩
    ↪ 点后的每个分量是一个大点
    2.1) 如果一个大点的度数为 1 也就是这个大点只和一个割点连通
    ↪ 那么就必须要在大点内部, 而且不是割点的点中随便放一个
        因为割点坏掉, 那么大点就内部封死了 方案数是 cnt-1
    2.2) 如果一个大点的度数大于 1 那么无需在该大点内部 设置出口
    ↪ 忽略这个大点即可
    从缩点的过程来看 每个缩点后的大点的度数 就是这个分量内部的割
    ↪ 点的个数 因为割点会被独立出去
    缩点的时候, 每个割点都会向所有包含它的 v-DCC 连边
```

```
    从树的本质理解：缩点后的图是一个无环图 也就是一棵树，度数为
↪   0 的点都是叶子节点
*/
const int N = 1010, M = 510;
int n, m;
int h[N], e[M], ne[M], idx;
int dfn[N], stk[N], low[N], top;
int timestamp;
int dcc_cnt;
vector<int> dcc[N];
bool cut[N]; //判断是否是割点
int root;
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
void tarjan(int u) {
    dfn[u] = low[u] = ++timestamp;
    stk[++top] = u;

    if (u == root && h[u] == -1) {
        dcc_cnt ++; //孤立点
        dcc[dcc_cnt].push_back(u);
        return;
    }
    int cnt = 0;
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (!dfn[j]) {
            tarjan(j);
            low[u] = min(low[u], low[j]);
            if (dfn[u] <= low[j]) {
                cnt ++; //u 节点的子树的数量
                //判断割点
                if (u != root || cnt > 1) cut[u] = true;
                ++dcc_cnt;
                int y;
                do {
                    y = stk[top --];
```

```
                    dcc[dcc_cnt].push_back(y);
                } while (y != j); //注意此处
                dcc[dcc_cnt].push_back(u);
            }
        }
        else low[u] = min(low[u], dfn[j]);
    }
}
int main() {
    int t = 1;  // 输出样例计数器
    while (cin >> m && m) {
        for (int i = 1; i <= dcc_cnt; i++) dcc[i].clear();
        memset(h, -1 , sizeof h);
        idx = timestamp = n = dcc_cnt = top = 0;
        memset(dfn, 0, sizeof dfn);
        memset(cut, 0, sizeof cut);
        while (m --) {
            int a, b;
            cin >> a >> b;
            n = max(n, a), n = max(n, b);
            add(a, b); add(b, a);
        }
        for (root = 1; root <= n; root++) //以每个点为 root 深
        ↪ 搜一次
            if (!dfn[root]) tarjan(root);
        ull res = 0; //最少需要的出口的数量
        ull num = 1; //方案数
        //求出割点的数量 遍历每一个 dcc
        for (int i = 1; i <= dcc_cnt; i++) {
            int cnt = 0;
            for (int j = 0; j < sz(dcc[i]); j++)
                if (cut[dcc[i][j]])
                    cnt ++;
            //处理当前的 dcc 分类讨论 注意要特判只有一个孤立点的
            ↪ 情况
            if (cnt == 0) {
                if (sz(dcc[i]) > 1) res += 2, num *= sz(dcc[i])
                ↪ * (sz(dcc[i]) - 1) / 2;
                else res ++;
            }
```

```
            else if (cnt == 1) {
                res ++;
                if (sz(dcc[i]) > 1) num *= sz(dcc[i]) - 1;
                else num *= 1;
            }
        }
        printf("Case %d: %llu %llu\n", t++, res, num);
    }
    return 0;
}
```

4.8 欧拉回路/路径

```
/*
给定一张图，请你找出欧拉回路，即在图中找一个环使得每条边都在环上出现恰好一次。
输入格式
第一行包含一个整数 t，t {1,2}，如果 t=1，表示所给图为无向图，如果 t=2，表示所给图为有向图。
第二行包含两个整数 n,m，表示图的结点数和边数。
接下来 m 行中，第 i 行两个整数 vi,ui，表示第 i 条边，从 1 开始
如果 t=1 则表示 vi 到 ui 有一条无向边。
如果 t=2 则表示 vi 到 ui 有一条有向边。
图中可能有重边也可能有自环，点的编号从 1 到 n。
*/
//无向图：连通图 而且所有点度数都是偶数
//有向图：连通图 而且所有点的入度等于出度
//一般没必要加删表头的优化 但是小概率会被卡掉
//欧拉回路和孤立点没有关系 只和边有关系
const int N = 100010, M = 400010;//无向图 二倍边
int h[N], e[M], ne[M], idx;
bool used[M]; //标记边是否被用过
int ans[M], cnt; //cnt 记录答案路径的长度
int din[N], dout[N]; //度数
int type;
int n, m;
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
```

```
}
void dfs(int u) {
    while (h[u] != -1) {
        int i = h[u];
        if (used[i]) {
            h[u] = ne[i]; //删边
            continue;
        }
        h[u] = ne[i];
        used[i] = true;
        if (type == 1) used[i ^ 1] = true; //无向图标记反向边
        dfs(e[i]);
        //加边时候第一条边的编号是 0 开始 但是题目是边从 1 开始
        if (type == 1) {
            int t = i / 2 + 1; //无向图边的编号 t 因为每次都是
            ↪ 加成对的边 但是输入只输入一条
            if (i & 1) t *= -1; //边的编号是奇数 说明是反向边
            ↪ 因为加边的时候是从 0 开始成对加
            ans[++cnt] = t;
        }
        else ans[++cnt] = i + 1;
    }
}
int main() {
    scanf("%d", &type);
    scanf("%d%d", &n, &m);
    memset(h, -1 , sizeof h);
    for (int i = 0; i < m; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b);
        if (type == 1) add(b, a);
        din[b] ++, dout[a] ++;
    }
    //提前判断无解的情况
    if (type == 1) {
        for (int i = 1; i <= n; i++)
            if (din[i] + dout[i] & 1) {
                puts("NO");
                return 0;
```

```
            }
        }
        else {
            for (int i = 1; i <= n; i++)
                if (din[i] != dout[i]) {
                    puts("NO");
                    return 0;
                }
        }
        //找一个包含边的起点
        for (int i = 1; i <= n; i++)
            if (h[i] != -1) {
                dfs(i);
                break;
            }
        if (cnt < m) { //有孤立边
            puts("NO");
            return 0;
        }
        puts("YES");
        for (int i = cnt ; i ; i --) printf("%d ", ans[i]);
        puts("");
        return 0;
}
/*
给出 n 个单词，是否能收尾相连成一个串第一行包含整数 T，表示共有 T
↪ 组测试数据。
每组数据第一行包含整数 N，表示盘子数量。接下来 N 行，每行包含一个
↪ 小写字母字符串，表示一个单词。一个单词可能出现多次。
*/
//26 个字母作为点 然后单词作为边 求是否存在欧拉路径 注意这是有向
↪ 图 因为字母是点
//除了起点终点 其余点都满足出度等于入度 而且所有边都连在一起
↪ (dfs/并查集) 判断连通即可
//边上的字母不重要 不用记录 只需要记录端点的 2 个字母即可
const int N = 30;
int n, m;
int p[N];
bool st[N];
int din[N], dout[N];
```

```
int find(int x) {
    if (x != p[x]) p[x] = find(p[x]);
    return p[x];
}
int main() {
    char str[1010];
    int t;
    cin >> t;
    while (t --) {
        scanf("%d", &n);
        memset(din, 0, sizeof din);
        memset(dout, 0, sizeof dout);
        memset(st, 0, sizeof st);
        for (int i = 0; i < 26; i++) p[i] = i; //初始化并查集
        for (int i = 0; i < n; i++) {
            scanf("%s", str);
            int len = strlen(str);
            int a = str[0] - 'a', b = str[len - 1] - 'a';
            st[a] = st[b] = true;
            dout[a] ++, din[b] ++;
            p[find(a)] = find(b);
        }
        int start = 0, end = 0;
        bool success = true;
        for (int i = 0; i < 26; i++)
            if (din[i] != dout[i]) {
                if (din[i] == dout[i] + 1) end ++;
                else if (din[i] + 1 == dout[i]) start++;
                else {
                    success = false;
                    break;
                }
            }
        //如果上边的 if 没有进入 说明所有点都是入度 == 出度
        if (success && !(!start && !end || start == 1 && end ==
        ↪  1)) success = false; //有多个起点和终点 那就不满足
        //判断连通性
        int rep = -1;
        for (int i = 0; i < 26; i++)
```

```
            if (st[i]) { //st[i] 是已经有单词 也就是右边的前提
            ↪ 下 而且没有在并查集中 说明是孤立边
                if (rep == -1) rep = find(i);
                else if (rep != find(i)) {
                    success = false;
                    break;
                }
            }
        if (success) puts("Ordering is possible.");
        else puts("The door cannot be opened.");
    }
    return 0;
}
```

4.9 拓扑排序

```
/*
时间复杂度 O(n+m)O(n+m)，nn 表示点数，mm 表示边数
第一行包含两个整数 n 和 m
接下来 m 行，每行包含两个整数 x 和 y，表示存在一条从点 x 到点 y
↪ 的有向边 (x, y)。
输出格式：共一行，如果存在拓扑序列，则输出任意一个合法的即可
否则输出-1。
*/
queue<int> q;
int n, m;
const int N = 100010;
int h[N], e[N], ne[N], idx;
int d[N], path[N];
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
bool topsort() {
    for (int i = 1; i <= n; i++)
        if (d[i] == 0) q.push(i);
    int cnt = 0;
    while (q.size()) {
        int t = q.front();
```

```
        path[cnt++] = t;
        q.pop();
        for (int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            if (-- d[j] == 0) q.push(j);
        }
    }
    return cnt == n ;
}
int main() {
    memset(h, -1, sizeof h);
    cin >> n >> m;
    int a, b;
    while (m --) {
        cin >> a >> b;
        add(a, b);
        d[b]++;
    }
    if (topsort()) {
        for (int i = 0; i < n; i++) cout << path[i] << ' ';
        cout << endl;
    } else cout << -1 << endl;
    return 0;
}
/*
例题：可达性统计
给定一张 N 个点 M 条边的有向无环图，分别统计从每个点出发能够到达
↪ 的点的数量。
输入格式
第一行两个整数 N,M，接下来 M 行每行两个整数 x,y，表示从 x 到 y
↪ 的一条有向边。
输出格式：输出共 N 行，表示每个点能够到达的点的数量。
*/
const int N = 30010;
int n, m;
int h[N], e[N], ne[N], idx;
int d[N], seq[N];
bitset<N> f[N];
void add(int a, int b) {
    e[idx] = b; ne[idx] = h[a], h[a] = idx++;
```

```
}
void topsort() {
    queue<int> q;
    for (int i = 1; i <= n; i++)
        if (!d[i]) q.push(i);
    int k = 0;
    while (q.size()) {
        int t = q.front();
        q.pop();
        seq[k ++] = t; //储存排序后的序列
        for (int i = h[t]; ~i; i = ne[i]) {
            int j = e[i];
            if (-- d[j] == 0) q.push(j);
        }
    }
}
int main() {
    cin >> n >> m;
    memset(h , -1, sizeof h);
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        d[b] ++;  // b 点的入度加一
        add(a, b);
    }
    topsort();
    for (int i = n - 1; ~i; i--) { //从末尾开始
        int j = seq[i];  //拓扑序列的点 开始往后走
        f[j][j]  = 1; //自己可到达自己
        for (int p = h[j]; ~p; p = ne[p])
            f[j] |= f[e[p]];  //e[p] 代表这条边指向的点  求并集
    }
    for (int i = 1; i <= n; i++) cout << f[i].count() << endl;
}
```

5 数学

5.1 质数/筛法

```
//试除法判定质数 O(sqrt(n))
bool is_prime(int x) {
    if (x < 2) return false;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
            return false;
    return true;
}
//试除法分解质因数
void divide(int x) {
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0) {
            int s = 0;
            while (x % i == 0) x /= i, s ++ ;
            cout << i << ' ' << s << endl;
        }
    if (x > 1) cout << x << ' ' << 1 << endl;
    cout << endl;
}
/*
例题：阶乘分解：给定整数 N，试把阶乘 N! 分解质因数，按照算术基本
↪ 定理的形式输出分解结果中的 pi 和 ci 即可
1~N 中质因子 p 的个数和为 (N/p+N/(p^2)+....)
*/
const int N = 1000010;
int primes[N], cnt;
bool st[N];
void init(int n) { //线性筛
    for (int i = 2; i <= n; i++) {
        if (!st[i]) primes[cnt ++] = i;
        for (int j = 0; primes[j] <= n / i; j++) {
            st[i * primes[j]] = 1;
            if (i % primes[j] == 0) break;
        }
    }
}
int main() {
```

```
    int n;
    scanf("%d", &n);
    init (n);
    for (int i = 0; i < cnt; i++) {
        int p = primes[i];
        int s = 0;
        for (int j = n; j; j /= p) s += j / p;
        printf("%d %d\n", p , s);
    }
    return 0;
}
//埃氏筛 (1~n 中所有的质数)
int primes[N], cnt; // primes[] 存储所有素数
bool st[N];   // st[x] 存储 x 是否被筛掉
void get_primes(int n) {
    for (int i = 2; i <= n; i ++ ) {
        if (st[i]) continue;
        primes[cnt ++ ] = i;//cnt 记录素数个数
        for (int j = i + i; j <= n; j += i)
            st[j] = true;
    }
}
//线性筛 (筛出 1~n 中所有的质数)
int primes[N], cnt; // primes[] 存储所有素数
bool st[N]; // st[x] 存储 x 是否被筛掉
void get_primes(int n) {
    for (int i = 2; i <= n; i ++ ) {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ ) {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}
/*
例题：给出一个闭区间，求出区间内相邻最近和最远的质数对
如果存在多组，输出第一组 区间长度最多 1e6 范围 INT_MAX
*/
//左右端点是 int 范围 不能线性筛一次 会超时
//而且线性筛不能只筛一个区间 做不到
```

```cpp
//切入点在于相邻质数区间长度不会超过 1e6
//处理所有数的质因子,sqrt(INT_MAX)<50000 处理 50000 内的质数
//然后用埃氏筛筛掉所有 [2~sqrt(R)] 的合数 那么剩下的就是区间内所有的质数 然后逐个相邻比较即可
const int N = 1000010; //区间长度
int primes[N], cnt;
bool v[N];
void prime(int n) { //线性筛
    memset(v, 0, sizeof v);
    for (int i = 2; i <= n; i++) {
        if (!v[i]) primes[cnt++] = i;
        for (int j = 0; primes[j] <= n / i; j++) {
            v[primes[j] * i] = 1;
            if (i % primes[j] == 0) break;
        }
    }
}
int main() {
    int l, r;
    while (cin >> l >> r) {
        prime(50000);
        memset(v, 0, sizeof v);
        for (int i = 0; i < cnt; i++) {
            ll p = primes[i]; //质数的倍数可能会爆掉 int
            // L / P 上取整 * p 就是第一个大于 L 的 p 的倍数
            //上取整可以 ceil 函数 也可以 (L + p - 1) / p
            //计算出第一个倍数 j 之后然后每次 +p 取 max 是因为最少要二倍的倍数
            for (ll j = max(p * 2, (l + p - 1) / p * p); j <= r; j += p)
                v[j - l] = true;
        }
        cnt = 0;
        for (int i = 0; i <= r - l; i++)
            //过滤掉 1 不是素数的特殊情况
            if (!v[i] && i + l >= 2) primes[cnt++] = i + l; //复用 primes 数组 此时保存的是区间内的素数
        if (cnt < 2) puts("There are no adjacent primes."); //不足 2 个素数 就无解
        else {
```

```
            int minp = 0, maxp = 0; //枚举所有的相邻素数
            for (int i = 0; i + 1 < cnt; i++) {
                int d = primes[i + 1] - primes[i];
                if (d < primes[minp + 1] - primes[minp])minp=i;
                if (d > primes[maxp + 1] - primes[maxp])maxp=i;
            }
            printf("%d,%d are closest, %d,%d are most
            ↪  distant.\n", primes[minp], primes[minp + 1],
            ↪  primes[maxp], primes[maxp + 1]);
        }
    }
    return 0;
}
```

5.2 约数

```
/*
倍数法求 (1~N) 每个数的正约数集合
复杂度 O(N*logN)
*/
const int N = 10000;
vector<int> factor[N];
int n;
void calc(int n) {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n / i; j++)
            factor[i * j].push_back(i);
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < factor[i].size(); j++)
            printf("%d ", factor[i][j]);
        puts("");
    }
}
/*
试除法求一个数的正约数集合 O(sqrt(n))
*/
vector<int> get_divisors(int x) {
    vector<int> res;
    for (int i = 1; i <= x / i; i ++ )
        if (x % i == 0) {
            res.push_back(i);
```

```
            if (i != x / i) res.push_back(x / i);
        }
    sort(res.begin(), res.end());
    return res;
}
/*
约数个数：
如果 N = p1^c1 * p2^c2 * ... *pk^ck
约数个数：(c1 + 1) * (c2 + 1) * ... * (ck + 1)
*/
//给定 n 个正整数 ai，输出这些数的乘积的约数个数，答案对 1e9+7 取模。
const int mod = 1e9 + 7;
//将每一个 ai 分解 然后把指数相加就是最后的乘积的质因数分解结果
//约数个数就是指数加 1 再相乘即可
int main() {
    int n; cin >> n; int x;
    unordered_map<int, int> primes;
    while (n --) {
        cin >> x;
        for (int i = 2; i <= x / i; i++)
            while (x % i == 0) {
                x /= i;
                primes[i] ++;
            }

        if (x > 1) primes[x] ++;
    }
    ll res  = 1;
    for (auto prime : primes) res = res * (prime.second + 1) %
    ↪  mod;
    cout << res << endl;
    return 0;
}
/*
约数之和：(p1^0+p1^1+...+p1^c1)*...* (pk^0 + pk^1 + ... +
↪  pk^ck)
给定 n 个正整数 ai，请你输出这些数的乘积的约数之和，答案对 1e9+7
↪  取模
*/
```

```
typedef long long ll;
const int mod = 1e9 + 7;
int main() {
    unordered_map<int , int > primes;
    int n, x; cin >> n;
    while (n --) {
        cin >> x;
        for (int i = 2; i <= x / i; i++)
            while (x % i == 0) {
                x /= i;
                primes[i] ++;
            }
        if (x > 1) primes[x] ++;
    }
    ll res = 1;
    for (auto prime : primes) {
        int p = prime.first,  a = prime.second;
        ll t = 1;  //等比数列求和 递推法
        while (a--) t = (t * p + 1) % mod;
        res  = res * t % mod;
    }
    cout << res << endl;
    return 0;
}
```

5.3 欧拉函数

```
//分解质因子求某个数欧拉函数 复杂度 sqrt(n)
ll get_euler(ll c) {
    ll res = c;
    for (int i = 2; i <= c / i; i++)
        if (c % i == 0) {
            while (c % i == 0) c /= i;
            res = res / i  * (i - 1);
        }
    if (c > 1) res = res / c * (c - 1);
    return res;
}
//线性筛 求 1 ~ N 中所有数欧拉函数模板
const int N = 1010;
int n;
```

```cpp
int primes[N], cnt;
int phi[N]; //欧拉函数
bool v[N];
void get_euler(int n) { //线性筛求欧拉函数
    memset(v, 0, sizeof v);
    cnt = 0;
    phi[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!v[i]) {
            primes[cnt++] = i;
            phi[i] = i - 1;
        }
        for (int j = 0; primes[j] <= n / i; j++) {
            int t = primes[j] * i;
            v[t] = 1;
            if (i % primes[j] == 0) {
                phi[t] = phi[i] * primes[j];
                break;
            }
            phi[t] = phi[i] * (primes[j] - 1);
        }
    }
}
/*
例题：给定整数 N，求 1<=x,y<=N 且 GCD(x,y) 为素数的数对 (x,y)
↪ 有多少对。
*/
//gcd(a,b) == p，那么 gcd(a/p, b/p) == 1，枚举 n 内所有素数
//而求 1~n 中有序互质对 x，y 的个数，可以令 y >= x，当 y = x 时，
↪ 有且只有 y = x = 1 互质，
//当 y > x 时，确定 y 以后符合条件的个数 x 就是 phiy
// 所以有序互质对的个数为（1 ~ n/p）的欧拉函数之和乘 2 减 1（要
↪ 求的是有序互质对，乘 2 以后减去（1, 1）多算的一次）
// 那么就只需要先筛出欧拉函数再求个前缀和就可以了
const int N = 1e7 + 10;
int n;
int primes[N], cnt;
int phi[N]; //欧拉函数
bool v[N];
ll s[N];  //欧拉函数值的前缀和
```

```
void get_euler(int n) { //线性筛求欧拉函数
    memset(v, 0, sizeof v);
    cnt = 0;
    phi[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!v[i]) {
            primes[cnt++] = i;
            phi[i] = i - 1;
        }
        for (int j = 0; primes[j] <= n / i; j++) {
            int t = primes[j] * i;
            v[t] = 1;
            if (i % primes[j] == 0) {
                phi[t] = phi[i] * primes[j];
                break;
            }
            phi[t] = phi[i] * (primes[j] - 1);
        }
    }
    for (int i = 1; i <= n; i++) s[i] = s[i - 1] + phi[i];
}
int main() {
    cin >> n;
    get_euler(n);
    ll res = 0;
    for (int i = 0; i < cnt; i++) {
        int p = primes[i];
        res += s[n / p] * 2 - 1;
    }
    cout << res << endl;
    return 0;
}
```

5.4 快速幂/龟速乘

```
//求 m^k mod p, 时间复杂度 O(logk)。
ll qmi(ll a, ll b, int p) {
    ll res = 1; a %= p;
    while (b) {
        if (b & 1) res = res * a % p;
        b >>= 1;
```

```
        a = a * a % p;
    }
    return res;
}
//龟速乘 a * b % p 复杂度 log(b)
ll powmod(ll a, ll b, ll mod) {
    ll res = 1; a %= mod;
    assert(b >= 0);
    for (; b; b >>= 1) {
        if (b & 1)
            res = res * a % mod;
        a = a * a % mod;
    }
    return res;
}
/*
快速幂求乘法逆元:
给定 n 组 ai,pi, 其中 pi 是质数, 求 ai 模 pi 的乘法逆元返回在
↪ 0 p-1 之间的逆元
b 存在乘法逆元的充要条件是 b 与模数 m 互质。当模数 m 为质数时,
↪ b^(m-2) 即为 b 的乘法逆元。
*/
int n;
int main() {
    cin >> n;
    int a, p;
    while (n--) {
        cin >> a >> p;
        if (__gcd(a, p) != 1) cout << "impossible" << endl;
        else cout << qmi(a, p - 2 , p) << endl;
    }
    return 0;
}
```

5.5 扩展欧几里得算法

```
// 求 x, y, 使得 ax + by = gcd(a, b)
int exgcd(int a, int b, int &x, int &y) {
    if (!b) {
        x = 1; y = 0;
```

```
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return d; //最大公约数
}
//void 版本
void exgcd(ll a, ll b, ll &x, ll & y) {
    if (!b)
        x = 1, y = 0;
    else {
        exgcd(b, a % b, y, x);
        y -= a / b * x;
    }
}
/*
关于 x 的同余方程 ax  b(mod m) 的最小正整数解是
↪ (x*b/d%(m/d)+(m/d))%(m/d)
通解是 x0 + k * (m/d), d 是 gcd(a,m)
*/
int exgcd(int a, int b, int &x, int &y) {
    if (!b) {
        x = 1, y  = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return d;
}
int main() {
    cin >> n;
    int a, b , m;
    int x, y;
    while (n --) {
        cin >> a >> b >> m;
        int d = exgcd(a, m, x, y);
        if (b % d) puts("impossible");
        else cout << (ll)x * (b / d) % (m / d) << endl;
        //如果直接用 int 存再取模 答案错误 因为 int 会溢出
    }
```

```
    return 0;
}
```

5.6 中国剩余定理

```
//中国剩余定理模板
const int N = 11;
int n;
int A[N], B[N];
void exgcd(ll a, ll b, ll &x, ll & y) {
    if (!b)
        x = 1, y = 0;
    else {
        exgcd(b, a % b, y, x);
        y -= a / b * x;
    }
}
int main() {
    scanf("%d", &n);
    ll M = 1;
    //方程组为 x 同余 B[i] mod (A[i])
    for (int i = 0; i < n; i++) {
        scanf("%d%d", &A[i], &B[i]);
        M *= A[i];
    }
    ll res = 0;
    for (int i = 0; i < n; i++) {
        ll Mi = M / A[i];
        //求 Mi*ti 同余 1(mod mi) 的 ti
        ll ti, x;
        exgcd(Mi, A[i], ti, x);
        res += (B[i] * Mi * ti);
    }
    //中国剩余定理给出的通解是 x + k * M 对 M 取模即可得到最小正
    ↪ 整数解
    cout << (res % M + M )  % M << endl;
}
```

5.7 高斯消元

```
/*
a11*x1 + a12*x2 + ... + a1n*xn = b1
a21*x1 + a22*x2 + ... + a2n*xn = b2
       ... .... ... ..
am1*x1 + am2*x2 + ... + amn*xn = bm

第一行包含整数 n。
接下来 n 行，每行包含 n+1 个实数，表示一个方程的 n 个系数以及等号右侧的常数。
如果给定线性方程组存在唯一解，则输出共 n 行，其中第 i 行输出第 i 个未知数的解，结果保留两位小数。
如果给定线性方程组存在无数解，则输出 "Infinite group solutions"。
如果给定线性方程组无解，则输出 "No solution"。
*/
const int N = 110;
const double eps = 1e-6;
double a[N][N];
int n;
int gauss() {
    int c, r;  //列和行
    for (c = 0, r = 0; c < n; c++) { //列向右移动 因为要变成上三角矩阵 注意：该矩阵是 n * (n + 1) 的
        int t = r;
        for (int i = r; i < n; i++) //求出绝对值最大的行
            if (fabs(a[i][c]) > fabs(a[t][c]))
                t = i;
        if (fabs(a[t][c]) < eps) continue; //绝对值最大也是 0 说明这一列全 0 直接下一列 c++ 即可
        for (int i = c; i <= n; i++) swap(a[t][i], a[r][i]); //把这行换到最上面 (第 r 行) 也就是是固定行的下一行
        for (int i = n; i >= c; i--) a[r][i] /= a[r][c]; //第一个数变成 1 后边的同时改变
        for (int i = r + 1; i < n; i++) //把第一个数下面该列的元素 都消成 0 行下标是 0 ~ n-1
            if (fabs(a[i][c]) > eps) // 这一行该列元素不是 0 才需消掉
                for (int j = n; j >= c; j--)
```

```
                    a[i][j] -= a[r][j] * a[i][c]; //被消去的元素减去 第 r 行元素 * 该行第一个元素，因为 r 行第一个是 1
        r ++;
    }
    if (r < n) { //不是有唯一解
        for (int i = r; i < n; i++)
            if (fabs(a[i][n]) > eps) //0== 非 0
                return 2;  //无解
        return 1;//无穷多解
    }
    //有唯一解 将增广矩阵化为简化阶梯矩阵 每行最后一个元素就是唯一解了
    //这里就是把每行 元素 1 后边的非 0 元素全部改为 0  每行的 1 将该 1 所在列上方元素全部消成 0
    //但是事实上并没有消 只对于每行的最后一个元素在模拟这个过程 输出最后的矩阵 不是简化阶梯矩阵
    for (int i = n - 1; i >= 0; i -- )
        for (int j = i + 1; j < n; j ++ )
            a[i][n] -= a[i][j] * a[j][n];
    return 0;  // 唯一解
}
int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n + 1; j++) //每行 n + 1 个元素
            cin >> a[i][j];
    int t = gauss();
    if (t == 0) {
        for (int i = 0; i < n; i++) printf("%.2f\n", a[i][n]); //最后矩阵系数都是 1 最后列常数列就是解
    } else if (t == 1) puts("Infinite group solutions"); //无数解
    else puts("No solution");  //无解
    return 0;
}
/*
异或线性方程组
M[1][1]x[1] ^ M[1][2]x[2] ^ ··· ^ M[1][n]x[n] = B[1]
M[2][1]x[1] ^ M[2][2]x[2] ^ ··· ^ M[2][n]x[n] = B[2]
```

```
...
M[n][1]x[1] ^ M[n][2]x[2] ^ ... ^ M[n][n]x[n] = B[n]
第一行包含整数 n。
接下来 n 行，每行包含 n+1 个整数 0 或 1，表示一个方程的 n 个系数
↪ 以及等号右侧的常数。
*/
const int N  = 110;
int a[N][N]; int n;
int gauss() {
    int r, c;
    for (r = c = 0; c < n; c++) { //外层枚举列
        int t = r;
        for (int i = r; i < n; i++)
            if (a[i][c]) {
                t = i;
                break;  //找到一个非 0 行即可
            }
        if (!a[t][c]) continue;  //该列全部都是 0 直接下一列
        //该行交换上去 最上面 未固定的
        for (int i = c; i <= n; i++) swap(a[t][i], a[r][i]);
        ↪ //从左往右扫 交换 2 行的每一个元素
        for (int i = r + 1; i < n; i++) //将 1 下边的全部消成 0
            if (a[i][c])
                for (int j = c; j <= n; j++)
                    a[i][j] ^= a[r][j];   //  下边如果是 0 那异
                    ↪ 或 1 还是 0  如果是 1  那和 1 异或之后
                    ↪ 就是 0 了
        r++;
    }
    if (r < n) {
        for (int i = r; i < n; i++) //此时的 r 所在的行 就是只
        ↪ 有一个未知数的方程 开始判断 0 = 非 0
            if (a[i][n]) return 2; //无解
        return 1;  //无数解
    }
    //用每一行的主元 消去该列的上边的数 全部为 0
    for (int i = n - 1; i >= 0; i--)
        for (int j = i + 1; j < n; j++)
            a[i][n] ^= a[i][j] & a[j][n];
    return 0;  //唯一解
```

```
}
int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        for (int j = 0 ; j < n + 1; j++)
            cin >> a[i][j];
    int res = gauss();
    if (res == 0) { //唯一解
        for (int i = 0; i < n; i++) cout << a[i][n] << endl;
    } else if (res == 1) { //无数解
        cout << "Multiple sets of solutions" << endl;
    } else cout << "No solution" << endl;
    return 0;
}
```

5.8 组合数

```
/*
递推法 O(n ^ 2) n 是组合数 a
给 n 组询问每组询问两个整数 a,b; 输出 C(a,b)mod(1e9+7) 的值
1 <= n <= 10000, 1 <= b <= a <= 2000
*/
const int N = 2010, mod = 1e9 + 7;
int n;
int c[N][N];
void init() {
    for (int i = 0; i < N; i++)
        for (int j = 0; j <= i; j++)
            if (!j) c[i][j] = 1;
            else c[i][j] = (c[i - 1][j - 1] + c[i - 1][j]) %
             ↪  mod;
}
int main() {
    cin >> n; int a, b;
    init();
    while (n --) {
        cin >> a >> b;
        cout << c[a][b] << endl;
    }
    return 0;
}
```

```cpp
/*
预处理逆元 1 <= b <= a <= 1e5
1e9 + 7 是质数  复杂度 n * log n
*/
const int N = 100010, mod = 1e9 + 7;
int fact[N], infact[N];  //阶乘和阶乘逆元 模 mod 的值 预处理
ll qmi(int a, int k, int p) {
    int res = 1;
    while (k) {
        if (k & 1) res = (ll)res * a % p;
        a = (ll)a * a % p;
        k >>= 1;
    }
    return res;
}
int main() {
    int n; int a, b;
    cin >> n;
    fact[0] = infact[0] = 1;
    for (int i = 1; i <= N; i++) { //预处理
        fact[i] = 1ll * i * fact[i - 1] % mod;
        infact[i] = 1ll * infact[i - 1] * qmi(i, mod - 2, mod)
        ↪  % mod;
    }
    while (n -- ) {
        cin >> a >> b;
        cout << (1ll * fact[a] * infact[b]  % mod * infact[a -
        ↪  b]) % mod << endl;
    }
    return 0;
}
/*
lucas 定理 1<=b<=a<=1e18,1<=p<=1e5,p 是质数
c(m,n)mod p, 复杂度 O(p*logm)
可以预处理阶乘和逆元达到 O(p+logm);
*/
int p;
ll qmi(ll a, ll b) {
    ll res = 1; a %= p;
    while (b) {
```

```
        if (b & 1) res = res * a % p;
        b >>= 1;
        a = a * a % p;

    }
    return res;
}
int c(int a, int b) {
    //逆元求组合数
    if (a < b) return 0;
    int down = 1, up = 1;
    for (int i = a, j = 1; j <= b; i--, j++) {
        down = (ll)down * j % p;
        up = (ll)up * i % p;
    }
    return (ll)up * qmi(down, p - 2) % p;
}
ll lucas(ll a, ll b) { //递归求 lucas
    if (a < p && b < p) return c(a, b);
    return c(a % p, b % p) * lucas(a / p, b / p) % p;
}
int main() {
    int n; cin >> n;
    ll a, b;
    while (n--) {
        cin >> a >> b >> p;
        cout << lucas(a, b) << endl;
    }
    return 0;
}
/*
当需要求出组合数的真实值，而非对某个数的余数时，用分解质因数
    1. 筛法求出范围内的所有质数
    2. 通过 C(a, b) = a! / b! / (a - b)! 求出每个质因子的次数
       n! 中 p 的次数是 n / p + n / p^2 + n / p^3 + ...
    3. 用高精度乘法将所有质因子相乘
*/
const int N = 5010;
int primes[N], cnt;
int sum[N];
```

```cpp
bool st[N];
int p;
void get_primes(int n) {
    for (int i = 2; i <= n; i++) {
        if (!st[i]) primes[cnt++] = i;
        for (int j = 0; primes[j] <= n / i; j++) {
            st[primes[j] * i] = 1;
            if (i % primes[j] == 0) break;
        }
    }
}
int get(int n) { //求出 n! 中包含的 p 的个数
    int res = 0;
    while (n) {
        res += n / p;
        n /= p;
    }
    return res;
}
ll qmi(int a, int b) {
    ll res = 1;
    while (b) {
        if (b & 1) res *= a;
        b >>= 1;
        a = a * a;
    }
    return res;
}
//高精度乘法
vector<int> mul(vector<int> &a, int b) {
    vector<int> c;
    int t = 0;
    for (int i = 0; i < a.size(); i++) {
        t += a[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }
    while (t) {
        c.push_back(t % 10);
        t /= 10;
```

```
    }
    return c;
}
int main() {
    int a, b; cin >> a >> b;
    get_primes(a);  //求出 1 ~ a 中所有质数
    for (int i = 0; i < cnt; i++) {
        p = primes[i];
        sum[i] = get(a) - get(b) - get(a - b); //整个组合数计算
        ↪ 除式中的 p 的次数 0 就相当于不减
    }
    vector<int> res;  //高精度乘法
    res.push_back(1);
    for (int i = 0; i < cnt; i++)
        res = mul(res, qmi(primes[i], sum[i]));
    //输出答案
    for (int i = res.size() - 1 ; i >= 0; i--) cout << res[i];
    cout << endl;
    return 0;
}
/*
卡特兰数：Cat(n)=c(2n,n)/(n+1) 对质数取模
给定 n 个 0 和 n 个 1，它们将按照某种顺序排成长度为 2n 的序列，
↪ 求它们能排列成的所有序列中，能够满足任意前缀序列中 0 的个数
↪ 都不少于 1 的个数的序列有多少个。输出的答案对 1e9+7 取模。
*/
const int p = 1e9 + 7;
typedef long long ll;
ll qmi(ll a, ll b, int p) {
    ll res = 1; a %= p;
    while (b) {
        if (b & 1) res = res * a % p;
        b >>= 1;
        a = a * a % p;
    }
    return res;
}
int c(int a, int b) {
    // return res;
```

```
    if (a < b) return 0; //不管为了正确性还是时间来说都请加上这
    ↪ 句话
    int down = 1, up = 1;
    for (int i = a, j = 1; j <= b; i--, j++) {
        down = (ll)down * j % p;
        up = (ll)up * i % p;
    }
    return (ll)up * qmi(down, p - 2) % p;
}
ll lucas(ll a, ll b) { //递归求 lucas
    if (a < p && b < p) return c(a, b);
    return c(a % p, b % p) * lucas(a / p, b / p) % p;
}
int main() {
    int n; cin >> n;
    //除法取模，计算逆元，n+1<p，所以和 p 互质
    cout << lucas(2 * n, n) % p * qmi(n + 1, p - 2, p) % p <<
    ↪ endl;
    return 0;
}
/*
A(0,0) 点出发，只能向正右方或者正上方行走，且不能经过 y=x 左上方
↪ 的点，即任何途径的点 (x,y) 都要满足 x y，问到达 B(n,m) 有多
↪ 少种走法。
*/
/*
找出 (n,m) 关于 y=x+1 的对称点 (m-1,n+1)
每一条非法路径都对应一条 (0,0) 到 (m-1,n+1) 的路径，因此合法路径
↪ = 总路径数-非法路径
答案就是 C(n + m, n) - C(n + m, m - 1)
但是此题不是组合数取模，而且 N 上界 5000，需要分解质因子 + 高精
↪ 度乘法
*/
const int N = 100010; //高精度位数
int primes[N], cnt;
bool st[N]; //筛质数筛到 10000 组合数最大 n+m
int a[N], b[N];
void get_primes(int n) {
    for (int i = 2; i <= n; i++) {
        if (!st[i]) primes[cnt++] = i;
```

```
        for (int j = 0; primes[j] <= n / i; j++) {
            st[primes[j] * i] = 1;
            if (i % primes[j] == 0) break;
        }
    }
}
int get(int n, int p) { //求出 n! 中包含的 p 的个数
    int res = 0;
    while (n) {
        res += n / p;
        n /= p;
    }
    return res;
}
void mul(int r[], int &len, int x) { //高精度乘法
    int t = 0;
    for (int i = 0; i < len; i++) {
        t += r[i] * x;
        r[i] = t % 10;
        t /= 10;

    }
    while (t) r[len++] = t % 10, t /= 10;
}
void sub(int a[], int al, int b[], int bl) { //高精度减法
    for (int i = 0, t = 0; i < al; i++) {
        a[i] -= t + b[i];
        if (a[i] < 0) a[i] += 10, t = 1;
        else t = 0;
    }
}
int C(int x, int y, int r[N]) {
    int len = 1;
    r[0] = 1;
    for (int i = 0; i < cnt; i++) {
        int p = primes[i];
        int s = get(x, p) - get(y, p) - get(x - y, p);
        while (s --) mul(r, len, p); //直接边求边乘
    }
    return len;
```

```
}
int main() {
    get_primes(N - 1);
    int n, m;
    cin >> n >> m;
    int al = C(n + m, n, a); //结果放入 a 中 al 记录高精度结果
    ↪ 的长度 便于减法高精度
    int bl = C(n + m, m - 1, b);
    sub(a, al, b, bl);
    int k = al - 1;
    while (!a[k] && k > 0) k--;
    while (k >= 0) printf("%d", a[k --]);
    return 0;
}
/*
卡特兰数, 对非质数取模 1<=n<=1e6,1<=p<=1e9 模数是 p
分解质因数 + 快速幂取模 + 组合数减法 C(2n,n) - C(2n,n-1)
阶乘分解质因数复杂度 N*loglogN 该算法的复杂度不会超过 O(N)
*/
const int N = 2e6 + 10;
ll p; //模数
int primes[N], cnt;
bool st[N]; int n;
void init(int n) {
    for (int i = 2; i <= n; i++) {
        if (!st[i]) primes[cnt++] = i;
        for (int j = 0; primes[j] <= n / i; j++) {
            st[i * primes[j]] = true;
            if (i % primes[j] == 0)  break;
        }
    }
}
int get(int n, int p) { //求出 n! 中包含的 p 的个数
    int res = 0;
    while (n) {
        res += n / p;
        n /= p;
    }
    return res;
}
```

```
ll C(int x, int y) {
    ll res = 1;
    for (int i = 0; i < cnt; i++) {
        int pi = primes[i];
        int s = get(x, pi) - get(y, pi) - get(x - y, pi);
        while (s --) res  = res * pi % p; //一边求一边乘，不用
        ↪ 写快速幂了
    }
    return res;
}
int main() {
    scanf("%d", &n);
    init(n * 2);
    scanf("%d", &p);
    cout << ((C(n * 2, n) -  C(n * 2, n - 1)) % p + p) % p <<
    ↪ endl;
    return 0;
}
```

5.9 莫比乌斯函数

```
/*
对于给定的整数 a,b 和 d，有多少正整数对 x,y，满足 x<=a，y<=b，并
↪ 且 gcd(x,y)=d
*/
const int N = 50010;
int primes[N], cnt;
bool st[N];
int mobius[N], sum[N];
//线性筛处理 mobius 函数
void init(int n) {
    mobius[1] =  1;//0 个质因子 0 是偶数
    for (int i = 2; i <= n; i++) {
        if (!st[i]) {
            primes[cnt++] = i;
            mobius[i] = -1; //质数本身自身质因子
        }
        for (int j = 0; primes[j] <= n / i; j++) {
            int t = primes[j] * i;
            st[t] = true;
```

```
            if (i % primes[j] ==  0) {
                mobius[t] = 0; //有至少 2 个质因子 primes[j]
                break;
            }
            //primes[j] 是质数，所以 t 的质因子个数取决于 i
            //t 比 i 多了一个 primes[j] 而已
            mobius[t] = mobius[i] * -1;
        }
    }
    //求 mobius 前缀和
    for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] +
    ↪  mobius[i];
}
int main() {
    init(N - 1);
    int T; scanf("%d", &T);
    while (T--) {
        int a, b, d;
        scanf("%d%d%d", &a, &b, &d);
        a /= d, b /= d;
        ll res = 0;
        int n = min(a, b);
        for (int l = 1, r; l <= n; l = r + 1) {
            //l 是每一小段的左端，r 是 g(x) 经典函数
            r = min(n, min(a / (a / l), b / (b / l)));
            res += (sum[r] - sum[l - 1]) * 1ll * (a / l) * (b /
            ↪  l);
        }
        printf("%lld\n", res);
    }
    return 0;
}
```

5.10 矩阵乘法

```
//矩阵乘法模板
const int N = 3;
void mul(int c[], int a[], int b[][N]) {//c=a*b 一维乘二维
    int temp[N] = {0};
    for (int i = 0; i < N; i++) //列
        for (int j = 0; j < N; j++) //行
```

```
            temp[i] = (temp[i] + (ll)a[j] * b[j][i]); //这里可
            ↪ 选是否取模
    memcpy(c, temp, sizeof temp); //这里要写 temp 因为传进来的
    ↪ c 只是一个指针
}
//对于一行的矩阵 * 二维的矩阵
//可以将一行的矩阵初始化为二维的, 除了第一行其余 行为 0 即可
void mul(int c[][N], int a[][N], int b[][N]) { //c = a * b
↪ //二维相乘
    int temp[N][N] = {0};
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                temp[i][j] = (temp[i][j] + (ll)a[i][k] *
                ↪ b[k][j]);//此处可选取模 % m;
    memcpy(c, temp, sizeof temp);
}
/*
例题：f1=1,f2=1,f3=2,f4=3,⋯,fn=fn-1+fn-2。
输入 n 和 m, 求 fn 的前 n 项和
*/
//矩阵乘法复杂度 O(N ^ 3 * log N)
//构造 3x3 矩阵来递推 [f[n], f[n + 1], s[n]] -> [f[n + 1], f[n
↪ + 2], s[n + 1]]
//[0, 1, 0]
//[1, 1, 1]
//[0, 0, 1]
const int N = 3;
int n, m;
void mul(int c[], int a[], int b[][N]) { //c = a * b
    int temp[3] = {0};
    for (int i = 0; i < N; i++) //列
        for (int j = 0; j < N; j++) //行
            temp[i] = (temp[i] + (ll)a[j] * b[j][i]) % m;
    memcpy(c, temp, sizeof temp); //这里要写 temp 因为传进来的
    ↪ c 只是一个指针
}
void mul(int c[][N], int a[][N], int b[][N]) { //c = a * b
↪ //二维相乘
    int temp[N][N] = {0};
```

```
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                temp[i][j] = (temp[i][j] + (ll)a[i][k] *
                ↪  b[k][j]) % m;
    memcpy(c, temp, sizeof temp);
}
int main() {
    cin >> n >> m;
    int f1[3] = {1, 1, 1}; //初始状态 [f[1], f[2], s[1]]
    int a[3][3] = {
        {0, 1, 0},
        {1, 1, 1},
        {0, 0, 1}
    };
    n -- ; //从 1 开始 递推 n - 1 次
    while (n) {
        if (n & 1) mul(f1, f1, a); //res = res * a
        mul(a, a, a); // a = a * a
        n >>= 1;
    }
    //最后递推之后的 f1 就是 [f[n], f[n + 1], s[n]]
    cout << f1[2] << endl;
    return 0;
}
```

5.11 容斥原理

```
/*
给定一个整数 n 和 m 个不同的质数 p1,p2,⋯,pm
求 1~n 中能被 p1,p2,⋯,pm 中的至少一个数整除的整数有多少
*/
//集合 S[i]:1 到 n 中能被 i 整除的数的集合 然后 一共有 m 个集合
↪  处理一个集合需要 m 次 最后复杂度 O(2 ^ m * m)
//容斥原理 + 二进制状压
const int N = 20;
int n, m;
int p[N];
int main() {
    cin >> n >> m;
```

```
    for (int i = 0; i < m; i++) cin >> p[i];
    int res = 0;
    for (int i = 1; i < 1 << m; i++) {
        int t = 1, cnt = 0;
        for (int j = 0; j < m; j++)
            if (i >> j & 1) {
                cnt ++;
                if (1ll * t * p[j] > n) {
                    t = -1;
                    break;
                }
                t *= p[j];
            }
        if (t != -1) {
            if (cnt & 1) res += n / t;
            else res -= n / t;
        }
    }
    cout << res << endl;
    return 0;
}
//多重集组合数质数取模 n 堆取 m 个的不同集合数量
//对于组合数 C(a,b) 计算的次数由 b 大小决定，此题 N 很小，直接用组合数定义计算即可
const int N = 20;
const ll mod = 1e9 + 7;
ll a[N];
int down = 1; //组合数的不变部分
int qmi(int a, int b, int p) {
    int res = 1;
    while (b) {
        if (b & 1) res = 1ll * res * a % p;
        a = 1ll * a * a % p;
        b >>= 1;
    }
    return res;
}
int C(ll a, ll b) { //逆元求组合数
    if (a < b) return 0;
    int up = 1;
```

```
    for (ll i = a; i > a - b; i--) up = 1ll * i % mod * up %
    ↪  mod;
    return 1ll * up * down % mod;
}
int calc(ll n, ll m) { //参数 ll 还是 int 由题而定
    for (int j = 1; j <= n - 1; j++) down = 1ll * j * down %
    ↪  mod;
    down = qmi(down, mod - 2, mod);
    int res = 0; //答案
    //容斥原理一共由 1<<N 项 直接枚举即可
    for (int i = 0; i  < 1 << n; i++) {
        ll aa = m + n - 1, b = n - 1;
        int sign = 1; //项的正负号
        for (int j = 0; j < n; j++)
            if (i >> j & 1) {
                sign *= -1;
                aa -= a[j] + 1;
            }
        res = (res + C(aa, b) * sign) % mod;
    }
    return (res + mod) % mod;
}
int main() {
    ll n, m;
    cin >> n >> m;
    for (int i = 0; i < n; i++) cin >> a[i];
    //n 堆取 m 个组成的不同集合数量
    cout << calc(n , m) << endl;
    return 0;
}
```

5.12 概率期望

```
/*
给出一个有向无环的连通图，起点为 1，终点为 N，每条边都有一个长度。
数据保证从起点出发能到达图中所有的点，所有的点也都能到终点
到达每一个顶点时，如果有 K 条离开该点的道路，绿豆蛙可以选择任意一
↪  条道路离开该点，并且走向每条路的概率为 1/K 。
现在绿豆蛙想知道，从起点走到终点所经过的路径总长度的期望是多少？
输入格式：第一行：两个整数 N，  M，代表图中有 N 个点、M 条边。
```

```
第二行到第 1+M 行：每行 3 个整数 a, b, c, 代表从 a 到 b 有一条长
↪ 度为 c 的有向边。
输出格式
输出从起点到终点路径总长度的期望值，结果四舍五入保留两位小数。
数据范围 1<=N 1e5,1<=M<=2N
*/
const int N = 1e5 + 10, M = 2 * N;
int n, m;
int h[N], e[M], w[M], ne[M], idx;
int dout[N]; //出度
double f[N]; //DP
void add(int a, int b, int c) {
    e[idx] = b, w[idx] = c, ne[idx] = h[a];
    h[a] = idx ++;
}
double dp(int u) {
    if (f[u] >= 0) return f[u]; //记忆化
    if (u == n) return f[u] = 0; //终点
    f[u] = 0; //当前点先初始置为 0 然后再加
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        f[u] += (w[i] + dp(j)) / dout[u];  //递归反向搜
    }
    return f[u];
}
int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    for (int i = 0; i < m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
        dout[a]++;
    }
    memset(f, -1, sizeof f);
    printf("%.2lf\n", dp(1));
    return 0;
}
/*
把一副扑克牌 (54 张) 随机洗开，倒扣着放成一摞。
```

```
然后从上往下依次翻开每张牌，每翻开一张黑桃、红桃、梅花或者方块，
↪ 就把它放到对应花色的堆里去。
问得到 A 张黑桃、B 张红桃、C 张梅花、D 张方块需要翻开的牌的张数的
↪ 期望值 E 是多少？
如果翻开的牌是大王或者小王，将会把它作为某种花色的牌放入对应堆
↪ 中，使得放入之后 E 的值尽可能小。
输入格式：输入仅由一行，包含四个用空格隔开的整数，A,B,C,D。
输出格式
输出需要翻开的牌数的期望值 E，四舍五入保留 3 位小数。
如果不可能达到输入的状态，输出-1.000。
*/
const int N = 14;
const double INF = 1e20;
int A, B, C, D;
double f[N][N][N][N][5][5];
double dp(int a, int b, int c, int d, int x, int y) {
    double &v = f[a][b][c][d][x][y];
    if (v >= 0) return v; //记忆化
    int as = a + (x == 0) + (y == 0);
    int bs = b + (x == 1) + (y == 1);
    int cs = c + (x == 2) + (y == 2);
    int ds = d + (x == 3) + (y == 3);
    if (as >= A && bs >= B && cs >= C && ds >= D) return v =
    ↪ 0; //终点
    if (a > 13 || b > 13 || c > 13 || d > 13) return v = INF;
    ↪ //输入有误数据
    int sum = a + b + c + d + (x != 4) + (y != 4); //当前已经翻
    ↪ 开的牌总数
    v = 1;  //初始化当前状态 开始计算
    sum = 54 - sum; //剩余牌数
    if (sum <= 0) return v = INF;
    if (a < 13) v += (13.0 - a) / sum * dp(a + 1, b, c, d, x,
    ↪ y);
    if (b < 13) v += (13.0 - b) / sum * dp(a, b + 1, c, d, x,
    ↪ y);
    if (c < 13) v += (13.0 - c) / sum * dp(a, b, c + 1, d, x,
    ↪ y);
    if (d < 13) v += (13.0 - d) / sum * dp(a, b, c, d + 1, x,
    ↪ y);
    if (x == 4) {
```

```
        double t = INF;
        for (int i = 0; i < 4; i++) t = min(t, 1.0 / sum *
        ↪  dp(a, b, c, d, i, y));
        v += t;
    }
    if (y == 4) {
        double t = INF;
        for (int i = 0; i < 4; i++) t = min(t, 1.0 / sum *
        ↪  dp(a, b, c, d, x, i));
        v += t;
    }
    return v;
}
int main() {
    cin >> A >> B >> C >> D;
    memset(f, -1, sizeof f);
    double  t = dp(0, 0, 0, 0, 4, 4);
    if (t > INF / 2) t = -1;
    printf("%.3lf\n", t);
    return 0;
}
```

5.13 博弈论

```
/*
NIM 博弈：给定 n 堆石子，两位玩家轮流操作，每次操作可以从任意一堆
↪  石子中拿走任意数量的石子（可以拿完，但不能不拿），最后无法进行
↪  操作的人视为失败。
*/
int n;
int main() {
    cin >> n; int x;
    ll res = 0;
    while (n--) {
        cin >> x;
        res ^= x;
    }
    if (res == 0) cout << "No" << endl;
    else cout << "Yes" << endl;
    return 0;
}
```

```
/*
台阶-NIM 博弈
有一个 n 级台阶的楼梯，每级台阶上都有若干个石子，其中第 i 级台阶
↪ 上有 ai 个石子 (i>=1)。
两位玩家轮流操作，每次操作可以从任意一级台阶上拿若干个石子放到下
↪ 一级台阶中（不能不拿）
已经拿到地面上的石子不能再拿，最后无法进行操作的人视为失败。
问如果两人都采用最优策略，先手是否必胜。
*/
int n;
int main() {
    cin >> n;
    int x;
    ll res = 0;
    for (int i = 1; i <= n; i++) {
        cin >> x;
        if (i & 1) res ^= x;
    }
    if (res == 0) cout << "No" << endl;
    else cout << "Yes" << endl;
    return 0;
}
/*
集合 NIM 博弈：给定 n 堆石子以及一个由 k 个不同正整数构成的数字
↪ 集合 S，有两位玩家轮流操作，每次操作可以从任意一堆石子中拿取
↪ 石子，每次拿取的石子数量必须包含于集合 S，最后无法进行操作的
↪ 人视为失败。问如果两人都采用最优策略，先手是否必胜。
*/
const int N = 110, M = 10010;
//每一堆石子都可以看为一个有向图，然后求初始节点的 SG 函数 每一
↪ 个节点的出边对应能取的所有状态
int n, m;
int s[N], f[M];  //s 表示集合的元素 , f 是 sg 函数值
int sg(int x) {
    if (f[x] != -1) return f[x];
    unordered_set<int> S;
    for (int i = 0; i < m; i++) {
        int sum = s[i];
        if (x >= sum) S.insert(sg(x - sum));
    }
```

```
    for (int i = 0; ; i++) {
        if (!S.count(i)) return f[x] = i;
    }
}
int main() {
    cin >> m;
    for (int i = 0; i < m; i++) cin >> s[i];
    cin >> n;
    memset(f, -1, sizeof f);
    int res = 0;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        res ^= sg(x);//求出所有有向图的 sg 函数然后全部异或
    }
    if (res == 0) cout << "No" << endl;
    else cout << "Yes" << endl;
    return 0;
}
/*
拆分 NIM 博弈：给定 n 堆石子，两位玩家轮流操作，每次操作可以取走
↪ 其中的一堆石子，然后放入两堆规模更小的石子（新堆规模可以为 0，
↪ 且两个新堆的石子总数可以大于取走的那堆石子数），最后无法进行操
↪ 作的人视为失败。
问如果两人都采用最优策略，先手是否必胜。
*/
//石子堆的最大值在减小 所以博弈一定会结束
//每堆石子看坐有向图 然后全部求 SG 函数
const int N = 110;
int f[N];
int sg(int x) {
    if (f[x] != -1) return f[x];
    unordered_set<int> S;
    //枚举可以到达的状态
    for (int i = 0; i < x; i++) //第一堆
        for (int j = 0; j <= i; j++) //第二堆
            S.insert(sg(i) ^ sg(j));
    //mex 函数
    for (int i = 0;; i++)
        if (!S.count(i))
```

```
            return f[x] = i;

}
int main() {
    int n; cin >> n;
    int res = 0;
    memset(f, -1, sizeof f);
    while (n --) {
        int x;
        cin >> x;
        res ^= sg(x);
    }
    if (res == 0) cout << "No" << endl;
    else cout << "Yes" << endl;
    return 0;
}
/*
有向图博弈：给定一个有 N 个节点的有向无环图，图中某些节点上有棋
↪ 子，两名玩家交替移动棋子。
玩家每一步可将任意一颗棋子沿一条有向边移动到另一个点，无法移动者
↪ 输掉游戏。
对于给定的图和棋子初始位置，双方都会采取最优的行动，询问先手必胜
↪ 还是先手必败。
输入格式
第一行，三个整数 N,M,K，N 表示图中节点总数，M 表示图中边的条数，K
↪ 表示棋子的个数。
接下来 M 行，每行两个整数 X,Y 表示有一条边从点 X 出发指向点 Y。
接下来一行， K 个空格间隔的整数，表示初始时，棋子所在的节点编号。
节点编号从 1 到 N。
输出格式：若先手胜，输出 win，否则输出 lose
*/
const int N = 2010, M = 6010;
int n, m, k;
int h[N], e[M], ne[M], idx;
int sg[N];
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
```

```
int SG(int u) {
    if (sg[u] != -1) return sg[u];
    set<int> s;
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        s.insert(SG(j));
    }
    for (int i = 0; ; i++)
        if (!s.count(i)) {
            sg[u] = i;
            break;
        }
    return sg[u];
}
int main() {
    scanf("%d%d%d", &n, &m, &k);
    memset(h, -1, sizeof h);
    memset(sg, -1, sizeof sg);
    for (int i = 0; i < m; i++) {
        int a, b ;
        scanf("%d%d", &a, &b);
        add(a, b);
    }
    int res = 0;
    for (int i = 0; i < k; i++) {
        int u; scanf("%d", &u);
        res ^= SG(u);
    }
    if (res) puts("win");
    else puts("lose");
    return 0;
}
```

6 动态规划

6.1 背包

6.1.1 01 背包

朴素 01 背包

```
/*
有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。
第 i 件物品的体积是 vi 价值是 wi。
求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总
↪ 价值最大。
输出最大价值。复杂度 O(N * V)
*/
const int N = 1010;
int n, m;   //m 表示背包的容积
int v[N], w[N];
int f[N];
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> v[i] >> w[i];
    for (int i = 1; i <= n; i++)
        //这里的 j <= m 就保证了决策的边界 一定不会超出背包容积
        for (int j = m; j >= v[i]; j--) {

            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    cout << f[m] << endl;
    return 0;
}
```

01 背包求方案数

```
/*
有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。
第 i 件物品的体积是 vi，价值是 wi。
求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总
↪ 价值最大。
输出 最优选法的方案数。输出答案模 109+7 的结果。
输入格式
第一行两个整数，N，V，用空格隔开，分别表示物品数量和背包容积。
```

```
接下来有 N 行，每行两个整数 vi,wi，用空格隔开，分别表示第 i 件物品的体积和价值。
输出格式：输出一个整数，表示方案数模 1e9+7 的结果。
数据范围
0 <= N,V <= 1000
0 <= vi,wi <= 1000
*/
const int N = 1010, mod = 1e9 + 7;
int f[N], g[N];  //g 数组记录 f[] 取到最大 时候的方案数
int n, m;
int main() {
    cin >> n >> m;
    memset(f, 0xcf, sizeof f);
    f[0] = 0;
    g[0] = 1;
    for (int i = 0; i < n; i++) {
        int v, w;
        cin >> v >> w;
        for (int j = m; j >= v; j--) {
            int maxv = max(f[j], f[j - v] + w);
            int cnt = 0;
            if (maxv == f[j]) cnt += g[j]; //不选第 i 个物品的最优方案数
            if (maxv == f[j - v] + w) cnt += g[j - v]; //选第 i 个物品的最优方案数
            g[j] = cnt % mod;
            f[j] = maxv;
        }
    }
    int res = 0;
    for (int i = 0; i <= m; i++) res = max(res, f[i]);
    int cnt = 0; //找出最大的方案数 求和
    for (int i = 0; i <= m; i++)
        if (res == f[i]) cnt = (cnt + g[i]) % mod;
    cout << cnt << endl;
    return 0;
}
```

01 背包求具体方案

```
/*
有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。
第 i 件物品的体积是 vi，价值是 wi。
求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出 字典序最小的方案。这里的字典序是指：所选物品的编号所构成的序列。物品的编号范围是 1…N。
输入格式
第一行两个整数，N，V，用空格隔开，分别表示物品数量和背包容积。
接下来有 N 行，每行两个整数 vi,wi，用空格隔开，分别表示第 i 件物品的体积和价值。
输出格式
输出一行，包含若干个用空格隔开的整数，表示最优解中所选物品的编号序列，且该编号序列的字典序最小。
物品编号范围是 1…N。
*/
const int N = 1010;
int n, m;
int v[N], w[N];
int f[N][N];
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) scanf("%d%d", & v[i], &w[i]);
    for (int i = n; i >= 1; i--) {
        for (int j = 0 ; j <= m; j++) { //2 维 循环顺序无所谓 不优化空间
            f[i][j] = f[i + 1][j];
            if (j >= v[i]) f[i][j] = max(f[i][j], f[i + 1][j - v[i]] + w[i]);
        }
    }
    //f[1][m] 是最最大价值
    int j = m;
    for (int i = 1; i <= n; i++)
        if (j >= v[i] &&  f[i][j] == f[i + 1][j - v[i]] + w[i]) { //相等 能选就必选 才使字典序最小
            printf("%d ", i);
            j -= v[i];
        }
    return 0;
```

```
}
```

6.1.2 完全背包

```
/*
有 N 种物品和一个容量是 V 的背包，每种物品都有无限件可用。
第 i 种物品的体积是 vi，价值是 wi。
求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总
↪ 价值最大。输出最大价值。
*/
const int N = 1010;
int v[N], w[N];
int f[N];
int n, m;
int main() { //一维优化
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> v[i] >> w[i];
    for (int i = 1; i <= n; i++)
        for (int j = v[i]; j <= m; j++) {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    cout << f[m] << endl;
    return 0;
}
```

6.1.3 多重背包

```
/*
有 N 种物品和一个容量是 V 的背包。
第 i 种物品最多有 si 件，每件体积是 vi，价值是 wi。
求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总
↪ 和最大。输出最大价值。 O(N^3)
输入格式
第一行两个整数，N，V (0<N 1000，0<V 20000)，用空格隔开，分别表示
↪ 物品种数和背包容积。
接下来有 N 行，每行三个整数 vi,wi,si，用空格隔开，分别表示第 i
↪ 种物品的体积、价值和数量。
*/
const int N = 110;
int v[N], w[N], s[N];
int f[N][N];
```

```
int n, m;
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> v[i] >> w[i] >> s[i];

    for (int i = 1; i <= n; i++)
        for (int j = 0; j <= m; j++)
            for (int k = 0; k <= s[i] && k * v[i] <= j; k++)
                f[i][j] = max(f[i][j], f[i - 1][j - v[i] * k] +
                ↪  k * w[i]);
    cout << f[n][m] << endl;
    return 0;
}
/*
二进制优化多重背包
*/
const int N = 25000;  //数组开 N * log V
int v[N], s[N], w[N];
int f[N];
int n, m;
int main() {
    cin >> n >> m;
    int cnt = 0;
    for (int i = 1; i <= n; i++) {
        int a, b, s;
        cin >> a >> b >> s;
        int k = 1;
        while (k <= s) {
            cnt ++;
            v[cnt] = a * k;
            w[cnt] = b * k;
            s -= k;
            k *= 2;
        }
        if (s > 0) {
            cnt ++;
            v[cnt] = a * s;
            w[cnt] = b * s;
        }
    }
```

```
    n = cnt;
    for (int i = 1; i <= n; i++)
        for (int j = m; j >= v[i]; j --)
            f[j] = max(f[j], f[j - v[i]] + w[i]);
    cout << f[m] << endl;
    return 0;
}
/*
单调队列优化多重背包
*/
const int N = 20010;
int n, m;
int f[N], q[N], g[N];
int main() {
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        int v, w, s;
        cin >> v >> w >> s;
        memcpy(g, f, sizeof f);
        for (int j = 0; j < v; j++) {
            int hh = 0, tt = -1;  //单调队列
            for (int k = j; k <= m; k += v) {
                if (hh <= tt && q[hh] < k - s * v) hh++; //出队
                if (hh <= tt) f[k] = max(f[k], g[q[hh]] + (k -
                ↪  q[hh]) / v * w);
                while (hh <= tt && g[q[tt]] - (q[tt] - j) / v *
                ↪  w <= g[k] - (k - j) / v * w) tt--;
                q[++tt] = k;
            }
        }
    }
    cout << f[m] << endl;
    return 0;
}
```

6.1.4 分组背包

```
/*
有 N 组物品和一个容量是 V 的背包。
每组物品有若干个，同一组内的物品最多只能选一个。
每件物品的体积是 vij，价值是 wij，其中 i 是组号，j 是组内编号。
```

```
求将哪些物品装入背包可使物品总体积不超过背包容量，且总价值最大
输出最大价值。
输入格式
第一行有两个整数 N, V, 用空格隔开，分别表示物品组数和背包容量。
接下来有 N 组数据：
每组数据第一行有一个整数 Si, 表示第 i 个物品组的物品数量；
每组数据接下来有 Si 行，每行有两个整数 vij,wij, 用空格隔开，分别
↪ 表示第 i 个物品组的第 j 个物品的体积和价值
*/
const int N = 110;
int v[N][N], w[N][N], s[N];
int n, m;
int f[N];
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        cin >> s[i];
        for (int j = 0; j < s[i]; j++)
            cin >> v[i][j] >> w[i][j];
    }
    for (int i = 1; i <= n; i++)
        for (int j = m ; j >= 0; j--)
            for (int k = 0; k < s[i]; k++)
                if (j >= v[i][k])
                    f[j] = max(f[j], f[j - v[i][k]] + w[i][k]);
    cout << f[m] << endl;
    return 0;
}
```

6.1.5 混合背包

```
/*
有 N 种物品和一个容量是 V 的背包。
物品一共有三类：
第一类物品只能用 1 次（01 背包）
第二类物品可以用无限次（完全背包）
第三类物品最多只能用 si 次（多重背包）
每种体积是 vi, 价值是 wi。
求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总
↪ 和最大。输出最大价值。
输入格式
```

```
第一行两个整数，N，V，用空格隔开，分别表示物品种数和背包容积。
接下来有 N 行，每行三个整数 vi,wi,si，用空格隔开，分别表示第 i 种物品的体积、价值和数量。
si=-1 表示第 i 种物品只能用 1 次；
si=0 表示第 i 种物品可以用无限次；
si>0 表示第 i 种物品可以使用 si 次；
输出格式
输出一个整数，表示最大价值。
*/
const int N = 1010;
int n, m;
int f[N];
//注意没有分组背包 只有其余 3 种背包混合
int main() {
    scanf("%d%d", & n, & m);
    for (int i = 0; i < n; i++) {
        int v, w, s;
        scanf("%d%d%d", &v, &w, &s);
        if (s == 0) { //完全背包
            for (int j = v; j <= m; j++) f[j] = max(f[j], f[j - v] + w);
        } else {
            if (s == -1) s = 1; //01 背包是多重背包的特殊情况 直接合并一起写
            for (int k = 1; k <= s; k *= 2) { //把 si 次 等价于 si 个同样的物品 然后 01 背包 每次倍增
                for (int j = m; j >= k * v; j--)
                    f[j] = max(f[j], f[j - k * v] + k * w);
                s -= k;
            }
            if (s) { //二进制打包之后 剩余的物品 收尾
                for (int j = m; j >= s * v; j --)
                    f[j] = max(f[j], f[j - s * v] + s * w);
            }
        }
    }
    cout << f[m] << endl;
    return 0;
}
```

6.1.6 二维费用背包

```
/*
有 N 件物品和一个容量是 V 的背包，背包能承受的最大重量是 M。
每件物品只能用一次。体积是 vi，重量是 mi，价值是 wi。
求解将哪些物品装入背包，可使物品总体积不超过背包容量，总重量不超过背包可承受的最大重量，且价值总和最大。
输出最大价值。
输入格式
第一行两个整数，N，V,M，用空格隔开，分别表示物品件数、背包容积和背包可承受的最大重量。
接下来有 N 行，每行三个整数 vi,mi,wi，用空格隔开，分别表示第 i 件物品的体积、重量和价值。
输出格式
输出一个整数，表示最大价值。
*/
int N, V, M;
const int P = 110;
int f[P][P];
int main() {
    cin >> N >> V >> M;
    for (int i = 0 ; i < N; i++) {
        int v, m, w;
        cin >> v >> m >> w;
        for (int j = V; j >= v; j--)
            for (int k = M; k >= m; k--)
                f[j][k] = max(f[j][k], f[j - v][k - m] + w);
    }
    cout << f[V][M] << endl;
    return 0;
}
```

6.1.7 有依赖的背包

```
/*
有 N 个物品和一个容量是 V 的背包。
物品之间具有依赖关系，且依赖关系组成一棵树的形状。如果选择一个物品，则必须选择它的父节点。
每件物品的编号是 i，体积是 vi，价值是 wi，依赖的父节点编号是 pi。物品的下标范围是 1…N。
```

```
求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最
↪ 大。输出最大价值。
输入格式
第一行有两个整数 N, V, 用空格隔开，分别表示物品个数和背包容量。
接下来有 N 行数据，每行数据表示一个物品。
第 i 行有三个整数 vi,wi,pi, 用空格隔开，分别表示物品的体积、价值
↪ 和依赖的物品编号。
如果 pi=-1, 表示根节点。数据保证所有物品构成一棵树。
*/
const int N = 110;
int h[N], ne[N], e[N];
int idx;
int v[N], w[N];
int n, m;
int f[N][N];
void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++;
}
void dfs(int u) { //遍历 u 为根的子树
    for (int i = h[u]; ~i; i = ne[i]) {
        int son = e[i];
        dfs(e[i]);
        // 分组背包
        for (int j = m - v[u]; j >= 0; j--) //循环体积 分组
            for (int k = 0; k <= j; k++) //决策
                f[u][j] = max(f[u][j], f[u][j - k] +
                ↪ f[son][k]);
    }
    //将 u 加进去
    for (int i = m; i >= v[u]; i--) f[u][i] = f[u][i - v[u]] +
    ↪ w[u];
    for (int i = 0; i < v[u]; i++) f[u][i] = 0;
}
//f[u][j] 表示 u 为 root 的子树中选 总体积不超过 j 的方案
int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1 , sizeof h);
    int root;
```

```
    for (int i = 1; i <= n; i++) {
        int p;
        scanf("%d%d%d", &v[i], &w[i], &p);
        if (p == -1) root = i;
        else add(p, i); //有向边

    }
    dfs(root);
    printf("%d\n", f[root][m]) ;
    return 0;
}
```

6.2 线性 DP

6.2.1 LIS/Dilworth 定理

```
/*
给定一个长为 N 的数列，求数值严格单递的子序列的长度最长是多少
输入格式
第一行包含整数 N。 1<=N<=1000, -1e9<= 数列中的数 <=1e9
第二行包含 N 个整数，表示完整序列。
输出格式
输出一个整数，表示最大长度。暴力 O(n^2)
*/
const int N = 1010;
int a[N], f[N];
int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> a[i];
    for (int i = 1; i <= n; i++) {
        f[i] = 1;
        for (int j = 1; j < i; j++)
            if (a[j] < a[i]) f[i] = max(f[i], f[j] + 1);
    }
    int res = 0;
    for (int i = 1; i <= n; i++) res = max(res, f[i]);
    cout  << res << endl;
    return 0;
}
/*
```

```
O(n * logn) 做法 1<=N<=100000
*/
const int N = 1e5 + 10;
int n, a[N];
int q[N];  //储存上升子序列 结尾的最小值
int main() {
    cin >> n;
    for (int i = 0; i < n; i++) cin >> a[i];
    int len = 0;  //q 元素个数
    q[0] = -2e9;  //边界
    for (int i = 0 ; i < n; i++) {
        int l = 0, r = len;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (q[mid] < a[i]) l = mid;
            else r = mid - 1;
        }
        len = max(len, r + 1);
        q[r + 1] = a[i];
    }
    cout << len << endl;
    return 0;
}
/*
Dilworth 定理 例题：拦截导弹
输入格式：共一行，输入导弹依次飞来的高度。
输出格式
第一行包含一个整数，表示最多能拦截的导弹数。
第二行包含一个整数，表示要拦截所有导弹最少要配备的系统数。
*/
const int N = 1010;
int f[N];
int rf[N];
vector<int> a;
/*
Dilworth 定理：
    “能覆盖整个序列的最少的不上升子序列的个数”等价于“该序列的
↪   最长上升子序列长度”
    同理即有：
```

```
    “能覆盖整个序列的最少的不下降子序列的个数”等价于“该序列的
↪   最长下降子序列长度”
*/
int main() {
    int x;
    while (cin >> x) {
        a.push_back(x);
    }
    for (int i = a.size() - 1; i >= 0; i--) {
        f[i] = 1;
        for (int j = a.size() - 1; j > i; j--)
            if (a[j] <= a[i]) f[i] = max(f[i], f[j] + 1); //注
            ↪  意相等也可以拦截
    }
    int res1 = 0;
    for (int i = 0; i < a.size(); i++) res1 = max(res1, f[i]);
    int res2 = 0;
    for (int i = 0; i < a.size(); i++) {
        rf[i] = 1;
        for (int j = 0; j < i; j++)
            if (a[j] < a[i]) rf[i] = max(rf[i], rf[j] + 1);

    }
    for (int i = 0; i < a.size(); i++) res2 = max(res2,
    ↪  rf[i]);
    cout << res1 << endl;
    cout << res2 << endl;
    return 0;
}
/* 最大上升子序列和
一个数的序列 bi，当 b1<b2<…<bS 的时候，我们称这个序列是上升的。
对于给定的一个序列 (a1,a2,…,aN)，我们可以得到一些上升的子序列
↪  (ai1,ai2,…,aiK)，这里 1 i1<i2<…<iK N。
比如，对于序列 (1,7,3,5,9,4,8)，有它的一些上升子序列，如
↪  (1,7),(3,4,8) 等等。
这些子序列中和最大为 18，为子序列 (1,3,5,9) 的和。
你的任务，就是对于给定的序列，求出最大上升子序列和。
注意，最长的上升子序列的和不一定是最大的，比如序列 (100,1,2,3)
↪  的最大上升子序列和为 100，而最长上升子序列为 (1,2,3)。
输入格式：输入的第一行是序列的长度 N。
```

```
第二行给出序列中的 N 个整数，这些整数的取值范围都在 0 到
↪ 10000(可能重复)。
输出格式：输出一个整数，表示最大上升子序列和。
*/
typedef pair<int, int> pii;
const int N = 1010;
vector<pii> a;
//用第一关键字保存元素的值
//第二关键字维护上升子序列的最大值
int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        a.push_back({x, 0});
    }
    for (int i = 0 ; i < n; i++) {
        a[i].second = a[i].first;
        for (int j = 0 ; j < i; j++)
            if (a[j].first < a[i].first) a[i].second =
            ↪ max(a[i].second, a[j].second + a[i].first);
    }
    int res = 0;
    for (int i = 0; i < n; i++) res = max(res, a[i].second);
    cout << res << endl;
    return 0;
}
```

6.2.2 LCS

```
/*
给定两个长度分别为 N 和 M 的字符串 A 和 B，求既是 A 的子序列又是
↪ B 的子序列的字符串长度最长是多少。
输入格式
第一行包含两个整数 N 和 M。
第二行包含一个长度为 N 的字符串，表示字符串 A。
第三行包含一个长度为 M 的字符串，表示字符串 B。
字符串均由小写字母构成。
输出格式：输出一个整数，表示最大长度。
数据范围 1<=N,M<=1000
```

```
*/
const int N = 1010;
int n, m;
char a[N], b[N];
int f[N][N];
int main() {
    cin >> n >> m;
    cin >> a + 1 >> b + 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) {
            f[i][j] = max(f[i - 1][j], f[i][j - 1]);
            if (a[i] == b[j]) f[i][j] = max(f[i][j], f[i - 1][j
            ↪  - 1] + 1);
        }
    cout << f[n][m] << endl;
    return 0;
}
```

6.3 区间 DP

```
/*
设有 N 堆石子排成一排，其编号为 1，2，3，…，N。
每堆石子有一定的质量，可以用一个整数来描述，现在要将这 N 堆石子合
↪  并成为一堆。
每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后
↪  与这两堆石子相邻的石子将和新堆相邻，合并时由于选择的顺序不同，
↪  合并的总代价也不相同。
问题是：找出一种合理的方法，使总的代价最小，输出最小代价。
输入格式：第一行一个数 N 表示石子的堆数 N。
第二行 N 个数，表示每堆石子的质量（均不超过 1000）
*/
const int N = 1010;
int a[N], f[N][N];
int sum[N];
int main() {
    memset(f, 0x3f, sizeof f);
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> a[i];
    for (int i = 1; i <= n; i++) { //前缀和
```

```
        f[i][i] = 0;
        sum[i] = sum[i - 1] + a[i];
    }
    for (int len = 2; len <= n; len++) //枚举阶段
        for (int l = 1; l + len - 1 <= n;  l++) {
            int r = l + len - 1;
            for (int k = l; k < r; k++)
                f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r]);
                ↪ //只需要枚举 k 的位置 然后求出最小的分割
            f[l][r] += sum[r] - sum[l - 1]; //最后加上的前缀和
            ↪ 都是一样的 所以没必要在枚举 k 的时候加上
        }
    cout << f[1][n] << endl;
    return 0;
}
/* 环形石子合并
将 n 堆石子绕圆形操场排放，现要将石子有序地合并成一堆。
规定每次只能选相邻的两堆合并成新的一堆，并将新的一堆的石子数记做
↪ 该次合并的得分。
请编写一个程序，读入堆数 n 及每堆的石子数，并进行如下计算：
选择一种合并石子的方案，使得做 n-1 次合并得分总和最大。
选择一种合并石子的方案，使得做 n-1 次合并得分总和最小。
输入格式：第一行包含整数 n，表示共有 n 堆石子。
第二行包含 n 个整数，分别表示每堆石子的数量。
输出格式 输出共两行：
第一行为合并得分总和最小值，
第二行为合并得分总和最大值。
*/
const int N = 410; //开 2 倍的链长度
const int INF = 0x3f3f3f3f;
int n;
int s[N], w[N];
int f[N][N];  //最小值  f[i][j] 表示合并区间 i~j 所需要的最小值
int g[N][N];  // 最大值
//朴素的石子合并 一排的 复杂度是 n^3
//环形的思路是枚举环上的缺口 然后再用朴素 这样本质是求 n-1 个 长
↪ 度为 n 的链合并 复杂度 n^4 要优化
//优化 直接把环形展开 复制一份 2 个拼起来 一起 然后在 2n 长度区
↪ 间 求区间长度为 n 的区间即可
int main() {
```

```
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> w[i];
        w[i + n] = w[i];
    }
    //前缀和
    for (int i = 1; i <= 2 * n; i++) s[i] = s[i - 1] + w[i];
    memset(f, 0x3f, sizeof f);
    memset(g, 0xcf, sizeof g);
    for (int len = 1; len <= n; len ++) //区间长度
        for (int l = 1; l + len - 1 <= n * 2; l++) { //左端点
            int r = l + len - 1; //右端点
            if (len == 1) f[l][r] = g[l][r] = 0;
            else {
                for (int k = l; k < r; k++) { //分界点
                    f[l][r] = min(f[l][r], f[l][k] + f[k +
                    ↪  1][r] + s[r] - s[l - 1]);
                    g[l][r] = max(g[l][r], g[l][k] + g[k +
                    ↪  1][r] + s[r] - s[l - 1]);
                }
            }
        }
    int minv = INF, maxv = -INF;
    //由于是 2 倍的长度 所以要遍历一次 求出最值
    for (int i = 1; i <= n; i++) {
        minv = min(minv, f[i][i + n - 1]);
        maxv = max(maxv, g[i][i + n - 1]);
    }
    cout << minv << endl << maxv << endl;
    return 0;
}
```

6.4 计数 DP

```
/*
一个正整数 n 可以表示成若干个正整数之和，形如：n=n1+n2+…+nk，其
↪  中 n1 n2 … nk,k 1。
我们将这样的一种表示称为正整数 n 的一种划分。
现在给定一个正整数 n，请你求出 n 共有多少种不同的划分方法。
输入格式：共一行，包含一个整数 n。
```

```
输出格式
共一行，包含一个整数，表示总划分数量。
由于答案可能很大，输出结果请对 109+7 取模。
*/
const int N = 1010, M = 1e9 + 7;
//整数划分 第二种 思路；
//f[i][j] 表示总和是 i 用了 j 个数加起来得到了 N 的所有方案数
//分为两类 一种是 这 j 个数 中最小值是 1，另外一种是 最小值大于
↪ 1
// f[i][j] = f[i - 1][j-1] + f[i - j][j]  第一项是把开头的 1
↪ 去掉 第二项是 j 个数 每个都减去 1
//最后的答案 ans = f[n][1] + f[n][2] + ..... f[n][n]
int n;
int f[N][N];
int main() {
    cin >> n;
    f[0][0] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++)
            f[i][j] = (f[i - 1][j - 1] + f[i - j][j]) % M;
    int res = 0;
    for (int i = 1; i <= n; i++) res = (res + f[n][i]) % M;
    cout << res  << endl;
    return 0;
}
```

6.5 数位 DP

```
/*
给定两个整数 a 和 b，求 a 和 b 之间的所有数字中 0~9 的出现次数。
输入格式：输入包含多组测试数据。
每组测试数据占一行，包含两个整数 a 和 b。
当读入一行为 0 0 时，表示输入终止，且该行不作处理。
输出格式
每组数据输出一个结果，每个结果占一行。
每个结果包含十个用空格隔开的数字，第一个数字表示‘0’出现的次数，
↪ 第二个数字表示‘1’出现的次数，以此类推。
*/
//计算每个数字 在所有数字的每一位上出现的次数 然后累加起来即可
int a, b;
```

```
int get(vector<int> num, int l, int r) { //求前面这些位数组成的
↪  数字 是多大 下标从右到左递增
    int res = 0;
    for (int i = l; i >= r; i--)
        res = res * 10 + num[i];
    return res;
}
int power(int i) { //计算 10 ^ i
    int res = 1;
    while (i --) res *= 10;
    return res ;
}
int count(int n, int x) { //统计 1 ~ n 里面有多少个 x 题目的 a
↪  ,b 范围都是正整数 注意
    if (!n) return 0;
    vector<int> num;
    while (n) {
        num.push_back(n % 10);
        n /= 10;
    }
    n = num.size(); //数字 n 的位数
    //假设数字为 abc d efg 分为 d 前面的 <abc 以及等于 abc 讨论
    int res = 0;
    for (int i = n - 1 - !x; i >= 0; i --) { //从最高位枚举
        if (i < n - 1) {
            res += get(num, n - 1, i + 1) * power(i); //计算 i
            ↪  前面的数字  注意 vector 存放的数字是反的
            if (x == 0) res -= 1 * power(i); //当 d 是 0 的时候
            ↪  则 abc 最小为 001  因为不能有前导 0 不合法 减去
            ↪  这种不合法的数量
        }
        if (num[i] == x) res += get(num, i - 1, 0) + 1; // d
        ↪  == x
        else if (num[i] > x) res += power(i); //d > x
    }
    return res;
}
int main() {
    while (cin >> a >> b && (a || b)) {
        if (a > b) swap(a, b);
```

```
        for (int i = 0; i <= 9; i++)
            cout << count(b, i) - count(a - 1, i) << ' ';
        cout << endl;
    }
    return 0;
}
```

6.6 状压 DP

```
/*
给定一张 n 个点的带权无向图，点从 0~n-1 标号，求起点 0 到终点
↪ n-1 的最短 Hamilton 路径。 Hamilton 路径的定义是从 0 到 n-1
↪ 不重不漏地经过每个点恰好一次。
输入格式：第一行输入整数 n。
接下来 n 行每行 n 个整数，其中第 i 行第 j 个整数表示点 i 到 j 的
↪ 距离（记为 a[i,j]）。
对于任意的 x,y,z，数据保证 a[x,x]=0，a[x,y]=a[y,x] 并且
↪ a[x,y]+a[y,z]>=a[x,z]。
输出格式
输出一个整数，表示最短 Hamilton 路径的长度。
*/
const int N = 20, M = 1 << N ;
int w[N][N];
int f[M][N];  // f[i][j] 表示所有从 0 走到 j 走过的所有点是二进
↪ 制表示的 i 的路径
int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> w[i][j];
    memset(f, 0x3f, sizeof f);
    f[1][0] = 0; //最初是第 0 位 第 0 位的状态就是 1
    for (int i = 0 ; i < 1 << n; i++)
        for (int j = 0; j < n; j++)
            if (i >> j & 1)
                for (int k = 0; k < n; k++) //枚举转移的点 也就
                ↪ 是倒数第二个点
                    if ((i - (1 << j)) >> k & 1) //包含第 k 位
```

```
                        f[i][j] = min(f[i][j], f[i - (1 <<
                        ↪  j)][k] + w[k][j]);
    cout << f[(1 << n) - 1][n - 1] << endl;
    return 0;
}
```

6.7 树形 DP

```
/*
给定一棵树，树中包含 n 个结点（编号 1~n）和 n-1 条无向边，每条边
↪  都有一个权值。
现在请你找到树中的一条最长路径。
换句话说，要找到一条路径，使得使得路径两端的点的距离最远。
注意：路径中可以只包含一个点。
输入格式：第一行包含整数 n。
接下来 n-1 行，每行包含三个整数 ai,bi,ci，表示点 ai 和 bi 之间存
↪  在一条权值为 ci 的边。
输出格式：输出一个整数，表示树的最长路径的长度。
*/
int n;
const int N = 10010;
const int M = 2 * N; //边数 无向边 两遍加
//最长路径 也叫树的直径
//做法：遍历每一个点 然后求出穿过这个点的所有路径 中最长的
//具体求法：求出距离某节点最远的 2 个叶节点 然后加起来 不断更新
int h[N], ne[M], e[M], w[M], idx;
int ans;
void add(int a, int b, int x) {
    e[idx] = b;
    w[idx] = x;
    ne[idx] = h[a];
    h[a] = idx ++;
}
int dfs(int u, int father) {
    int dist = 0;  //表示当前点往下走的最大长度
    //求出所有挂在这个 root 上的最长和第二长的距离
    int d1 = 0, d2 = 0;
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        if (j == father) continue;
```

```
        int d = dfs(j, u) + w[i]; //u 是 j 的父节点
        dist = max(dist, d); //用子节点更新 dist
        if (d >= d1) d2 = d1, d1 = d; //比最大值大 那就把最大值
        ↪ 换位第二大值
        else if (d > d2) d2 = d; //比最大的小 但是比第二大的大
        ↪ 就换位第二大的值
    }
    ans = max(ans, d1 + d2);
    return dist;
}
int main() {
    memset(h, -1 , sizeof h);
    cin >> n;
    for (int i = 0 ; i < n - 1; i++) { //共有 n-1 条边
        int a, b, x;
        cin >> a >> b >> x;
        add(a, b , x);
        add(b, a, x);
    }
    dfs(1, -1);  //第一个参数是根节点 由于是连通图 任意一个点都
    ↪ 可以为 root 这里取 1 号随意
    //dfs 的第二参数是父节点, 是为了让遍历的顺序 从上往下 根节点
    ↪ 设为-1 即可 一个不存咋的数
    cout << ans << endl;
    return 0;
}
```

6.8 单调队列优化 DP

```
/*
输入一个长度为 n 的整数序列, 从中找出一段长度不超过 m 的连续子序
↪ 列, 使得子序列中所有数的和最大。
注意：子序列的长度至少是 1。
输入格式: 第一行输入两个整数 n,m。
第二行输入 n 个数, 代表长度为 n 的整数序列。
同一行数之间用空格隔开。
输出格式
输出一个整数, 代表该序列的最大子序和。
*/
//滑动窗口
```

```cpp
const int N = 300010;
int q[N];  //单调队列下标
ll s[N]; //前缀和
int n, m;
//对于单调队列的下标问题：
//如果是从 0 开始写的代码，那么 i 就是窗口的最后一个元素 则判断条件是 i - q[hh] + 1 > m
//就要 hh++ 对于本题 下标从 1 开始，窗口的结尾元素下标是 i - 1，而 i 是窗口后的一个元素
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++ ) {
        cin >> s[i];
        s[i] += s[i - 1];
    }
    int hh = 1, tt = 1;  //队头 队尾
    ll res = INT_MIN;
    for (int i = 1; i <= n; i++) {
        if (i - q[hh] > m) hh++;
        res = max(res, s[i] - s[q[hh]]);
        while (hh <= tt && s[q[tt]] >= s[i]) tt--;
        q[++tt] = i; //队尾下标
    }
    cout << res << endl;
    return 0;
}
/*
烽火台是重要的军事防御设施，一般建在交通要道或险要处。
一旦有军情发生，则白天用浓烟，晚上有火光传递军情。
在某两个城市之间有 n 座烽火台，每个烽火台发出信号都有一定的代价。
为了使情报准确传递，在连续 m 个烽火台中至少要有一个发出信号。
现在输入 n,m 和每个烽火台的代价，请计算在两城市之间准确传递情报所需花费的总代价最少为多少。
输入格式：第一行是两个整数 n,m，具体含义见题目描述；
第二行 n 个整数表示每个烽火台的代价 ai
输出格式：输出仅一个整数，表示最小代价
*/
const int N = 200010;
int w[N], f[N], q[N];
int n, m;
```

```
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> w[i];
    int hh = 0, tt = 0;
    f[0] = 0;
    for (int i = 1; i <= n; i++) {
        if (q[hh] < i - m) hh ++;
        f[i] = f[q[hh]] + w[i];
        while (hh <= tt && f[q[tt]] > f[i]) tt--;
        q[++tt] = i;
    }
    int res = 1e9;
    for (int i = n - m + 1; i <= n; i++) res = min(res, f[i]);
    cout << res << endl;
    return 0;
}
```

6.9 斜率优化 DP

```
/*
有 N 个任务排成一个序列在一台机器上等待执行，它们的顺序不得改变。
机器会把这 N 个任务分成若干批，每一批包含连续的若干个任务。
从时刻 0 开始，任务被分批加工，执行第 i 个任务所需的时间是 Ti。
另外，在每批任务开始前，机器需要 S 的启动时间，故执行一批任务所需
↪ 的时间是启动时间 S 加上每个任务所需时间之和。
一个任务执行后，将在机器中稍作等待，直至该批任务全部执行完毕。
也就是说，同一批任务将在同一时刻完成。
每个任务的费用是它的完成时刻乘以一个费用系数 Ci。
为机器规划一个分组方案，使得总费用最小。
输入格式：第一行包含整数 N，第二行包含整数 S。
接下来 N 行每行有一对整数，分别为 Ti 和 Ci，表示第 i 个任务单独
↪ 完成所需的时间 Ti 及其费用系数 Ci。
输出格式：输出一个整数，表示最小总费用
*/
/*
    数据范围加大 不能朴素 N ^ 2
    需要斜率优化 也叫凸包优化
    查找凸包的下界时候可以直接二分找出大于等于当前斜率的第一个点
    但是此题特殊 直线的斜率为正 且点的横坐标递增 且含有前缀和
    所以斜率也是单调递增的 所以可以线性求出 总体复杂度 O(N)
```

```
    在加点的过程中同时维持凸包 用队列维护
    队头所有斜率小于当前斜率的点删掉 队尾不在凸包上的点删掉 其实
↪   就是格雷厄姆算法
*/
//f[i] = min{f[j] + sumt[i]*(sumc[i] - sumc[j]) + s *
↪   (sumc[n] - sumc[j])}
//移项得到 f[j] = (sumt[i] + s) * sumc[j] + f[i] - sumt[i] *
↪   sumc[i] - s * sumc[n]
// f[j] 就是因变量 y, sumc[j] 就是自变量 x  斜率是 sumt[i] + s
//当 i 固定的时候 截距是 f[i]  - sumt[i] * sumc[i] - s *
↪   sumc[n]
//目的就是让截距中的 f[i] 最小
const int N = 300010;
int n, s;
ll c[N], t[N];
int q[N];
ll f[N];
int main() {
    cin >> n >> s;
    for (int i = 1; i <= n; i++) {
        cin >> t[i] >> c[i];
        t[i] += t[i - 1];
        c[i] += c[i - 1];
    }
    f[0] = 0;
    int hh = 0, tt = 0;
    q[0] = 0;
    for (int i = 1; i <= n; i++) {
        //队列中至少要 2 个元素 所以是 hh < tt  维护队头
        while (hh < tt && (f[q[hh + 1]] - f[q[hh]]) <= (t[i] +
        ↪   s) * (c[q[hh + 1]] - c[q[hh]])) hh++;
        int j = q[hh];
        f[i] = f[j] - (t[i] + s) * c[j] + t[i] * c[i] + s *
        ↪   c[n];
        //维护队尾 队尾最后 2 个点的斜率大于当前点和队尾点的斜率
        ↪   那么就队尾弹出 然后加入新的 维护凸包
        while (hh < tt && (f[q[tt]] - f[q[tt - 1]]) * (c[i] -
        ↪   c[q[tt]]) >= (f[i] - f[q[tt]]) * (c[q[tt]] - c[q[tt
        ↪   - 1]])) tt--;
        q[++tt] = i;
```

```
    }
    cout << f[n] << endl;
    return 0;
}
```

7 字符串

7.1 KMP

```
// s[] 是长文本，p[] 是模式串，n 是 s 的长度，m 是 p 的长度
//求短模式串的 Next 数组：
for (int i = 2, j = 0; i <= m; i ++ ) {
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j ++ ;
    ne[i] = j;
}
// 匹配
for (int i = 1, j = 0; i <= n; i ++ ) {
    while (j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j ++ ;
    if (j == m)
    {
        j = ne[j];
        // 匹配成功后的逻辑
    }
}

/*
例题：给定一个模式串 S，以及一个模板串 P，所有字符串中只包含大小
↪ 写英文字母以及阿拉伯数字。
模板串 P 在模式串 S 中多次作为子串出现。
求出模板串 P 在模式串 S 中所有出现的位置的起始下标。
第一行输入整数 N，表示字符串 P 的长度。第二行输入字符串 P。
第三行输入整数 M，表示字符串 S 的长度。第四行输入字符串 S。
输出：共一行，输出所有出现位置的起始下标（下标从 0 开始计数），整
↪ 数之间用空格隔开。
1 <= N < 1e5  1 <= M < 1e6
*/
const int N = 100010, M = 1000010;
int n, m;
char p[N], s[M];
int ne[N];  // Next 数组是对短串而言的
int main() {
    cin >> n >> p + 1 >> m >> s + 1;
    for (int i = 2, j = 0; i <= n; i++) {
```

```
        while (j && p[i] != p[j + 1]) j = ne[j];
        if (p[i] == p[j + 1]) j++;
        ne[i] = j;
    }
    for (int i = 1, j = 0; i <= m; i++) {
        while (j && s[i] != p[j + 1]) j = ne[j];
        if (s[i] == p[j + 1]) j++;
        if (j == n) {
            cout << i - n << ' ';
            j = ne[j];
        }
    }
    return 0;
}
```

7.2 字符串哈希

```
/*
核心思想：将字符串看成 P 进制数，P 的经验值是 131 或 13331，取这两个值的冲突概率低
技巧：取模的数用 2^64，这样直接用 unsigned long long 存储，溢出的结果就是取模的结果
*/
typedef unsigned long long ULL;
ULL h[N], p[N]; // h[k] 存储字符串前 k 个字母的哈希值，p[k] 存储 P^k mod 2^64
// 初始化
p[0] = 1;
for (int i = 1; i <= n; i ++ ) {
    h[i] = h[i - 1] * P + str[i];
    p[i] = p[i - 1] * P;
}
// 计算子串 str[l ~ r] 的哈希值
ULL get(int l, int r) {
    return h[r] - h[l - 1] * p[r - l + 1];
}
/*
例题：给定一个长度为 n 的字符串，再给定 m 个询问，每个询问包含四个整数 l1,r1,l2,r2，请你判断 [l1,r1] 和 [l2,r2] 这两个区间所包含的字符串子串是否完全相同。
```

```
串中只包含大小写英文字母和数字
第一行整数 n 和 m，表示字符串长度和询问次数。
第二行一个长度为 n 的字符串，串中只包含大小写英文字母和数字
接下来 m 行，每行包含四个整数 l1,r1,l2,r2，表示一次询问所涉及的两个区间。
注意，字符串的位置从 1 开始编号。 1 <= n,m <= 1e5
*/
typedef unsigned long long ull;
const int P = 131;
const int N = 100010;
ull p[N], h[N]; char s[N];
ull get(int l, int r) {
    return h[r] - h[l - 1] *  p[r - l + 1];
}
int main() {
    int n, m; p[0] = 1;
    cin >> n >> m >> s + 1;
    for (int i = 1; i <= n; i++) { //预处理 o(N)
        h[i] = h[i - 1] * P + s[i] ; //char 类型计算直接 ascii 码值
        p[i] = P * p[i - 1];
    }
    int l1, r1, l2, r2;
    while (m--) {
        cin >> l1 >> r1 >> l2 >> r2;
        if (get(l1, r1) == get(l2, r2))
            cout << "Yes" << endl;
        else cout << "No" << endl;
    }
    return 0;
}
```

7.3 后缀自动机

```
/*
 * 1 call init()
 * 2 call add(x) to add every character in order
 *
 * Args:
 * Return:
```

```
 *   an automaton
 *   link: link path pointer
 *    len: maximum length
 */
struct node {
  node* chd[26], *link;
  int len;
} a[3 * N], *head, *last;
int top;
void init()
{
  memset(a, 0, sizeof(a));
  top = 0;
  head = last = &a[0];
}
void add(int x)
{
  node *p = &a[++top], *mid;
  p->len = last->len + 1;
  mid = last, last = p;
  for (; mid && !mid->chd[x]; mid = mid->link)  mid->chd[x] =
  ↪  p;
  if (!mid)  p->link = head;
  else {
    if (mid->len + 1 == mid->chd[x]->len) {
      p->link = mid->chd[x];
    } else {
      node *q = mid->chd[x], *r = &a[++top];
      *r = *q, q->link = p->link = r;
      r->len = mid->len + 1;
      for (; mid && mid->chd[x] == q; mid = mid->link)
      ↪  mid->chd[x] = r;
    }
  }
}
```

8 比赛现场配置

8.1 VIM 现场赛配置

```
syntax on
set nu
set cin
set sw=4
set ts=4
set mouse=a
set noswapfile
set nobackup

inoremap { {}<Left>
inoremap {<CR> {<CR>}<Esc>O
inoremap {{ {
inoremap {} {}
```

8.2 Snippet 代码头文件

```
#include <bits/stdc++.h>
using namespace std;
#define pb push_back
#define all(x) (x).begin(), (x).end()
#define sz(x) ((int)(x).size())
typedef vector<int> vi;
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int, int> pii;
mt19937 mrand(random_device{}());
const ll mod = 1000000007;
int rnd(int x) { return mrand() % x;}
ll mulmod(ll a, ll b) {ll res = 0; a %= mod; assert(b >= 0);
↪  for (; b; b >>= 1) {if (b & 1)res = (res + a) % mod; a = 2
↪  * a % mod;} return res;}
ll powmod(ll a, ll b) {ll res = 1; a %= mod; assert(b >= 0);
↪  for (; b; b >>= 1) {if (b & 1)res = res * a % mod; a = a *
↪  a % mod;} return res;}
ll gcd(ll a, ll b) { return b ? gcd(b, a % b) : a;}
//snippet-head
```