Contents

Tree Vector 建樹, DFS 先序尋訪, 找尋最近 LCA 共同祖先

```cpp
//POJ1330
#include <iostream>
#include <vector>
#include <cstring>
using namespace std;

const int N = 10000; // 最大節點數
vector<int> tree[N]; // 用來儲存樹的鄰接表
int parent[N]; // 紀錄每個節點的父節點
int depth[N]; // 紀錄每個節點的深度

// 深度優先搜尋 (DFS) 計算每個節點的深度
void DFS(int node, int dep) {
    depth[node] = dep; // 設置當前節點的深度
    for (size_t i = 0; i < tree[node].size(); i++) { // 遍歷所有子節點
        int child = tree[node][i];
        DFS(child, dep + 1); // 遞迴處理子節點，深度加 1
    }
}

// 最近共同祖先 (LCA) 查找函數
int findLCA(int x, int y) {
    // 讓深度較大的節點向上移動，直到兩個節點在同一深度
    while (x != y) {
        if (depth[x] > depth[y]) {
            x = parent[x]; // x 上移到其父節點
        } else {
            y = parent[y]; // y 上移到其父節點
        }
    }
    return x; // 返回最近共同祖先
}

int main() {
    int casenum, n, i, x, y;
    scanf("%d", &casenum); // 輸入測試案例數量

    for (int num = 0; num < casenum; num++) {
        scanf("%d", &n); // 輸入節點數
        for (i = 0; i < n; i++) tree[i].clear(); // 清空鄰接表
        memset(parent, -1, sizeof(parent)); // 初始化父節點為 -1

        // 輸入 n-1 條邊來建樹
        for (i = 0; i < n - 1; i++) {
            scanf("%d %d", &x, &y);
            x--; y--; // 將節點編號轉為 0 開始的索引
            tree[x].push_back(y); // 將 y 加入 x 的子節點中
            parent[y] = x; // 樹中 y 的父節點為 x
        }

        // 找到樹的根節點 (父節點為 -1 的節點)
        int root = 0;
        for (i = 0; i < n; i++) {
            if (parent[i] == -1) {
                root = i;
                break;
            }
        }
        DFS(root, 0); // 從根節點開始進行先序遍歷，初始化深度

        // 查詢最近共同祖先
        scanf("%d %d", &x, &y);
        x--; y--; // 將輸入的節點轉為 0 開始的索引
        int lca = findLCA(x, y); // 找到最近共同祖先
        printf("%d\n", lca + 1); // 輸出結果，轉回 1 開始的編號
    }

    return 0;
}
```

Tree Disjoinset 並查集, 路徑壓縮

```cpp
//POJ1703
#include <iostream>
#include <vector>
#include <cstring>

const int maxn = 100000 + 5;

int parent[maxn * 2]; // 並查集的根節點陣列，用來儲存每個節點的父節點

// 初始化並查集，將每個節點的根設置為 -1
void initializeDisjointSet(int size) {
    for (int i = 0; i < size; ++i) {
        parent[i] = -1; // 將每個節點初始化為獨立集合
    }
}

// 查找集合的根，並進行路徑壓縮
int find(int x) {
    if (parent[x] < 0) // 若 x 是根節點，則返回 x
        return x;
    return parent[x] = find(parent[x]); // 路徑壓縮，將 x 直接連接到根節點
}

// 合併兩個集合，將 x 和 y 所在的集合合併
void unionSets(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);
    if (rootX != rootY) {
        parent[rootX] = rootY; // 將 x 的根連接到 y 的根
    }
}

// 處理查詢指令，判斷是否同一幫派或敵對
void handleQuery(int a, int b, int n) {
    if (find(a) != find(b) && find(a) != find(b + n)) {
        printf("Not sure yet.\n");
    } else if (find(a) == find(b)) {
        printf("In the same gang.\n");
    } else {
        printf("In different gangs.\n");
    }
}

// 處理合併指令，將 a 和 b 設置為敵對關係
void handleUnion(int a, int b, int n) {
    if (find(a) != find(b + n)) {
        unionSets(a, b + n); // 將 a 的敵人設為 b
        unionSets(b, a + n); // 將 b 的敵人設為 a
    }
```

```
}

int main() {
    int loop;
    scanf("%d", &loop); // 測試用例數量
    while (loop--) {
        int n, m;
        scanf("%d%d", &n, &m);

        // 初始化並查集，大小為 2 * n，用來表示敵對關係
        initializeDisjointSet(2 * n + 1);

        for (int i = 0; i < m; i++) {
            int a, b;
            char command[5];
            scanf("%s%d%d", command, &a, &b);

            if (command[0] == 'A') { // 查詢指令
                handleQuery(a, b, n);
            } else { // 標記敵對關係
                handleUnion(a, b, n);
            }
        }
    }
    return 0;
}
```

Tree Fenwick tree 鄰接表樹, 時間戳樹, 權值陣列, **lowbit** 修改查詢區間和

```
//POJ3321
#include <iostream>
#include <vector>
#include <cstring>

#define MAXN 100002
using namespace std;

struct Edge {
    int next, tail; // next 指向下一條邊，tail 表示當前邊的終點
} edge[MAXN];

struct NodeRange {
    int l, r; // 節點的左右時間戳
} nodeRange[MAXN];

int head[MAXN], cnt = 1, fenwickTree[MAXN],
    values[MAXN]; // `head` 為鄰接表頭，`cnt` 為時間戳計數器

// 初始化
void initialize(int n) {
    memset(head, -1, sizeof(head[0]) * (n + 1));
    memset(fenwickTree, 0, sizeof(fenwickTree[0]) * (n
        + 1));
    memset(nodeRange, 0, sizeof(nodeRange[0]) * (n +
        1));
}

// 新增一條邊至鄰接表
void addEdge(int u, int v, int edgeIndex) {
    edge[edgeIndex].tail = v;
    edge[edgeIndex].next = head[u];
    head[u] = edgeIndex;
}

// DFS 標記每個節點的時間戳
void DFS(int u) {
    nodeRange[u].l = cnt;
    for (int i = head[u]; i != -1; i = edge[i].next) {
        DFS(edge[i].tail);
    }
    nodeRange[u].r = cnt++;
}

// Fenwick Tree 的低位元計算，用於更新和查詢操作
inline int lowbit(int x) {
    return x & -x;
}

// 更新 Fenwick Tree 中的位置值
void updateFenwick(int x, int delta) {
    for (int i = x; i < cnt; i += lowbit(i)) {
        fenwickTree[i] += delta;
    }
}

// 在 Fenwick Tree 中計算區間和
int queryFenwick(int x) {
    int sum = 0;
    for (int i = x; i > 0; i -= lowbit(i)) {
        sum += fenwickTree[i];
    }
    return sum;
}

// 初始化每個節點的權值
void initializeValues(int n) {
    for (int i = 1; i <= n; i++) {
        values[i] = 1;
        updateFenwick(i, 1); // 每個節點初始值為 1
    }
}

// 翻轉節點的權值
void toggleValue(int x) {
    if (values[x]) {
        updateFenwick(x, -1);
    } else {
        updateFenwick(x, 1);
    }
    values[x] = 1 - values[x]; // 翻轉節點值 (0->1 或 1->0)
}

// 查詢指定區間的和
int queryRange(int l, int r) {
    return queryFenwick(r) - queryFenwick(l - 1);
}

int main() {
    int n, m, u, v, t;
    char command[3];

    // 讀取節點數
    scanf("%d", &n);
    initialize(n); // 初始化陣列和 Fenwick Tree

    // 建立樹的鄰接表
    for (int i = 0; i < n - 1; ++i) {
        scanf("%d%d", &u, &v);
```

```
        addEdge(u, v, i);
    }

    // 使用 DFS 標記節點的時間戳
    DFS(1);

    // 初始化每個節點的初始權值為 1
    initializeValues(n);

    // 讀取指令數量
    scanf("%d", &m);
    while (m--) {
        scanf("%s%d", command, &t);
        if (command[0] == 'Q') { // 查詢指令
            printf("%d\n", queryRange(nodeRange[t].l,
                nodeRange[t].r));
        } else { // 修改指令
            toggleValue(nodeRange[t].r); // 翻轉權值
        }
    }

    return 0;
}
```

Trie 建樹, 修改, 查詢

```
//POJ2001
#include <cstdio>
#include <cstring>
#include <algorithm>

const int MAXN = 1000 + 10; // 單詞的最大長度
const int maxnode = 100005; // Trie 樹的最大節點數量
const int sigma_size = 26; // 字母表的大小（假設只有小寫字母）
char str[MAXN][25]; // 儲存單詞的陣列
int ch[maxnode][sigma_size]; // Trie 樹的子節點指標
int val[maxnode]; // Trie 樹節點的取值次數

int sz = 1; // Trie 樹的節點數量，初始化為 1（包含根節點）

// 初始化 Trie 樹
void initializeTrie() {
    sz = 1; // 重置節點數量
    memset(ch[0], 0, sizeof(ch[0])); // 初始化根節點
}

// 將字母轉換成數字索引
int charToIndex(char c) {
    return c - 'a'; // 將字符轉換為 0-25 的索引
}

// 插入單詞到 Trie 樹中
void insert(char *s) {
    int u = 0; // 起始於根節點 u = 0
    int n = strlen(s); // 獲取單詞的長度
    for (int i = 0; i < n; i++) {
        int c = charToIndex(s[i]); // 計算字母的索引值
        if (!ch[u][c]) { // 若該節點不存在，則創建新節點
            memset(ch[sz], 0, sizeof(ch[sz])); //
                初始化新節點
            ch[u][c] = sz++; // 設置子節點並增加節點數量
        }
        u = ch[u][c]; // 移動到下一個節點
        val[u]++; // 計算到達該節點的次數
    }
}

// 查詢單詞在 Trie 中的最短唯一前綴
void query(char *s) {
    int u = 0; // 起始於根節點 u = 0
    int n = strlen(s); // 獲取單詞的長度
    for (int i = 0; i < n; i++) {
        putchar(s[i]); // 輸出當前字母
        int c = charToIndex(s[i]); // 計算字母的索引值
        if (val[ch[u][c]] == 1) return; //
            若當前子節點的次數為 1，則找到最短前綴
        u = ch[u][c]; // 移動到下一個節點
    }
}

int main() {
    int tot = 0; // 單詞數初始化
    initializeTrie(); // 初始化 Trie 樹

    // 讀取每個單詞並插入到 Trie
    while (scanf("%s", str[tot]) != EOF) {
        insert(str[tot]); // 插入單詞到 Trie
        tot++; // 單詞數累加
    }

    // 查詢每個單詞的最短唯一前綴
    for (int i = 0; i < tot; i++) {
        printf("%s ", str[i]); // 輸出單詞
        query(str[i]); // 查詢單詞的最短前綴
        printf("\n"); // 換行
    }

    return 0;
}
```

Trie AC 自動機, 多模式字串數

```
//HDOJ2222
#include <iostream>
#include <cstring>
#include <string>
#include <queue>
using namespace std;

const int MAXN = 1e6 + 6; // 最大字串大小，根據需要調整
int cnt; // 匹配模式字串的次數計數器

// Trie 節點結構定義
struct node {
    int sum; // 該節點的匹配次數
    node *next[26]; // 子節點的指標陣列（對應 a-z）
    node *fail; // 失敗指針

    node() : sum(0), fail(nullptr) {
        for(int i = 0; i < 26; i++) next[i] = nullptr;
    }
};
node *root; // Trie 樹的根節點
char key[70]; // 用於儲存模式字串
char pattern[MAXN]; // 用於儲存目標字串
int N; // 模式字串數量

// 初始化 Trie 樹，建立根節點
void initializeTrie() {
    root = new node(); // 建立根節點
```

```
}

// 將單個模式字串插入到 Trie 樹中
void insertToTrie(char *s)
{
    node *p = root; // 從 Trie 樹的根節點開始
    for (int i = 0; s[i]; i++) {
        int x = s[i] - 'a'; // 計算字母的索引值（0 到 25）
        if (p->next[x] == nullptr) {
            p->next[x] = new node(); //
                若該子節點不存在，則創建新節點
        }
        p = p->next[x]; // 移動到下一個節點
    }
    p->sum++; // 該節點的匹配次數加 1
}

// 建立 AC 自動機的失敗指針
void buildFailPointer()
{
    queue<node*> q; // 使用 BFS 的隊列
    root->fail = nullptr; // 根節點的失敗指針設為空

    // 初始化根節點的所有子節點的失敗指針
    for (int i = 0; i < 26; i++) {
        if (root->next[i] != nullptr) {
            root->next[i]->fail = root; //
                根的子節點的失敗指針設為根節點
            q.push(root->next[i]); // 將該子節點加入隊列
        }
        else {
            root->next[i] = root; // 若缺失該邊，則指向根節點
        }
    }

    // 使用 BFS 建立其餘節點的失敗指針
    while (!q.empty()) {
        node* current = q.front(); q.pop(); // 取出隊首節點
        for (int i = 0; i < 26; i++) {
            if (current->next[i] != nullptr) { //
                 若當前節點有子節點
                node* fail_node = current->fail; //
                     從當前節點的失敗指針開始
                // 找到祖先節點的匹配邊
                while (fail_node != nullptr &&
                     fail_node->next[i] == nullptr)
                    fail_node = fail_node->fail;
                // 若找不到匹配的祖先節點，設置為根
                current->next[i]->fail = fail_node ?
                     fail_node->next[i] : root;
                q.push(current->next[i]); // 將子節點加入隊列
            }
            else {
                current->next[i] =
                     current->fail->next[i]; //
                     缺少的邊指向失敗指針的相應子節點
            }
        }
    }
}

// 使用 AC 自動機在目標字串中進行多模式匹配
void acAutomation(char *ch)
{
    node *p = root; // 從根節點開始
    int len = strlen(ch); // 獲取目標字串的長度
    for (int i = 0; i < len; i++) {
        int x = ch[i] - 'a'; // 當前字母的索引值
        // 當前字母無法匹配且非根節點時，沿著失敗指針移動
        while (p->next[x] == root && p != root)
            p = p->fail;

        p = p->next[x];
        if (!p) p = root; // 若找不到則返回根

        node *temp = p;
        // 沿著失敗指針查找所有匹配的模式
        while (temp != root) { // 從當前節點沿著失敗指針向上移動
            if (temp->sum >= 0) { // 若為匹配節點
                cnt += temp->sum; // 累計匹配次數
                temp->sum = -1; // 設置為 -1 以避免重複計算
            } else {
                break;
            }
            temp = temp->fail; // 繼續沿失敗指針向上
        }
    }
}

int main()
{
    int T; // 測試案例數量
    cin >> T;
    while (T--)
    {
        initializeTrie(); // 初始化 Trie 樹並建立根節點
        cin >> N; // 讀取模式字串數量
        cin.ignore(); // 忽略換行符

        // 插入所有模式字串到 Trie 樹
        for (int i = 1; i <= N; i++)
        {
            cin.getline(key, sizeof(key)); // 讀取模式字串
            insertToTrie(key); // 插入到 Trie 樹
        }

        // 讀取目標字串
        cin.getline(pattern, sizeof(pattern));
        cnt = 0; // 重置匹配次數
        buildFailPointer(); // 建立失敗指針
        acAutomation(pattern); // 進行多模式字串匹配
        cout << cnt << "\n"; // 輸出匹配到的模式字串次數
    }
    return 0;
}
```

BST 有序樹轉二元樹, 數組模擬樹

```
//POJ3437
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

string s; // 輸入的字符串
int i, n = 0, height1, height2; // 迭代變量 i, 樹的編號 n,
    兩種高度變量

// 遞迴構建二元樹並計算高度
```

```
void calculateHeights(int level1, int level2) {
    int tempson = 0; // 當前節點的子節點數量
    while (s[i] == 'd') { // 如果當前字元是 'd'（代表向下擴展）
        i++; // 移動到下一個字元
        tempson++; // 計算子節點
        calculateHeights(level1 + 1, level2 + tempson);
            // 遞迴進入下一層
    }
    height1 = max(height1, level1); // 更新高度1（基於深度）
    height2 = max(height2, level2); //
        更新高度2（基於水平位置）
    if (s[i] == 'u') i++; // 若為 'u' 字元，則返回到父節點
}

// 初始化並輸出結果
void processTree() {
    i = height1 = height2 = 0; // 重置索引和高度
    calculateHeights(0, 0); // 計算樹的高度
    cout << "Tree " << ++n << ": " << height1 << " =>
        " << height2 << endl;
}

int main() {
    while (cin >> s && s != "#") {
        processTree(); // 對每棵樹進行處理
    }
    return 0;
}
```

BST Stack 後序轉二元樹

```
//POJ3367
#include <iostream>
#include <stack>
#include <queue>
#include <cstring>
#include <algorithm>
using namespace std;

const int maxn = 11000;

struct node {
    int l, r; // 左右子節點的索引
    char c; // 節點的字元值
} e[maxn];

int cnt; // 節點數量
char s[maxn]; // 存儲輸入的後序表達式

// 初始化二元樹節點結構
void initial() {
    int len = strlen(s);
    for (int i = 0; i <= len; i++) {
        e[i].l = e[i].r = -1; // 初始化左右子節點為
            -1，表示無子節點
    }
    cnt = 0; // 重置節點計數器
}

// 將後序表達式轉換為二元樹
void solve() {
    int len = strlen(s);
    stack<int> v; // 用來處理二元樹節點的堆疊
    for (int i = 0; i < len; i++) {
        if (s[i] >= 'a' && s[i] <= 'z') { // 若為字母節點
            e[cnt].c = s[i]; // 將字母存入節點
            v.push(cnt); // 將當前節點索引壓入堆疊
            cnt++; // 節點數加一
        } else { // 若為操作符節點
            int r = v.top(); // 彈出右子節點
            v.pop();
            int l = v.top(); // 彈出左子節點
            v.pop();
            e[cnt].l = l; // 設置當前節點的左子節點
            e[cnt].r = r; // 設置當前節點的右子節點
            e[cnt].c = s[i]; // 將操作符存入節點
            v.push(cnt); // 將當前節點索引壓入堆疊
            cnt++; // 節點數加一
        }
    }
}

// 層次遍歷輸出並反轉結果
void output() {
    string ans;
    queue<int> q;
    q.push(cnt - 1); // 將根節點（最後一個節點）加入隊列
    while (!q.empty()) {
        int st = q.front();
        q.pop();
        ans.push_back(e[st].c); // 將節點的字元加入結果字串
        if (e[st].l != -1) q.push(e[st].l); //
            若有左子節點，則加入隊列
        if (e[st].r != -1) q.push(e[st].r); //
            若有右子節點，則加入隊列
    }
    reverse(ans.begin(), ans.end()); // 反轉結果，取得所需順序
    printf("%s\n", ans.c_str()); // 輸出結果
}

int main() {
    int t;
    scanf("%d", &t); // 測試案例數
    while (t--) {
        scanf("%s", s); // 讀取後序表達式
        initial(); // 初始化樹節點
        solve(); // 將後序表達式轉換為二元樹
        output(); // 輸出結果
    }
    return 0;
}
```

BST 前序加中序找出後序

```
//POJ2255
#include <stdio.h>
#include <string.h>

char preord[30], inord[30]; // 儲存前序和中序遍歷的字串

// 讀取每個測試案例的前序和中序遍歷
int read_case() {
    if (scanf("%s %s", preord, inord) != 2) return 0;
        // 若輸入不為 2 個字串，返回 0
    return 1; // 成功讀取返回 1
}

// 根據前序和中序遍歷來遞迴構建樹並輸出後序遍歷
void recover(int preleft, int preright, int inleft,
     int inright) {
```

```
    int root, leftsize, rightsize;

    // 根據前序的根節點位置，在中序中找到對應位置，確定左右子樹範圍
    for (root = inleft; root <= inright; root++) {
        if (preord[preleft] == inord[root]) break; //
            找到中序中的根節點位置
    }

    leftsize = root - inleft; // 左子樹的節點數
    rightsize = inright - root; // 右子樹的節點數

    // 遞迴處理左子樹
    if (leftsize > 0)
        recover(preleft + 1, preleft + leftsize,
            inleft, root - 1);

    // 遞迴處理右子樹
    if (rightsize > 0)
        recover(preleft + leftsize + 1, preright, root
            + 1, inright);

    // 後序輸出根節點
    printf("%c", inord[root]);
}

// 解決每個測試案例，計算並輸出後序遍歷
void solve_case() {
    int n = strlen(preord); // 前序字串的長度即為節點總數
    recover(0, n - 1, 0, n - 1); // 遞迴計算後序遍歷
    printf("\n"); // 每個案例後序遍歷結束換行
}

int main() {
    // 讀取每個測試案例並解決
    while (read_case()) solve_case();
    return 0;
}
```

BST 後序二分搜尋樹還原

```
//URAL 1136
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

int n; // 節點總數
int a[3010]; // 儲存後序遍歷的節點值

// 從後序遍歷還原並輸出前序遍歷（右子樹-左子樹-根）
void solve(int l, int r) {
    // 如果範圍無效則返回，表示已無節點需要處理
    if (l > r) return;

    // 找到右子樹的起始位置，該位置的元素大於當前子樹根節點
    //（後序遍歷中的根節點是 a[r]）
    int i = l;
    while (i < r && a[i] < a[r]) i++; //
        找到左子樹與右子樹的分界點

    // 先遞迴處理右子樹（右子樹的範圍是 i 到 r-1）
    if (i < r) solve(i, r - 1);

    // 再遞迴處理左子樹（左子樹的範圍是 l 到 i-1）
    if (l < i) solve(l, i - 1);

    // 最後輸出根節點（後序遍歷的根節點）
    printf("%d\n", a[r]);
}

int main() {
    // 讀取節點總數
    scanf("%d", &n);

    // 讀取後序遍歷的節點數據
    for (int i = 0; i < n; ++i)
        scanf("%d", &a[i]);

    // 使用遞迴方法處理後序遍歷並輸出前序結果
    solve(0, n - 1);
    return 0;
}
```

BST 建立結構指標二元樹, 前序尋訪

```
//HDOJ3999
#include <stdio.h>

typedef struct binTreeNode { // 定義二元搜尋樹的節點結構
    int data;
    struct binTreeNode *lchild, *rchild;
} *BT;

// 向二元搜尋樹中插入新值
void add(BT &T, int val) {
    if (T == NULL) { // 若當前節點為空，則找到插入位置
        T = new binTreeNode(); // 建立新節點，並賦予初始值
        T->data = val;
        T->lchild = T->rchild = NULL; // 新節點沒有子節點
    } else if (T->data > val) { //
        若值小於當前節點，則插入左子樹
        add(T->lchild, val);
    } else { // 若值不小於當前節點，則插入右子樹
        add(T->rchild, val);
    }
}

// 前序遍歷輸出二元搜尋樹
void preOrder(BT T, bool flag) {
    if (T == NULL)
        return;

    if (!flag) // 若非首節點，則在輸出前添加空格
        printf(" ");

    printf("%d", T->data); // 輸出當前節點的值

    preOrder(T->lchild, false); // 遞迴處理左子樹，flag 設為
        false
    preOrder(T->rchild, false); // 遞迴處理右子樹，flag 設為
        false
}

// 建立二元搜尋樹並進行前序遍歷輸出
void buildAndPrintBST() {
    BT T = NULL; // 初始化根節點
    int n, v;
    if (scanf("%d", &n) != 1) return; //
        讀取二元搜尋樹的節點數

```

```
    for (int i = 0; i < n; i++) {
        scanf("%d", &v); // 讀取每個節點值
        add(T, v); // 插入二元搜尋樹
    }

    preOrder(T, true); // 按前序順序輸出二元搜尋樹
    printf("\n"); // 輸出後換行
}

int main() {
    // 不斷讀取測試案例，建立並輸出二元搜尋樹
    while (!feof(stdin)) {
        buildAndPrintBST();
    }
    return 0;
}
```

BST Heap priority queue 插入取出調整

```
//ZOJ2724
#include <iostream>
#include <cstdio>
#include <cstring>
#include <queue>

using namespace std;

const int maxn = 60000 + 10;
const int maxs = 100;

// 定義結構體來存儲佇列的資訊
struct Info {
    char name[maxs]; // 名稱
    int para; // 儲存的參數
    int pri; // 優先級
    int t; // 插入時間，用於維持先插入先處理
};

Info p[maxn]; // 儲存所有的 Info 物件
int used = 0; // `p` 的下一個可用索引
int cnt = 0; // 記錄插入的順序，維持穩定性

// 自定義比較器，用於在優先佇列中排序
struct Compare {
    bool operator()(const int a, const int b) const {
        if (p[a].pri != p[b].pri) //
            若優先級不同，根據優先級排序
            return p[a].pri > p[b].pri; // 優先級越小優先處理
        return p[a].t > p[b].t; //
            若優先級相同，插入時間早者優先
    }
};

// 定義優先佇列
priority_queue<int, vector<int>, Compare> pq;

// 處理 GET 命令，從優先佇列取出元素
void processGetCommand() {
    if (!pq.empty()) {
        int top_idx = pq.top(); // 獲取最高優先級的元素索引
        pq.pop(); // 從佇列中刪除該元素
        printf("%s %d\n", p[top_idx].name,
            p[top_idx].para); // 輸出元素資訊
    } else {
        printf("EMPTY QUEUE!\n"); // 若佇列為空，輸出 "EMPTY
            QUEUE!"
    }
}

// 處理 PUT 命令，將新元素插入優先佇列
void processPutCommand() {
    // 讀取並存儲新元素的 name、para、pri 資訊
    scanf("%s %d %d", p[used].name, &p[used].para,
        &p[used].pri);
    p[used].t = cnt++; // 設置插入順序並遞增
    pq.push(used++); // 將新元素的索引壓入優先佇列
}

// 主函式，處理輸入並根據指令執行相應操作
void processCommands() {
    char command[maxs]; // 儲存命令名稱

    // 循環讀取命令，直到文件結尾
    while (scanf("%s", command) != EOF) {
        if (strcmp(command, "GET") == 0) {
            processGetCommand(); // 處理 GET 命令
        } else { // 否則視為 PUT 命令
            processPutCommand(); // 處理 PUT 命令
        }
    }
}

int main() {
    processCommands(); // 處理所有輸入命令
    return 0;
}
```

BST Treap 樹堆積左旋右旋, 插入刪除

```
//POJ3481
#include <cstdio>
#include <cstdlib>
using namespace std;

// Treap 節點結構
struct Node {
    Node *ch[2]; // 左右子樹指標
    int v, r, info; // v 是優先順序，info 是客戶編號，r
        是節點的隨機優先級

    Node(int v, int info) : v(v), info(info) {
        r = rand(); // 隨機產生優先級
        ch[0] = ch[1] = NULL; // 初始化左右子樹為空
    }

    // 比較當前節點的優先順序和目標值 x
    int cmp(int x) {
        if (v == x) return -1; // 若相等，返回 -1 表示找到目標
        return x < v ? 0 : 1; // 小於當前值返回
            0（左子樹），大於返回 1（右子樹）
    }
};

// Treap 左右旋轉
void rotate(Node *&o, int d) {
    Node *k = o->ch[d^1]; // 獲取旋轉後的新根
    o->ch[d^1] = k->ch[d]; // 更新 o 的子樹指標
    k->ch[d] = o; // 完成旋轉
    o = k; // 更新根節點
```

```
}

// Treap 插入節點
void insert(Node *&o, int v, int info) {
    if (o == NULL) {
        o = new Node(v, info); // 若節點為空，創建新節點
    } else {
        int d = v < o->v ? 0 : 1; // 若 v
             小於當前節點值，放左子樹，否則右子樹
        insert(o->ch[d], v, info); // 遞迴插入左或右子樹
        if (o->ch[d]->r > o->r) // 若新節點的優先級高於根節點
            rotate(o, d^1); // 進行旋轉
    }
}

// Treap 刪除節點
void remove(Node *&o, int v) {
    if (!o) return;
    int d = o->cmp(v);
    if (d == -1) { // 若找到節點
        Node *u = o;
        if (o->ch[0] && o->ch[1]) { // 若節點有兩個子樹
            int d2 = o->ch[0]->r < o->ch[1]->r ? 1 : 0;
            rotate(o, d2); // 進行旋轉，維持 Treap 性質
            remove(o->ch[d2], v); // 繼續刪除節點
        } else { // 若只有一個子樹，將子樹提到當前位置
            o = o->ch[0] ? o->ch[0] : o->ch[1];
            delete u; // 刪除節點
        }
    } else {
        remove(o->ch[d], v); // 遞迴刪除左或右子樹
    }
}

// 查找 Treap 中優先順序最大的節點
int find_max(Node *o) {
    if (!o) return -1;
    while (o->ch[1] != NULL) o = o->ch[1]; //
         移動到最右子節點
    printf("%d\n", o->info); // 輸出最大優先順序的節點 info
    return o->v;
}

// 查找 Treap 中優先順序最小的節點
int find_min(Node *o) {
    if (!o) return -1;
    while (o->ch[0] != NULL) o = o->ch[0]; //
         移動到最左子節點
    printf("%d\n", o->info); // 輸出最小優先順序的節點 info
    return o->v;
}

// 處理用戶命令
void processCommands() {
    int op;
    Node *root = NULL;

    while (scanf("%d", &op) == 1 && op) {
        if (op == 1) { // 插入新客戶
            int v, info;
            scanf("%d%d", &info, &v);
            insert(root, v, info);
        } else if (op == 2) { // 查找最大優先順序的客戶
            if (root == NULL) {
                printf("0\n");
                continue;
            }
            int v = find_max(root);
            if (v != -1) remove(root, v);
        } else if (op == 3) { // 查找最小優先順序的客戶
            if (root == NULL) {
                printf("0\n");
                continue;
            }
            int v = find_min(root);
            if (v != -1) remove(root, v);
        }
    }
}

int main() {
    processCommands();
    return 0;
}
```

BST Treap rope 結構操作字串修改 Treap

```
//UVA12538
/*
  rope 函式庫提供的基本操作有:
  list.insert(p, str); // 將字串 str 插入到 rope 的 p 位置
  list.erase(p, c); // 刪除 rope 中從 p 位置開始的 c 個字元
  list.substr(p, c); // 擷取 rope 中從 p 位置開始長度為 c 的子字串
  list.copy(q, p, c); // 將 rope 中從 p 位置開始長度為 c 的子字串複製到
       q
*/
#include <iostream>
#include <ext/rope> // 使用 GNU C++ rope 函式庫
using namespace std;
using namespace __gnu_cxx; // rope 所在的命名空間

rope<char> ro; // 定義當前版本的 rope 物件
rope<char> l[50005]; // 紀錄每個版本
char str[205]; // 用於暫存輸入的字串
int cnt = 1, d = 0; // 版本計數器和偏移值

// 插入操作: 將字串插入 rope 的指定位置
void insertString(int p, const char* str) {
    p -= d; // 計算相對位置
    ro.insert(p, str); // 將字串 str 插入 rope 的 p 位置
    l[cnt++] = ro; // 儲存當前版本
}

// 刪除操作: 刪除 rope 中指定位置的指定長度的字元
void deleteString(int p, int c) {
    p -= d; c -= d; // 計算相對位置和長度
    ro.erase(p - 1, c); // 刪除 rope 中從 p 開始長度為 c 的字元
    l[cnt++] = ro; // 儲存當前版本
}

// 列印操作: 從指定版本擷取子字串並輸出，並計算 'c' 的次數
void printSubstring(int v, int p, int c) {
    p -= d; v -= d; c -= d; // 計算相對位置和長度
    rope<char> tmp = l[v].substr(p - 1, c); //
         擷取指定版本的子字串
    d += count(tmp.begin(), tmp.end(), 'c'); //
         計算子字串中 'c' 的出現次數並更新偏移
    cout << tmp << "\n"; // 輸出子字串
}
```

```cpp

// 主函式，處理所有命令
void processCommands() {
    int n, op, p, c, v;
    cin >> n;
    while (n--) {
        cin >> op;
        if (op == 1) { // 插入命令
            cin >> p >> str;
            insertString(p, str); // 呼叫插入操作
        } else if (op == 2) { // 刪除命令
            cin >> p >> c;
            deleteString(p, c); // 呼叫刪除操作
        } else if (op == 3) { // 列印命令
            cin >> v >> p >> c;
            printSubstring(v, p, c); // 呼叫列印操作
        }
    }
}

int main() {
    processCommands();
    return 0;
}
```

BST 霍夫曼樹最小代價 Min Heap

```cpp
//HDOJ5884
#include <iostream>
#include <queue>
#include <algorithm>
using namespace std;

const int maxn = 1e5 + 100;
typedef long long ll;
queue<ll> q1, q2; // 定義兩個佇列，用於模擬 k 元霍夫曼樹
ll a[maxn]; // 儲存序列元素
ll t, n; // 目標代價 t 和元素數量 n

// 清空佇列 q1 和 q2
void clearQueues() {
    while (!q1.empty()) q1.pop();
    while (!q2.empty()) q2.pop();
}

// 初始化佇列：根據虛葉節點和排序後的序列元素
void initializeQueue(int x) {
    int extraLeaves = (n - 1) % (x - 1) != 0 ? (x - 1) - (n - 1) % (x - 1) : 0;
    for (int i = 1; i <= extraLeaves; i++) q1.push(0); // 添加虛葉節點
    for (int i = 1; i <= n; i++) q1.push(a[i]); // 添加序列元素
}

// 模擬 k 元霍夫曼樹，判斷是否在限制代價 t 內
bool huffman(int x) {
    clearQueues(); // 清空佇列
    initializeQueue(x); // 初始化佇列

    ll totalCost = 0; // 儲存霍夫曼合併的總代價

    // 進行霍夫曼合併
    while (true) {
        ll currentSum = 0; // 當前合併的 x 個元素的總和

        for (int i = 1; i <= x; i++) { // 每次取出 x 個元素進行合併
            if (q1.empty() && q2.empty()) break;

            if (q1.empty()) {
                currentSum += q2.front();
                q2.pop();
            } else if (q2.empty()) {
                currentSum += q1.front();
                q1.pop();
            } else if (q1.front() < q2.front()) {
                currentSum += q1.front();
                q1.pop();
            } else {
                currentSum += q2.front();
                q2.pop();
            }
        }

        totalCost += currentSum; // 更新總代價
        if (q1.empty() && q2.empty()) break;

        q2.push(currentSum); // 將合併結果加入 q2 佇列
        if (totalCost > t) return false; // 若超過目標代價，返回失敗
    }

    return totalCost <= t; // 若總代價在限制內，返回成功
}

// 使用二分法查找最小的 k 值，確保代價不超過 t
int findMinK() {
    int start = 2, end = n;
    while (start < end) {
        int mid = (start + end) / 2;
        if (huffman(mid)) end = mid;
        else start = mid + 1;
    }
    return start; // 返回找到的最小 k 值
}

// 主函式，處理多組測試案例
void processCases() {
    int T;
    scanf("%d", &T);
    while (T--) {
        scanf("%lld%lld", &n, &t); // 讀取 n 和 t
        for (int i = 1; i <= n; i++) scanf("%lld", &a[i]); // 讀取序列元素
        sort(a + 1, a + 1 + n); // 將序列排序
        printf("%d\n", findMinK()); // 輸出最小 k 值
    }
}

int main() {
    processCases();
    return 0;
}
```

Graph BFS 狀態空間搜尋最短路徑

```cpp
//POJ3126
#include <iostream>
#include <queue>
```

```cpp
#include <cstring>
using namespace std;

// 定義節點結構，包含當前數字和步數
struct node {
    int k, step;
    node(int k_val, int step_val) : k(k_val), step(step_val) {}
};

// 使用埃拉托斯特尼篩法產生 10000 以內的質數
const int power10_arr[4] = {1000, 100, 10, 1}; // 用於數位位置變換
bool is_prime[10000], visited[10000]; // 儲存質數和訪問狀態

// 初始化篩法產生質數
void makeSieve(int n) {
    memset(is_prime, 0, sizeof(is_prime));
    is_prime[0] = is_prime[1] = 1; // 0 和 1 不是質數
    for (int i = 2; i <= n; i++) {
        if (!is_prime[i]) {
            for (int j = i * i; j <= n; j += i)
                is_prime[j] = 1; // 標記 i 的所有倍數為非質數
        }
    }
}

// 改變 x 的第 pos 位數字為 new_digit，並返回新數字
int changeDigit(int x, int pos, int new_digit) {
    int digits[4] = { x / 1000, (x / 100) % 10, (x / 10) % 10, x % 10 };
    digits[pos - 1] = new_digit;
    return digits[0] * 1000 + digits[1] * 100 + digits[2] * 10 + digits[3];
}

// 使用 BFS 搜索 x 到 y 的最短路徑，僅允許變換為質數
int bfsShortestPath(int x, int y) {
    // 初始化佇列和訪問狀態
    queue<node> q;
    memset(visited, 0, sizeof(visited));

    q.push(node(x, 0));
    visited[x] = 1;

    while (!q.empty()) {
        node cur = q.front();
        q.pop();

        if (cur.k == y) return cur.step; // 找到目標數字時返回步數

        // 遍歷每一位數進行數字變換
        for (int i = 1; i <= 4; i++) { // i 表示數位 (千位、百位、十位、個位)
            for (int j = 0; j <= 9; j++) {
                // 跳過不合法的情況：千位不為 0 和當前數字不變的情況
                if ((i == 1 && j == 0) || ((cur.k / power10_arr[i - 1]) % 10) == j)
                    continue;

                int new_number = changeDigit(cur.k, i, j);

                // 判斷新數字是否為質數，且在範圍內，且未被訪問過
                if (new_number >= 1000 && new_number <= 9999 && !is_prime[new_number] && !visited[new_number]) {
                    visited[new_number] = 1;
                    q.push(node(new_number, cur.step + 1)); // 新數字加入佇列
                }
            }
        }
    }
    return -1; // 無法從 x 轉換為 y 時，返回 -1
}

// 主程式，處理多組測試案例
void processCases() {
    int test_cases;
    cin >> test_cases;

    while (test_cases--) {
        int start, end;
        cin >> start >> end;

        // 驗證輸入是否在範圍內且為質數
        if (start < 1000 || start > 9999 || end < 1000 || end > 9999 || is_prime[start] || is_prime[end]) {
            cout << "Impossible" << endl;
        } else {
            int result = bfsShortestPath(start, end);
            if (result >= 0)
                cout << result << "\n";
            else
                cout << "Impossible\n";
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    makeSieve(9999); // 初始化質數篩法
    processCases(); // 處理多組測試案例
    return 0;
}
```

Graph DFS 走訪, 建無向相鄰矩陣

```cpp
//UVA291
#include <iostream>
#include <cstring>
using namespace std;

int map[6][6]; // 定義無向圖的相鄰矩陣，節點編號從 1 到 5

// 初始化無向圖的相鄰矩陣
void initializeMap() {
    memset(map, 0, sizeof(map)); // 將矩陣所有元素設置為 0，表示無邊
    for (int i = 1; i <= 5; i++) {
        for (int j = 1; j <= 5; j++) {
            if (i != j) map[i][j] = 1; // 除了自己，所有節點相鄰（設為 1 表示有邊）
        }
```

```
    }
    // 刪除節點間不相鄰的邊
    map[4][1] = map[1][4] = 0;
    map[4][2] = map[2][4] = 0;
}

// 使用 DFS 搜索所有可能的訪問路徑
void dfsTraversal(int currentNode, int depth, string path) {
    path += char(currentNode + '0'); // 將當前節點加入路徑中

    if (depth == 8) { // 如果已遍歷 8 條邊，輸出該路徑
        cout << path << endl;
        return;
    }

    // 遍歷所有相鄰節點
    for (int nextNode = 1; nextNode <= 5; nextNode++) {
        if (map[currentNode][nextNode]) { // 如果存在相鄰邊
            map[currentNode][nextNode] = map[nextNode][currentNode] = 0; // 標記邊為已訪問
            dfsTraversal(nextNode, depth + 1, path); // 遞迴搜索相鄰節點
            map[currentNode][nextNode] = map[nextNode][currentNode] = 1; // 回溯，恢復邊的狀態
        }
    }
}

// 主函式
void runTraversal() {
    initializeMap(); // 初始化無向圖的相鄰矩陣
    dfsTraversal(1, 0, ""); // 從節點 1 開始，深度 0，初始路徑為空字串
}

int main() {
    runTraversal();
    return 0;
}
```

Graph DFS 剪枝回朔, 狀態空間搜尋

```
//POJ1011
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int sticks[65]; // 儲存木棍的長度
int used[65]; // 標記木棍是否被使用
int n, len, sum; // 木棍數量、目標長度、所有木棍的總長度

// 初始化木棍數據和標記
void initialize() {
    sum = 0;
    for (int i = 0; i < n; ++i) {
        cin >> sticks[i];
        sum += sticks[i]; // 計算木棍總長度
        used[i] = 0; // 初始化使用標記
    }
    sort(sticks, sticks + n, greater<int>()); // 按木棍長度降序排列
}

// DFS 搜尋木棍的切割可能性
bool dfs(int index, int remainingLength, int remainingSum) {
    // 若已組成一根長度為 len 的木棍
    if (remainingLength == 0) {
        remainingSum -= len; // 減去一根已組成木棍的長度
        if (remainingSum == 0) return true; // 若沒有剩餘長度則成功

        // 找到下一個未使用的木棍，並嘗試組成下一根木棍
        int i = 0;
        while (used[i]) ++i;
        used[i] = 1; // 標記該木棍已使用
        if (dfs(i + 1, len - sticks[i], remainingSum)) return true;
        used[i] = 0;
        return false;
    }

    // 嘗試使用剩餘木棍組成當前目標長度
    for (int j = index; j < n; ++j) {
        // 當前木棍已被使用，
        //或與前一木棍相同且前一木棍未被使用則跳過
        if (used[j] || (j > 0 && sticks[j] == sticks[j - 1] && !used[j - 1])) continue;

        // 如果木棍長度不超過當前剩餘長度，嘗試將其使用
        if (sticks[j] <= remainingLength) {
            used[j] = 1;
            if (dfs(j + 1, remainingLength - sticks[j], remainingSum)) return true;
            used[j] = 0;

            // 如果無法完成當前木棍的組合，則結束該分支
            if (sticks[j] == remainingLength) break;
        }
    }
    return false;
}

// 找到最小的木棍長度，使得所有木棍可以被完全拼接成該長度的木棍
int findMinimumStickLength() {
    for (len = sticks[0]; len <= sum / 2; ++len) { // 在 [最大木棍長度..sum/2] 區間搜尋
        if (sum % len == 0) { // 若總長度能被 len 整除
            if (dfs(0, len, sum)) return len; // 若可以組成目標長度，返回此長度
        }
    }
    return sum; // 若無法組成，返回總長度
}

int main() {
    while (cin >> n && n) { // 讀取木棍數量，直到輸入 0 為止
        initialize(); // 初始化木棍數據
        cout << findMinimumStickLength() << endl; // 找到並輸出最小的木棍可能長度
    }
    return 0;
}
```

Graph DFS 回朔找尋拓樸排序, 建相鄰有向邊

```cpp
//POJ1094
#include <iostream>
#include <vector>
#include <cstring>
#include <queue>
#include <string>

using namespace std;

const int MAXN = 100; // 最大節點數

// 深度優先搜尋 (DFS)：檢查從 current 到 target 是否可達
bool is_reachable(int current, int target, int N, int g[][MAXN], bool visited[]) {
    if (current == target) return true; // 如果到達目標節點，返回 true
    visited[current] = true;
    for (int i = 0; i < N; i++) {
        if (g[current][i] && !visited[i]) { // 如果有連接且未訪問，遞迴檢查
            if (is_reachable(i, target, N, g, visited))
                return true;
        }
    }
    return false;
}

// 檢查並構建拓樸排序
bool topological_sort(int N, int g[][MAXN], vector<int>& in_degree, vector<int>& result) {
    vector<int> in_copy = in_degree; // 複製 in-degree 以供修改
    queue<int> Q;
    result.clear();

    // 初始入度為 0 的節點入佇列
    for (int i = 0; i < N; i++) {
        if (in_copy[i] == 0) Q.push(i);
    }

    bool multiple_sequences = false; // 用於檢查多個拓樸序

    while (!Q.empty()) {
        if (Q.size() > 1) multiple_sequences = true; // 多個節點入度為 0

        int current = Q.front();
        Q.pop();
        result.push_back(current);

        // 更新鄰居的入度
        for (int i = 0; i < N; i++) {
            if (g[current][i]) {
                in_copy[i]--;
                if (in_copy[i] == 0) Q.push(i);
            }
        }
    }

    // 判斷是否完成唯一排序
    return result.size() == N && !multiple_sequences;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int N, K;
    while (cin >> N >> K) {
        if (N == 0 && K == 0) break; // 當 N 與 K 都為 0 時結束

        // 初始化相鄰矩陣和入度計數
        int g[MAXN][MAXN];
        memset(g, 0, sizeof(g));
        vector<int> in_degree(N, 0);

        bool determined = false;
        bool inconsistent = false;
        string relation;

        for (int i = 0; i < K; i++) {
            cin >> relation;

            char a = relation[0];
            char b = relation[2];
            char op = relation[1];
            int x, y;

            // 根據關係設定方向
            if (op == '<') {
                x = a - 'A';
                y = b - 'A';
            } else { // op == '>'
                x = b - 'A';
                y = a - 'A';
            }

            // 使用 DFS 檢查是否形成迴圈
            bool visited[MAXN];
            memset(visited, false, sizeof(visited));
            if (is_reachable(y, x, N, g, visited)) {
                cout << "Inconsistency found after " << (i + 1) << " relations." << endl;
                for (int j = i + 1; j < K; j++) cin >> relation; // 忽略剩餘關係
                inconsistent = true;
                break;
            }

            // 新增有向邊 x -> y 並更新入度
            g[x][y] = 1;
            in_degree[y]++;

            // 嘗試拓樸排序
            vector<int> result;
            if (topological_sort(N, g, in_degree, result)) {
                cout << "Sorted sequence determined after " << (i + 1) << " relations: ";
                for (size_t idx = 0; idx < result.size(); idx++) {
                    cout << char('A' + result[idx]);
                }
                cout << "." << endl;

                for (int j = i + 1; j < K; j++) cin >> relation; // 忽略剩餘關係
```

```
            determined = true;
            break;
        }
    }

    // 若無法判定唯一序列或發現不一致
    if (!inconsistent && !determined) {
        cout << "Sorted sequence cannot be determined." << endl;
    }
}

return 0;
}
```

Graph DFS 計算圖連接性

```
//POJ1562
#include <iostream>
#include <cstring>
using namespace std;

int map[105][105]; // 儲存油田的圖形，1 表示油田，0 表示空地
int vis[105][105]; // 紀錄是否已訪問
int n, m; // 圖形的行數和列數

// 初始化訪問標記的函式
void initializeVisited() {
    memset(vis, 0, sizeof(vis)); // 將訪問標記陣列清零
}

// 深度優先搜索 (DFS) 用於探測一個油田的連接範圍
void depthFirstSearch(int x, int y) {
    vis[x][y] = 1; // 標記當前節點為已訪問

    // 探索所有 8 個方向
    if (x + 1 < n && !vis[x + 1][y] && map[x + 1][y])
        depthFirstSearch(x + 1, y); // 下
    if (x - 1 >= 0 && !vis[x - 1][y] && map[x - 1][y])
        depthFirstSearch(x - 1, y); // 上
    if (y + 1 < m && !vis[x][y + 1] && map[x][y + 1])
        depthFirstSearch(x, y + 1); // 右
    if (y - 1 >= 0 && !vis[x][y - 1] && map[x][y - 1])
        depthFirstSearch(x, y - 1); // 左
    if (x + 1 < n && y + 1 < m && !vis[x + 1][y + 1]
        && map[x + 1][y + 1]) depthFirstSearch(x + 1,
        y + 1); // 右下
    if (x - 1 >= 0 && y - 1 >= 0 && !vis[x - 1][y - 1]
        && map[x - 1][y - 1]) depthFirstSearch(x - 1,
        y - 1); // 左上
    if (x + 1 < n && y - 1 >= 0 && !vis[x + 1][y - 1]
        && map[x + 1][y - 1]) depthFirstSearch(x + 1,
        y - 1); // 左下
    if (x - 1 >= 0 && y + 1 < m && !vis[x - 1][y + 1]
        && map[x - 1][y + 1]) depthFirstSearch(x - 1,
        y + 1); // 右上
}

// 計算油田數量的函式
int countOilDeposits() {
    int count = 0; // 用於儲存油田的總數量
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (!vis[i][j] && map[i][j]) { //
                若節點未訪問且是油田
                depthFirstSearch(i, j); // 執行 DFS 探索該油田
                count++; // 每探索一次，計算一個獨立的油田
            }
        }
    }
    return count; // 返回油田的總數
}

int main() {
    char ch;
    while (cin >> n >> m) { // 輸入圖形的大小
        if (n == 0 && m == 0) break;

        initializeVisited(); // 初始化訪問標記陣列

        // 填充圖形的數據
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                cin >> ch;
                map[i][j] = (ch == '@') ? 1 : 0; // '@'
                    表示油田，'*' 表示空地
            }
        }

        int result = countOilDeposits(); // 計算油田數量
        cout << result << endl; // 輸出油田的總數
    }
    return 0;
}
```

Graph BFS 計算圖連接性

```
//POJ1562
#include <iostream>
#include <cstring>
using namespace std;

struct Position {
    int i, j; // 網格位置
} bfsQueue[10000]; // BFS 的佇列，重新命名為 bfsQueue
    避免與標準庫衝突

int m, n; // 網格的行數 m 和列數 n
char map[101][101]; // 相鄰矩陣，'*' 表示牆，'@' 表示油田
int directions[8][2] = { {-1, 0}, {1, 0}, {0, -1}, {0,
    1}, {-1, -1}, {-1, 1}, {1, -1}, {1, 1} }; // 8 個方向

// 初始化網格
void initializeMap() {
    memset(map, 0, sizeof(map)); // 將網格 map 初始化
}

// 執行 BFS 遍歷找到所有相連的油田
void performBFS(int startRow, int startCol) {
    int front = 0, rear = 1; // 初始化 BFS 佇列的首尾指標
    bfsQueue[front].i = startRow;
    bfsQueue[front].j = startCol;
    map[startRow][startCol] = '*'; // 將起始點設為已訪問狀態

    while (front != rear) {
        int currentRow = bfsQueue[front].i;
        int currentCol = bfsQueue[front].j;
        front++; // 出佇列，佇列首指標 +1

        // 探索 8 個可能方向
```

```
        for (int k = 0; k < 8; k++) {
            int newRow = currentRow + directions[k][0];
            int newCol = currentCol + directions[k][1];

            if (newRow >= 0 && newRow < m && newCol >=
                0 && newCol < n && map[newRow][newCol]
                == '@') {
                bfsQueue[rear].i = newRow;
                bfsQueue[rear].j = newCol;
                map[newRow][newCol] = '*'; //
                    將新位置設為已訪問
                rear++; // 進佇列，佇列尾指標 +1
            }
        }
    }
}

// 計算網格中所有獨立油田的數量
int countOilDeposits() {
    int oilCount = 0; // 計算油田的總數
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (map[i][j] == '@') { // 若找到新的油田
                oilCount++; // 油田數量加一
                performBFS(i, j); // 使用 BFS
                    探索整個油田區域並標記
            }
        }
    }
    return oilCount; // 返回油田的總數量
}

int main() {
    while (scanf("%d %d", &m, &n) && m) { // 重複讀取行數 m
        和列數 n，直到 m 為 0
        initializeMap(); // 初始化網格資料
        for (int i = 0; i < m; i++) {
            scanf("%s", map[i]); // 讀取網格中的所有元素
        }
        int result = countOilDeposits(); //
            計算所有的油田數量
        printf("%d\n", result); // 輸出結果
    }
    return 0;
}
```

Graph 有向邊並查集, 速通性檢查, 樹判斷

```
//POJ1308
#include <cstdio>
#include <memory>

const int MAX_SIZE = 105;
int parent[MAX_SIZE]; // 每個點的根節點
bool flag[MAX_SIZE]; // 標記每個點是否被取用

// 初始化集合，每個節點的父節點指向自己，標記為未使用
void make_set() {
    for (int x = 1; x < MAX_SIZE; x++) {
        parent[x] = x;
        flag[x] = false;
    }
}

// 查找集合的根，並使用路徑壓縮優化查找
int find_set(int x) {
    if (x != parent[x])
        parent[x] = find_set(parent[x]);
    return parent[x];
}

// 合併兩個集合（樹），若它們的根不同，將 y 的根設為 x 的根
void union_set(int x, int y) {
    if (x < 1 || x >= MAX_SIZE || y < 1 || y >=
        MAX_SIZE) return; // 加入範圍檢查
    x = find_set(x);
    y = find_set(y);
    if (x != y)
        parent[y] = x;
}

// 檢查所有使用的節點是否屬於同一個集合（單一根節點）
bool single_root(int n) {
    int i = 1;
    while (i <= n && !flag[i]) i++; // 找到第一個被使用的節點
    if (i > n) return true; //
        如果範圍內沒有使用的節點，則為單一根

    int root = find_set(i); // 設定第一個使用的節點的根為判斷基準
    while (i <= n) {
        if (flag[i] && find_set(i) != root) //
            檢查是否所有使用的節點共享同一根
            return false;
        ++i;
    }
    return true;
}

// 判斷輸入是否構成樹的主函式
void process_input() {
    int x, y;
    bool is_tree = true;
    int range = 0;
    int case_num = 1;

    make_set();

    while (scanf("%d %d", &x, &y) != EOF) {
        if (x < 0 && y < 0)
            break;

        if (x == 0 && y == 0) {
            if (is_tree && single_root(range)) //
                檢查結束條件
                printf("Case %d is a tree.\n",
                    case_num++);
            else
                printf("Case %d is not a tree.\n",
                    case_num++);

            is_tree = true; // 重置為新案例
            range = 0;
            make_set(); // 重置並查集
            continue;
        }

        if (x >= MAX_SIZE || y >= MAX_SIZE) { // 檢查範圍
            is_tree = false;
            continue;
```

```
        }

        range = (x > range) ? x : range;
        range = (y > range) ? y : range;
        flag[x] = flag[y] = true;

        if (find_set(x) == find_set(y)) // 若 x 和 y 在同一集合中則形成循環，不構成樹
            is_tree = false;
        else
            union_set(x, y); // 否則合併 x 和 y
    }
}

int main() {
    process_input(); // 呼叫主函式處理輸入
    return 0;
}
```

MST Kuskal 計算最小樹新增無向邊權和

```
//POJ2421
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// 定義邊結構
struct Edge {
    int u; // 起點
    int v; // 終點
    int weight; // 權重
};

// 定義並查集（Union-Find）類別
class UnionFind {
private:
    vector<int> parent; // 父節點

public:
    // 建構子，初始化並查集，每個節點的父節點為自己
    UnionFind(int n) : parent(n + 1) { // 使用 1-based 索引
        for(int i = 0; i <= n; ++i)
            parent[i] = i;
    }

    // 查找集合，並進行路徑壓縮
    int find_set(int x) {
        if(parent[x] != x)
            parent[x] = find_set(parent[x]);
        return parent[x];
    }

    // 合併兩個集合
    void union_set(int x, int y) {
        int fx = find_set(x);
        int fy = find_set(y);
        if(fx != fy)
            parent[fx] = fy;
    }
};

// 比較兩條邊的權重，用於排序
bool compare_edges(const Edge &a, const Edge &b) {
    return a.weight < b.weight;
}
}

// 計算最小生成樹總權重的函式
int calculateMSTWeight(int N, const vector<vector<int>> &P, int M, const vector<pair<int, int>> &preConnected) {
    // 初始化並查集
    UnionFind uf(N);

    // 將已經預先連接的邊加入並查集
    for(int i = 0; i < preConnected.size(); ++i) {
        int a = preConnected[i].first;
        int b = preConnected[i].second;
        uf.union_set(a, b);
    }

    // 收集所有可能的邊
    vector<Edge> edges;
    for(int i = 1; i <= N; i++) {
        for(int j = i + 1; j <= N; j++) { // 確保 j > i 避免重複邊
            if(P[i][j] > 0) { // 只考慮權重大於 0 的邊
                Edge e;
                e.u = i;
                e.v = j;
                e.weight = P[i][j];
                edges.push_back(e);
            }
        }
    }

    // 對所有邊按照權重進行排序
    sort(edges.begin(), edges.end(), compare_edges);

    int total_weight = 0;
    // 遍歷所有邊，選擇不形成環路的邊加入 MST
    for(int i = 0; i < edges.size(); ++i) {
        Edge edge = edges[i];
        if(uf.find_set(edge.u) != uf.find_set(edge.v)) {
            uf.union_set(edge.u, edge.v);
            total_weight += edge.weight;
        }
    }

    return total_weight;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0); // 提高輸入輸出效率

    int N;
    while(cin >> N) { // 讀取節點數量
        // 讀取鄰接矩陣
        vector<vector<int>> P(N + 1, vector<int>(N + 1, 0));
        for(int i = 1; i <= N; i++) { // 1-based 索引
            for(int j = 1; j <= N; j++) {
                cin >> P[i][j];
            }
        }

        // 讀取已經預連接的邊
        int M;
```

```
        cin >> M;
        vector<pair<int, int>> preConnected;
        for(int i = 0; i < M; ++i) {
            int a, b;
            cin >> a >> b;
            preConnected.push_back(make_pair(a, b));
        }

        // 計算最小生成樹的總權重
        int total_MST_weight = calculateMSTWeight(N, P,
            M, preConnected);

        // 輸出 MST 的總權重
        cout << total_MST_weight << "\n";
    }
    return 0;
}
```

MST Prim 計算權和, 線性掃描最小邊, 密稠圖

```
//POJ1258
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

// 取得最小生成樹權重的函式（Prim 算法）
int calculateMSTWeightPrim(int n, const
    vector<vector<int>> &v) {
    int tot = 0; // 用於累計 MST 的總權重
    vector<int> dist(n, INT_MAX); // 距離陣列，初始化為最大值
    vector<bool> use(n, false); // 記錄節點是否已被加入 MST
        的布林陣列

    dist[0] = 0; // 設定起始節點的距離為 0

    // 初始化從起始節點到其他節點的距離
    for (int i = 1; i < n; i++) {
        dist[i] = v[0][i];
    }

    // 主迴圈，用來擴展最小生成樹，直到包含所有節點
    for (int i = 1; i < n; i++) { // 擴展 MST 需要 n - 1 條邊
        int tmp = -1;

        // 線性掃描找出未加入 MST 的節點中距離最小的節點
        for (int k = 1; k < n; k++) {
            if (!use[k] && (tmp == -1 || dist[k] <
                dist[tmp])) {
                tmp = k;
            }
        }

        use[tmp] = true; // 將選中的節點標記為已使用
        tot += dist[tmp]; // 累加選中邊的權重到 MST 的總權重

        // 更新距離陣列 dist，對於所有未加入 MST 的節點
        for (int k = 1; k < n; k++) {
            if (!use[k]) {
                dist[k] = min(dist[k], v[k][tmp]);
            }
        }
    }

    return tot; // 返回最小生成樹的總權重
}

int main() {
    int n;
    while (cin >> n) { // 讀取節點數量
        vector<vector<int>> v(n, vector<int>(n)); //
            鄰接矩陣表示圖
        // 讀取鄰接矩陣
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cin >> v[i][j];
            }
        }

        // 計算最小生成樹的權重
        int total_MST_weight =
            calculateMSTWeightPrim(n, v);

        // 輸出 MST 的總權重
        cout << total_MST_weight << endl;
    }

    return 0;
}
```

SP Warshell 閉包遞移, 二分法計算最長邊最小路徑

```
//UVA534
#include <iostream>
#include <cmath>
#include <iomanip>
#include <vector>

using namespace std;

const int MAX_N = (1 << 9) + 1;
double L[MAX_N][MAX_N]; // 距離矩陣
bool con[MAX_N][MAX_N]; // 連接矩陣，用於表示節點之間是否可達

// 計算青蛙距離的函式（Floyd-Warshall 閉包遞移 + 二分法）
double calculateFrogDistance(int N, const
    vector<double>& x, const vector<double>& y) {
    // 計算距離矩陣 L
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            L[i][j] = sqrt(pow(x[i] - x[j], 2) +
                pow(y[i] - y[j], 2));
        }
    }

    // 使用二分法找最小的最大距離，使得第一塊石頭和第二塊石頭連通
    double l = 0, r = 1e5;
    while (r - l > 1e-5) { // 精度為 1e-5
        double mid = (l + r) / 2;

        // 初始化連接矩陣，判斷在當前 mid 距離下的節點連通情況
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < N; ++j) {
                con[i][j] = (L[i][j] <= mid);
            }
        }

        // Floyd-Warshall 演算法，計算所有節點之間的可達性
        for (int k = 0; k < N; ++k) {
            for (int i = 0; i < N; ++i) {
```

```
            for (int j = 0; j < N; ++j) {
                con[i][j] = con[i][j] || (con[i][k] && con[k][j]);
            }
        }
    }

    // 檢查第一和第二塊石頭是否連通
    if (con[0][1]) {
        r = mid;
    } else {
        l = mid;
    }
}

return l; // 返回最小的最大距離，即青蛙距離
}

int main() {
int N, testCase = 0;
while (cin >> N && N != 0) {
    vector<double> x(N), y(N);

    // 讀取每塊石頭的座標
    for (int i = 0; i < N; ++i) {
        cin >> x[i] >> y[i];
    }

    // 計算並輸出青蛙距離
    double frogDistance = calculateFrogDistance(N, x, y);

    // 輸出結果，保留三位小數
    cout << "Scenario #" << ++testCase << endl;
    cout << "Frog Distance = " << fixed << setprecision(3) << frogDistance << endl;
    cout << endl;
}
return 0;
}
```

SP Dijkstra 重邊判斷, 找最短路徑

```
//POJ2387
#include <iostream>
#include <cstdio>
#include <algorithm>

#define MAX_N 1010
#define INF 1e9

using namespace std;

int w[MAX_N][MAX_N]; // 圖的權重矩陣
int d[MAX_N]; // 距離陣列，用於 Dijkstra 演算法
bool visited[MAX_N]; // 記錄節點是否已訪問
int n, m; // n 表示節點數量，m 表示邊數量

// 使用 Dijkstra 演算法找從起點到各節點的最短路徑
void dijkstraShortestPath(int s) {
    // 初始化距離和訪問狀態
    for (int i = 1; i <= n; ++i) {
        d[i] = INF; // 將所有節點距離初始化為無限大
        visited[i] = false; // 將所有節點標記為未訪問
    }
    d[s] = 0; // 起始節點的距離設為 0

    // 主要迴圈：逐步擴展最短路徑
    for (int i = 1; i <= n; ++i) {
        int x = -1;
        // 找出距離最小的未訪問節點
        for (int j = 1; j <= n; ++j) {
            if (!visited[j] && (x == -1 || d[j] < d[x])) {
                x = j;
            }
        }
        visited[x] = true; // 將選中的節點標記為已訪問

        // 更新與節點 x 直接相連的其他未訪問節點的最短距離
        for (int j = 1; j <= n; ++j) {
            if (!visited[j] && w[x][j] != INF) { // 確保節點 j 未訪問且有路徑
                d[j] = min(d[j], d[x] + w[x][j]); // 更新最短距離
            }
        }
    }
}

int main() {
    scanf("%d%d", &m, &n); // 輸入邊數和節點數
    // 初始化權重矩陣，所有邊的初始權重設為無限大（無邊連接）
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j) {
            w[i][j] = INF;
        }
    }

    // 讀取邊的資料，並考慮重邊情況，保留權重最小的邊
    for (int i = 0; i < m; ++i) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c); // 輸入邊的兩端節點和權重
        if (w[a][b] > c) {
            w[a][b] = w[b][a] = c; // 更新為最小權重，適用於無向圖
        }
    }

    dijkstraShortestPath(1); // 從節點 1 開始運行 Dijkstra 演算法

    printf("%d\n", d[n]); // 輸出從節點 1 到節點 n 的最短距離
    return 0;
}
```

SP Dijkstra 二分搜尋最佳初始值

```
//UVA1027
#include <bits/stdc++.h>
using namespace std;

// 將字母字符轉換成索引
int turn(char x) {
    if (x >= 'A' && x <= 'Z') return x - 'A' + 1; // 'A'-'Z' -> 1-26
    if (x >= 'a' && x <= 'z') return x - 'a' + 27; // 'a'-'z' -> 27-52
    return -1; // 無效字符
}
```

```cpp

// 檢查從起點到終點的貨物量，並計算經過給定的貨物量後能到達的最大貨物量
int check(int from, int to, int o, bool go[][55]) {
    int g[55];
    memset(g, 0, sizeof(g)); // 初始化每個節點的貨物量為 0
    bool flag[55];
    memset(flag, false, sizeof(flag)); // 初始化訪問標記為 false
    g[from] = o; // 設定起點的初始貨物量

    while (true) {
        int w = 0, next = -1;
        // 找到貨物量最大的未標記節點
        for (int i = 1; i <= 52; i++) {
            if (!flag[i] && g[i] > w) {
                next = i;
                w = g[i];
            }
        }

        if (next == -1) break; // 無節點可以處理，退出
        flag[next] = true; // 標記當前節點為已處理

        // 更新相鄰節點的貨物量
        for (int i = 1; i <= 52; i++) {
            if (go[next][i]) {
                int reduction;
                if (i < 27) {
                    reduction = (w + 19) / 20; // 對應於 ceil(w / 20)
                } else {
                    reduction = 1;
                }
                int tmp = w - reduction;
                if (tmp > g[i]) {
                    g[i] = tmp; // 更新節點貨物量
                }
            }
        }
    }

    return g[to]; // 返回到達目標節點的貨物量
}

// 使用二分搜尋法找到最小的初始貨物量，使得起始貨物量從 from 到 to 的最小貨物量達到 Tot
int findMinimalStartingCargo(int from, int to, int Tot, bool go[][55]) {
    int l = 1, r = (1 << 20); // 搜尋範圍設置為 1 到 1048576

    while (l < r) {
        int mid = (l + r - 1) / 2; // 計算中間值
        int cargo = check(from, to, mid, go); // 對應貨物量進行檢查
        if (cargo >= Tot) {
            r = mid; // 若符合要求，嘗試更小的初始貨物量
        } else {
            l = mid + 1; // 不符合要求，增加初始貨物量
        }
    }

    // 驗證結果：確認二分搜尋找到的貨物量能滿足要求
    if (check(from, to, l, go) >= Tot) {
        return l; // 找到最小貨物量
    } else {
        return -1; // 如果無法滿足，返回 -1 表示不可能
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0); // 快速輸入輸出

    int T; // 連接數
    int tot = 0; // 測試案例計數

    while (cin >> T) {
        if (T == -1) break; // 結束條件

        // 初始化鄰接矩陣
        bool go[55][55];
        memset(go, false, sizeof(go));

        // 讀取 T 條連接
        for (int i = 0; i < T; i++) {
            char x, y;
            cin >> x >> y;
            int a = turn(x);
            int b = turn(y);
            if (a == -1 || b == -1) continue; // 無效字符，跳過該連接
            go[a][b] = go[b][a] = true; // 雙向連接
        }

        // 讀取 Tot（需求的最小貨物量）
        int Tot;
        cin >> Tot;

        // 讀取起點和終點字符
        char fromChar, toChar;
        cin >> fromChar >> toChar;
        int from = turn(fromChar);
        int to = turn(toChar);

        if (from == -1 || to == -1) {
            tot++;
            cout << "Case " << tot << ": Impossible\n";
            continue;
        }

        // 使用二分搜尋法計算最小初始貨物量
        int result = findMinimalStartingCargo(from, to, Tot, go);
        tot++;
        if (result != -1) {
            cout << "Case " << tot << ": " << result << "\n";
        } else {
            cout << "Case " << tot << ": Impossible\n";
        }
    }

    return 0;
}
```

SP SPFA 求負權最短路徑

```cpp
//POJ3259
#include <iostream>
```

```cpp
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;

const int MAX = 501;
const int INF = 1e9; // 定義一個大常數作為無限大
int map[MAX][MAX]; // 鄰接矩陣，用來儲存圖的權重
int dis[MAX]; // 距離陣列
int n, m, w, s, e, t; // 節點數、邊數、起點和終點以及權重

// 使用 SPFA 演算法檢測圖中是否存在負環的函式
bool detectNegativeCycleUsingSPFA(int start) {
    bool flag[MAX] = {0}; // 標記節點是否在隊列中
    int count[MAX] = {0}; // 每個節點進入隊列的次數
    queue<int> q;
    q.push(start); // 從起點開始
    dis[start] = 0; // 起點的距離設為 0

    while (!q.empty()) {
        int curr = q.front();
        q.pop();

        // 遍歷所有相鄰節點，更新最短距離
        for (int i = 1; i <= n; i++) {
            // 如果節點相連且存在有效邊
            if (map[curr][i] < INF) {
                // 檢查從當前節點到相鄰節點的最短距離
                if (dis[i] > map[curr][i] + dis[curr]) {
                    dis[i] = map[curr][i] + dis[curr]; // 更新最短距離
                    if (!flag[i]) {
                        q.push(i); // 將相鄰節點加入隊列
                        flag[i] = true; // 標記節點在隊列中
                    }
                    count[i]++; // 記錄節點的進入次數

                    // 如果節點進入隊列次數超過節點數，則存在負環
                    if (count[i] >= n) return false;
                }
            }
        }
        flag[curr] = false; // 將當前節點標記為不在隊列中
    }
    return true; // 如果無負環，返回 true
}

int main() {
    int f;
    scanf("%d", &f);
    while (f--) {
        // 重置陣列
        memset(dis, 63, sizeof(dis)); // 初始化距離陣列為大值
        memset(map, 127, sizeof(map)); // 初始化鄰接矩陣為大值，表示無連接

        // 輸入節點、邊、起點和終點數
        scanf("%d %d %d", &n, &m, &w);
        for (int i = 0; i < m; i++) {
            scanf("%d %d %d", &s, &e, &t);
            // 若存在重邊，保留最小的權重
            map[s][e] = min(map[s][e], t);
            map[e][s] = min(map[e][s], t);
        }

        for (int i = 0; i < w; i++) {
            scanf("%d %d %d", &s, &e, &t);
            map[s][e] = -t; // 設定負權邊代表 wormhole
        }

        // 使用 SPFA 檢測負環
        if (detectNegativeCycleUsingSPFA(1))
            printf("NO\n"); // 沒有負環
        else
            printf("YES\n"); // 存在負環
    }
    return 0;
}
```

BG HA(匈牙利算法) 二分圖最大匹配

```cpp
//POJ1469
#include <iostream>
#include <cstdio>
#include <cstring>

using namespace std;

int a[110][310]; // 二分圖的鄰接矩陣，a[i][j] 表示課程 i 和學生 j 之間的連接
int n, m; // n 表示課程數量，m 表示學生數量
int vis[310]; // 訪問標記陣列，用於記錄當前匹配過程中訪問過的學生
int pre[310]; // 匹配陣列，pre[j] 表示與學生 j 匹配的課程

// 使用 DFS 尋找增廣路，試圖找到未匹配的課程
bool dfs(int x) {
    for (int t = 1; t <= m; ++t) {
        if (a[x][t] && !vis[t]) { // 如果課程 x 與學生 t 有邊且學生 t 未訪問
            vis[t] = 1; // 將學生 t 標記為已訪問
            if (pre[t] == 0 || dfs(pre[t])) { // 若學生 t 尚未匹配或學生 t 原先匹配的課程可以找到替代路徑
                pre[t] = x; // 設定 (x, t) 為匹配
                return true; // 匹配成功
            }
        }
    }
    return false; // 匹配失敗
}

// 使用匈牙利算法進行二分圖的最大匹配
int maxBipartiteMatching() {
    memset(pre, 0, sizeof(pre)); // 初始化匹配儲存陣列 pre
    int matchCount = 0; // 匹配計數器

    for (int i = 1; i <= n; ++i) {
        memset(vis, 0, sizeof(vis)); // 對每個課程重新初始化訪問標記陣列
        if (dfs(i)) matchCount++; // 如果課程 i 可以匹配，匹配計數增加
    }
    return matchCount; // 返回最大匹配數量
}

int main() {
    int T;
    scanf("%d", &T); // 輸入測試案例數量
    while (T--) {
        // 輸入課程和學生數量
```

```
        scanf("%d%d", &n, &m);
        memset(a, 0, sizeof(a)); // 初始化二分圖的鄰接矩陣

        // 建立二分圖的鄰接矩陣
        for (int i = 1; i <= n; ++i) {
            int t;
            scanf("%d", &t); // 輸入每門課程的學生數量
            while (t--) {
                int j;
                scanf("%d", &j); // 輸入學生編號
                a[i][j] = 1; // 標記課程 i 和學生 j 之間的連接
            }
        }

        // 使用匈牙利算法進行最大匹配
        int result = maxBipartiteMatching();

        // 檢查匹配數量是否等於課程數量
        if (result == n) {
            printf("YES\n"); // 若所有課程都能匹配，輸出 "YES"
        } else {
            printf("NO\n"); // 否則，輸出 "NO"
        }
    }
    return 0;
}
```

BG 最大匹配數求邊覆蓋

```
//URAL1109
#include <iostream>
#include <cstring>
using namespace std;

const int V = 1100; // 節點最大數量

int n, m, k; // n: A 集合的代表數, m: B 集合的代表數, k: 代表之間的配對數
int pre[V]; // 匹配儲存陣列，pre[j] 表示與 B 集合中 j 節點匹配的 A 集合節點
bool v[V]; // 訪問標記陣列，紀錄當前匹配過程中訪問過的 B 集合節點
bool a[V][V]; // 二分圖的鄰接矩陣，a[i][j] 表示 A 集合中的 i 與 B 集合中的 j 之間是否有邊

// DFS 尋找增廣路來進行最大匹配
bool dfs(int i) {
    for (int j = 1; j <= m; j++) {
        // 檢查節點 j 是否未訪問，且 i 和 j 之間有邊
        if (!v[j] && a[i][j]) {
            v[j] = 1; // 標記 j 為已訪問
            // 若 j 未匹配，或 j 的匹配節點可以找到其他匹配
            if (pre[j] == 0 || dfs(pre[j])) {
                pre[j] = i; // 設定 (i, j) 為匹配
                return true; // 匹配成功
            }
        }
    }
    return false; // 匹配失敗
}

// 計算二分圖的最小邊覆蓋數
int minEdgeCover() {
    memset(pre, 0, sizeof(pre)); // 初始化匹配儲存陣列
    int matchCount = 0; // 匹配計數

    for (int i = 1; i <= n; i++) {
        memset(v, 0, sizeof(v)); // 初始化訪問標記陣列
        if (dfs(i)) matchCount++; // 若 i 可以匹配，則匹配計數增加
    }

    // 最小邊覆蓋數 = N + M - 最大匹配數
    return n + m - matchCount;
}

int main() {
    cin >> n >> m >> k; // 輸入 A 和 B 的代表數量以及配對數
    memset(a, 0, sizeof(a)); // 初始化鄰接矩陣為 0，表示無邊

    // 讀取配對資料並設置鄰接矩陣
    for (int i = 1; i <= k; i++) {
        int x, y;
        cin >> x >> y; // 輸入 A 集合的代表 x 與 B 集合的代表 y 的配對
        a[x][y] = 1; // 標記 i 與 j 之間有邊
    }

    // 計算二分圖的最小邊覆蓋數
    int result = minEdgeCover();

    // 輸出最小邊覆蓋數
    cout << result << endl;

    return 0;
}
```

BG 二分圖匹配最大化最小值

```
//HDOJ6667
#include <stdio.h>
#include <algorithm>
using namespace std;

const int N = 1e6 + 5;
long long a[N], b[N]; // 數據陣列，a[i] 和 b[i] 分別表示每個學生的兩種數據值

// 計算二分圖匹配的最大化最小值
long long maximizeMinMatchValue(int n) {
    long long ans1 = 0, ans2 = 0;

    // 計算 a[i] 和 b[i] 的累積和
    for (int i = 1; i <= n; i++) {
        ans1 += a[i]; // 累加所有 a[i] 的值
        ans2 += b[i]; // 累加所有 b[i] 的值
    }

    // 初始答案為兩個累積和的最小值
    long long ans = min(ans1, ans2);

    // 遍歷每個學生，計算匹配方案中的最小可能值
    for (int i = 1; i <= n; i++) {
        // 計算在選擇特定匹配方案下的最小值，並更新 ans
        ans = min(ans, ans1 - (a[i] - (ans2 - b[i])));
    }

    return ans; // 返回最大化的最小值
}

int main() {
```

```
    int T, n;
    scanf("%d", &T); // 輸入測試案例數量
    while (T--) {
        scanf("%d", &n); // 輸入學生數量

        // 讀取每個學生的數據 a[i] 和 b[i]
        for (int i = 1; i <= n; i++) {
            scanf("%lld %lld", &a[i], &b[i]);
        }

        // 計算並輸出結果
        printf("%lld\n", maximizeMinMatchValue(n));
    }
    return 0;
}
```

BG KM 求二分圖最小權和 (負邊)

```
//POJ2195
#include <iostream>
#include <algorithm>
#include <cstring>
#include <climits>

using namespace std;

#define MAXN 102
#define max(x, y) ((x) > (y) ? (x) : (y))

int a[MAXN][MAXN]; // 權重矩陣
int lx[MAXN], ly[MAXN]; // X 集合和 Y 集合的標籤（用於 KM 演算法）
int slack[MAXN], maty[MAXN]; // slack 用於減少更新量，maty 用於紀錄 Y 集合的匹配情況
bool vx[MAXN], vy[MAXN]; // 標記陣列，用於記錄當前匹配過程中的 X 和 Y 集合訪問情況
int lenx, leny; // X 集合和 Y 集合的有效元素數量

// DFS 搜索增廣路，試圖找到最佳匹配
bool search(int u) {
    vx[u] = 1;
    for (int i = 0; i < leny; ++i) {
        if (!vy[i]) {
            int t = lx[u] + ly[i] - a[u][i]; // 計算是否符合增廣路條件
            if (t == 0) { // 若滿足匹配條件
                vy[i] = 1;
                if (maty[i] == -1 || search(maty[i])) {
                    maty[i] = u; // 設定 (u, i) 為匹配
                    return true; // 匹配成功
                }
            } else if (slack[i] > t) {
                slack[i] = t; // 更新 slack 值
            }
        }
    }
    return false; // 匹配失敗
}

// KM 演算法，計算二分圖的最小權和
int minWeightBipartiteMatching(int lenx, int leny) {
    memset(maty, -1, sizeof(maty)); // 初始化 Y 集合的匹配狀態
    memset(ly, 0, sizeof(ly)); // 初始化 Y 集合的標籤
    for (int i = 0; i < lenx; ++i) {
        lx[i] = -INT_MAX;
        for (int j = 0; j < leny; ++j) {
            lx[i] = max(lx[i], a[i][j]); // 初始化 X 集合的標籤為最大權重
        }
    }

    // 對每個 X 集合的元素進行匹配
    for (int i = 0; i < lenx; ++i) {
        fill(slack, slack + leny, INT_MAX); // 初始化 slack 值
        while (1) {
            memset(vx, 0, sizeof(vx));
            memset(vy, 0, sizeof(vy));
            if (search(i)) break; // 若成功匹配則跳出迴圈
            int d = INT_MAX;
            for (int j = 0; j < leny; ++j) {
                if (!vy[j] && d > slack[j]) d = slack[j]; // 找到最小的 slack 值
            }
            for (int j = 0; j < lenx; ++j) {
                if (vx[j]) lx[j] -= d; // 更新 X 集合的標籤
            }
            for (int j = 0; j < leny; ++j) {
                if (vy[j]) ly[j] += d; // 更新 Y 集合的標籤
            }
        }
    }

    int ans = 0;
    for (int i = 0; i < leny; ++i) {
        if (maty[i] != -1) ans += a[maty[i]][i]; // 計算最小權和
    }
    return -ans; // 因為 a[i][j] 存的是負權重，所以返回 -ans 作為最小權和
}

int main() {
    int n, m;
    while (~scanf("%d%d", &n, &m) && n + m) {
        lenx = leny = 0;
        char map[MAXN][MAXN];
        int lx_pos[MAXN], ly_pos[MAXN];

        // 讀取輸入並建立二分圖
        for (int i = 0; i < n; ++i) {
            scanf("%s", map[i]);
            for (int j = 0; j < m; ++j) {
                if (map[i][j] == 'H') {
                    lx_pos[lenx] = i;
                    slack[lenx++] = j;
                } else if (map[i][j] == 'm') {
                    ly_pos[leny] = i;
                    maty[leny++] = j;
                }
            }
        }

        // 初始化權重矩陣 a[i][j]
        for (int i = 0; i < lenx; ++i) {
            for (int j = 0; j < leny; ++j) {
                a[i][j] = -abs(lx_pos[i] - ly_pos[j]) - abs(slack[i] - maty[j]);
            }
```

```
        }

        // 使用 KM 演算法計算最小權和
        printf("%d\n", minWeightBipartiteMatching(lenx,
            leny));
    }
    return 0;
}
```

Flow EK 求最大流

```
//POJ1459
#include <stdio.h>
#include <math.h>
#include <memory.h>

int n, np, nc, m, s, t;
int fa[104], q[104], f[104][104], c[104][104]; //
    fa[]儲存路徑，q[]為佇列，f[i][j]記錄流量，c[i][j]記錄容量

// Edmonds-Karp 演算法，用來計算最大流
int calculateMaxFlow() {
    int ans = 0; // 儲存最大流結果
    while (1) {
        int qs = 0, qt = 1;
        q[qt] = s;
        memset(fa, 0, sizeof(fa)); // 初始化增廣路徑
        fa[s] = s; // 起點的前驅設為自己

        // BFS 尋找增廣路徑
        while (qs < qt && fa[t] == 0) {
            int i = q[++qs];
            for (int j = 1; j <= t; j++) {
                if (fa[j] == 0) { // 若節點 j 尚未被訪問
                    if (f[i][j] < c[i][j]) { //
                        正向邊有剩餘容量
                        fa[j] = i; // 設定前驅節點
                        q[++qt] = j; // 將節點 j 加入佇列
                    } else if (f[j][i] > 0) { //
                        反向邊有可減少流量
                        fa[j] = -i; // 使用負號表示反向流
                        q[++qt] = j;
                    }
                }
            }
        }

        // 若無增廣路則結束
        if (fa[t] == 0) break;

        // 找到增廣路中最小容量的邊
        int doo = 1000000000;
        int i = t;
        while (i != s) {
            int d;
            if (fa[i] > 0) {
                d = c[fa[i]][i] - f[fa[i]][i];
            } else {
                d = f[i][-fa[i]];
            }
            if (d < doo) doo = d;
            i = abs(fa[i]);
        }

        // 沿增廣路增加流量
        ans += doo;
        i = t;
        while (i != s) {
            if (fa[i] > 0) {
                f[fa[i]][i] += doo; // 正向增加流量
            } else {
                f[i][-fa[i]] -= doo; // 反向減少流量
            }
            i = abs(fa[i]);
        }
    }
    return ans; // 返回最大流
}

int main() {
    int i, u, v, cc;
    while (scanf("%d%d%d%d", &n, &np, &nc, &m) == 4) {
        // 輸入節點數、源點數、匯點數和邊數
        s = n + 2; // 設置虛擬源點編號
        t = n + 1; // 設置虛擬匯點編號
        memset(f, 0, sizeof(f)); // 初始化流量矩陣
        memset(c, 0, sizeof(c)); // 初始化容量矩陣

        // 讀取邊的資訊，並設定容量
        for (i = 1; i <= m; i++) {
            while (getchar() != '(');
            scanf("%d,%d)%d", &u, &v, &cc);
            c[u + 1][v + 1] = cc;
        }

        // 讀取源點資訊，將虛擬源點連接到每個源點
        for (i = 1; i <= np; i++) {
            while (getchar() != '(');
            scanf("%d)%d", &u, &cc);
            c[s][u + 1] = cc;
        }

        // 讀取匯點資訊，將每個匯點連接到虛擬匯點
        for (i = 1; i <= nc; i++) {
            while (getchar() != '(');
            scanf("%d)%d", &u, &cc);
            c[u + 1][t] = cc;
        }

        // 執行 Edmonds-Karp 演算法，並輸出最大流
        printf("%d\n", calculateMaxFlow());
    }
    return 0;
}
```

Flow SPFA 求最小費用流, 帶權二分圖轉網路圖

```
//URAL1076
#include <iostream>
#include <vector>
#include <queue>
#include <cstring>
#include <climits>
using namespace std;

const int MAXN = 505;
const int INF = INT_MAX;

struct Edge {
    int to, cap, cost;
```

```
    Edge(int _to, int _cap, int _cost) : to(_to), cap(_cap), cost(_cost) {}
};

class MinCostMaxFlow {
private:
    int n, s, t;
    vector<Edge> edges;
    vector<vector<int>> adj;
    vector<int> dist, prev, prevEdge;
    vector<bool> inQueue;

    // 使用 SPFA 尋找最短路徑，並更新 dist 和 prev 陣列
    bool SPFA() {
        dist.assign(n + 1, INF);
        prev.assign(n + 1, -1);
        prevEdge.assign(n + 1, -1);
        inQueue.assign(n + 1, false);

        queue<int> q;
        q.push(s);
        dist[s] = 0;
        inQueue[s] = true;

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

            for (int eid : adj[u]) {
                Edge& e = edges[eid];
                int v = e.to;

                // 如果邊有剩餘容量，且找到更短路徑
                if (e.cap > 0 && dist[v] > dist[u] + e.cost) {
                    dist[v] = dist[u] + e.cost;
                    prev[v] = u;
                    prevEdge[v] = eid;

                    if (!inQueue[v]) {
                        q.push(v);
                        inQueue[v] = true;
                    }
                }
            }
        }

        return dist[t] != INF;
    }

    // 沿增廣路徑增加流量
    int augment() {
        int minFlow = INF;
        for (int v = t; v != s; v = prev[v]) {
            Edge& e = edges[prevEdge[v]];
            minFlow = min(minFlow, e.cap);
        }

        for (int v = t; v != s; v = prev[v]) {
            Edge& e = edges[prevEdge[v]];
            e.cap -= minFlow;
            edges[prevEdge[v] ^ 1].cap += minFlow;
        }

        return minFlow * dist[t];
    }

public:
    MinCostMaxFlow(int _n) : n(_n) {
        adj.resize(n + 1);
    }

    // 增加一條帶容量和費用的邊到網路
    void addEdge(int from, int to, int cap, int cost) {
        adj[from].push_back(edges.size());
        edges.emplace_back(to, cap, cost);
        adj[to].push_back(edges.size());
        edges.emplace_back(from, 0, -cost);
    }

    // 求解最小費用最大流，返回最小費用
    int minCostMaxFlow(int _s, int _t) {
        s = _s;
        t = _t;
        int totalCost = 0;

        while (SPFA()) { // 當存在增廣路徑時
            totalCost += augment(); // 增加流量並累計費用
        }

        return totalCost;
    }
};

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;

    MinCostMaxFlow mcmf(2 * n + 2); // 建立網路圖，包括 2*n 節點和虛擬源/匯點
    vector<vector<int>> cost(n + 1, vector<int>(n + 1));

    // 讀入成本矩陣，計算每個節點的成本差
    for (int i = 1; i <= n; i++) {
        int sum = 0;
        for (int j = 1; j <= n; j++) {
            cin >> cost[i][j];
            sum += cost[i][j];
        }
        for (int j = 1; j <= n; j++) {
            cost[i][j] = sum - cost[i][j]; // 調整成本使得每條邊的權值表示差異
        }
    }

    int s = 1, t = 2 * n + 2;

    // 建立二分圖到網路圖的映射
    for (int i = 1; i <= n; i++) {
        mcmf.addEdge(s, i + 1, 1, 0); // 源點到左側節點
        mcmf.addEdge(i + n + 1, t, 1, 0); // 右側節點到匯點

        for (int j = 1; j <= n; j++) {
```

```
            mcmf.addEdge(i + 1, j + n + 1, 1,
                cost[i][j]); // 左側節點到右側節點的連接
        }
    }

    cout << mcmf.minCostMaxFlow(s, t) << endl; //
        計算最小費用最大流並輸出結果

    return 0;
}
```

Convex Hull 凸包模板題

```
//UVA11626
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <cmath>
using namespace std;

const double eps = 1e-10;
const int maxn = 100000 + 10;

struct Point {
    double x, y;
    Point() {}
    Point(double x, double y) : x(x), y(y) {}

    // 判斷兩點是否相等
    bool operator==(const Point &rhs) const {
        return fabs(x - rhs.x) < eps && fabs(y - rhs.y)
            < eps;
    }

    // 比較兩點的大小，先比較 x 再比較 y
    bool operator<(const Point &rhs) const {
        return x < rhs.x || (fabs(x - rhs.x) < eps && y
            < rhs.y);
    }
};

typedef Point Vector;

// 向量相減
Vector operator-(Point A, Point B) {
    return Vector(A.x - B.x, A.y - B.y);
}

// 計算向量叉積，若結果為正表示逆時針，負為順時針，0 表示共線
double Cross(Vector A, Vector B) {
    return A.x * B.y - A.y * B.x;
}

// 使用 Andrew's Monotone Chain 算法計算凸包
int convexHull(Point *p, int n, Point *ch) {
    // 排序點集，並去除重複點
    sort(p, p + n);
    n = unique(p, p + n) - p;

    int m = 0; // 凸包頂點數

    // 構建下半部分凸包
    for (int i = 0; i < n; i++) {
        while (m > 1 && Cross(ch[m - 1] - ch[m - 2],
            p[i] - ch[m - 2]) < 0) m--;
        ch[m++] = p[i];
    }

    int k = m; // 下半部分結束的位置

    // 構建上半部分凸包
    for (int i = n - 2; i >= 0; i--) {
        while (m > k && Cross(ch[m - 1] - ch[m - 2],
            p[i] - ch[m - 2]) < 0) m--;
        ch[m++] = p[i];
    }

    // 若點數大於 1，則去除最後一個重複的點
    if (n > 1) m--;

    return m; // 返回凸包頂點數
}

int main() {
    int T;
    scanf("%d", &T); // 讀取測試案例數量
    while (T--) {
        int n;
        scanf("%d", &n); // 讀取點數
        Point p[maxn], ch[maxn]; // 點集和凸包頂點集

        // 讀取每個點的坐標
        for (int i = 0; i < n; i++) {
            scanf("%lf%lf %*c", &p[i].x, &p[i].y);
        }

        // 計算凸包
        int m = convexHull(p, n, ch);

        // 輸出凸包頂點數
        printf("%d\n", m);

        // 輸出凸包頂點
        for (int i = 0; i < m; i++) {
            printf("%.0lf %.0lf\n", ch[i].x, ch[i].y);
        }
    }
    return 0;
}
```