Contents

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

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

const int maxn = 100000 + 5;

class DisjointSet {
public:
    std::vector<int> parent;

    // º l¤ id¶°¡A±N¨C ` Iªº® [¸m¬° -1
    DisjointSet(int size) : parent(size, -1) {}

    // ¬dß䮢¦Xªº® A¨öi¦ |# Y
    int find(int x) {
        if (parent[x] < 0)
            return x;
        return parent[x] = find(parent[x]); // ¸ |# Y¡A±N x ª½±µ³s±µ¨ ` I
    }

    // ±N¨ °¦X¦X¨
    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            parent[rootX] = rootY; // ±N x ªº® s±µ¨ y ªº®
        }
    }
};

int n, m;

```

```
int main() {
    int loop;
    scanf("%d", &loop); // ´ ¸ Ψ q
    while (loop--) {
        scanf("%d%d", &n, &m);

        // º l¤ id¶°¡A¤j¤p¬° 2 * n¡AΨ Ĺ Y
        DisjointSet dsu(2 * n + 1);

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

            if (s[0] == 'A') { // ¬d¸O§_¦P¤@O¬£© Ĺ
                if (dsu.find(a) != dsu.find(b)
                    && dsu.find(a) != dsu.find(b
                    + n))
                    printf("Not sure yet.\n");
                else if (dsu.find(a) ==
                    dsu.find(b))
                    printf("In the same gang.\n");
                else
                    printf("In different
                        gangs.\n");
            } else { // ¼O a ©M b ¬°¼Ĺ O¬£
                if (dsu.find(a) != dsu.find(b +
                    n)) {
                    dsu.unionSets(a, b + n); // a
                        ºº¼ĤH¬O b
                    dsu.unionSets(b, a + n); // b
                        ºº¼ĤH¬O a
                }
            }
        }
    }
    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

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

const int maxn = 100000 + 5;

class DisjointSet {
public:
    std::vector<int> parent;

    // º l¤ id¶°¡A±N¨C ` Iºº® [¸m¬° -1
    DisjointSet(int size) : parent(size, -1) {}

    // ¬dŠ鞋¦X ºº® A¨öi¦ |# Y
    int find(int x) {
        if (parent[x] < 0)
            return x;
        return parent[x] = find(parent[x]); //
            ¸ |# Y¡A±N x º¼±µ³s±µ¨ ` I
    }

    // ±N¨ °¦X¦X¨
    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            parent[rootX] = rootY; // ±N x ºº® s±µ¨
                y ºº®
        }
    }
};

int n, m;

int main() {
    int loop;
    scanf("%d", &loop); // ´ ¸ Ψ q
    while (loop--) {
        scanf("%d%d", &n, &m);

        // º l¤ id¶°¡A¤j¤p¬° 2 * n¡AΨ Ĺ Y
        DisjointSet dsu(2 * n + 1);

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

            if (s[0] == 'A') { // ¬d¸O§_¦P¤@O¬£© Ĺ
                if (dsu.find(a) != dsu.find(b)
                    && dsu.find(a) != dsu.find(b
                    + n))
                    printf("Not sure yet.\n");
                else if (dsu.find(a) ==
                    dsu.find(b))
                    printf("In the same gang.\n");
                else
                    printf("In different
                        gangs.\n");
            } else { // ¼O a ©M b ¬°¼Ĺ O¬£
                if (dsu.find(a) != dsu.find(b +
                    n)) {
                    dsu.unionSets(a, b + n); // a
                        ºº¼ĤH¬O b
                    dsu.unionSets(b, a + n); // b
                        ºº¼ĤH¬O a
                }
            }
        }
    }
    return 0;
}
```

Trie 建樹, 修改, 查詢

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

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

// 定義 Trie 結構
struct Trie {
    int sz; // Trie 樹的節點數量

    // 初始化 Trie 樹
    Trie() { sz = 1; memset(ch[0], 0, sizeof(ch[0])); } // 根節點初始化

    // 將字母轉成數字索引
    int idx(char c) { return c - 'a'; }

    // 插入單詞到 Trie 樹
    void insert(char *s) {
        int u = 0, n = strlen(s); // 起始於根節點 u = 0
        for (int i = 0; i < n; i++) {
            int c = idx(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 中的最短前綴
    void query(char *s) {
        int u = 0, n = strlen(s); // 起始於根節點 u = 0
        for (int i = 0; i < n; i++) {
            putchar(s[i]); // 輸出當前字母
            int c = idx(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
    Trie trie; // 建立 Trie 的結構體變數

    // 讀取每個單詞並插入到 Trie
    while (scanf("%s", str[tot]) != EOF) {
        trie.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
        trie.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

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

const int MAXN = 1e6 + 6; // 適當調整大小，根據需要
int cnt; // 記錄匹配模式字串的次數

// 節點結構體定義
struct node {
    int sum; // 該節點的匹配次數
    node *next[26]; // 指向子節點的指標陣列
    node *fail; // 失敗指針

    node() : sum(0), fail(nullptr) {
        for(int i = 0; i < 26; i++) next[i] = nullptr;
    }
};
node *root;
char key[70];
char pattern[MAXN];
int N;

// 插入模式字串到 Trie 樹中
void Insert(char *s)
{
    node *p = root; // 開始於 Trie 樹的根節點
    for (int i = 0; s[i]; i++) {
        int x = s[i] - 'a'; // 計算字元索引
        if (p->next[x] == nullptr) {
            p->next[x] = new node(); // 創建新節點
        }
        p = p->next[x]; // 移動到子節點
    }
    p->sum++; // 該節點匹配次數加 1
```

```
}

// 建立失敗指針
void build_fail_pointer()
{
    queue<node*> q; // 使用 C++ 的隊列
    root->fail = nullptr; // 根節點的失敗指針指向空

    // 將根節點的子節點加入隊列並設置失敗指針
    for (int i = 0; i < 26; i++) {
        if (root->next[i] != nullptr) {
            //
                如果根的某子節點存在，將該子節點的失敗指針設為根節點
            root->next[i]->fail = root;
            q.push(root->next[i]); //
                將該子節點加入隊列
        }
        else {
            root->next[i] = root; //
                優化，缺失的邊指向根節點
        }
    }

    // BFS 建立失敗指針
    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) {
                // 若存在子節點
                // 設置失敗指針
                node* fail_node = current->fail;
                    // 從當前節點的失敗指針開始
                // 找到某個祖先節點的匹配邊
                while (fail_node != nullptr &&
                    fail_node->next[i] ==
                    nullptr)
                    fail_node = fail_node->fail;
                        // 繼續沿著失敗指針向上
                if (fail_node == nullptr)
                    current->next[i]->fail =
                        root; // 若找不到則指向根節點
                else
                    current->next[i]->fail =
                        fail_node->next[i]; //
                        否則設置為找到的節點
                q.push(current->next[i]); //
                    將該子節點加入隊列
            }
            else {
                //
                    若當前節點缺少某字母的邊，將其指向失敗指針的相應子節點
                current->next[i] =
                    current->fail->next[i];
            }
        }
    }
}


// 在目標字串中運行 AC 自動機，進行多模式匹配
void ac_automation(char *ch) {
    node *p = root; // 從根節點開始
    int len = strlen(ch); // 目標字串的長度
    for (int i = 0; i < len; i++) {
        int x = ch[i] - 'a'; // 當前字元索引
        while (p->next[x] == root && p != root)
            p = p->fail;

        p = p->next[x];
        if (!p)
            p = root;

        node *temp = p;
        while (temp != root) { //
            往上沿失敗指針累計所有匹配結果
            if (temp->sum >= 0) { // 如果是匹配節點
                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; // 沿失敗指針往上跳轉
        }
    }
}

int main()
{
    int T; // 測試案例數量
    cin >> T;
    while (T--)
    {
        // 建立根節點
        root = new node();
        // 讀取模式字串數量
        cin >> N;
        cin.ignore(); // 忽略換行符

        for (int i = 1; i <= N; i++)
        {
            // 讀取模式字串
            cin.getline(key, sizeof(key));
            Insert(key); // 將模式字串插入到 Trie 樹中
        }
        // 讀取目標字串
        cin.getline(pattern, sizeof(pattern));
        cnt = 0;
        build_fail_pointer(); // 建立失敗指針
        ac_automation(pattern); // 使用 AC
            自動機進行匹配
        cout << cnt << "\n"; //
            輸出匹配到的模式字串次數
    }
    return 0;
}
```

```
/*
輸入範例:
1
5
she
he
say
shr
her
yasherhs

預期輸出:
3
*/
```

BST 有序樹轉二元樹, 數組模擬樹

```
#include <iostream>
#include <string>
using namespace std;

string s;
int i, n = 0, height1, height2;

void work(int level1, int level2) {
    int tempson = 0;
    while (s[i] == 'd') {
        i++;
        tempson++;
        work(level1 + 1, level2 + tempson);
    }
    height1 = max(height1, level1);
    height2 = max(height2, level2);
    if (s[i] == 'u') i++;
}

int main() {
    while (cin >> s && s != "#") {
        i = height1 = height2 = 0;
        work(0, 0);
        cout << "Tree " << ++n << ": " <<
            height1 << " => " << height2 <<
            endl;
    }
    return 0;
}
```

BST Stack 後序轉二元樹

```
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

const int maxn = 11000;

struct node {
    int l, r;
    char c;
} e[maxn];

int cnt;
char s[maxn];

void initial() {
    int len = strlen(s);
    for (int i = 0; i <= len; i++) {
        e[i].l = e[i].r = -1;
    }
    cnt = 0;
}

void solve() {
    int len = strlen(s);
    stack<int> v;
    for (int i = 0; i < len; i++) {
        if (s[i] >= 'a' && s[i] <= 'z') {
            e[cnt].c = s[i];
            v.push(cnt);
            cnt++;
        } else {
            int r = v.top();
            v.pop();
            int l = v.top();
            v.pop();
            e[cnt].l = l;
            e[cnt].r = r;
            e[cnt].c = s[i];
            v.push(cnt);
            cnt++;
        }
    }
}

void output() {
    string ans;
    queue<int> q;
    q.push(cnt - 1);
    while (!q.empty()) {
        int st = q.front();
        q.pop();
        ans.push_back(e[st].c);
        if (e[st].l != -1) q.push(e[st].l);
        if (e[st].r != -1) q.push(e[st].r);
    }
    reverse(ans.begin(), ans.end());
    printf("%s\n", ans.c_str());
}

int main() {
    while (scanf("%s", s) != EOF) {
        initial();
        solve();
        output();
    }
    return 0;
```

```
}
```

BST 前序加中序找出後序

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

char preord[30], inord[30];

int read_case() {
    if (scanf("%s %s", preord, inord) != 2)
        return 0;
    return 1;
}

void recover(int preleft, int preright, int
    inleft, int inright) {
    // 首先根據前序字串中的根節點判斷樹結構，計算左右子樹
    int root, leftsize, rightsize;

    for (root = inleft; root <= inright;
        root++) {
        if (preord[preleft] == inord[root])
            break; // 找到根的位置
    }

    leftsize = root - inleft;
    rightsize = inright - root;

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

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

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

void solve_case() {
    int n = strlen(preord);
    recover(0, n - 1, 0, n - 1);
    printf("\n");
}

int main() {

    while (read_case()) solve_case();
    return 0;
}
```

BST 後序二分搜尋樹還原

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

int n; // 節點總數
int a[3010]; // 儲存後序遍歷

void solve(int l, int r) { // l 和 r
    是左子樹和右子樹的範圍

    if (l > r) return; // 如果範圍無效則返回

    int i = l;
    while (i < r && a[i] < a[r]) i++; //
        找到分界點 i，使得左子樹的元素都小於 a[r]

    if (i < r ) solve(i, r - 1); // 遞迴處理右子樹
    if (l < i ) solve(l, i - 1); // 遞迴處理左子樹

    printf("%d\n", a[r]); // 輸出根節點
}

int main() {
    scanf("%d", &n); // 輸入節點總數
    for (int i = 0; i < n; ++i) // 輸入後序遍歷
        scanf("%d", &a[i]);

    solve(0, n - 1); // 計算並輸出右子樹-左子樹-根的遍歷
    return 0;
}
```

BST 建立結構指標二元樹, 前序尋訪

```
#include <stdio.h>

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

void add(BT &T, int val) { // 將順序值 val
    插入二元搜尋樹
    if (T == NULL) { // 若 T 為空，則找到插入位置
        T = new binTreeNode(); //
            申請記憶體，建構儲存 val 的葉節點
        T->data = val;
        T->lchild = T->rchild = NULL;
    } else if (T->data > val) { // 若 val
        小於根節點值，則沿左子樹方向尋找插入點
        add(T->lchild, val);
    } else { // 若 val
        不小於根節點值，則沿右子樹方向尋找插入位置
        add(T->rchild, val);
    }
}

void preOrder(BT T, bool flag) { //
    前序輸出樹的順序，參數 flag 為首節點標誌
    if (T == NULL)
        return;
```

```
    else {
        if (!flag) // 若節點非首節點，則尾隨空格
            printf(" ");
        printf("%d", T->data); // 輸出 T 的順序值
        preOrder(T->lchild, 0); //
             分別遞迴左子樹和右子樹
        preOrder(T->rchild, 0);
    }
}

int main() {
    BT T;
    int n, v;
    while (~scanf("%d", &n)) { //
         二元搜尋樹的根節點數量
        T = NULL; // 初始化二元搜尋樹
        for (int i = 0; i < n; i++) {
            scanf("%d", &v); // 輸入一個個順序值
            add(T, v); // 插入二元搜尋樹
        }
        preOrder(T, 1); // 按照前序搜尋的順序輸出樹的順序
        printf("\n");
    }
    return 0;
}
```

BST Heap priority queue 插入取出調整

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

using namespace std;

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

// Structure to hold the information
struct Info {
    char name[maxs];
    int para, pri, t;
};

Info p[maxn];
int used = 0; // Next available index in p[]
int cnt = 0; // Counter to maintain insertion order

// Comparator for the priority queue
struct Compare {
    bool operator()(const int a, const int b)
         const {
        if (p[a].pri != p[b].pri)
            return p[a].pri > p[b].pri; //
                 Min-heap based on pri
        return p[a].t > p[b].t; // If pri equal,
             earlier t has higher priority
    }
};

int main(void) {
    char command[maxs];

    // Define the priority queue with the custom comparator
    priority_queue<int, vector<int>, Compare>
         pq;

    // Read commands until end of file
    while (scanf("%s", command) != EOF) {
        if (strcmp(command, "GET") == 0) { // GET
             command
            if (!pq.empty()) {
                int top_idx = pq.top(); // Get the
                     index of the highest priority element
                pq.pop(); // Remove it from the queue
                printf("%s %d\n",
                     p[top_idx].name,
                     p[top_idx].para);
            } else {
                printf("EMPTY QUEUE!\n");
            }
        } else { // Insert command
            // Read the name, para, and pri for the new
                 Info object
            scanf("%s %d %d", p[used].name,
                 &p[used].para, &p[used].pri);
            p[used].t = cnt++; // Assign and
                 increment the insertion order
            pq.push(used++); // Push the index into
                 the priority queue
        }
    }

    return 0;
}

/*
GET
PUT msg1 10 5
PUT msg2 10 4
GET
GET
GET
*/
```

BST Treap 樹堆積左旋右旋, 插入刪除

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

struct Node {
    Node *ch[2]; // 左右指標
    int v, r, info; // v 是客戶優先順序，info
         是客戶的編號，r 由 rand() 產生，作為節點的優先順序

```

```
    Node(int v, int info) : v(v), info(info) {
        r = rand(); // 隨機產生節點優先順序
        ch[0] = ch[1] = NULL; // 左右指標為空
    }

    int cmp(int x) { // 客戶優先順序 v 與 x 比較大小
        if (v == x) return -1;
        return x < v ? 0 : 1;
    }
};

void rotate(Node *&o, int d) { // 節點 o 旋轉，方向 d = 0 左旋，1 右旋
    Node *k = o->ch[d^1];
    o->ch[d^1] = k->ch[d];
    k->ch[d] = o;
    o = k;
}

void insert(Node *&o, int v, int info) { // 插入一個節點 info，優先順序為 v
    if (o == NULL) {
        o = new Node(v, info); // 若找到插入位置，則客戶作為節點插入
    }
    else {
        // **Corrected Insertion Direction**: Place higher 'v' to the right
        int d = v < o->v ? 0 : 1;
        insert(o->ch[d], v, info);
        if (o->ch[d]->r > o->r) rotate(o, d^1); // 若方向 d 的子樹優先順序較小，則進行旋轉
    }
}

void remove(Node *&o, int v) { // 在 o 為根的樹狀堆棧中，刪除優先順序 v 的節點
    if (!o) return;
    int d = o->cmp(v);
    if (d == -1) { // 如果找到該節點
        Node *u = o;
        if (o->ch[0] && o->ch[1]) { // 若 o 有左右子樹，則計算被刪除節點的方向
            int d2 = o->ch[0]->r < o->ch[1]->r ? 1 : 0;
            rotate(o, d2);
            remove(o->ch[d2], v);
        } else { // 若 o 節點僅有一個子樹，則將其子樹取代 o
            o = o->ch[0] ? o->ch[0] : o->ch[1];
            delete u;
        }
    } else {
        remove(o->ch[d], v); // 若 o 節點為葉節點，直接將其刪除
    }
}

int find_max(Node *o) { // 在 o 為根的樹狀堆棧中尋找最大優先順序
    if (!o) return -1; // Handle empty treap
    while (o->ch[1] != NULL) o = o->ch[1];
    printf("%d\n", o->info);
    return o->v;
}

int find_min(Node *o) { // 在 o 為根的樹狀堆棧中尋找最小優先順序
    if (!o) return -1; // Handle empty treap
    while (o->ch[0] != NULL) o = o->ch[0];
    printf("%d\n", o->info);
    return o->v;
}

int main() {
    int op;
    Node *root = NULL;
    while (scanf("%d", &op) == 1 && op) {
        if (op == 1) { // 若輸入為新增客戶
            int v, info;
            scanf("%d%d", &info, &v);
            insert(root, v, info);
        } else if (op == 2) { // 若輸入為最大優先順序
            if (root == NULL) {
                printf("0\n");
                continue;
            }
            int v = find_max(root);
            if (v != -1) remove(root, v);
        } else if (op == 3) { // 若輸入為最小優先順序
            if (root == NULL) {
                printf("0\n");
                continue;
            }
            int v = find_min(root);
            if (v != -1) remove(root, v);
        }
    }
    return 0;
}

/*
Sample Input:
2
1 20 14
1 30 3
2
1 10 99
3
2
2
0

Expected Output:
0
20
30
10
0
```

```
*/
```

BST Treap rope 結構操作字串修改 Treap

```cpp
#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 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
*/

using namespace std;

rope<char> ro, tmp; // 定義 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 main() {
    int n, op, p, c, d, v, cnt;
    cin >> n;
    d = 0;
    cnt = 1;
    while (n--) {
        cin >> op;

        if (op == 1) { // 插入命令
            cin >> p >> 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; // 將版本 ro 儲存到 l[cnt], 版本計數 + 1
        } else if (op == 2) { // 刪除命令
            cin >> p >> 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 位置開始的 c 個字元

            l[cnt++] = ro; // 將版本 ro 儲存到 l[cnt], 版本計數 + 1
        } else { // 列印命令
            cin >> v >> p >> c;

            p -= d; v -= d; c -= d; // 計算相對位置和長度
            tmp = l[v].substr(p-1, c); // 擷取版本 v 中從 p 位置開始長度為 c 的子字串
            d += count(tmp.begin(), tmp.end(), 'c'); // 計算子字串 tmp 中 'c' 的出現次數
            cout << tmp << "\n"; // 輸出子字串 tmp
        }
    }

    return 0;
}

/*
6
1 0 abcdefgh
2 4 3
3 1 2 5
*/
```

BST 霍夫曼樹最小代價 Min Heap

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

const int maxn = 1e5 + 100;
typedef long long ll;
queue<ll> q1, q2;
ll a[maxn];
ll t, n;
bool Hufman(int x) {
    // 清空 q1 和 q2 佇列
    while (!q1.empty()) q1.pop();
    while (!q2.empty()) q2.pop();

    int tt = 0;
    // 模擬 k 元霍夫曼樹: 計算要使用的虛葉節點數
    if ((n - 1) % (x - 1) != 0) tt = (x - 1) - (n - 1) % (x - 1);

    // 將虛葉節點加入 q1
    for (int i = 1; i <= tt; i++) q1.push(0);
    for (int i = 1; i <= n; i++) q1.push(a[i]); // 將序列元素加入 q1

    ll sum = 0;
    while (1) {
        ll tem = 0;
        for (int i = 1; i <= x; i++) { // 每次取出 x 個元素
            if (q1.empty() && q2.empty()) break;
            if (q1.empty()) {
                tem += q2.front();
                q2.pop();
            } else if (q2.empty()) {
                tem += q1.front();
                q1.pop();
            } else if (q1.front() < q2.front()) {
                tem += q1.front();
                q1.pop();
            } else {
                tem += q2.front();
```

```
            q2.pop();
        }
    }
    sum += tem;
    if (q1.empty() && q2.empty()) break;
    q2.push(tem);
    if (sum > t) return 0;
  }
  return sum <= t;
}

int main() {
  int T;
  scanf("%d", &T); // 測試案例數量
  while (T--) {
    scanf("%lld%lld", &n, &t); // 輸入 n 和 t
    for (int i = 1; i <= n; i++)
      scanf("%lld", &a[i]); // 輸入每個序列元素
    sort(a + 1, a + 1 + n); //
      排序以便進行霍夫曼合併

    int st = 2, en = n;
    while (st < en) { // 使用二分法找最小的 k 值
      int mid = (st + en) / 2;
      if (Hufman(mid)) en = mid;
      else st = mid + 1;
    }
    printf("%d\n", st); // 輸出最小的 k 值
  }
  return 0;
}
```

Graph BFS 狀態空間搜尋最短路徑

```
#include <iostream>
#include <queue>
#include <cstring>
// #include <cmath> // Removed since we're no longer using
    pow
using namespace std;

// Define the node struct with an explicit constructor
struct node {
  int k, step;
  node(int k_val, int step_val) : k(k_val),
      step(step_val) {}
};

// Precompute powers of 10 for digit manipulation
const int power10_arr[4] = {1000, 100, 10, 1};

bool p[10000], s_array[10000]; // Renamed 's' to
    's_array' to avoid confusion

// Sieve of Eratosthenes to generate prime numbers up to n
void make_sieve(int n) {
  memset(p, 0, sizeof(p));
  p[0] = 1; // 0 is not a prime
  p[1] = 1; // 1 is not a prime
  for (int i = 2; i <= n; i++) {
    if (!p[i]) {
      for (int j = i * i; j <= n; j += i)
        p[j] = 1; // Mark multiples of i as
            non-prime
    }
  }
}

// Function to change the digit at a specific position
int change_digit(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];
}

int main() {
  // Optimize I/O operations
  ios::sync_with_stdio(false);
  cin.tie(NULL);

  make_sieve(9999); // Generate primes up to 9999

  int tot;
  cin >> tot;
  while (tot--) {
    int x, y;
    cin >> x >> y;

    // Validate that both x and y are 4-digit primes
    if (x < 1000 || x > 9999 || y < 1000 ||
        y > 9999 || p[x] || p[y]) {
      cout << "Impossible" << endl;
      continue;
    }

    // Initialize the BFS queue and visited array
    queue<node> q;
    q.push(node(x, 0)); // Use constructor to
        initialize node
    memset(s_array, 0, sizeof(s_array));
    s_array[x] = 1;
    int ans = -1;

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

      if (cur.k == y) {
        ans = cur.step;
        break;
      }

      for (int i = 1; i <= 4; i++) { //
          Change each digit position
```

```
                for (int j = 0; j <= 9; j++) {
                    // Skip if trying to set the first digit to 0 or if the digit is the same
                    if ((i == 1 && j == 0) || ((cur.k / power10_arr[i - 1]) % 10) == j)
                        continue;

                    int tk = change_digit(cur.k, i, j);

                    // Check if the new number is a prime, within range, and hasn't been visited
                    if (tk >= 1000 && tk <= 9999 && !p[tk] && !s_array[tk]) {
                        s_array[tk] = 1;
                        q.push(node(tk, cur.step + 1)); // Use constructor to initialize node
                    }
                }
            }
        }

        if (ans >= 0)
            cout << ans << "\n";
        else
            cout << "Impossible\n";
    }

    return 0;
}
```

Graph DFS 走訪, 建無向相鄰矩陣

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

int map[6][6]; // 無向圖的相鄰矩陣

// 產生無向圖的相鄰矩陣
void makemap() {
    memset(map, 0, sizeof(map));
    for (int i = 1; i <= 5; i++) {
        for (int j = 1; j <= 5; j++) {
            if (i != j) map[i][j] = 1;
        }
    }
    map[4][1] = map[1][4] = 0;
    map[4][2] = map[2][4] = 0;
}

// 深度優先搜索，找到所有可能的存取路徑
void dfs(int x, int k, string s) {
    s += char(x + '0'); // 將當前節點 x 加入存取序列

    if (k == 8) { // 如果已完成一筆順序
        cout << s << endl;
        return;
    }

    for (int y = 1; y <= 5; y++) { // 依照節點順序訪問相鄰節點
        if (map[x][y]) {
            map[x][y] = map[y][x] = 0; // 設定邊為已訪問
            dfs(y, k + 1, s); // 遞迴搜索
            map[x][y] = map[y][x] = 1; // 恢復邊的狀態
        }
    }
}

int main() {
    makemap(); // 產生無向圖的相鄰矩陣
    dfs(1, 0, ""); // 從節點 1 開始計算所有可能的存取順序
    return 0;
}
```

Graph DFS 剪枝回朔, 狀態空間搜尋

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

int sticks[65]; // 用來存放木棍的長度
int used[65]; // 標記木棍是否被使用
int n, len, sum; // 木棍的數量，木棍的目標長度，所有木棍的總和

// 深度優先搜尋，檢查是否可以用目前的木棍切成長度為 len 的木棍
bool dfs(int i, int l, int t) {
    int j;

    // 若長度達到 len，則成功構成一根木棍
    if (l == 0) {
        t -= len;
        if (t == 0) return true;
        for (i = 0; used[i]; ++i); // 找到下一個未使用的木棍
        used[i] = 1; // 標記該木棍為已使用
        if (dfs(i + 1, len - sticks[i], t))
            return true; // 遞迴切割
        used[i] = 0;
        return false;
    } else {
        for (int j = i; j < n; ++j) { // 從長度遞減順序尋找木棍 j 到木棍 n - 1
            if (j > 0 && sticks[j] == sticks[j - 1] && !used[j - 1]) continue; // 若長度相同且木棍 j - 1 沒有被使用則跳過
```

```
            if (!used[j] && sticks[j] <= l) { // 若木棍沒有被使用且長度小於等於當前長度
                used[j] = 1;
                if (dfs(j + 1, l - sticks[j], t)) return true; // 遞迴嘗試
                used[j] = 0;
                if (sticks[j] == l) break; // 若木棍無法完成切割則跳過
            }
        }
        return false;
    }
}

// 木棍長度的比較函數
bool cmp(const int a, const int b) {
    return a > b;
}

int main() {
    while (cin >> n && n) { // 讀取木棍數量，直至輸入 0 為止
        int 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, cmp); // 按木棍長度降序排列
        bool flag = false;
        for (len = sticks[0]; len <= sum / 2; ++len) { // 在 [sticks[0]..sum/2] 區間搜尋
            if (sum % len == 0) { // 若總長度能被 len 整除
                if (dfs(0, len, sum)) { // 若長度為 len 的木棍能夠切成 n 根木棍，則標記成功
                    flag = true;
                    cout << len << endl; // 輸出木棍的最小可能長度並結束計算
                    break;
                }
            }
        }
        if (!flag) cout << sum << endl; // 若找不到符合條件的木棍長度，則輸出木棍的總長度
    }
    return 0;
}
```

Graph DFS 回朔找尋拓樸排序, 建相鄰有向邊

```
#include <iostream>
#include <vector>
#include <cstring>
#include <queue>
#include <string> // Ensure you have included <string>

using namespace std;

// Function to perform DFS to check if target is reachable from current
bool is_reachable(int current, int target, int N, int g[][100], bool visited[]) {
    if (current == target) return true;
    visited[current] = true;
    for (int i = 0; i < N; i++) {
        if (g[current][i] && !visited[i]) {
            if (is_reachable(i, target, N, g, visited))
                return true;
        }
    }
    return false;
}

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

    int N, K;
    while (cin >> N >> K) {
        if (N == 0 && K == 0) break; // Terminate when N and K are both zero

        // Initialize adjacency matrix and in-degree count
        int g[100][100];
        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;
            if (relation.size() < 3 || (relation[1] != '<' && relation[1] != '>')) {
                // Handle invalid input format
                cout << "Invalid relation format." << endl;
                inconsistent = true;
                break;
            }

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

            // Determine the direction of the relation
            if (op == '<') {
                x = a - 'A';
                y = b - 'A';
            }
            else { // op == '>'
```

```
            x = b - 'A';
            y = a - 'A';
        }

        // Check if adding edge x -> y creates a cycle
        bool visited[100];
        memset(visited, false, sizeof(visited));
        if (is_reachable(y, x, N, g, visited)) {
            // Adding edge x->y would create a cycle
            cout << "Inconsistency found after " << (i + 1) << " relations." << endl;

            // Read and discard remaining relations for this test case
            for (int j = i + 1; j < K; j++) {
                cin >> relation;
            }

            inconsistent = true;
            break;
        }

        // Add the edge x -> y
        g[x][y] = 1;
        in_degree[y]++;

        // Perform Topological Sort to check if a unique sequence is determined
        // Create a copy of in_degree to manipulate
        vector<int> in_copy = in_degree;
        queue<int> Q;
        vector<int> result;

        // Enqueue all nodes with in-degree 0
        for (int node = 0; node < N; node++) {
            if (in_copy[node] == 0) {
                Q.push(node);
            }
        }

        bool multiple = false; // Flag to check if multiple sequences are possible

        while (!Q.empty()) {
            if (Q.size() > 1) {
                multiple = true; // More than one node with in-degree 0 implies multiple sequences
            }

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

            // Decrease in-degree of neighboring nodes
            for (int neighbor = 0; neighbor < N; neighbor++) {
                if (g[current][neighbor]) {
                    in_copy[neighbor]--;
                    if (in_copy[neighbor] == 0) {
                        Q.push(neighbor);
                    }
                }
            }
        }

        // Check if a unique sorted sequence is determined
        if (result.size() == N && !multiple) {
            cout << "Sorted sequence determined after " << (i + 1) << " relations: ";
            // Replace range-based for loop with traditional for loop
            for (size_t idx = 0; idx < result.size(); idx++) {
                int node = result[idx];
                cout << char('A' + node);
            }
            cout << "." << endl;

            // Read and discard remaining relations for this test case
            for (int j = i + 1; j < K; j++) {
                cin >> relation;
            }

            determined = true;
            break;
        }
    }

    // If no inconsistency or unique sequence was found after processing all relations
    if (!inconsistent && !determined) {
        cout << "Sorted sequence cannot be determined." << endl;
    }
  }

  return 0;
}
```

Graph DFS 計算圖連接性

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

int map[105][105]; // Grid map indicating oil and empty spots
```

```cpp
int vis[105][105]; // Visited marker array
int n, m; // Rows and columns of the grid

// DFS function to explore connected components
void dfs(int x, int y) {
    vis[x][y] = 1; // Mark current cell as visited

    // Explore all 8 possible directions
    if (x + 1 < n && y < m && !vis[x + 1][y] && map[x + 1][y]) dfs(x + 1, y);
    if (x - 1 >= 0 && y < m && !vis[x - 1][y] && map[x - 1][y]) dfs(x - 1, y);
    if (x < n && y + 1 < m && !vis[x][y + 1] && map[x][y + 1]) dfs(x, y + 1);
    if (x < n && y - 1 >= 0 && !vis[x][y - 1] && map[x][y - 1]) dfs(x, y - 1);
    if (x + 1 < n && y + 1 < m && !vis[x + 1][y + 1] && map[x + 1][y + 1]) dfs(x + 1, y + 1);
    if (x - 1 >= 0 && y - 1 >= 0 && !vis[x - 1][y - 1] && map[x - 1][y - 1]) dfs(x - 1, y - 1);
    if (x + 1 < n && y - 1 >= 0 && !vis[x + 1][y - 1] && map[x + 1][y - 1]) dfs(x + 1, y - 1);
    if (x - 1 >= 0 && y + 1 < m && !vis[x - 1][y + 1] && map[x - 1][y + 1]) dfs(x - 1, y + 1);
}

// Function to initialize visited array to zero
void init() {
    memset(vis, 0, sizeof(vis));
}

int main() {
    char ch;
    while (cin >> n >> m) { // Read grid dimensions
        if (n == 0 && m == 0) break;

        init(); // Clear the visited markers

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                cin >> ch; // Read each cell of the grid
                if (ch == '*')
                    map[i][j] = 0; // Mark empty spot
                else
                    map[i][j] = 1; // Mark oil spot
            }
        }

        int count = 0; // Initialize oil deposit count

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (!vis[i][j] && map[i][j]) {
                    dfs(i, j); // Run DFS from each unvisited oil spot
                    count++; // Increment oil deposit count
                }
            }
        }
        cout << count << endl; // Output the count of different oil deposits
    }
    return 0;
}
```

Graph BFS 計算圖連接性

```cpp
#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 a[8][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}, {-1, -1}, {-1, 1}, {1, -1}, {1, 1}}; // 8 個方向的移動

void BFS(int i, int j) {
    int front = 0, rear = 1; // 佇列的首尾標誌初始化
    bfsQueue[front].i = i;
    bfsQueue[front].j = j;
    map[i][j] = '*'; // 將起點設為無油狀態

    while (front != rear) {
        int ii = bfsQueue[front].i;
        int jj = bfsQueue[front].j;
        front++; // 佇列首指標 +1

        for (int k = 0; k < 8; k++) { // 8 個相鄰方向
            int t1 = ii + a[k][0];
            int t2 = jj + a[k][1];

            if (map[t1][t2] == '@') { // 若 (t1, t2) 是油田
                bfsQueue[rear].i = t1;
                bfsQueue[rear].j = t2;
                map[t1][t2] = '*'; // 將 (t1, t2) 設為無油狀態
                rear++; // 佇列尾指標 +1
            }
        }
    }
}

int main() {
```

```
    int i, j;
    int num;

    while (scanf("%d %d", &m, &n) && m) { // 反覆輸入行數 m 和列數 n，直到 m 為 0
        num = 0;
        for (i = 0; i < m; i++)
            scanf("%s", map[i]); // 自上而下，從左至右讀取每個網格

        for (i = 0; i < m; i++)
            for (j = 0; j < n; j++)
                if (map[i][j] == '@') { // 若 (i, j) 為油田
                    num++; // 不同的油田數量 +1
                    BFS(i, j); // 透過 BFS 將 (i, j) 可達的所有油田設為無油狀態
                }

        printf("%d\n", num); // 輸出油田數
    }

    return 0;
}
```

Graph 有向邊並查集, 速通性檢查, 樹判斷

```
#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

void make_set() { // 初始化
    for (int x = 1; x < MAX_SIZE; x++) {
        parent[x] = x;
        flag[x] = false;
    }
}

int find_set(int x) { // 尋找根節點，帶路徑壓縮
    if (x != parent[x])
        parent[x] = find_set(parent[x]);
    return parent[x];
}

void union_set(int x, int y) { // 合併兩個節點的集合
    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;
}

bool single_root(int n) { // 檢查是否只有一個根
    int i = 1;
    while (i <= n && !flag[i]) i++;
    if (i > n) return true; // 如果範圍內沒有使用的節點
    int root = find_set(i);
    while (i <= n) {
        if (flag[i] && find_set(i) != root)
            return false;
        ++i;
    }
    return true;
}

int main() {
    int x, y;
    bool is_tree = true;
    int range = 0;
    int idx = 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))
                printf("Case %d is a tree.\n", idx++);
            else
                printf("Case %d is not a tree.\n", idx++);

            is_tree = true;
            range = 0;
            make_set();
            continue;
        }

        if (x >= MAX_SIZE || y >= MAX_SIZE) { // 檢查 x 和 y 是否在範圍內
            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))
            is_tree = false;
        else
            union_set(x, y);
    }

    return 0;
}
```

MST Kuskal 計算最小樹新增無向邊權和

```
#include <iostream>
```

```cpp
#include <vector>
#include <algorithm>
using namespace std;

// µ²ºc  Ψ v«
struct Edge {
    int u;
    int v;
    int weight;
};

// Union-Find (Disjoint Set Union - DSU)  §O
class UnionFind {
private:
    vector<int> parent;
public:
    // º l¤ DSU¡A¸`I¼° n
    UnionFind(int n) : parent(n + 1) { // °²³]¸`I±q 1 ¶}©l
        for(int i = 0; i <= n; ++i)
            parent[i] = i;
    }

    // ¬dß麾¦X º ®A¨öi¦ |# Y
    int find_set(int x) {
        if(parent[x] != x)
            parent[x] = find_set(parent[x]);
        return parent[x];
    }

    // ¦X¨ °¦X
    void union_set(int x, int y) {
        int fx = find_set(x);
        int fy = find_set(y);
        if(fx != fy)
            parent[fx] = fy;
    }
};

// ¤ ¨禡¡A¥Ω
bool compare_edges(const Edge &a, const Edge &b) {
    return a.weight < b.weight;
}

// °  Kruskal ºtº k¨ê ^³p¥¨¼ `v«
int kruskal(int N, vector<Edge> &edges, UnionFind &uf) {
    // « v«¹ i¦ ā]±q¤p¨ j¡^
    sort(edges.begin(), edges.end(), compare_edges);

    int total_weight = 0;
    for(int i = 0; i < edges.size(); ++i) {
        Edge edge = edges[i];
        // ¦pºG頠¨ `I  Pºº¶°¦X¡A«h¥[¤J³p¥¨¼
        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); // ¨  0 ¥N´ nullptr

    int N;
    while(cin >> N) { // Ū¨¸`I¼ N
        // Ū¨¼F±µ¯x°}
        vector<vector<int>> P(N + 1, vector<int>(N + 1, 0));
        for(int i = 1; i <= N; ++i) { // 1-based indexing
            for(int j = 1; j <= N; ++j) {
                cin >> P[i][j];
            }
        }

        // º l¤ Union-Find
        UnionFind uf(N);

        // Ū¨¤w¸g³s±µºº  M ¨œX¨o¨
        int M;
        cin >> M;
        for(int i = 0; i < M; ++i) {
            int a, b;
            cin >> a >> b;
            uf.union_set(a, b);
        }

        // ¦¬¶°©³ A K«½d]µL¦V¹ ^
        vector<Edge> edges;
        for(int i = 1; i <= N; ++i) {
            for(int j = i + 1; j <= N; ++j) { // j ±q i+1 ¶}©l K«½
                if(P[i][j] > 0) { // °²³] 0 º S¦³
                    edges.push_back(Edge{ i, j, P[i][j] });
                }
            }
        }

        // °  Kruskal ºtº k¥Hpº p¥¨¼ `v«
        int total_MST_weight = kruskal(N, edges, uf);

        // ¨ std::endl ¥H½T«O¨C ¸ ¿Y X
        cout << total_MST_weight << endl;
    }
    return 0;
}

/*
½d¨ J:
3
```

```
0 990 692
990 0 179
692 179 0
1
1 2

'w´- X:
179
*/
```

MST prim 計算權和, 線性掃描最小邊, 密稠圖

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

int min(int i, int j) { // Return the index with the minimum value
    return i < j ? i : j;
}

int main() {
    int n;
    while (cin >> n) {
        int tot = 0;
        vector<vector<int>> v(n, vector<int>(n)); // Adjacency matrix for the graph
        vector<int> dist(n, INT_MAX); // Distance array, initialized to maximum
        vector<bool> use(n, false); // Boolean array to mark visited nodes

        // Reading the adjacency matrix
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cin >> v[i][j];
            }
        }

        dist[0] = 0; // Starting node has distance 0

        // Prim's algorithm to find MST
        for (int i = 1; i < n; i++) {
            dist[i] = v[0][i]; // Initialize the distance from the starting node
        }

        for (int i = 1; i < n; i++) { // Expand the MST with n - 1 edges
            int tmp = -1;
            for (int k = 1; k < n; k++) { // Find the minimum edge weight to add to MST
                if (!use[k] && (tmp == -1 || dist[k] < dist[tmp])) {
                    tmp = k;
                }
            }

            use[tmp] = true; // Mark the node as part of MST
            tot += dist[tmp]; // Add the minimum edge weight to the total weight

            // Update distances to nodes outside the MST
            for (int k = 1; k < n; k++) {
                if (!use[k]) {
                    dist[k] = min(dist[k], v[k][tmp]);
                }
            }
        }

        cout << tot << endl; // Output the total weight of the MST
    }

    return 0;
}
```

SP Warshell 閉包遞移, 二分法計算最長邊最小路徑

```cpp
#include <iostream>
#include <cmath>
#include <iomanip>
#include <vector>

using namespace std;

const int MAX_N = (1 << 9) + 1;
double L[MAX_N][MAX_N]; // Distance matrix
bool con[MAX_N][MAX_N]; // Connection matrix

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

        // Read coordinates of each stone
        for (int i = 0; i < N; ++i) {
            cin >> x[i] >> y[i];
        }

        // Calculate the distance matrix 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));
            }
        }

        // Binary search for the minimum distance
        double l = 0, r = 1e5;
        while (r - l > 1e-5) {
            double mid = (l + r) / 2;
```

```

            // Initialize the connection matrix based on mid distance
            for (int i = 0; i < N; ++i) {
                for (int j = 0; j < N; ++j) {
                    con[i][j] = (L[i][j] <= mid);
                }
            }

            // Floyd-Warshall algorithm to determine reachability
            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]);
                    }
                }
            }

            // Check if the first and second stones are connected
            if (con[0][1]) {
                r = mid;
            } else {
                l = mid;
            }
        }

        // Output the result with three decimal precision
        cout << "Scenario #" << ++testCase << endl;
        cout << "Frog Distance = " << fixed << setprecision(3) << l << endl;
        cout << endl;
    }
    return 0;
}
```

SP Dijkstra 重邊判斷, 找最短路徑

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

#define MAX_N 1010
#define MAX_M 2010
#define INF 1e9

using namespace std;

int w[MAX_N][MAX_N]; // Weight matrix for the graph
int d[MAX_N]; // Distance array for Dijkstra's algorithm
bool visited[MAX_N]; // Visited array for Dijkstra's algorithm
int n, m; // n = number of nodes, m = number of edges

void dijkstra(int s) { // Dijkstra's algorithm starting from node s
    for (int i = 1; i <= n; ++i) {
        d[i] = INF; // Initialize all distances to infinity
        visited[i] = false; // Mark all nodes as unvisited
    }
    d[s] = 0; // Starting node distance is 0

    for (int i = 1; i <= n; ++i) {
        int x = -1;
        for (int j = 1; j <= n; ++j) {
            if (!visited[j] && (x == -1 || d[j] < d[x])) x = j;
        }
        visited[x] = true;

        for (int j = 1; j <= n; ++j) {
            if (!visited[j] && w[x][j] != INF) {
                d[j] = min(d[j], d[x] + w[x][j]);
            }
        }
    }
}

int main() {
    scanf("%d%d", &m, &n); // Input the number of edges and nodes
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j) {
            w[i][j] = INF; // Initialize weight matrix with INF
        }
    }

    for (int i = 0; i < m; ++i) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c); // Input edge endpoints and weight
        if (w[a][b] > c) w[a][b] = w[b][a] = c; // Update to minimum weight for undirected graph
    }

    dijkstra(1); // Run Dijkstra's algorithm starting from node 1

    printf("%d\n", d[n]); // Output the shortest path distance to node n
    return 0;
}
```

SP Dijkstra 二分搜尋最佳初始值

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

```

```cpp
// Function to convert character to index
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; // Invalid character
}

// Check function: Determines the maximum cargo that can reach 'to' from 'from' with starting cargo 'o'
int check(int from, int to, int o, bool go[][55]){
    int g[55];
    memset(g, 0, sizeof(g)); // Initialize cargo for each node to 0
    bool flag[55];
    memset(flag, false, sizeof(flag)); // Initialize visit flags to false
    g[from] = o; // Set starting cargo at 'from' node

    while(true){
        int w = 0, next = -1;
        // Find the unflagged node with the highest cargo
        for(int i = 1; i <= 52; i++){
            if(!flag[i] && g[i] > w){
                next = i;
                w = g[i];
            }
        }

        if(next == -1) break; // No more nodes to process
        flag[next] = true; // Mark the node as processed

        // Update cargo for connected nodes
        for(int i = 1; i <= 52; i++){
            if(go[next][i]){
                int reduction;
                if(i < 27){
                    reduction = (w + 19) / 20; // Equivalent to ceil(w / 20)
                }
                else{
                    reduction = 1;
                }
                int tmp = w - reduction;
                if(tmp > g[i]){
                    g[i] = tmp; // Update cargo if the new value is higher
                }
            }
        }
    }

    return g[to]; // Return the cargo at the destination node
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0); // Fast I/O

    int T; // Number of connections
    int tot = 0; // Test case counter

    while(cin >> T){
        if(T == -1) break; // Termination condition

        // Initialize adjacency matrix
        bool go[55][55];
        memset(go, false, sizeof(go));

        // Read T connections
        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){
                // Invalid characters, skip this connection
                continue;
            }
            go[a][b] = go[b][a] = true; // Bidirectional connection
        }

        // Read Tot (required cargo)
        int Tot;
        cin >> Tot;

        // Read source and destination characters
        char fromChar, toChar;
        cin >> fromChar >> toChar;
        int from = turn(fromChar);
        int to = turn(toChar);

        if(from == -1 || to == -1){
            // Invalid source or destination nodes
            tot++;
            cout << "Case " << tot << ": Impossible\n";
            continue;
        }

        // Binary search to find the minimal starting cargo 'o' such that check(from, to, o) >= Tot
        int l = 1, r = (1 << 20); // Search range: 1 to 1048576
        while(l < r){
            int mid = (l + r - 1) / 2; // Midpoint calculation
            int cargo = check(from, to, mid, go);
            if(cargo >= Tot){
                r = mid; // Possible answer found, search lower half
            }
```

```
            else{
                l = mid + 1; // Need a higher starting cargo, search upper half
            }
        }

        // After binary search, 'l' should be the minimal 'o' such that check(from, to, l) >= Tot
        // Verify the result
        int finalCargo = check(from, to, l, go);
        if(finalCargo >= Tot){
            tot++;
            cout << "Case " << tot << ": " << l << "\n";
        }
        else{
            // If even the maximum cargo doesn't meet the requirement
            tot++;
            cout << "Case " << tot << ": Impossible\n";
        }
    }

    return 0;
}

/*
1
a Z
19 a Z
5
A D
D X
A b
b c
c X
39 A X
-1
*/
```

SP SPFA 求負權最短路徑

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

const int MAX = 501;
const int INF = 1e9; // Define a large constant for infinity
int map[MAX][MAX]; // Adjacency matrix to store graph weights
int dis[MAX]; // Distance array
int n, m, w, s, e, t; // Variables for nodes, edges, and endpoints
bool spfa(); // Function declaration for SPFA algorithm

// SPFA function to detect if there is a negative cycle in the graph
bool spfa() {
    bool flag[MAX] = {0}; // Flags to mark nodes in the queue
    int count[MAX] = {0}; // Counter for each node's occurrences in queue
    queue<int> q;
    q.push(s); // Start from source node
    dis[s] = 0; // Distance to source is 0
    int curr;
    int i;

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

        for (int i = 1; i <= n; i++) {
            if(map[curr][i]<100000)
            {
                if(dis[i] > map[curr][i] + dis[curr])
                {
                    dis[i] = map[curr][i] + dis[curr];
                    if(flag[i] == 0)
                        q.push(i);
                    count[i]++;
                    flag[i] = 1;
                    if(count[i]>=n)
                        return 0;
                }
            }
        }
        flag[curr] = 0;
    }
    return true;
}

int main() {
    int f;
    scanf("%d", &f);
    while (f--) {
        // Reset arrays between test cases
        memset(dis, 63, sizeof(dis)); // Initialize distance array to a large number
        memset(map, 127, sizeof(map));

        scanf("%d %d %d", &n, &m, &w); // Read number of nodes, edges, and start/end points
        int i;
        for (i = 0; i < m; i++) {
            scanf("%d %d %d", &s, &e, &t);
                        // Update to the smallest weight if multiple edges exist
            map[s][e] = map[s][e]>t? t: map[s][e];
```

```
            map[e][s] = map[e][s]>t? t:
                map[e][s];
        }

        for (int i = 0; i < w; i++) {
            scanf("%d %d %d", &s, &e, &t);
            map[s][e] = -t; // Set the edge with
                negative weight for wormhole
        }

        if (spfa()) // Use SPFA to detect negative
            cycles
            printf("NO\n");
        else
            printf("YES\n");
    }
    return 0;
}
```

BG HA(匈牙利算法) 二分圖最大匹配

```
#include <iostream>
#include <cstdio>
#include <cstring>

using namespace std;

int a[110][310];
int n, m, vis[310], pre[310];

bool dfs(int x) {
    int t;
    for (t = 1; t <= m; ++t) {
        if (a[x][t] && !vis[t]) { // if there is an
            edge and t is not visited
            vis[t] = 1; // mark t as visited
            if (pre[t] == 0 || dfs(pre[t])) { //
                if t has no previous match or previous
                match can find an alternate path
                pre[t] = x; // set (x, t) as a match
                return true; // matching success
            }
        }
    }
    return false; // matching failed
}

int main() {
    int T, i, t, j, s;
    scanf("%d", &T); // input number of test cases
    while (T--) {
        scanf("%d%d", &n, &m); // input number of
            courses and students
        memset(a, 0, sizeof(a)); // initialize
            adjacency matrix for bipartite graph
        memset(pre, 0, sizeof(pre)); // initialize
            matching storage array

        for (i = 1; i <= n; ++i) {
            scanf("%d", &t); // input number of
                students corresponding to each course
            while (t--) {
                scanf("%d", &j); // input each student
                a[i][j] = 1; // mark relation between
                    course i and student j as an edge
            }
        }

        s = 0;
        for (i = 1; i <= n; ++i) {
            memset(vis, 0, sizeof(vis)); //
                initialize visit array for each course
            if (dfs(i)) s++; // if course i can be
                matched, increment match count
        }

        if (s == n) printf("YES\n"); // if maximum
            match count equals course count, team can be
            formed
        else printf("NO\n"); // otherwise, team
            formation fails
    }
    return 0;
}
```

BG 最大匹配數求邊覆蓋

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

const int V = 1100;

int n, m, k, x, y, pre[V];
bool v[V], a[V][V];

bool dfs(int i) {
    for (int j = 1; j <= m; j++) {
        if (!v[j] && a[i][j]) { // check if j is
            unvisited and there is an edge between i and j
            v[j] = 1; // mark j as visited
            if (pre[j] == 0 || dfs(pre[j])) { //
                if j has no previous match or previous
                match can find an alternate path
                pre[j] = i; // set (i, j) as a match
                return 1; // matching success
            }
        }
    }
    return 0; // matching failed
}

int main() {
    cin >> n >> m >> k; // input number of
        representatives from A and B, and the number of
        pairs
```

```
    memset(a, 0, sizeof(a)); // initialize adjacency matrix for bipartite graph
    memset(pre, 0, sizeof(pre)); // initialize matching storage array

    for (int i = 1; i <= k; i++) {
        cin >> x >> y; // input pair representing relationship between A's representative x and B's representative y
        a[x][y] = 1; // mark the relationship as an edge
    }

    int ans = 0;
    for (int i = 1; i <= n; i++) {
        memset(v, 0, sizeof(v)); // initialize visit array for each representative
        if (dfs(i)) ans++; // if representative i can be matched, increment match count
    }

    cout << n + m - ans << endl; // output minimum number of telephone lines required, N + M - max matching
    return 0;
}
```

BG 二分圖匹配最大化最小值

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

const int MAX = 501;
const int INF = 1e9; // Define a large constant for infinity
int map[MAX][MAX]; // Adjacency matrix to store graph weights
int dis[MAX]; // Distance array
int n, m, w, s, e, t; // Variables for nodes, edges, and endpoints
bool spfa(); // Function declaration for SPFA algorithm

// SPFA function to detect if there is a negative cycle in the graph
bool spfa() {
    bool flag[MAX] = {0}; // Flags to mark nodes in the queue
    int count[MAX] = {0}; // Counter for each node's occurrences in queue
    queue<int> q;
    q.push(s); // Start from source node
    dis[s] = 0; // Distance to source is 0
    int curr;
    int i;

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

        for (int i = 1; i <= n; i++) {
            if(map[curr][i]<100000)
            {
                if(dis[i] > map[curr][i] + dis[curr])
                {
                    dis[i] = map[curr][i] + dis[curr];
                    if(flag[i] == 0)
                        q.push(i);
                    count[i]++;
                    flag[i] = 1;
                    if(count[i]>=n)
                        return 0;
                }
            }
        }
        flag[curr] = 0;
    }
    return true;
}

int main() {
    int f;
    scanf("%d", &f);
    while (f--) {
        // Reset arrays between test cases
        memset(dis, 63, sizeof(dis)); // Initialize distance array to a large number
        memset(map, 127, sizeof(map));

        scanf("%d %d %d", &n, &m, &w); // Read number of nodes, edges, and start/end points
        int i;
        for (i = 0; i < m; i++) {
            scanf("%d %d %d", &s, &e, &t);
                        // Update to the smallest weight if multiple edges exist
            map[s][e] = map[s][e]>t? t: map[s][e];
            map[e][s] = map[e][s]>t? t: map[e][s];
        }

        for (int i = 0; i < w; i++) {
            scanf("%d %d %d", &s, &e, &t);
            map[s][e] = -t; // Set the edge with negative weight for wormhole
        }

        if (spfa()) // Use SPFA to detect negative cycles
            printf("NO\n");
        else
            printf("YES\n");
    }
```

```
    return 0;
}
```

BG KM 求二分圖最小權和 (負邊)

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

using namespace std;

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

int n, m, a[MAXN][MAXN], lx[MAXN], ly[MAXN],
    slack[MAXN], maty[MAXN];
int lenx, leny;
bool vx[MAXN], vy[MAXN];
char map[MAXN][MAXN];

bool search(int u) {

    int i, t;
    vx[u] = 1;
    for (int i = 0; i < leny; ++i) {
        if (!vy[i]) {
            int t = lx[u] + ly[i] - a[u][i];
            if (t == 0) {
                vy[i] = 1;
                if (maty[i] == -1 ||
                    search(maty[i])) {
                    maty[i] = u;
                    return 1;
                }
            }
            else if (slack[i] > t) {
                slack[i] = t;
            }
        }
    }
    return 0;
}

int KM() {
    int i, j, ans = 0;
    for (i = 0; i < lenx; ++i) {
        lx[i] = -INT_MAX;
        for (j = 0; j < leny; ++j) {
            lx[i] = max(lx[i], a[i][j]);
        }
    }
    memset(maty, -1, sizeof(maty));
    memset(ly, 0, sizeof(ly));
    for (i = 0; i < lenx; ++i) {
        fill(slack, slack + leny, INT_MAX);
        while (1) {
            memset(vx, 0, sizeof(vx));
            memset(vy, 0, sizeof(vy));
            if (search(i)) break;
            int d = INT_MAX;
            for (j = 0; j < leny; ++j) {
                if (!vy[j] && d > slack[j]) d =
                    slack[j];
            }
            for (j = 0; j < lenx; ++j) {
                if (vx[j]) lx[j] -= d;
            }
            for (j = 0; j < leny; ++j) {
                if (vy[j]) ly[j] += d;
            }
        }
    }
    for (i = 0; i < leny; ++i) {
        if (maty[i] != -1) ans += a[maty[i]][i];
    }
    return -ans;
}

int main() {
    int i, j;
    while (~scanf("%d%d", &n, &m) && n + m) {
        lenx = leny = 0;
        for (i = 0; i < n; ++i) {
            scanf("%s", map[i]);
            for (j = 0; j < m; ++j) {
                if (map[i][j] == 'H') {
                    lx[lenx] = i;
                    slack[lenx++] = j;
                } else if (map[i][j] == 'm') {
                    ly[leny] = i;
                    maty[leny++] = j;
                }
            }
        }
        for (i = 0; i < lenx; ++i) {
            for (j = 0; j < leny; ++j) {
                a[i][j] = -abs(lx[i] - ly[j]) -
                    abs(slack[i] - maty[j]);
            }
        }
        printf("%d\n", KM());
    }
    return 0;
}
```

Flow EK 求最大流

```
#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
```

```

void Edmonds_Karp() { // Edmonds-Karp演算法求最大流
    int qs, qt, d, doo, i, j, ans = 0;
    fa[t] = 1;
    while (fa[t] != 0) { // 若增廣路徑存在
        qs = 0; qt = 1;
        q[qt] = s;
        memset(fa, 0, sizeof(fa)); // 初始化增廣路徑
        fa[s] = s;
        while (qs < qt && fa[t] == 0) {
            i = q[++qs];
            for (j = 1; j <= t; j++) {
                if (fa[j] == 0) {
                    if (f[i][j] < c[i][j]) { //
                        若(i, j)的流量可增加
                        fa[j] = i;
                        q[++qt] = j;
                    }
                    else if (f[j][i] > 0) { //
                        若(i, j)的反向流量可減少
                        fa[j] = -i;
                        q[++qt] = j;
                    }
                }
            }
        }
        if (fa[t] != 0) { // 如果找到增廣路徑
            doo = 1000000000;
            i = t;
            while (i != s) { //
                從終點逆向尋找增廣路徑中最小容量
                if (fa[i] > 0) {
                    if((d = c[fa[i]][i] -
                        f[fa[i]][i]) < doo)
                        doo = d;
                } else {
                    if (f[i][-fa[i]] < doo)
                        doo = f[i][-fa[i]];
                }
                i = abs(fa[i]);
            }
            ans += doo;
            i = t;
            while (i != s) { // 增加增廣路徑上流量
                if (fa[i] > 0)
                    f[fa[i]][i] += doo;
                else
                    f[i][-fa[i]] -= doo;
                i = abs(fa[i]);
            }
        }
    }
    printf("%d\n", ans); // 輸出最大流
}

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; t = n + 1;
        memset(f, 0, sizeof(f));
        memset(c, 0, sizeof(c));
        for (i = 1; i <= m; i++) { // 讀取邊的資訊
            while(getchar() != '(');
            scanf("%d,%d)%d", &u, &v, &cc);
            c[u + 1][v + 1] = cc;
        }
        for (i = 1; i <= np; i++) { //
            讀取每個源點的流量容量
            while (getchar() != '(');
            scanf("%d)%d", &u, &cc);
            c[s][u + 1] = cc;
        }
        for (i = 1; i <= nc; i++) { //
            讀取每個匯點的流量容量
            while (getchar() != '(');
            scanf("%d)%d", &u, &cc);
            c[u + 1][t] = cc;
        }
        Edmonds_Karp(); // 執行 Edmonds-Karp
            演算法求最大流
    }
    return 0;
}
```

Flow SPFA 求最小費用流, 帶權二分圖轉網路圖

```
#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;

    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;

                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;
    }

    void 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;
        }
    }

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

    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);
    }

    int solve(int _s, int _t) {
        s = _s;
        t = _t;
        int totalCost = 0;

        while (SPFA()) {
            totalCost += dist[t];
            augment();
        }

        return totalCost;
    }
};

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

    int n;
    cin >> n;

    MinCostMaxFlow mcmf(2 * n + 2);
    vector<vector<int>> cost(n + 1, vector<int>(n + 1));

    // 讀入成本矩陣
    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];
        }
        // 調整成本
        for (int j = 1; j <= n; j++) {
            cost[i][j] = sum - cost[i][j];
        }
    }

    // 建圖
    int s = 1, t = 2 * n + 2;
    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]); // 左側到右側的連接
        }
    }

    cout << mcmf.solve(s, t) << endl;

    return 0;
}
```