**National Collegiate Programming Competition**

Millitary Institute of Science and Technology

**CODEBOOK**

Perimeter(sq): s^4

Perimeter(rect): 2 \* (l \* w)

Perimeter(tri): a + b + c

Area(sq): s \* s

Rect: l \* w

Tri: (b \* h) / 2

Trap: (b1 + b2) \* (h / 2)

Vol. of Cube: s^3

Vol. of Box: l \* w \* h

Vol. of sphere: (4/3) \* pi \* r^3

Vol. of cylinder: pi \* r^2 \* height

Projectile:

y = y0 + v \* y0 \* t – ½(g \* t^2)

**Common Header Files**

#include <algorithm>

// var.reverse()

// find(first, last, key)

// count(first, last, key)

// count\_if(first, last, condition)

// swap(index1, index2)

// replace(first, last, old\_value, new\_value)

// sort(first, last)

// is\_sorted(first, last)

// min(value1, value2)

// max(value1, value2)

#include <functional>

#include <climits>

#include <cmath>

// cos(value)

// sin(value)

// tan(value)

// exp(value)

// log(value)

// log10(value)

// pow(value, power)

// sqrt(value)

// ceil(value)

// floor(value)

// fabs(value)

// abs(value)

// isnan(value)

// isinf(value)

// isfinite(value)

#include <cstdio>

#include <cstdlib>

// atof(str)

// atoi(str)

// atol(str)

// atoll(str)

// strtod(str)

// strtof(str)

// strtol(str)

// strtold(str)

// strtoll(str)

// strroul(str)

// strtoull(str)

// rand()

#include <ctime>

#include <iostream>

// cin(value)

// cout(str)

#include <sstream>

#include <iomanip>

#include <deque>

#include <list>

#include <queue>

// var.empty()

// var.size()

// var.front()

// var.back()

// var.push(value)

// var.pop()

#include <stack>

#include <cstring>

// strcpy(destination, source)

// strncpy(destination, source, sizeof(destination))

// strcmp(string1, string2)

#include <vector>

**Useful Constant**

INT\_MIN

INT\_MAX

LONG\_MIN

LONG\_MAX

LLONG\_MIN

LLONG\_MAX

**Math Tricks**

// when the number is too large. use powl instead of pow.

// will provide you more accuracy.

powl(a, b)

(int)round(p, (1.0/n)) // nth root of p

**Prime Factor**

// smallest prime factor of a number.

void factor(int n) {

int a;

if (n % 2 == 0) return 2;

for (a = 3; a <= sqrt(n); a++++)

if (n%a==0) return a;

return n;

// complete factorization

int r;

while (n > 1) {

r = factor(n);

printf("%d", r); n /= r;

}

}

**a^b mod p**

long powmod(long base, long exp, long modulus) {

base %= modulus;

long result = 1;

while (exp > 0) {

if (exp & 1) result = (result \* base) % modulus;

base = (base \* base) % modulus;

exp >>= 1;

}

return result;

}

**Graph Representation**

// The most common way to define graph is to use adjacency matrix

// example:

// (1) (2) (3) (4) (5)

// (1) 2 0 5 0 0

// (2) 4 2 0 0 1

// (3) 3 0 0 1 4

// (4) 6 9 0 0 0

// (5) 1 1 1 1 5

// it’s always a square matrix.

// suppose a graph has n nodes, if given exactly adjacency matrix

for (int i = 1; i <= n; i++)

for (int j = 1; i <= n; j++)

cin << a[i][j] << endl;

// Usually will go like this representation in data

// start\_node end\_node weight

// suppose m lines

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

int x = 0, y = 0, t = 0;

cin >> x >> y >> t;

a[x][y]=t; // if undirected graph a[y][x]=t;

}

// another variant: on the ith line, has data as

// end\_node weight

// when you read data, you can assign matrix as

a[i][x]=t;

// if undirected graph

a[x][i]=t;

// Initialization of graph !!!IMPORTANT

// Depends on usage, normally initialize as 0 for all elements in matrix.

// so that 0 means no connection, non-0 means connection

// (for problem without weight, use weight as 1)

// If weights are important in this context (especially searching for path)

// Initialize graph as infinity for all elements in matrix.

// Another way to store graph is Adjacency list

// No space advantage if using array (unknown maximum number for in-degree). // Big space advantage if using dynamic data structure (like list, vector).

// each row represent a node and its connectivity.

// we don’t need it so much due to it’s search efficiency.

// let’s define a node as

struct Node {

int id; // node id

int w; // weight

};

// suppose n nodes and m lines of inputs as

// start\_node end\_node weight

// assume using <vector> in this example

// g is a vector, and each element of g is also a vector of Node

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

int x = 0, y = 0, t = 0;

cin >> x >> y >> t;

Node temp;

temp.id=y;

temp.w=t;

g[x].push\_back(temp);

// if undirected

temp.id=x;

g[y].push\_back(temp);

}

// Note that you don’t need this node structure if graph has only connectivity information.

/\*\*\*\* Special Structure \*\*\*\*/

// Special structure here is usually not a typical graph, like city-blocks, triangles

// They are represented in 2-d array and shows weights on nodes instead of edges.

// Note that in this case travel through edge has no cost, but visit node has cost.

// Triangles: Read data like this

// 1

// 1 2

// 4 2 7

// 7 3 1 5

// 6 2 9 4 6

for (int i = 1; i <= n; i++)

for (int j = i; j <= n; j++)

cin >> a[i][j];

// More complex data structures: typical city-block structure may has some constraints on

// questions, but it has no boundaries. However, some questions requires to form a maze.

// In these cases, data structures can be very flexible, it totally depends on how the question

// presents the data. A usual way is to record it’s adjacent blocks information:

struct Block{

bool l[4];

// if has 8 neighbors then use bool l[8];

// label them as your favor, e.x.

// 1 123

// 4x2 8x4

// 3 765

// true if there is path, false if there is boundary

// other informations (optional)

int weight;

int component\_id;

// etc.

};

// Note that usually we use array from index 1 instead of 0 because sometimes

// you need index 0 as your boundary, and start from index 1 will give you

// advantage on locating nodes or positions

**Prim’s Algorithm**

int d[1001] = {0};

bool v[1001] = {0};

int a[1001][1001] = {0};

int main(void) {

int n=0;

cin >> n;

for (int i = 1; i <= n; i++) {

int x = 0, y = 0, z = 0;

cin >> x >> y >> z;

a[x][y] = z;

}

for (int i = 1; i <= n; i++)

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

if (a[i][j] == 0)

a[i][j] = INT\_MAX;

cout << prim(1, n) << endl;

}

int prim(int u, int n) {

int mst = 0, k;

for (int i = 0; i < d.length; i++)

d[i] = INT\_MAX;

for (int i = 0; i < v.length; i++)

v[i] = false;

d[u] = 0;

int i = u;

while (i != 0) {

v[i] = true;

k = 0;

mst += d[i];

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

if (!v[j]) {

if (a[i][j] < d[j])

d[j] = a[i][j];

if (d[j] < d[k])

k = j;

}

i = k;

}

return mst;

}

**GDC**

int gcd(int n, int m) {

if (n % m == 0) return m;

return gcd(m, n % m);

}

**LCM**

int lcm(int a, int b) { return (a \* b) / gcd(a, b); }

**Leap Year**

bool isLeapYear(int year) {

if ((year % 400 == 0) || (year % 100 != 0 && year % 4 == 0)) {

return true;

} else {

return false;

}

}

**math.h**

» sqrt(n) // square root

» fabs(n) // absolute

» sin(n), cos(n), tan(n)

» asin(n), acos(n), atan(n) // inverse

» atan2(y, x)

» pow(n, m)

» exp(n)

» log(n), log10(n)

» floor(n), ceil(n)

**prime number**

bool isPrime(int n) {

if (n <= 1) return false;

for (int i = 2; i < n; i++)

if (n % i == 0)

return false;

return true;

}

**BigMod**

int bigMod(int a, int b, int M) {

if (b == 0) return 1 % M;

int x = bigMod(a, b / 2, M);

x = (x \* x) % M;

if (b % 2 == 1) x = (x \* a) % M;

return x;

}

**Moduler Inverse**

int modInverse(int a, int m) {

a = a%m;

for (int x=1; x<m; x++)

if ((a\*x) % m == 1) return x;

}

**Factorial**

int fact(int n) {

if (n == 0 || n == 1) return 1;

else return n \* fact(n - 1);

}

**Combination & Permutation**

comb = fact(n) / (fact(r) \* fact(n-r));

per = fact(n) / fact(n-r);

**Fibonacci Number**

int fibonacci(int n) {

if (n == 0) return 0;

if (n == 1) return 1;

return fibonacci(n - 1) + fibonacci(n - 2);

}

**Insertion sort**

void insertionSort(int num[n]) {

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

int x = num[i];

int j = i - 1;

while (j >= 1 && num[j] > x) {

num[j + 1] = num[j];

j--;

}

num[j + 1] = x;

}

}

**Selection sort**

#include "algorithm.h"

void selectionSort(int num[n]) {

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

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

if (num[i] > num[j]) swap(num[i], num[j]);

}

}

}

**Bubble sort**

void bubbleSort(int num[n]) {

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

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

if (num[j + 1] > num[j]) {

int temp = num[j];

num[j] = num[j + 1];

num[j + 1] = temp;

}

}

}

}

**Merge sort**

int num[100000], temp[100000];

void mergeSort(int \_low, int \_high) {

if (\_low == \_high) return;

int mid = (\_low + \_high) / 2;

mergeSort(\_low, mid);

mergeSort(mid + 1, \_high);

int i, j, k;

for (i = \_low, j = mid + 1, k = \_low; k <= \_high; k++) {

if (i == mid + 1) temp[k] = num[j++];

else if (j == \_high + 1) temp[k] = num[i++];

else if (num[i] < num[j]) temp[k] = num[i++];

else temp[k] = num[j++];

}

for (k = \_low; k <= \_high; k++) num[k] = temp[k];

}

**String sort**

#include <string>

#include <algorithm>

#include <vector>

using namespace std;

void stringSort(int s[10000]) {

int n, i;

vector<string> V;

cin >> n;

for (i = 0; i < n; i++) {

cin >> s;

V.push\_back(s);

}

sort(V.begin(), V.end());

}

**Binary search**

int binarySearch(int arr[], int l, int r, int x) {

if (r >= l) {

int mid = l + (r - l) / 2;

if (arr[mid] == x) return mid;

if (arr[mid] > x) return binarySearch(arr, l, mid - 1, x);

return binarySearch(arr, mid + 1, r, x);

}

return -1;

}

**Ternary search**

int ternarySearch(int l, int r, int key, int ar[]) {

if (r >= l) {

int mid1 = l + (r - l) / 3;

int mid2 = r - (r - l) / 3;

if (ar[mid1] == key) return mid1;

if (ar[mid2] == key) return mid2;

if (key < ar[mid1]) return ternarySearch(l, mid1 - 1, key, ar);

else if (key > ar[mid2]) return ternarySearch(mid2 + 1, r, key, ar);

else return ternarySearch(mid1 + 1, mid2 - 1, key, ar);

}

return -1;

}

**Backtracking**

**Knight’s Tour Problem**

#include <stdio.h>

#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N], int xMove[], int yMove[]);

int isSafe(int x, int y, int sol[N][N]) {

return ( x >= 0 && x < N && y >= 0 && y < N && sol[x][y] == -1);

}

void printSolution(int sol[N][N]) {

for (int x = 0; x < N; x++) {

for (int y = 0; y < N; y++) printf(" %2d ", sol[x][y]);

printf("\n");

}

}

int solveKT() {

int sol[N][N];

for (int x = 0; x < N; x++)

for (int y = 0; y < N; y++)

sol[x][y] = -1;

int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };

int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

sol[0][0] = 0;

if (solveKTUtil(0, 0, 1, sol, xMove, yMove) == 0) {

printf("Solution does not exist");

return 0;

} else printSolution(sol);

return 1;

}

int solveKTUtil(int x, int y, int movei, int sol[N][N], int xMove[N], int yMove[N]) {

int k, next\_x, next\_y;

if (movei == N\*N) return 1;

for (k = 0; k < 8; k++) {

next\_x = x + xMove[k];

next\_y = y + yMove[k];

if (isSafe(next\_x, next\_y, sol)) {

sol[next\_x][next\_y] = movei;

if (solveKTUtil(next\_x, next\_y, movei+1, sol, xMove, yMove) == 1) return 1;

else sol[next\_x][next\_y] = -1;

}

}

return 0;

}

**Rat in Maze**

#include <stdio.h>

#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N]);

void printSolution(int sol[N][N]) {

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

printf(" %d ", sol[i][j]);

printf("\n");

}

}

bool isSafe(int maze[N][N], int x, int y) {

if (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1) return true;

return false;

}

bool solveMaze(int maze[N][N]) {

int sol[N][N] = { { 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 } };

if (solveMazeUtil(maze, 0, 0, sol) == false) {

printf("Solution doesn't exist");

return false;

}

printSolution(sol);

return true;

}

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N]) {

if (x == N - 1 && y == N - 1) {

sol[x][y] = 1;

return true;

}

if (isSafe(maze, x, y) == true) {

sol[x][y] = 1;

if (solveMazeUtil(maze, x + 1, y, sol) == true) return true;

if (solveMazeUtil(maze, x, y + 1, sol) == true) return true;

sol[x][y] = 0;

return false;

}

return false;

}

**N Queen Problem**

#define N 4

#include <stdbool.h>

#include <stdio.h>

void printSolution(int board[N][N]) {

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

printf(" %d ", board[i][j]);

printf("\n");

}

}

bool isSafe(int board[N][N], int row, int col) {

int i, j;

for (i = 0; i < col; i++)

if (board[row][i])

return false;

for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j])

return false;

for (i = row, j = col; j >= 0 && i < N; i++, j--)

if (board[i][j])

return false;

return true;

}

bool solveNQUtil(int board[N][N], int col) {

if (col >= N) return true;

for (int i = 0; i < N; i++) {

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

bool solveNQ() {

int board[N][N] = { { 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 } };

if (solveNQUtil(board, 0) == false) {

printf("Solution does not exist");

return false;

}

printSolution(board);

return true;

}

**m Coloring Problem**

#include<stdio.h>

#include<stdbool.h>

#define V 4

void printSolution(int color[]);

bool isSafe (int v, bool graph[V][V], int color[], int c) {

for (int i = 0; i < V; i++)

if (graph[v][i] && c == color[i])

return false;

return true;

}

bool graphColoringUtil(bool graph[V][V], int m, int color[], int v) {

if (v == V) return true;

for (int c = 1; c <= m; c++) {

if (isSafe(v, graph, color, c)) {

color[v] = c;

if (graphColoringUtil (graph, m, color, v+1) == true) return true;

color[v] = 0;

}

}

return false;

}

bool graphColoring(bool graph[V][V], int m) {

int color[V];

for (int i = 0; i < V; i++) color[i] = 0;

if (graphColoringUtil(graph, m, color, 0) == false) {

printf("Solution does not exist");

return false;

}

printSolution(color);

return true;

}

void printSolution(int color[]) {

printf("Solution Exists:"

" Following are the assigned colors \n");

for (int i = 0; i < V; i++) printf(" %d ", color[i]);

printf("\n");

}

**Hamiltonian Cycle**

#define V 5

void printSolution(int path[]);

bool isSafe(int v, bool graph[V][V], int path[], int pos) {

if (graph [path[pos - 1]][ v ] == 0) return false;

for (int i = 0; i < pos; i++)

if (path[i] == v)

return false;

return true;

}

bool hamCycleUtil(bool graph[V][V], int path[], int pos) {

if (pos == V) {

if (graph[path[pos - 1]][path[0]] == 1) return true;

else return false;

}

for (int v = 1; v < V; v++) {

if (isSafe(v, graph, path, pos)) {

path[pos] = v;

if (hamCycleUtil (graph, path, pos + 1) == true) return true;

path[pos] = -1;

}

}

return false;

}

bool hamCycle(bool graph[V][V]) {

int \*path = new int[V];

for (int i = 0; i < V; i++) path[i] = -1;

path[0] = 0;

if (hamCycleUtil(graph, path, 1) == false ) {

cout << "\nSolution does not exist";

return false;

}

printSolution(path);

return true;

}

void printSolution(int path[]) {

cout << "Solution Exists:"

" Following is one Hamiltonian Cycle \n";

for (int i = 0; i < V; i++)

cout << path[i] << " ";

cout << path[0] << " ";

cout << endl;

}

**Sudoku**

#define UNASSIGNED 0

#define N 9

bool FindUnassignedLocation(int grid[N][N], int &row, int &col);

bool isSafe(int grid[N][N], int row, int col, int num);

bool SolveSudoku(int grid[N][N]) {

int row, col;

if (!FindUnassignedLocation(grid, row, col)) return true;

for (int num = 1; num <= 9; num++) {

if (isSafe(grid, row, col, num)) {

grid[row][col] = num;

if (SolveSudoku(grid)) return true;

grid[row][col] = UNASSIGNED;

}

}

return false;

}

bool FindUnassignedLocation(int grid[N][N], int &row, int &col)

{

for (row = 0; row < N; row++)

for (col = 0; col < N; col++)

if (grid[row][col] == UNASSIGNED) return true;

return false;

}

bool UsedInRow(int grid[N][N], int row, int num) {

for (int col = 0; col < N; col++)

if (grid[row][col] == num) return true;

return false;

}

bool UsedInCol(int grid[N][N], int col, int num) {

for (int row = 0; row < N; row++)

if (grid[row][col] == num) return true;

return false;

}

bool UsedInBox(int grid[N][N], int boxStartRow, int boxStartCol, int num) {

for (int row = 0; row < 3; row++)

for (int col = 0; col < 3; col++)

if (grid[row + boxStartRow][col + boxStartCol] == num)

return true;

return false;

}

bool isSafe(int grid[N][N], int row, int col, int num) {

return !UsedInRow(grid, row, num) && !UsedInCol(grid, col, num) && !UsedInBox(grid, row - row % 3 , col - col % 3, num) && grid[row][col] == UNASSIGNED;

}

void printGrid(int grid[N][N]) {

for (int row = 0; row < N; row++) {

for (int col = 0; col < N; col++) cout << grid[row][col] << " ";

cout << endl;

}

}

**Permutation**

void permute(string a, int l, int r) {

if (l == r) cout<<a<<endl;

else {

for (int i = l; i <= r; i++) {

swap(a[l], a[i]);

permute(a, l+1, r);

swap(a[l], a[i]);

}

}

}

**Linked list**

struct Node {

int data;

struct Node \*next;

};

struct Node\* head = NULL;

void insert(int new\_data) {

struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

new\_node->data = new\_data;

new\_node->next = head;

head = new\_node;

}

void display() {

struct Node\* ptr;

ptr = head;

while (ptr != NULL) {

cout<< ptr->data <<" ";

ptr = ptr->next;

}

}

**Graph**

void addEdge(vector<int> adj[], int u, int v) {

adj[u].push\_back(v);

adj[v].push\_back(u);

}

void printGraph(vector<int> adj[], int V) {

for (int v = 0; v < V; ++v) {

cout << "\n Adjacency list of vertex " << v << "\n head ";

for (auto x : adj[v]) cout << "-> " << x;

printf("\n");

}

}

**Tree**

**Segment Tree (Build)**

void build(int at, int L, int R) {

sum[at] = 0;

if (L == R) return;

int mid = (L + R) / 2;

build(at \* 2, L, mid);

build(at \* 2 + 1, mid + 1, R);

}

**Segment Tree (Update)**

void update(int at, int L, int R, int pos, int u) {

if (L == R) {

sum[at] += u;

return;

}

int mid = (L + R) / 2;

if (pos <= mid) update(at \* 2, L, mid, pos, u);

else update(at \* 2 + 1, mid + 1, R, pos, u);

sum[at] = sum[at \* 2] + sum[at \* 2 + 1];

}

**Segment Tree (Query)**

int query(int at, int L, int R, int l, int r) {

if (r < L || R < l) return 0;

if (l <= L && R <= r) return sum[at];

int mid = (L + R) / 2;

int x = query(at \* 2, L, mid, l, r);

int y = query(at \* 2 + 1, mid + 1, R, l, r);

return x + y;

}

**Huffman**

#include <vector>

#include <queue>

#include <functional>

int n, freq[100];

int huffman() {

priority\_queue<int, vector<int>, greater<int>> PQ;

for (int i = 0; i < n; i++) PQ.push(freq[i]);

while (PQ.size() != 1) {

int a = PQ.top(); PQ.pop();

int b = PQ.top(); PQ.pop();

PQ.push(a + b);

}

return PQ.top();

}

**BFS**

#include <vector>

#include <queue>

vector<int> adj[100];

int visited[100];

void bfs(int s, int n) {

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

queue<int> Q;

Q.push(s);

visited[s] = 1;

while (!Q.empty()) {

int u = Q.front();

Q.pop();

for (int i = 0; i < adj[u].size(); i++) {

if (visited[adj[u][i]] == 0) {

int v = adj[u][i];

visietd[v] = 1;

Q.push(v);

}

}

}

}

**DFS**

#include <vector>

vector<int> adj[100];

int vis[100];

void dfs(int at) {

if (vis[at]) return;

vis[at] = 1;

for (int i = 0; i < vis[at].size(); i++) dfs(vis[at][i]);

}

**Hashing**

#include<iostream>

#include <list>

using namespace std;

class Hash {

int BUCKET;

list<int> \*table;

public:

Hash(int V);

void insertItem(int x);

void deleteItem(int key);

int hashFunction(int x) {

return (x % BUCKET);

}

void displayHash();

};

Hash::Hash(int b) {

this->BUCKET = b;

table = new list<int>[BUCKET];

}

void Hash::insertItem(int key) {

int index = hashFunction(key);

table[index].push\_back(key);

}

void Hash::deleteItem(int key) {

int index = hashFunction(key);

list <int> :: iterator i;

for (i = table[index].begin(); i != table[index].end(); i++) {

if (\*i == key)

break;

}

if (i != table[index].end()) table[index].erase(i);

}

void Hash::displayHash() {

for (int i = 0; i < BUCKET; i++) {

cout << i;

for (auto x : table[i])

cout << " --> " << x;

cout << endl;

}}

**Matrix Inversion**

#include <vector>

#include <cmath>

#include <algorithm>

const double EPS = 1e-10;

typedef vector<int> VI;

typedef double T;

typedef vector<T> VT;

typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {

const int n = a.size();

const int m = b.size();

VI irow(n), icol(n), ipiv(n);

T det = 1;

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

int pj = -1, pk = -1;

for (int j = 0; j < n; j++) if (!ipiv[j])

for (int k = 0; k < n; k++) if (!ipiv[k])

if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) {pj = j; pk = k;}

if (fabs(a[pj][pk]) < EPS) {

cerr << "Matrix is singular." << endl;

exit(0);

}

ipiv[pk]++;

swap(a[pj], a[pk]);

swap(b[pj], b[pk]);

if (pj != pk) det \*= -1;

irow[i] = pj;

icol[i] = pk;

T c = 1.0 / a[pk][pk];

det \*= a[pk][pk];

for (int p = 0; p < n; p++) a[pk][p] \*= c;

for (int p = 0; p < m; p++) b[pk][p] \*= c;

for (int p = 0; p < n; p++)

if (p != pk) {

c = a[p][pk];

a[p][pk] = 0;

for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] \* c;

for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] \* c;

}

for (int p = n - 1; p >= 0; p--)

if (irow[p] != icol[p]) {

for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);

}

return det;

}

}