Câu 1:

**I. Ý TƯỞNG BÀI TOÁN**

Cho N hình tròn, mỗi hình tròn có tâm (x, y) và bán kính r.  
Người chơi có thể di chuyển giữa hai hình tròn nếu thỏa mãn điều kiện:

* Nếu khoảng cách giữa mép hai hình tròn ≤ 50 cm → có thể bước sang (loại 0).
* Nếu khoảng cách giữa mép hai hình tròn > 50 và ≤ 60 cm → có thể nhảy sang (loại 1).
* Các trường hợp khác: không thể di chuyển.

→ Mỗi hình tròn được xem là một đỉnh trong đồ thị.  
→ Các cạnh nối biểu diễn việc có thể “bước” hoặc “nhảy” giữa hai hình tròn.  
→ Mục tiêu: Tìm đường đi tốt nhất từ hình tròn S đến T sao cho:

1. Ít số lần nhảy nhất.
2. Nếu số lần nhảy bằng nhau → chọn đường đi qua ít hình tròn hơn.

**II. PHÂN TÍCH CHƯƠNG TRÌNH**

**1. Cấu trúc dữ liệu**

* Dùng struct Circle lưu tâm (x, y) và bán kính r của từng hình tròn.
* Dùng danh sách kề adj để lưu các hình tròn có thể đi được với nhau.  
  Mỗi cạnh gồm: số hiệu hình tròn và loại di chuyển (0: bước, 1: nhảy).

**2. Hàm tính khoảng cách**Hàm edgeDist(a, b) tính khoảng cách giữa mép hai hình tròn:  
√((x₁ - x₂)² + (y₁ - y₂)²) - (r₁ + r₂).  
Nếu ≤ 50 → bước, nếu ≤ 60 → nhảy.

**3. Nhập dữ liệu**

* Nhập N, S, T.
* Sau đó nhập (x, y, r) cho từng hình tròn.

**4. Tạo đồ thị**

* Duyệt qua mọi cặp (i, j):
  + Nếu khoảng cách ≤ 50 → thêm cạnh loại 0.
  + Nếu ≤ 60 → thêm cạnh loại 1.
* Cạnh được nối hai chiều.

**5. Tìm đường đi tốt nhất**

* Dùng Dijkstra để tìm đường đi có số lần nhảy ít nhất.
* Nếu số lần nhảy bằng nhau → chọn đường đi ngắn hơn.
* Mảng parent[] lưu đỉnh cha, moveType[] lưu loại di chuyển**.**

**6. Truy vết và in kết quả**

* Nếu không có đường đi → in 1.
* Nếu có:
  + In 0.
  + In số lần nhảy và số hình tròn đi qua.
  + In thứ tự các hình tròn và loại di chuyển (0: bước, 1: nhảy).

III. Kết quả code:

A black background with white text

AI-generated content may be incorrect.

Câu 2:

**I. Ý TƯỞNG BÀI TOÁN**

* Ta có một bảng kích thước N × M, mỗi ô chứa giá trị A[i][j].
* Chỉ được di chuyển qua các ô có giá trị khác 0, và chỉ đi kề cạnh (trên, dưới, trái, phải).
* Chi phí đường đi = tổng các giá trị trên các ô đi qua, không tính ô đầu tiên và ô cuối cùng.
* Cần tìm đường đi ngắn nhất (tổng chi phí nhỏ nhất) từ ô (XI, YI) đến (XJ, YJ).

→ Bài này là bài toán đường đi ngắn nhất trên lưới, có trọng số.  
→ Giải bằng thuật toán Dijkstra.

**II. PHÂN TÍCH CHƯƠNG TRÌNH**

**1. Cấu trúc dữ liệu**

struct Node {

int x, y, cost;

bool operator>(const Node &o) const { return cost > o.cost; }

};

Dùng struct Node để lưu tọa độ (x, y) và chi phí tạm thời (cost).

Toán tử > giúp ưu tiên chi phí nhỏ hơn khi dùng trong hàng đợi ưu tiên (priority\_queue).

**2. Nhập dữ liệu**

cin >> N >> M >> XI >> YI >> XJ >> YJ;

* Nhập kích thước bảng, và hai ô đầu – cuối.
* Sau đó nhập toàn bộ ma trận A[i][j].

Nếu **ô bắt đầu hoặc ô kết thúc có giá trị 0** → không thể đi, in 1 và dừng.

1. **Khởi tạo:**

vector<vector<int>> dist(N + 1, vector<int>(M + 1, 1e9));

vector<vector<pair<int,int>>> parent(N + 1, vector<pair<int,int>>(M + 1, {-1,-1}));

* dist[i][j]: chi phí ngắn nhất từ ô đầu đến ô (i, j).
* parent[i][j]: lưu **ô trước đó** để truy vết lại đường đi.
* Mảng dx, dy dùng để di chuyển 4 hướng.

1. **Thuật toán Dijkstra**

priority\_queue<Node, vector<Node>, greater<Node>> pq;

pq.push({XI, YI, 0});

dist[XI][YI] = 0;

1. Dùng **priority\_queue** để chọn ô có chi phí nhỏ nhất hiện tại.
2. Mỗi lần lấy ô (x, y), xét 4 ô kề (nx, ny):

* Nếu **vượt biên hoặc gặp ô 0** → bỏ qua.
* Tính **chi phí thêm**:

int add = (nx == XJ && ny == YJ) ? 0 : A[nx][ny];

→ Không tính ô cuối.

* Nếu nc < dist[nx][ny] → cập nhật chi phí và cha.

**5. Xuất kết quả**

* Nếu không có đường đi (dist[XJ][YJ] == 1e9) → in 1.
* Ngược lại:
  + In 1 (có đường đi).
  + In **độ dài ngắn nhất** = dist[XJ][YJ].
  + Truy ngược parent[][] từ ô đích về ô đầu để in ra toàn bộ đường đi.

**III. Kết quả code:**

**A screenshot of a computer

AI-generated content may be incorrect.**