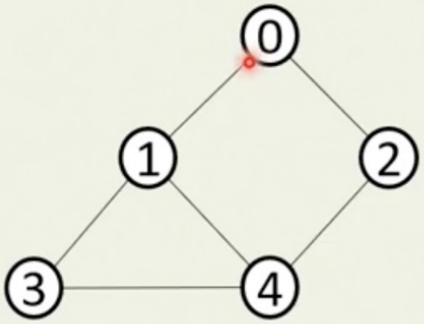
BFS – 큐(Queue)

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

#define MAX_N 10
int N, E;
int Graph[MAX_N][MAX_N];

int main() {
  cin >> N >> E;
  memset(Graph, 0, sizeof(Graph));
  for (int i = 0; i < E; ++i) {
    int u, v;
    cin >> u >> v;
    Graph[u][v] = Graph[v][u] = 1;
  }
  bfs(0);
  return 0;
}
```

```cpp
void bfs(int node) {
  bool visited[MAX_N] = { false };

  queue<int> myqueue;
  visited[node] = true;
  myqueue.push(node);

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

    cout << curr << ' ';

    for (int next = 0; next < N; ++next) {
      if (!visited[next] && Graph[curr][next]) {
        visited[next] = true;
        myqueue.push(next);
      }
    }
  }
}
```



<입력>

```
5 6
0 1 0 2 1 3 1 4 2 4 3 4
```

BFS 활용 - Shortest path → 최단경로를 구할 때

```cpp
#include <iostream>
#include <queue>
using namespace std;
#define MAX_N 10
struct Point {
  int row, col, dist;
};
int D[4][2] = { {-1,0},{1,0},{0,-1},{0,1} };
int N, Board[MAX_N][MAX_N];

int main() {
  cin >> N;
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      cin >> Board[i][j];

  int srcRow, srcCol, dstRow, dstCol;
  cin >> srcRow >> srcCol >> dstRow >> dstCol;

  cout << bfs(srcRow, srcCol, dstRow, dstCol);
  return 0;
}
```

```cpp
int bfs(int srcRow, int srcCol, int dstRow, int dstCol) {
  bool visited[MAX_N][MAX_N] = { false };
  queue<Point> myqueue;
  visited[srcRow][srcCol] = true;
  myqueue.push({ srcRow, srcCol, 0 });
  while (!myqueue.empty()) {
    Point curr = myqueue.front();
    myqueue.pop();
    if (curr.row == dstRow && curr.col == dstCol)
      return curr.dist;

    for (int i = 0; i < 4; ++i) {
      int nr = curr.row + D[i][0], nc = curr.col + D[i][1];
      if (nr < 0 || nr > N - 1 || nc < 0 || nc > N - 1) continue;
      if (visited[nr][nc]) continue;
      if (Board[nr][nc] == 1) continue;
      visited[nr][nc] = true;
      myqueue.push({ nr, nc, curr.dist + 1 });
    }
  }
  return -1;
}
```

<입력>

```
5
0 0 0 0 0
0 1 1 1 1
0 0 0 0 0
1 1 1 1 0
0 0 0 0 0
0 1 4 2
```

<출력>

```
11
```

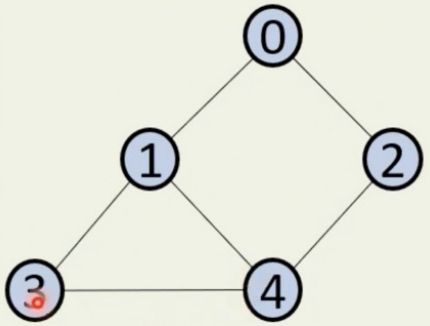
방문가능 여부

1. 이미 방문X
2. 값 범위 안에 있을 때
3. 방문할 수 있는 상황

DFS – 재귀 호출 → 다양 방문할 곳이 없을 때까지

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

#define MAX_N 10
int N, E;
int Graph[MAX_N][MAX_N];
bool Visited[MAX_N];

int main() {
  cin >> N >> E;
  memset(Visited, 0, sizeof(Visited));
  memset(Graph, 0, sizeof(Graph));
  for (int i = 0; i < E; ++i) {
    int u, v;
    cin >> u >> v;
    Graph[u][v] = Graph[v][u] = 1;
  }
  dfs(0);
  return 0;
}
```

```cpp
void dfs(int node) {
  Visited[node] = true;
  cout << node << ' ';

  for (int next = 0; next < N; ++next) {
    if (!Visited[next] && Graph[node][next])
      dfs(next);
  }
}
```



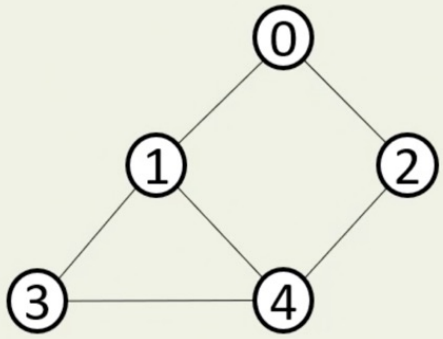
<입력>

```
5 6
0 1 0 2 1 3 1 4 2 4 3 4
```

DFS – 스택(Stack)

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

#define MAX_N 10
int N, E;
int Graph[MAX_N][MAX_N];

int main() {
  cin >> N >> E;
  memset(Graph, 0, sizeof(Graph));

  for (int i = 0; i < E; ++i) {
    int u, v;
    cin >> u >> v;
    Graph[u][v] = Graph[v][u] = 1;
  }
  dfs(0);
  return 0;
}
```

```cpp
void dfs(int node) {
  bool visited[MAX_N] = { false };

  stack<int> mystack;
  mystack.push(node);

  while (!mystack.empty()) {
    int curr = mystack.top();
    mystack.pop();

    if (visited[curr]) continue;

    visited[curr] = true;
    cout << curr << ' ';

    for (int next = 0; next < N; ++next) {
      if (!visited[next] && Graph[curr][next])
        mystack.push(next);
    }
  }
}
```



<입력>

```
5 6
0 1 0 2 1 3 1 4 2 4 3 4
```

DFS 활용 - Flood fill → 시작점과 연결된 모든 칸, 채우기에 활용

```
struct Point {
  int row, col;
};
int D[4][2] = {{-1,0},{1,0},{0,-1},{0,1}};
int N, Board[MAX_N][MAX_N];
int main() {
  cin >> N;
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      cin >> Board[i][j];
  int sr, sc, color;
  cin >> sr >> sc >> color;
  dfs(sr, sc, color);
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      cout << Board[i][j] << ' ';
    }
    cout << endl;
  }
  return 0;
}
```

```
void dfs(int r, int c, int color) {
  bool visited[MAX_N][MAX_N] = { false };
  stack<Point> mystack;
  mystack.push({ r, c });

  while (!mystack.empty()) {
    Point curr = mystack.top();
    mystack.pop();
    if (visited[curr.row][curr.col]) continue;

    visited[curr.row][curr.col] = true;
    Board[curr.row][curr.col] = color;

    for (int i = 0; i < 4; ++i) {
      int nr = curr.row + D[i][0], nc = curr.col + D[i][1];
      if (nr < 0 || nr > N-1 || nc < 0 || nc > N-1) continue;
      if (visited[nr][nc]) continue;
      if (Board[nr][nc] == 1) continue;
      mystack.push({ nr, nc });
    }
  }
}
```

<입력>

```
5
0 0 0 0 0
0 0 0 1 1
0 0 0 1 0
1 1 1 1 0
0 0 0 0 0
1 1 3
```

<출력>

```
3 3 3 3 3
3 3 3 1 1
3 3 3 1 0
1 1 1 1 0
0 0 0 0 0
```

Dijkstra – 모든 정점까지 거리 구하기 (1)

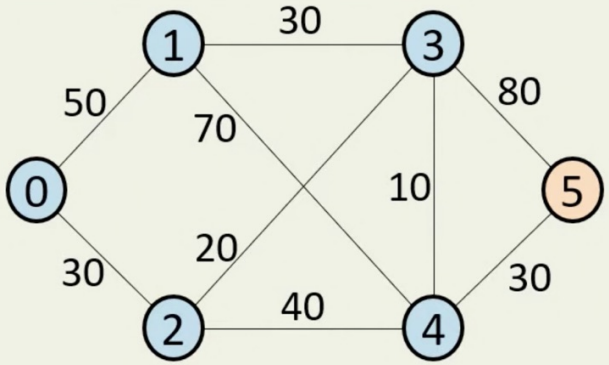
```cpp
#include <iostream>
#include <queue>

using namespace std;

#define INF 987654321
#define MAX_N 10

typedef pair<int, int> pii;

int N, E;
int Graph[MAX_N][MAX_N], Dist[MAX_N];

int main() {
  cin >> N >> E;
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      if (i == j) Graph[i][j] = 0;
      else        Graph[i][j] = INF;
    }
  }
  for (int i = 0; i < E; ++i) {
    int u, v, cost;
    cin >> u >> v >> cost;
    Graph[u][v] = Graph[v][u] = cost;
  }

  dijkstra(0);

  for (int i = 0; i < N; ++i)
    cout << Dist[i] << ' ';
  cout << endl;
  return 0;
}
```

<입력>

```
6 9
0 1 50
0 2 30
1 3 30
1 4 70
2 3 20
2 4 40
3 4 10
3 5 80
4 5 30
```

Dijkstra – 모든 정점까지 거리 구하기 (2)

```cpp
void dijkstra(int src) {
  priority_queue<pii, vector<pii>, greater<pii>> pq;
  bool visited[MAX_N] = { false };
  for (int i = 0; i < N; ++i) Dist[i] = INF;
  Dist[src] = 0;
  pq.push(make_pair(0, src));

  while (!pq.empty()) {
    int u = pq.top().second;
    pq.pop();
    if (visited[u]) continue;

    visited[u] = true;
    for (int v = 0; v < N; ++v) {
      if (Dist[v] > Dist[u] + Graph[u][v]) {
        Dist[v] = Dist[u] + Graph[u][v];
        pq.push(make_pair(Dist[v], v));
      }
    }
  }
```



Dist

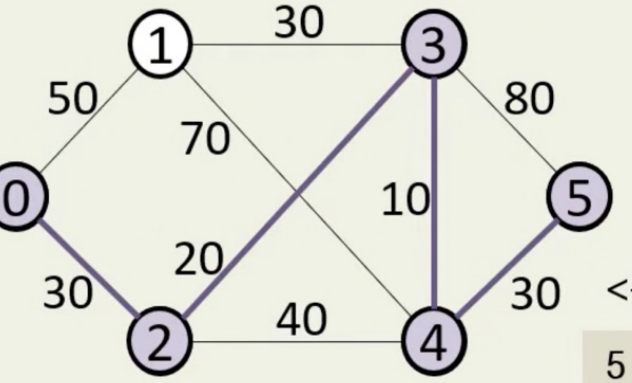
0	1	2	3	4	5
0	50	30	50	60	90

visited

0	1	2	3	4	5
T	T	T	T	T	T

Dijkstra – 경로가 필요한 경우

```
...
int Graph[MAX_N][MAX_N], Dist[MAX_N], Prev[MAX_N];
int main() {
  ...
  dijkstra(0);
  int curr = 5;
  while (curr != -1) {
    cout << curr << " < ";
    curr = Prev[curr];
  }
  return 0;
}
```



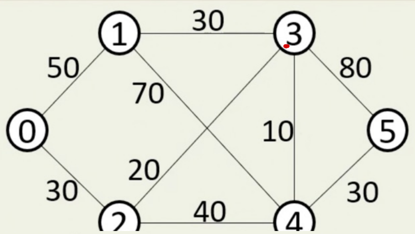
<출력>

5 < 4 < 3 < 2 < 0 <

```
void dijkstra(int src) {
  priority_queue<pii, vector<pii>, greater<pii>> pq;
  bool visited[MAX_N] = { false };
  for (int i = 0; i < N; ++i) {
    Prev[i] = -1;
    Dist[i] = INF;
  }
  Dist[src] = 0;
  pq.push(make_pair(0, src));
  while (!pq.empty()) {
    int u = pq.top().second;
    pq.pop();
    if (visited[u]) continue;
    visited[u] = true;
    for (int v = 0; v < N; ++v) {
      if (Dist[v] > Dist[u] + Graph[u][v]) {
        Prev[v] = u;
        Dist[v] = Dist[u] + Graph[u][v];
        pq.push(make_pair(Dist[v], v));
      }
    }
  }
}
```

Dijkstra – 특정 도착점까지 거리 구하기 (1)

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

using namespace std;

#define INF 987654321
#define MAX_N 10

typedef pair<int, int> pii;

int N, E;
int Graph[MAX_N][MAX_N], Dist[MAX_N];
```




```cpp
int main() {
  cin >> N >> E;
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      if (i == j) Graph[i][j] = 0;
      else        Graph[i][j] = INF;
    }
  }
  for (int i = 0; i < E; ++i) {
    int u, v, cost;
    cin >> u >> v >> cost;
    Graph[u][v] = Graph[v][u] = cost;
  }

  for (int i = 0; i < N; ++i) {
    cout << dijkstra(0, i) << endl;
  }
  return 0;
}
```

초기화

값입력

<입력>

```
6 9
0 1 50
0 2 30
1 3 30
1 4 70
2 3 20
2 4 40
3 4 10
3 5 80
4 5 30
```

출력

0 → 0-0
50 → 0-1
30 → 0-2
50 → 0-3
60 → 0-4
90 → 0-5

갈 때 최단거리.

Dijkstra – 특정 도착점까지 거리 구하기 (2)

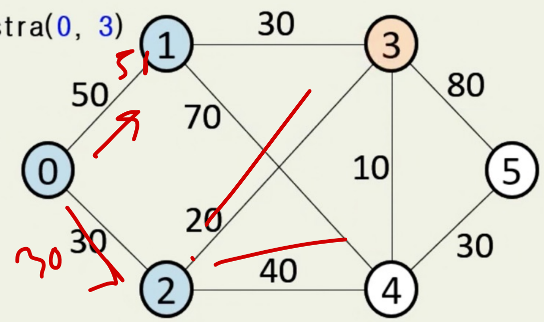
```cpp
int dijkstra(int src, int dst) {
  priority_queue<pii, vector<pii>, greater<pii>> pq;
  bool visited[MAX_N] = { false };
  for (int i = 0; i < N; ++i) Dist[i] = INF;
  Dist[src] = 0;
  pq.push(make_pair(0, src));
  while (!pq.empty()) {
    int u = pq.top().second;
    if (u == dst) return pq.top().first;
    pq.pop();
    if (visited[u]) continue;

    visited[u] = true;
    for (int v = 0; v < N; ++v) {
      if (Dist[v] > Dist[u] + Graph[u][v]) {
        Dist[v] = Dist[u] + Graph[u][v];
        pq.push(make_pair(Dist[v], v));
      }
    }
  }
  return INF;
}
```

목적지

제일 좋은 거리

현재위치가 목적지와 같으면 stop

dijkstra(0, 3)



Dist

0	1	2	3	4	5
0	50	30	50	70	INF

visited

0	1	2	3	4	5
T	T	T	F	F	F