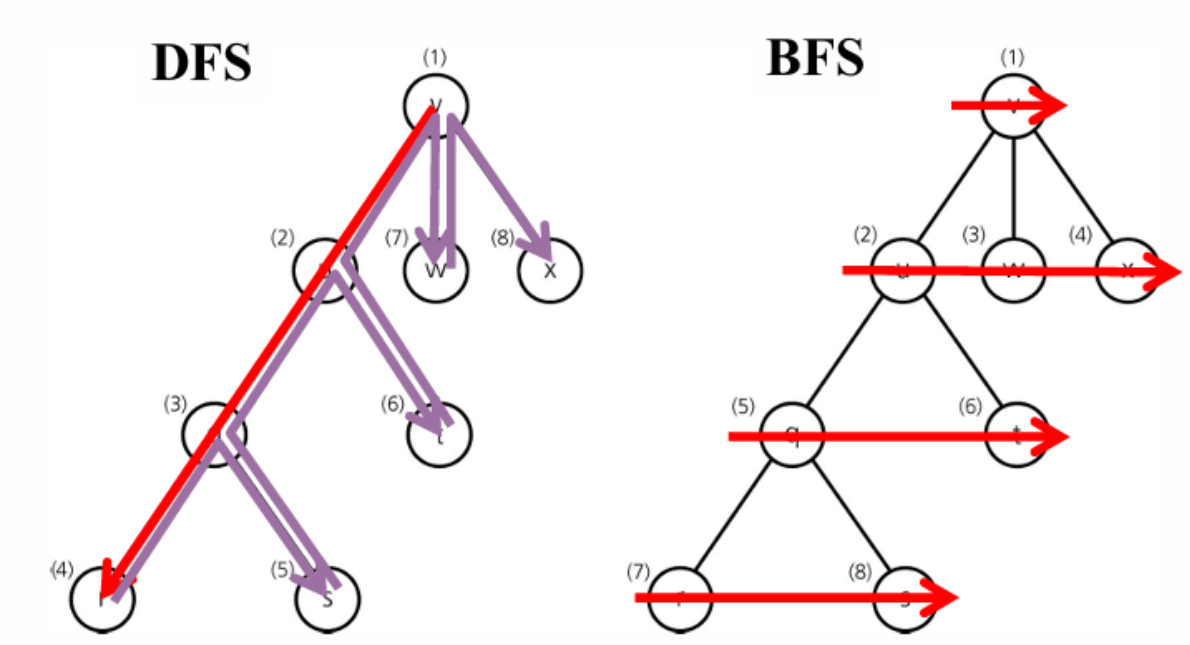
Dijkstra Algorithm

우선 다익스트라 알고리즘을 이해하기 전에 DFS와 BFS를 이해하고 넘어가고자 한다,

DFS : 깊이 우선 탐색으로 한 노드가 리프 노드에 도달할 때 까지 타고 내려간다.

BFS : 너비 우선 탐색으로 같은 높이 노드들을 탐사하고 내려가는 방식으로 전개한다.



주로 DFS는 Recursive + Back Tracking 방식이 사용되고, BFS는 queue를 활용한다.

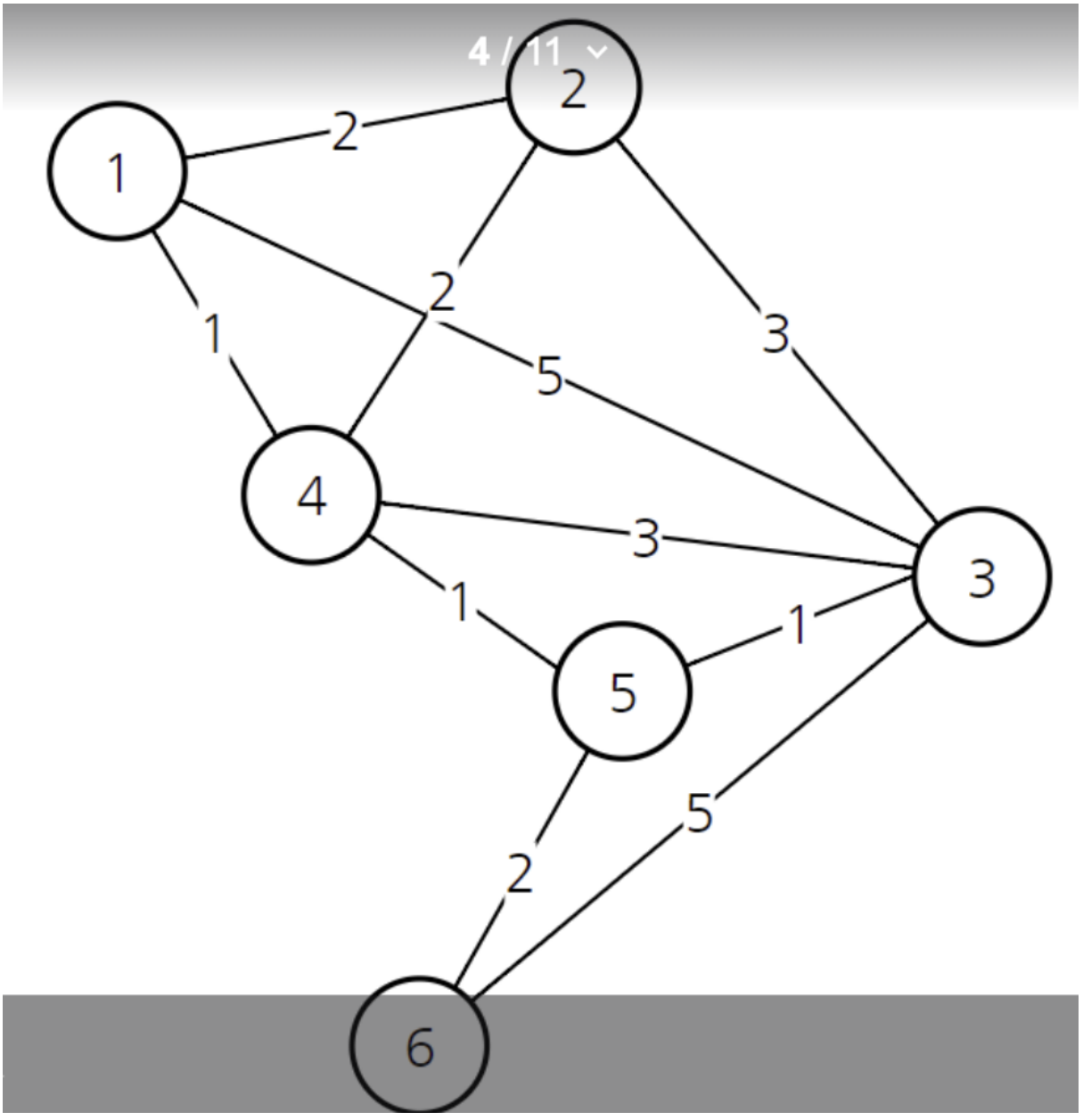
대표적인 방식을 활용하여 노드들을 탐사하는데 이 때 최적의 노드 탐사 방법으로 Dijkstra 알고리즘과 Bellman-Ford 알고리즘이 있다.

Dijkstra Algorithm은 노드들의 가중치가 음수가 아닐 때 적용할 수 있는 최적의 노드 탐사 알고리즘으로 한 지점에서 시작한다.

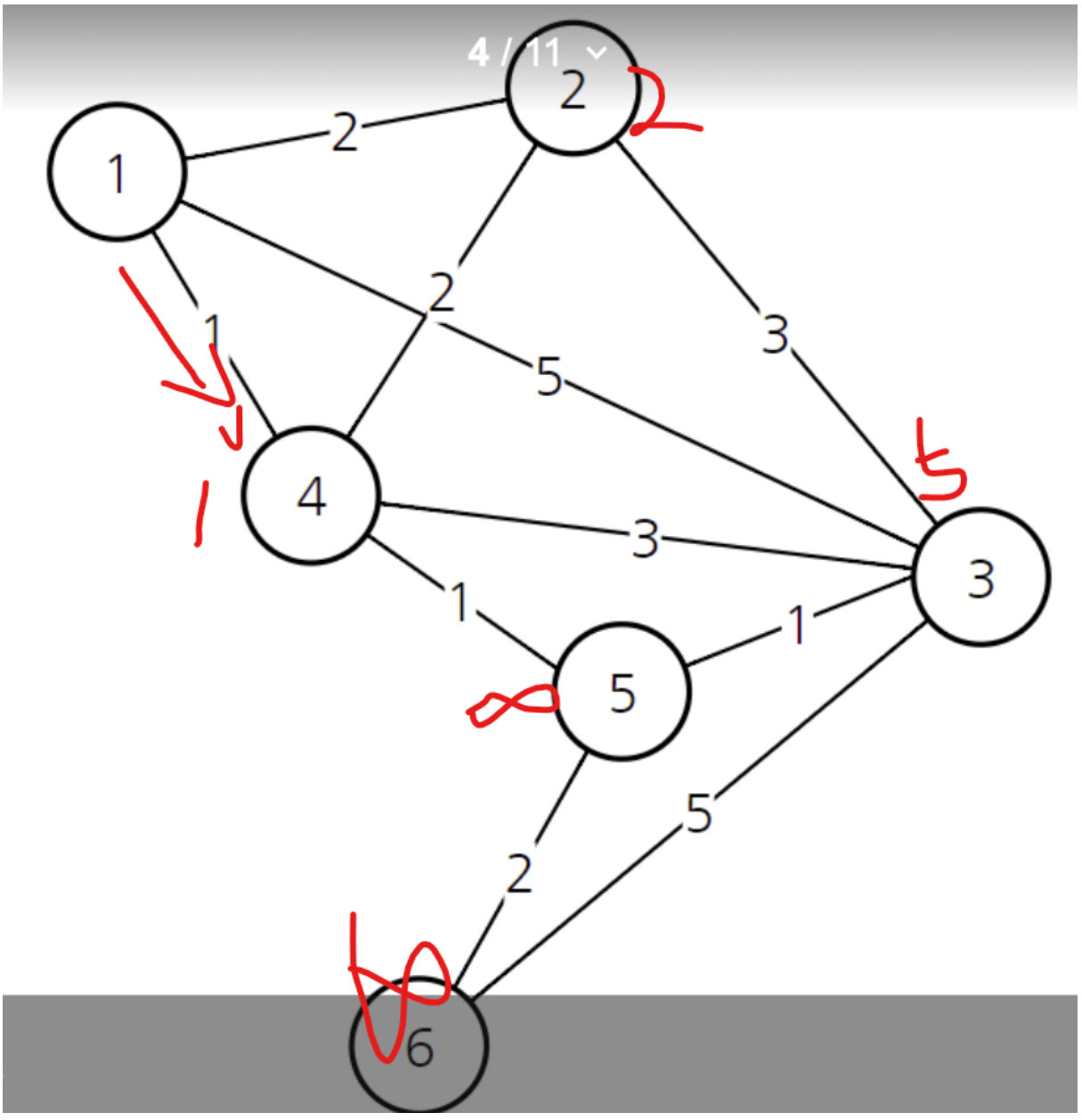
알고리즘은 다음과 같다.

- 시작 지점 노드에서 방문하지 않은 인접한 노드 중 가장 짧은 노드를 택한다.
- 가장 짧은 노드를 통과할 때, 시작 노드로부터 각 노드 사이의 거리를 갱신한다.
- 갱신한 거리 중 가장 짧은 거리를 택하여 모든 노드를 방문할 때 까지 계속 반복한다.'
- 단 방문한 노드로 다시 이동할 순 없다.

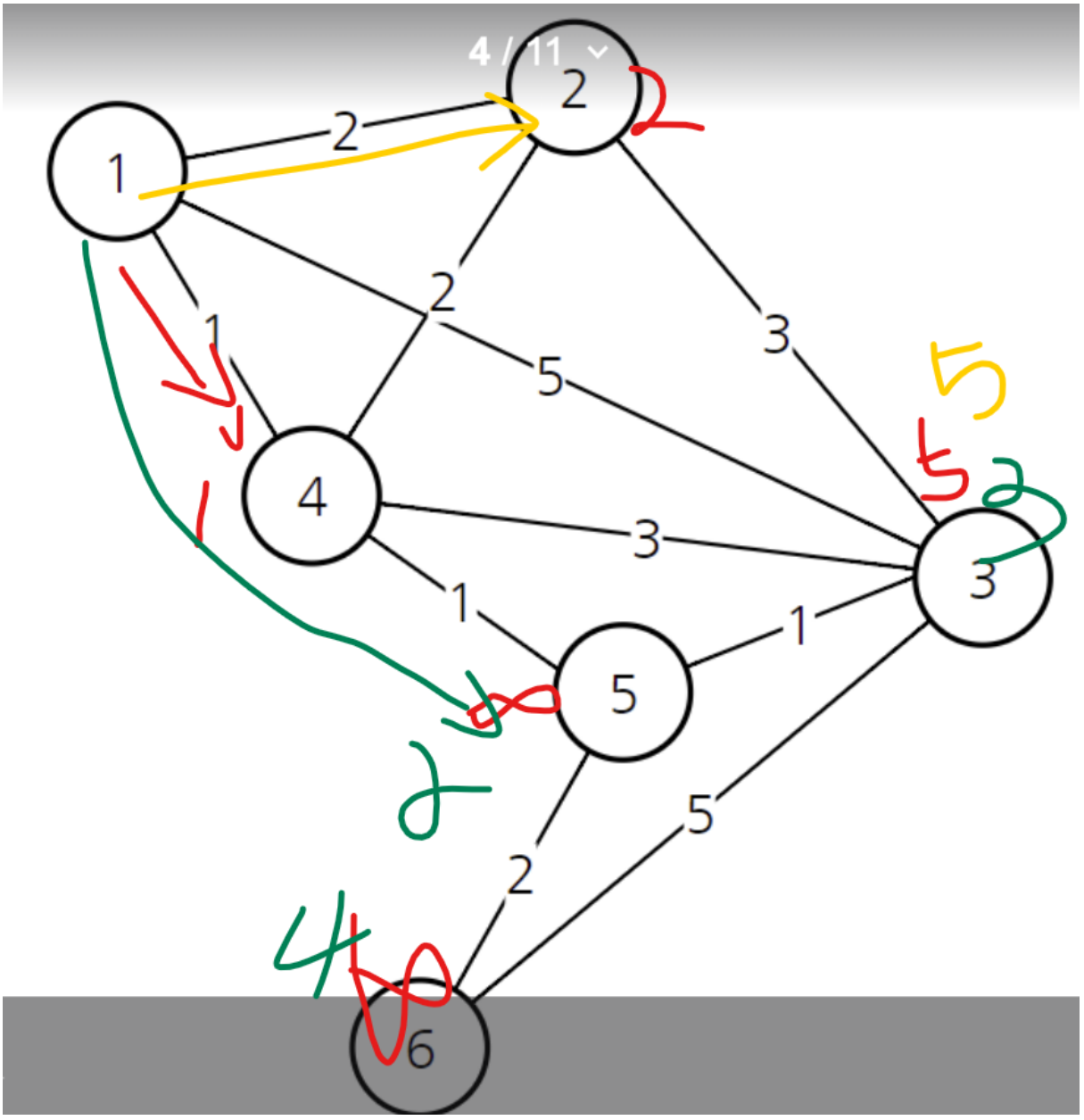
그림으로 이해를 해보자.



1번에서 6번 노드 까지 존재할 때, 6번 노드까지 최단 거리로 모든 노드를 탐색할 수 있는 방법을 알아보도록 한다. 1번에서 인접한 노드는 그대로 거리 작성하되, 인접하지 않은 노드들은 무한대로 처리해준다.



이제, 1번 노드에서 가장 가까운 거리가 1인 4번 노드로 이동한다. 4번 노드로 이동하게 되면 1번노드를 기준으로 5번노드를 1→4→5로 이동할 수 있기 때문에 거리가 2로 갱신 된다. 그러면 1번 노드를 기점으로 2, 3, 5번 노드가 각각 2, 5, 2가 되기 때문에 2번 노드나 5번 노드로 이동한다.



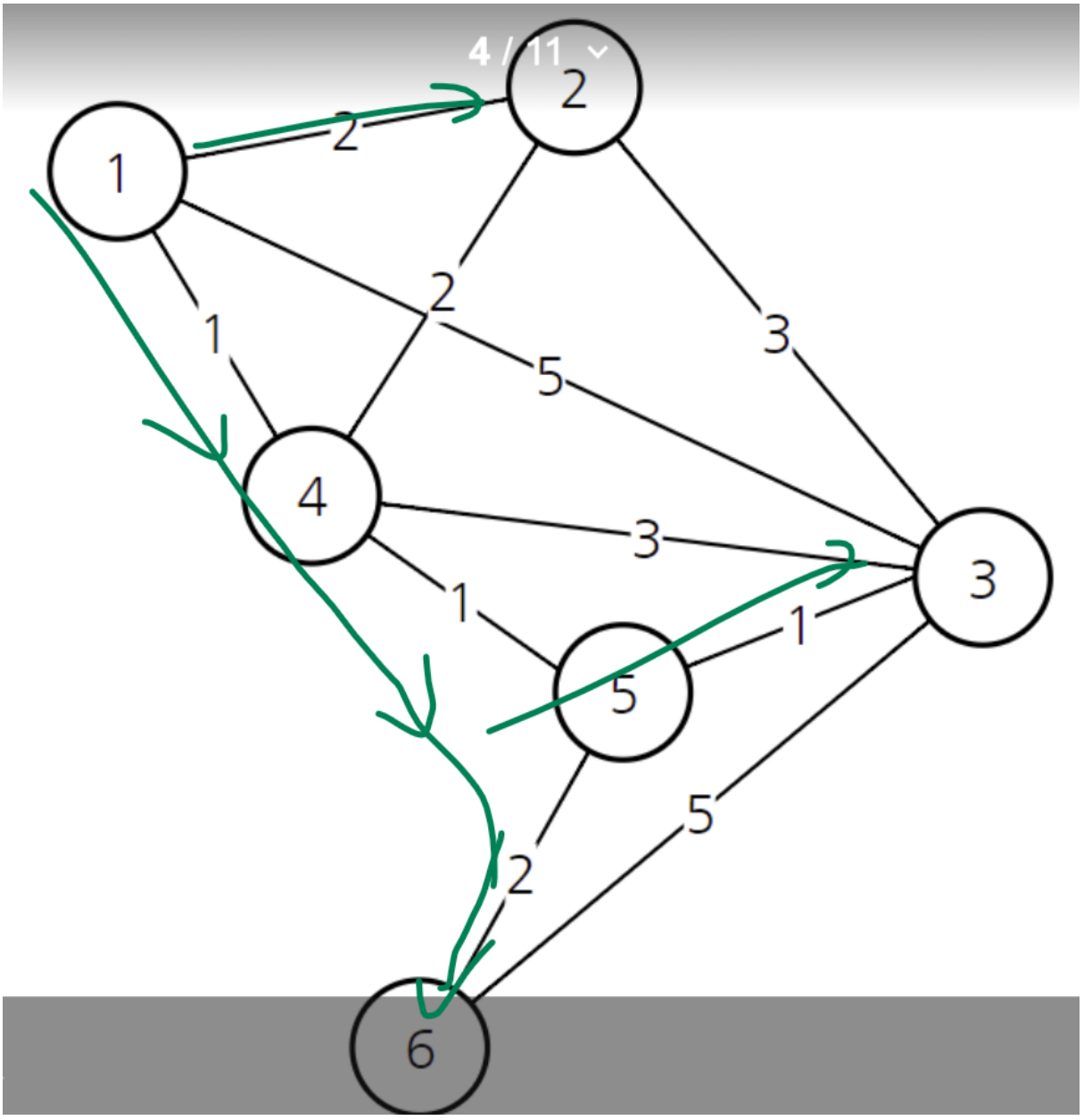
2번 노드로 이동할 경우, 1번 노드에서 방문하지 않은 3번 노드, 5번 노드, 6번 노드의 거리가 5 2 무한대가 된다.

그래서 그 다음 거리가 짧은 노드인 5번 노드로 이동한다.

5번 노드로 이동하면 3번 노드까지의 거리가 5→3으로 갱신되고, 6번 노드가 무한대→4로 갱신된다.

따라서 아직 방문하지 않은 노드는 3번, 6번 노드가 되고 이 때 거리는 각각 3, 4이다.

즉, 3번 노드로 이동하게 되고, 1→3→6인 거리와, 1→4→5→6으로 이동하는 거리를 비교하여 가장 짧은 거리로 6번 노드를 이동한다. 즉 전반적인 노드 경로는 다음과 같다.



이제 코드로 짜는 방법은 BFS와 DP이다.

우선 DP는 각 노드마다 연결된 인접한 점들 사이의 거리를 Array로 저장한 다음, 시작 지점에서의 거리 Array의 거리 값을 계속 수정시켜 주면서 진행하면 된다.

BFS같은 경우는 queue를 설정하여 인접한 노드에서의 가장 짧은 노드를 넣어주고, 거리 벡터를 초기화 시켜주는 과정으로 진행한다. 각각 코드는 다음과 같다.

1. Dijkstra Algorithm Using DP

```
#include <iostream>
#include <vector>
#include <climits>

std::vector<std::vector<int>>weight(6, std::vector<int>(6));
std::vector<bool> visited(6,false);
std::vector<int>distance;
void dijkstra(int start, int n){
    distance = weight[start];
    visited[start] = true;

    distance[start] = 0;
    int cnt = 0;
    while(cnt!=n-2){
        int Min = 9999;
        int u;
        for(int i=1; i<distance.size(); i++){
            if(!visited[i]){
                if(Min > distance[i]){
                    Min = distance[i];
                    u = i;
                }
            }
            std::cout <<"Min : " << Min <<"\n";
        }
        std::cout << "u : " << u <<"\n";
        visited[u] = true;

        for(int w = 0; w < n; w++){
            if(!visited[w]){
                if(distance[w] > distance[u] + weight[u][w]) 
            }
        }
        for(int i=0; i<distance.size(); i++) std::cout << dis
        std::cout << "\n\n";
        cnt++;
    }
}
```

```
int main(){
    weight = {{0,2,5,1,9999,9999},
              {2,0,3,2,9999,9999},
              {5,3,0,3,1,5},
              {1,2,3,0,1,9999},
              {9999,9999,5,1,0,2},
              {9999,9999,5,9999,2,0}};

    dijkstra(0,6);

}
```

2. Dijkstra Algorithm Using BFS

```
#include <iostream>
#include <algorithm>
#include <queue>
#include <vector>
#include <climits>
int n, m;
std::vector<std::pair<int, int>> graph[1001];
std::vector<int> distance(1001, INT_MAX);

std::vector<int> dijkstra(int st){

    distance[st] =0;

    std::priority_queue<std::pair<int, int>> pq;
    pq.push({distance[st],st});

    while(!pq.empty()){
        int cost = pq.top().first;
        int currentVertex = pq.top().second;
        pq.pop();

        if(distance[currentVertex]<cost) continue;
```

```
        for(int i=0; i<graph[currentVertex].size(); i++){
            int nxt = graph[currentVertex][i].first;
            int nxtdist = graph[currentVertex][i].second+cost

            if(nxtdist < distance[nxt]){
                distance[nxt] = nxtdist;
                pq.push({nxtdist, nxt});
            }
        }
    }
}

int main(){
    std::cin >> n >> m;
    while (m--){
        int start, end, cost;
        std::cin >> start >> end >> cost;
        graph[start].push_back({end,cost});
    }

    int dept, destination;
    std::cin >> dept >> destination;

    dijkstra(dept);

    std::cout << distance[destination];
}
```