# 데이터 구조 11장 실습과제

20223100 박신조

## 11.8 kruskal의 최소 비용 신장 트리 프로그램

```c
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

#define MAX_VERTICES 100
#define INF 1000

// 각 정점의 부모를 저장하는 배열 (Union-Find 구조 사용)
int parent[MAX_VERTICES];

// Union-Find: 집합 초기화 함수
void set_init(int n)
{
    for (int i = 0; i < n; i++)
        parent[i] = -1;  // 모든 노드를 루트로 초기화
}

// Union-Find: 해당 노드가 속한 집합의 루트(대표)를 찾는 함수
int set_find(int curr)
{
    if (parent[curr] == -1)
        return curr; // 자신이 루트이면 그대로 반환
    while (parent[curr] != -1)
        curr = parent[curr]; // 루트를 찾을 때까지 따라 올라감
    return curr;
}

// Union-Find: 두 집합을 합치는 함수
void set_union(int a, int b)
{
    int root1 = set_find(a); // a가 속한 집합의 루트를 찾음
    int root2 = set_find(b); // b가 속한 집합의 루트를 찾음
    if (root1 != root2)
        parent[root1] = root2; // 루트를 합쳐 하나의 집합으로 만듦
}

// 간선을 표현하는 구조체
struct Edge {
    int start, end, weight; // 시작 정점, 끝 정점, 가중치
};

// 그래프를 표현하는 구조체
typedef struct GraphType {
    int n; // 간선의 수
    int nvertex; //정점의 수/
    struct Edge edges[2 * MAX_VERTICES]; // 간선 배열
} GraphType;

// 그래프 초기화 함수
void graph_init(GraphType* g)
{
    g->n = 0;
    for (int i = 0; i < 2 * MAX_VERTICES; i++) {
        g->edges[i].start = 0;
        g->edges[i].end = 0;
        g->edges[i].weight = INF; // 기본적으로 큰 값으로 초기화
    }
}

// 간선을 그래프에 삽입하는 함수
void insert_edge(GraphType* g, int start, int end, int w)
{
    g->edges[g->n].start = start;
    g->edges[g->n].end = end;
    g->edges[g->n].weight = w;
    g->n++; // 간선 개수 증가
}
```

```c
// qsort에서 사용할 비교 함수 (가중치 기준 오름차순 정렬)
int compare(const void* a, const void* b)
{
    struct Edge* x = (struct Edge*)a;
    struct Edge* y = (struct Edge*)b;
    return (x->weight - y->weight);
}

// 크루스칼 알고리즘 함수
void kruskal(GraphType* g)
{
    int edge_accepted = 0;  // 현재까지 선택된 간선 수
    int uset, vset;         // 각 정점이 속한 집합 번호
    struct Edge e;

    set_init(g->n); // Union-Find 초기화
    qsort(g->edges, g->n, sizeof(struct Edge), compare); // 간선을 가중치 기준 정렬

    printf("크루스칼 최소 신장 트리 알고리즘 \n");
    int i = 0;
    while (edge_accepted < (g->nvertex - 1)) // MST는 간선이 (정점 수 - 1)개
    {
        e = g->edges[i]; // 가장 가중치가 낮은 간선부터 확인
        uset = set_find(e.start); // 시작 정점의 집합
        vset = set_find(e.end);   // 끝 정점의 집합
        if (uset != vset) { // 서로 다른 집합이면 사이클이 아님
            printf("간선 (%d,%d) %d 선택\n", e.start, e.end, e.weight);
            edge_accepted++; // 간선 선택
            set_union(uset, vset); // 두 정점을 같은 집합으로 합침
        }
        i++;
    }
}
// 메인 함수: 그래프 구성 및 Kruskal 실행
int main(void)
{
    GraphType* g;
    g = (GraphType*)malloc(sizeof(GraphType)); // 그래프 메모리 동적 할당
    graph_init(g); // 그래프 초기화

    g->nvertex = 7;
    // 그래프에 간선 추가 (총 9개)
    insert_edge(g, 0, 1, 29);
    insert_edge(g, 1, 2, 16);
    insert_edge(g, 2, 3, 12);
    insert_edge(g, 3, 4, 22);
    insert_edge(g, 4, 5, 27);
    insert_edge(g, 5, 0, 10);
    insert_edge(g, 6, 1, 15);
    insert_edge(g, 6, 3, 18);
    insert_edge(g, 6, 4, 25);

    kruskal(g); // 최소 신장 트리 생성
    free(g); // 동적 할당된 메모리 해제
    return 0;
}
```

```
크루스칼 최소 신장 트리 알고리즘
간선 (5,0) 10 선택
간선 (2,3) 12 선택
간선 (6,1) 15 선택
간선 (1,2) 16 선택
간선 (3,4) 22 선택
간선 (4,5) 27 선택
```

union 연산과 find 연산을 이용하여 Kruskal의 알고리즘을 구현한 코드
정렬 알고리즘으로는 C언어에서 기본적으로 제공 되는 qsort() 함수를 사용

크루스칼 알로리즘이란?
그래프에서 사이클을 만들지 않는 가장 가중치가 작은 간선을 선택하여 최소 비용의 신장 트리(MST)를 구성하는 알고리즘

최소 신장 트리(MST)란?
신장 트리: 그래프 내 모든 정점을 싸이클 없이 연결하는 트리
최소 신장 트리: 가능한 모든 신장 트리 중 간선의 가중치 합이 최소인 트리

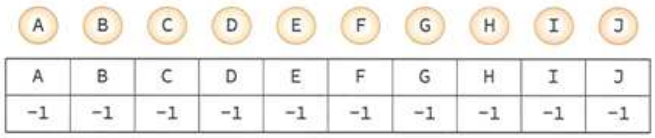
크루스칼 알고리즘의 핵심
간선의 가중치가 작은 순서대로 하나씩 선택
단, 선택한 간선이 싸이클을 만들면 제외
총 (정점 수 - 1)개의 간선이 선택될 때까지 반복

유니온 파인드 알고리즘이란?
여러 노드가 존재할 때 어떤 두 개의 노드를 같은 집합으로 묶어 주고, 어떤 두 노드가 같은 집합에 있는지 확인하는 알고리즘

Union: 서로 다른 두 개의 집합을 하나의 집합으로 병합하는 연산
Find: 하나읜 원소가 어떤 집합에 속해 있는지를 판단

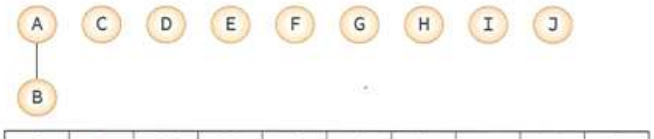
알고리즘 11.3 union-find 알고리즘

```
UNION(a, b):
    root1 = FIND(a);    // 노드 a의 루트를 찾는다.
    root2 = FIND(b);    // 노드 b의 루트를 찾는다.
    if root1 ≠ root2    // 합친다.
        parent[root1] = root2;

    FIND(curr):              // curr의 루트를 찾는다.
    if (parent[curr] == -1)
            return curr;     // 루트
    while (parent[curr] != -1) curr = parent[curr];
    return curr;
```




| A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

여기서 union(A, B)가 실행되었다면 다음과 같이 변경된다.



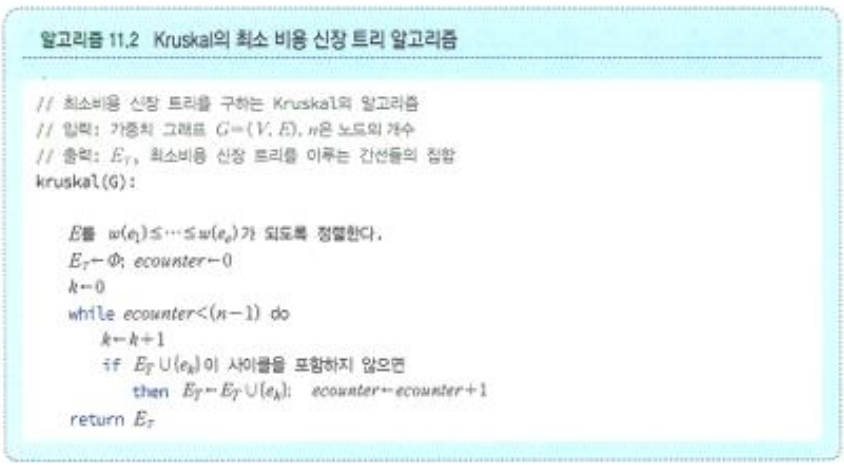

| A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|
| -1 | 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

이어서 union(C, H)가 호출되면 다음과 같이 변경된다.


| A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|
| -1 | 0 | -1 | -1 | -1 | -1 | -1 | 2 | -1 | -1 |

크루스칼 알고리즘의 절차

1. 모든 간선을 가중치 기준으로 오름차순 정렬
2. 초기에는 모든 정점이 각각 독립된 집합
3. 작은 간선부터 하나씩 확인:
4. 두 정점이 서로 다른 집합에 속하면, 그 간선을 선택하고 두 집합을 합침
5. 같은 집합이면 사이클이 생기므로 선택하지 않음
6. 간선 선택 개수가 정점 수 - 1이 되면 종료



알고리즘 11.2 Kruskal의 최소 비용 신장 트리 알고리즘

```
// 최소비용 신장 트리를 구하는 Kruskal의 알고리즘
// 입력: 가중치 그래프 G=(V, E), n은 노드의 개수
// 출력: E_T, 최소비용 신장 트리를 이루는 간선들의 집합
kruskal(G):

    E를 w(e_1)≤…≤w(e_e)가 되도록 정렬한다.
    E_T←Φ; ecounter←0
    k←0
    while ecounter<(n−1) do
        k←k+1
        if E_T∪{e_k}이 사이클을 포함하지 않으면
            then E_T←E_T∪{e_k}; ecounter←ecounter+1
    return E_T
```

ex)

정점: A, B, C, D

간선: A-B (1), A-C (3), B-C (2), B-D (4), C-D (5)

간선 정렬: AB(1), BC(2), AC(3), BD(4), CD(5)

AB 선택 → A와 B 연결

BC 선택 → C와 연결 (사이클 없음)

AC는 A, B, C가 이미 연결되어 사이클 생김 → 제외

BD 선택 → D 연결됨 → 완료

## void set_init(int n)

각 정점의 부모를 -1로 초기화 (자기 자신이 루트라는 뜻)

## int set_find(int curr)

curr 정점이 속한 집합의 루트를 찾아 반환
curr이 어떤 집합에 속해 있는지를 파악

## void set_union(int a, int b)

두 정점이 속한 집합을 하나로 합침

## struct Edge { . . . };

간선을 나타내는 구조체
멤버 : start, end, weight

## typedef struct GraphType { . . . } GraphType

간선들을 저장하는 그래프 구조체
멤버 : n, nvertex, edges[]

## void graph_init(GraphType* g)

간선 수 초기화, 배열 내 간선 초기화

## void insert_edge(GraphType* g, int start, int end, int w)

그래프에 간선 1개를 추가

## int compare(const void* a, const void* b)

qsort 함수에서 간선들을 가중치 기준으로 정렬하기 위해 사용

## void kruskal(GraphType *g)

집합 초기화
간선들을 가중치 오름차순으로 정렬
가장 짧은 간선부터 시작해서, 양 끝 정점이 서로 다른 집합에 있으면 MST에 추가
같은 집합이면 사이클 생기므로 건너뜀
(정점 수 - 1)개의 간선이 선택되면 종료

## int main(void)

그래프 생성, 초기화
정점의 수 설정
간선 삽입
Kruskal 알고리즘 실행

# 11.9 prim의 최소 비용 신장 트리 프로그램

```c
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0
#define MAX_VERTICES 100
#define INF 1000L

// 인접 행렬을 이용한 그래프 구조체 정의
typedef struct GraphType {
    int n;  // 정점의 개수
    int weight[MAX_VERTICES][MAX_VERTICES]; // 인접 행렬: 가중치 저장
} GraphType;

// Prim 알고리즘에서 사용되는 배열들
int selected[MAX_VERTICES];  // 해당 정점이 MST에 포함되었는지 여부
int distance[MAX_VERTICES];  // 시작 정점으로부터의 최소 가중치 저장


int get_min_vertex(int n)
{
    int v, i;

    // 첫 번째 미선택 정점을 초기값으로 설정
    for (i = 0; i < n; i++)
        if (!selected[i]) {
            v = i;
            break;
        }

    // 나머지 정점 중 더 작은 distance 값을 가진 정점을 찾음
    for (i = 0; i < n; i++)
        if (!selected[i] && (distance[i] < distance[v]))
            v = i;

    return v;  // 선택된 정점 인덱스 반환
}

// g: 그래프, s: 시작 정점
void prim(GraphType* g, int s)
{
    int i, u, v;

    // 모든 정점의 distance 값을 무한대로 초기화
    for (u = 0; u < g->n; u++)
        distance[u] = INF;

    distance[s] = 0;  // 시작 정점의 distance 값은 0

    // 정점의 수만큼 반복 (최대 g->n 번)
    for (i = 0; i < g->n; i++) {
        u = get_min_vertex(g->n);   // 현재 최소 가중치 정점을 선택
        selected[u] = TRUE;         // 해당 정점을 MST에 포함

        // distance 값이 여전히 INF라면 연결되지 않은 정점이므로 종료
        if (distance[u] == INF)
            return;

        // 선택된 정점을 출력 (MST에 포함된 정점)
        printf("정점 %d 추가₩n", u);

        // 인접한 정점 v에 대해
        for (v = 0; v < g->n; v++)
            // 간선이 존재하고 아직 선택되지 않았으며,
            // 현재 u를 통해 v로 가는 경로가 기존 경로보다 짧다면 갱신
            if (g->weight[u][v] != INF)
                if (!selected[v] && g->weight[u][v] < distance[v])
                    distance[v] = g->weight[u][v];
    }
}
// ◇ main 함수
// 그래프를 초기화하고 prim 알고리즘을 호출하는 부분
int main(void)
{
    // 인접 행렬로 구성된 그래프 초기화
    GraphType g = {
        7,  // 정점 개수
        {
            { 0, 29, INF, INF, INF, 10, INF },
            { 29,  0, 16, INF, INF, INF, 15 },
            { INF, 16, 0, 12, INF, INF, INF },
            { INF, INF, 12, 0, 22, INF, 18 },
            { INF, INF, INF, 22, 0, 27, 25 },
            { 10, INF, INF, INF, 27, 0, INF },
            { INF, 15, INF, 18, 25, INF, 0 }
        }
    };

    // 시작 정점을 0번 정점으로 설정하여 Prim 알고리즘 수행
    prim(&g, 0);

    return 0;
}
```

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

프림 알고리즘을 구현한 코드

프림 알고리즘(Prim's Algorithm)이란?
시작 정점에서부터 신장 트리 집합을 단계적으로 확장해가는 방법

**알고리즘 11.5 Prim의 최소 비용 신장 트리 알고리즘 #2**

```
// 최소 비용 신장 트리를 구하는 Prim의 알고리즘
// 입력: 네트워크 G=(V, E), s는 시작 정점
// 출력: 최소 비용 신장 트리를 이루는 정점들의 집합
Prim(G, s):

    for each u∈V do
        distance[u]←∞
    distance[s]←0
    우선 순위큐 Q에 모든 정점을 삽입(우선순위는 dist[])
    for i←0 to n-1 do
        u←delete_min(Q)
        화면에 u를 출력
        for each v∈(u의 인접 정점)
            if( v∈Q and weight[u][v]<dist[v] )
                then dist[v]←weight[u][v]
```

프림 알고리즘의 절차
모든 distance[]를 초기화
시작 정점 s의 distance[s] = 0
모든 정점을 우선순위 큐 Q에 삽입
루프 시작 (while Q is not empty)
Q에서 distance[] 값이 가장 작은 정점 u를 선택
u를 트리 집합에 추가 (selected[u] = TRUE)
인접 정점 거리 갱신
정점 u에 인접한 모든 정점 v에 대해:
아직 MST에 포함되지 않았고 (!selected[v])
distance[v] > weight[u][v] 이면
distance[v] = weight[u][v] 로 갱신
모든 정점이 MST에 포함되면 종료 (큐가 비면 종료)

## typedef struct GraphType { . . . } GraphType;

인접 행렬을 이용한 그래프 구조체
멤버 : n, weight[]

## int get_min_vertex(int n)

selected 되지 않은 정점 중 distance 가 가장 작은 정점을 찾아 반환.

## void prim(GraphType* g, int s)

모든 distance[]를 INF로 초기화
시작 정점 s의 distance[s] = 0
모든 정점을 우선순위 큐 Q에 삽입
정점의 수만큼 반복
Q에서 distance[] 값이 가장 작은 정점 u를 선택
u를 트리 집합에 추가 (selected[u] = TRUE)
정점 u에 인접한 모든 정점 v에 대해:
아직 MST에 포함되지 않았고 (!selected[v])
distance[v] > weight[u][v] 이면
distance[v] = weight[u][v] 로 갱신

## int main(void)

인접 행렬로 구성된 그래프 초기화
Prim 알고리즘 수행(0번 정점을 시작정점으로 지정)

# 11.10 최단 경로 Dijkstra 프로그램

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>  // INT_MAX 상수를 사용하기 위해 포함

// 상수 정의
#define TRUE 1
#define FALSE 0
#define MAX_VERTICES 100
#define INF 1000000

// 인접 행렬 기반 그래프 구조체 정의
typedef struct GraphType {
    int n;  // 정점의 개수
    int weight[MAX_VERTICES][MAX_VERTICES];  // 인접 행렬 (간선 가중치)
} GraphType;

// 전역 배열
int distance[MAX_VERTICES]; // 시작 정점으로부터의 최단 거리 저장
int found[MAX_VERTICES];    // 해당 정점의 방문 여부 (MST에 포함 여부와 유사)

// ◇ 가장 가까운(최소 거리) 정점 선택
int choose(int distance[], int n, int found[])
{
    int i, min, minpos;
    min = INT_MAX;  // 최소값 초기화
    minpos = -1;    // 선택된 정점이 없는 상태

    // 아직 방문하지 않은 정점 중에서 가장 작은 거리값을 갖는 정점 찾기
    for (i = 0; i < n; i++)
        if (distance[i] < min && !found[i]) {
            min = distance[i];
            minpos = i;
        }
    return minpos;  // 선택된 정점의 인덱스 반환
}
// ◇ 현재 알고리즘 진행 상황 출력 함수
void print_status(GraphType* g)
{
    static int step = 1;
    printf("STEP %d: ", step++);
    printf("distance: ");
    for (int i = 0; i < g->n; i++) {
        if (distance[i] == INF)
            printf(" * ");
        else
            printf("%2d ", distance[i]);
    }
    printf("\n");

    printf("        found:    ");
    for (int i = 0; i < g->n; i++)
        printf("%2d ", found[i]);
    printf("\n\n");
}

// g: 그래프 포인터, start: 시작 정점
void shortest_path(GraphType* g, int start)
{
    int i, u, w;

    // 시작 정점으로부터 각 정점까지의 초기 거리 설정
    for (i = 0; i < g->n; i++) {
        distance[i] = g->weight[start][i]; // 직행 경로 가중치
        found[i] = FALSE;                  // 초기에는 아무 정점도 방문하지 않음
    }

    found[start] = TRUE;     // 시작 정점 방문 처리
    distance[start] = 0;     // 자기 자신까지 거리는 0
    // 정점 수 - 1 번 반복
    for (i = 0; i < g->n - 1; i++) {
        print_status(g);                  // 현재 상태 출력
        u = choose(distance, g->n, found); // 가장 가까운 정점 선택
        found[u] = TRUE;                  // 선택된 정점을 방문 처리

        // u 정점의 인접 정점들을 확인하여 거리 갱신
        for (w = 0; w < g->n; w++) {
            if (!found[w]) { // 아직 방문하지 않은 정점에 대해
                if (distance[u] + g->weight[u][w] < distance[w]) {
                    // u를 경유한 경로가 더 짧으면 거리 갱신
                    distance[w] = distance[u] + g->weight[u][w];
                }
            }
        }
    }
}
```

```c
// ◇ 메인 함수: 그래프 초기화 및 실행
int main(void)
{
    GraphType g = {
        7,  // 정점 수
        {   // 인접 행렬 (가중치 그래프)
            {  0,  7, INF, INF,  3, 10, INF },
            {  7,  0,   4,  10,  2,  6, INF },
            {INF,  4,   0,   2, INF, INF, INF },
            {INF, 10,   2,   0, 11,  9,   4 },
            {  3,  2, INF,  11,  0, INF,  5 },
            { 10,  6, INF,   9, INF,  0, INF },
            {INF, INF, INF,  4,  5, INF,  0 }
        }
    };

    shortest_path(&g, 0);  // 정점 0에서 시작하여 최단 거리 계산

    return 0;
}
```

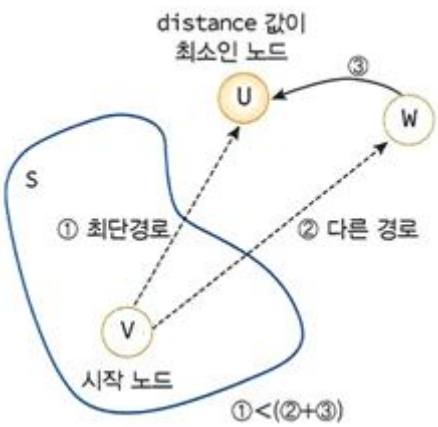
```
STEP 1: distance:  0  7  *  *  3 10  *
        found:     1  0  0  0  0  0  0

STEP 2: distance:  0  5  * 14  3 10  8
        found:     1  0  0  0  1  0  0

STEP 3: distance:  0  5  9 14  3 10  8
        found:     1  1  0  0  1  0  0

STEP 4: distance:  0  5  9 12  3 10  8
        found:     1  1  0  0  1  0  1

STEP 5: distance:  0  5  9 11  3 10  8
        found:     1  1  1  0  1  0  1

STEP 6: distance:  0  5  9 11  3 10  8
        found:     1  1  1  0  1  1  1
```

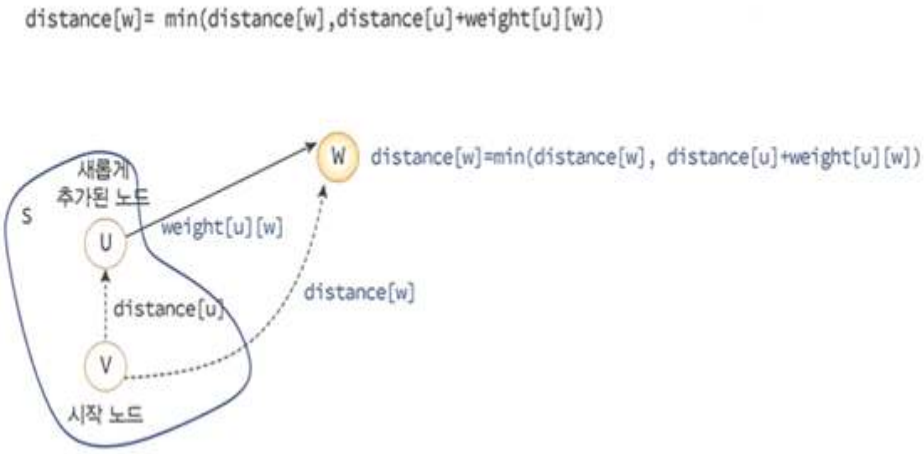
다익스트라(Dijkstra)의 최단거리 알고리즘을 구현한 코드

다익스트라 알고리즘이란?
네트워크에서 하나의 시작 정점으로부터 모든 다른 정점까지의 최단 경로를 찾는 알고리즘

**알고리즘 11.6 최단거리 알고리즘**

```
// 입력: 가중치 그래프 G, 가중치는 음수가 아님.
// 출력: distance 배열, distance[u]는 v에서 u까지의 최단 거리이다.
shortest_path(G, v):

    S←{v}
    for 각 정점 w∈G do
        distance[w]←weight[v][w];
    while 모든 정점이 S에 포함되지 않으면 do
        u←집합 S에 속하지 않는 정점 중에서 최소 distance 정점;
        S←S∪{u}
        for u에 인접하고 S에 있는 각 정점 z do
            if distance[u]+weight[u][z] < distance[z]
                then distance[z]←distance[u]+weight[u][z];
```



shortest_path 알고리즘의 절차
1. distance[w] = weight[v][w] (직접 연결된 정점은 그 가중치로, 아니면 INF)
2. 정점 선택 반복 (총 n번) - 모든 정점이 집합 S에 포함될 때까지 위 반복을 수행
S에 속하지 않은 정점들 중에서 distance 값이 가장 작은 정점 u를 선택
u를 집합 S에 추가 (최단 거리가 확정되었음)
u에 인접한 모든 정점 w에 대해 다음 조건을 만족하면 distance[w]를 갱신:
if distance[u] + weight[u][w] < distance[w]
→ distance[w] = distance[u] + weight[u][w]

## typedef struct GraphType { . . . } GraphType;

인접 행렬을 이용한 그래프 구조체
멤버 : n, weight[]

## int choose(int distance[], int n, int found[])

가장 가까운(최소 거리) 정점 선택

## void print_status(GraphType* g)

현재 알고리즘 진행 상황 출력 함수

## void shortest_path(GraphType* g, int start)

시작 정점으로부터 각 정점까지의 초기 거리 설정
시작 정점 방문 처리, 자기 자신까지 거리는 0
정점의 수-1 만큼 반복
가장 가까운 정점 선택, 선택된 정점을 방문 처리
u 정점의 인접 정점들을 확인하여 거리 갱신
아직 방문하지 않은 정점에 대해
distance[u] + g->weight[u][w] < distance[w] 이면
distance[w] = distance[u] + g->weight[u][w]로 변경

## int main(void)

인접 행렬로 구성된 그래프 초기화
최단거리 알고리즘 수행(0번 정점을 시작정점으로 지정)

# 11.11 Floyd의 최단 경로 프로그램

```c
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0
#define MAX_VERTICES 100
#define INF 1000000

// 그래프 구조체 정의 (가중치 인접 행렬 사용)
typedef struct GraphType {
    int n;  // 정점의 개수
    int weight[MAX_VERTICES][MAX_VERTICES];  // 가중치 인접 행렬
} GraphType;

// A[i][j]는 i에서 j로 가는 현재까지의 최단 거리
int A[MAX_VERTICES][MAX_VERTICES];

void printA(GraphType* g)
{
    int i, j;
    printf("============================\n");
    for (i = 0; i < g->n; i++) {
        for (j = 0; j < g->n; j++) {
            if (A[i][j] == INF)
                printf("  * ");  // 무한대는 별표로 표시
            else
                printf("%3d ", A[i][j]);  // 유한 값은 정수 출력
        }
        printf("\n");
    }
    printf("============================\n");
}

// 플로이드-워셜 알고리즘
void floyd(GraphType* g)
{
    int i, j, k;

    // 초기화: A 행렬을 그래프의 가중치 행렬로 복사
    for (i = 0; i < g->n; i++)
        for (j = 0; j < g->n; j++)
            A[i][j] = g->weight[i][j];

    printA(g);  // 초기 상태 출력

    // 핵심 알고리즘: 중간 정점 k를 경유하여 i->j로 가는 더 짧은 경로가 있는지 확인
    for (k = 0; k < g->n; k++) {          // 모든 정점을 중간 노드로 시도
        for (i = 0; i < g->n; i++) {
            for (j = 0; j < g->n; j++) {
                // 경유했을 때 더 짧으면 갱신
                if (A[i][k] + A[k][j] < A[i][j])
                    A[i][j] = A[i][k] + A[k][j];
            }
        }
        printA(g);  // 각 단계마다 행렬 출력
    }
}

// 메인 함수: 그래프 초기화 및 알고리즘 실행
int main(void)
{
    // 7개의 정점으로 구성된 가중치 인접 행렬
    GraphType g = { 7,
    {{  0,  7, INF, INF,   3,  10, INF },
    {  7,  0,   4,  10,   2,   6, INF },
    {INF,  4,   0,   2, INF, INF, INF },
    {INF, 10,   2,   0,  11,   9,   4 },
    {  3,  2, INF,  11,   0, INF,   5 },
    { 10,  6, INF,   9, INF,   0, INF },
    {INF, INF, INF,  4,   5, INF,   0 }}
    };

    floyd(&g);  // 플로이드 알고리즘 실행
    return 0;
}
```

```
============================
  0   7   *   *   3  10   *
  7   0   4  10   2   6   *
  *   4   0   2   *   *   *
  *  10   2   0  11   9   4
  3   2   *  11   0   *   5
 10   6   *   9   *   0   *
  *   *   *   4   5   *   0
============================
============================
  0   7  11  13   3  10  17
  7   0   4   6   2   6  10
 11   4   0   2   6  10   6
 13   6   2   0   8   9   4
  3   2   6   8   0   8   5
 10   6  10   9   8   0  13
 17  10   6   4   5  13   0
============================
============================
  0   7   *   *   3  10   *
  7   0   4  10   2   6   *
  *   4   0   2   *   *   *
  *  10   2   0  11   9   4
  3   2   *  11   0  13   5
 10   6   *   9  13   0   *
  *   *   *   4   5   *   0
============================
============================
  0   7  11  13   3  10   8
  5   0   4   6   2   6   7
  9   4   0   2   6  10   6
 11   6   2   0   8   9   4
  3   2   6   8   0   8   5
 10   6  10   9   8   0  13
  8   7   6   4   5  13   0
============================
============================
  0   7  11  17   3  10   *
  7   0   4  10   2   6   *
 11   4   0   2   6  10   *
 17  10   2   0  11   9   4
  3   2   6  11   0   8   5
 10   6  10   9   8   0   *
  *   *   *   4   5   *   0
============================
============================
  0   5   9  11   3  10   8
  5   0   4   6   2   6   7
  9   4   0   2   6  10   6
 11   6   2   0   8   9   4
  3   2   6   8   0   8   5
 10   6  10   9   8   0  13
  8   7   6   4   5  13   0
============================
============================
  0   7  11  13   3  10   *
  7   0   4   6   2   6   *
 11   4   0   2   6  10   *
 13   6   2   0   8   9   4
  3   2   6   8   0   8   5
 10   6  10   9   8   0   *
  *   *   *   4   5   *   0
============================
============================
  0   5   9  11   3  10   8
  5   0   4   6   2   6   7
  9   4   0   2   6  10   6
 11   6   2   0   8   9   4
  3   2   6   8   0   8   5
 10   6  10   9   8   0  13
  8   7   6   4   5  13   0
============================
```

floyd 알고리즘을 구현한 코드

floyd알고리즘 이란?

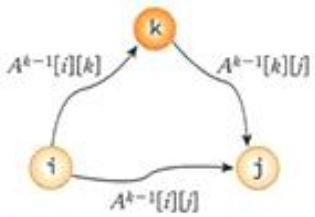
그래프에 존재하는 모든 정점 사이의 최단 경로를 한 번에 모두 찾아주는 알고리즘

알고리즘 11.8 Floyd의 최단 경로 알고리즘

```
floyd(G):

    for k ← 0 to n - 1
        for i ← 0 to n - 1
            for j ← 0 to n - 1
                A[i][j] = min(A[i][j], A[i][k] + A[k][j])
```

floyd 알고리즘의 핵심
$A^k[i][j]$는 정점 i에서 정점 j로 가는 경로 중에서,
0부터 k번 정점까지만을 중간 경유지로 사용할 수 있을 때의 최단 거리이다.
$A^{-1}[i][j]$: 아무 경유지도 사용하지 않음 → 즉, weight[i][j]
$A^0[i][j]$: 정점 0만 중간에 사용할 수 있음
$A^1[i][j]$: 정점 0, 1을 경유지로 사용할 수 있음
…
$A^{n-1}[i][j]$: 모든 정점을 경유지로 사용할 수 있음 → 최종 답

k번째 정점까지 고려된 상황이라면,
정점 i에서 j까지의 최단 경로는 다음 두 경우 중 더 짧은 것:
(1) 정점 k를 거치지 않는 경우
이전 단계에서 이미 구한 최단 거리 $A^{k-1}[i][j]$를 그대로 유지
(2) 정점 k를 거치는 경우
i → k까지의 최단 거리 $A^{k-1}[i][k]$
k → j까지의 최단 거리 $A^{k-1}[k][j]$
= $A^{k-1}[i][k] + A^{k-1}[k][j]$




[그림 11-12] Floyd 알고리즘

(1) 정점 $k$를 거쳐서 가지 않는 경우:
$A^k[i][j]$는 $k$보다 큰 정점은 통과하지 않으므로 이경우 최단 거리는 $A^{k-1}[i][j]$가 된다.

(2) 정점 $k$를 통과하는 경우:
이 경우 $i$에서 $k$까지의 최단거리 $A^{k-1}[i][k]$에다가 $k$에서 $j$까지의 최단거리인 $A^{k-1}[j][k]$를 더한 값이 될 것이다.

## typedef struct GraphType { . . . } GraphType;

가중치 인접 행렬을 이용한 그래프 구조체
멤버 : n, weight[]

## void printA(GraphType *g)

최단거리 행렬 A 출력
A[i][j] == INF라면 *, 정수라면 정수값 출력

## void floyd(GraphType* g)

플로이드-워셜 알고리즘 함수
A 행렬을 그래프의 가중치 행렬로 복사, 초기 상태 출력
정점의 수 만큼 3중 for반복
중간 정점 k를 경유하여 i->j로 가는 더 짧은 경로가 있는지 확인
A[i][k] + A[k][j] < A[i][j] 라면
A[i][j] = A[i][k] + A[k][j]로 변경
각 단계마다 행렬 A 출력

## int main(void)

가중치 인접 행렬 그래프 초기화
플로이드 알고리즘 실행

# 11.13 그래프 위상정렬 프로그램

```c
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0
#define MAX_VERTICES 50

// ---------- 인접 리스트를 위한 자료구조 정의 ----------
typedef struct GraphNode {
    int vertex;                 // 연결된 정점 번호
    struct GraphNode* link;     // 다음 인접 정점을 가리키는 포인터
} GraphNode;

typedef struct GraphType {
    int n;                      // 정점의 개수
    GraphNode* adj_list[MAX_VERTICES]; // 각 정점의 인접 리스트
} GraphType;

// ---------- 그래프 초기화 및 삽입 함수 ----------
void graph_init(GraphType* g)
{
    int v;
    g->n = 0;
    for (v = 0; v < MAX_VERTICES; v++)
        g->adj_list[v] = NULL;
}

void insert_vertex(GraphType* g, int v)
{
    if (((g->n) + 1) > MAX_VERTICES) {
        fprintf(stderr, "그래프: 정점의 개수 초과");
        return;
    }
    g->n++;
}

// u → v 방향 간선 추가
void insert_edge(GraphType* g, int u, int v)
{
    GraphNode* node;
    if (u >= g->n || v >= g->n) {
        fprintf(stderr, "그래프: 정점 번호 오류");
        return;
    }
    node = (GraphNode*)malloc(sizeof(GraphNode));
    node->vertex = v;
    node->link = g->adj_list[u];
    g->adj_list[u] = node;
}

// ---------- 위상 정렬 수행 함수 ----------
int topo_sort(GraphType* g)
{
    int i;
    StackType s;
    GraphNode* node;

    // 진입 차수 배열 동적 생성 및 초기화
    int* in_degree = (int*)malloc(g->n * sizeof(int));
    for (i = 0; i < g->n; i++)
        in_degree[i] = 0;

   // 모든 정점의 진입 차수를 계산
   for (i = 0; i < g->n; i++) {
       node = g->adj_list[i];
       while (node != NULL) {
           in_degree[node->vertex]++;
           node = node->link;
       }
   }

   // 진입 차수가 0인 정점들을 스택에 삽입
   init(&s);
   for (i = 0; i < g->n; i++)
       if (in_degree[i] == 0) push(&s, i);
```

```c
    // 위상 정렬 수행
    int count = 0;  // 정렬된 정점 수 추적
    while (!is_empty(&s)) {
        int w = pop(&s);
        printf("정점 %d -> ", w);
        count++;

        node = g->adj_list[w];
        while (node != NULL) {
            int u = node->vertex;
            in_degree[u]--;  // 진입 차수 감소
            if (in_degree[u] == 0)
                push(&s, u); // 새로 진입 차수가 0이 된 정점 추가
            node = node->link;
        }
    }

    printf("\n");
    free(in_degree);
    return (count == g->n);  // 모든 정점이 정렬되었는지 확인
}
// ---------- 스택 구조 및 연산 정의 ----------
#define MAX_STACK_SIZE 100
typedef int element;

typedef struct {
    element stack[MAX_STACK_SIZE];
    int top;
} StackType;

void init(StackType* s) { s->top = -1; }
int is_empty(StackType* s) { return (s->top == -1); }
int is_full(StackType* s) { return (s->top == MAX_STACK_SIZE - 1); }

void push(StackType* s, element item)
{
    if (is_full(s)) {
        fprintf(stderr, "스택 포화 에러\n");
        return;
    }
    s->stack[++(s->top)] = item;
}

element pop(StackType* s)
{
    if (is_empty(s)) {
        fprintf(stderr, "스택 공백 에러\n");
        exit(1);
    }
    return s->stack[(s->top)--];
}

// ---------- 메인 함수 ----------
int main(void)
{
    GraphType g;

    graph_init(&g);
    // 정점 0-5 삽입
    insert_vertex(&g, 0);
    insert_vertex(&g, 1);
    insert_vertex(&g, 2);
    insert_vertex(&g, 3);
    insert_vertex(&g, 4);
    insert_vertex(&g, 5);

    // 간선 삽입 (방향 그래프)
    insert_edge(&g, 0, 2);
    insert_edge(&g, 0, 3);
    insert_edge(&g, 1, 3);
    insert_edge(&g, 1, 4);
    insert_edge(&g, 2, 3);
    insert_edge(&g, 2, 5);
    insert_edge(&g, 3, 5);
    insert_edge(&g, 4, 5);

    // 위상 정렬 실행
    topo_sort(&g);

    return 0;
}
```

```
정점 1 -> 정점 4 -> 정점 0 -> 정점 2 -> 정점 3 -> 정점 5 ->
```

인접리스트, 스택을 사용하여 위상 정렬 알고리즘을 구현한 코드

위상 정렬 알고리즘이란?
그래프에 존재하는 각 정점들의 선행 순서를 위배하지 않으면서 모든 정점을 나열하는 것
즉, 간선 <u, v>가 있을 경우 정점 u는 반드시 정점 v보다 먼저 나와야 함

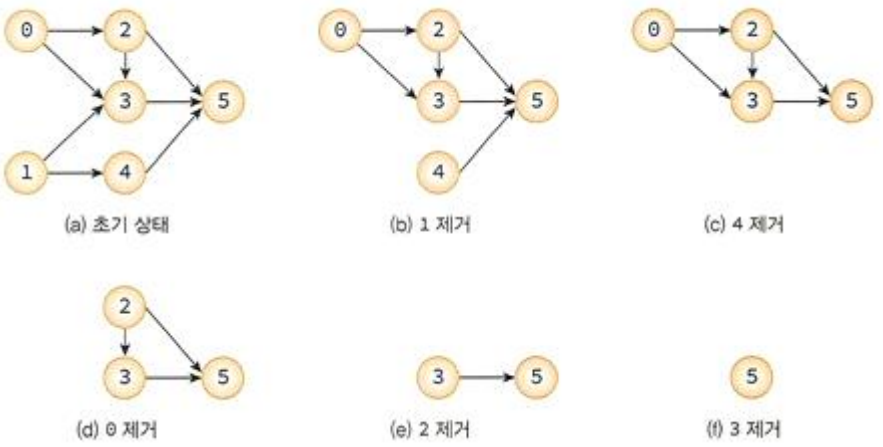
**알고리즘 11.9 그래프 위상 정렬 알고리즘 #1**

```
// Input: 그래프 G=(V,E)
// Output: 위상 정렬 순서

topo_sort(G)

for i←0 to n-1 do
    if( 모든 정점이 선행 정점을 가지면 )
        then 사이클이 존재하고 위상 정렬 불가;
    선행 정점을 가지지 않는 정점 v 선택;
    v를 출력;
    v와 v에서 나온 모든 간선들을 그래프에서 삭제;
```

진입 차수가 0인 정점 v를 선택하여 v와 v에 부착된 모든 간선을 삭제
이와 같은 진입 차수 0인 정점의 선택과 삭제 과정을 반복해서
모든 정점이 선택 · 삭제되면 알고리즘이 종료

이때 진입 차수 0인 정점이 여러 개 존재할 경우 어느 정점을 선택하여도 무방
(따라서, 하나의 그래프에는 복수의 위상순서가 있을 수 있다), 이 과정에서 선택되는 정점의 순서를 위상 순서(topological order)라 한다.

위의 과정 중에 그래프에 남아 있는 정점 중에 진입 차수 0인 정점이 없다면,
위상 정렬 알고리즘은 중단



## typedef struct GraphNode{ . . . } GraphNode

## typedef struct GraphType { . . . } GraphType

인접 리스트를 위한 구조체
GraphNode - 멤버 : vertex, link
GraphType - 멤버 : n, adj_list[]

## void graph_init(GraphType *g)

그래프 초기화
정점 개수를 0으로 하고 인접 리스트를 모두 NULL로 설정

## void insert_vertex(GraphType *g, int v)

정점을 그래프에 추가
((g->n) + 1) > MAX_VERTICES 가 아니라면
g->n를 1 증가

## void insert_edge(GraphType *g, int u, int v)

u → v 방향의 간선을 동적으로 할당하여 삽입
v를 u의 인접 리스트에 노드로 추가합니다.

## typedef struct { . . . } StackType

스택을 구현한 구조체
멤버 : stack[], top

## void init(StackType *s)

스택 초기화
top 를 -1로 지정

## int is_empty(StackType *s)

스택 공백상태 확인
top 가 -1이면 true, 아니면 false

## int is_full(StackType *s)

스택 포화상태 확인
top 가 MAX_STACK_SIZE - 1 이면 true, 아니면 false

## void push(StackType *s, element item)

스택에 item삽입
++ top한 후 스택 배열에 추가

## element pop(StackType *s)

스택 마지막 요소 삭제
스택 배열의 top인덱스를 리턴 후 top 1 감소

## int topo_sort(GraphType *g)

모든 정점의 진입 차수(in-degree)를 계산
진입 차수가 0인 정점들을 스택 에 삽입
while 스택가 비어있지 않으면:

a. 정점 u = 스택에서 꺼냄
b. u를 위상 순서에 추가
c. u와 연결된 모든 정점 v에 대해:
- in-degree[v] -= 1
- if in-degree[v] == 0:
→ v를 스택/큐에 삽입

모든 정점이 처리되었으면 위상 정렬 성공
→ 아니라면 그래프에 사이클이 존재함 (실행 불가능)

## int main(void)

그래프 초기화
총 6개 정점을 추가
여러 방향 간선을 추가
위상 정렬 알고리즘 실행