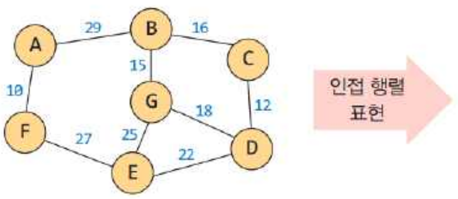
12차시

```
# 가중치 그래프
# 간선에 가중치가 할당된 그래프
# G = (V, E, w) , w : 비용, 길이 등

# 가중피 그래프의 표현
# 1. 인접행렬을 이용한 표현
# 2. 인접 리스트를 이용한 표현
```



	A	B	C	D	E	F	G
A	0	29	∞	∞	∞	10	∞
B	29	0	16	∞	∞	∞	15
C	∞	16	0	12	∞	∞	∞
D	∞	∞	12	0	22	∞	18
E	∞	∞	∞	22	0	27	25
F	10	∞	∞	∞	27	0	∞
G	∞	15	∞	18	25	∞	0

```
# 가중치의 총합을 구하기 위한 함수
def weightSum(vlist, W): # 매개 변수 : 정점 리스트, 인접 행렬
    sum = 0 # 가중치 합을 계산할 변수
    for i in range(len(vlist)): # 모든 정점에 대해서
        for j in range(i+1, len(vlist)): # 하나의 행에 대해서(삼각영역)
            if W[i][j] != None: # 만약 간선이 있으면
                sum += W[i][j] # 합계 변수에 추가
    return sum

# 인접행렬에서 모든 간선을 출력하는 함수
def printAllEdges(vlist, W): # 매개 변수 : 정점 리스트, 인접 행렬
    for i in range(len(vlist)) :
        for j in range(i+1, len(W[i])): # 모든 간선 W[i][j]에 대해
            if W[i][j] != None and W[i][j] != 0: # 간선이 있으면
                print("(%s, %s, %d)"%(vlist[i], vlist[j], W[i][j]), end=" ")
    print()

if __name__ == "__main__":
    # 1. 인접행렬을 이용한 표현
    vertex = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
    # 이차원 배열에 인접 행렬을 표시
    weight = [[None, 29, None, None, None, 10, None],
              [29, None, 16, None, None, None, 15],
              [None, 16, None, 12, None, None, None],
              [None, None, 12, None, 22, None, 18],
              [None, None, None, 22, None, 27, 25],
              [10, None, None, None, 27, None, None],
              [None, 15, None, 18, 25, None, None]]
    graph = (vertex, weight) # 전체 그래프: 튜플 사용

    # 인접행렬에서의 가중치의 합 계산
    print('AM : weight sum =', weightSum(vertex, weight))
    print()
    # 인접행렬에서의 모든 간선 출력
    printAllEdges(vertex, weight)

# 출력
# AM : weight sum = 174
#
# (A, B, 29) (A, F, 10) (B, C, 16) (B, G, 15) (C, D, 12) (D, E, 22) (D, G, 18) (E, F, 27) (E, G, 25)
```

```
## 2. 인접리스트를 이용한 방법

# 가중치의 총합을 구하기 위한 함수
def weightSum(graph):
    sum = 0
    for v in graph: # 그래프의 모든 정점 v에 대해
        for e in graph[v]: # v의 모든 간선 e에 대해 e=('F',10) , ('B',29) , ("G",15) , ... 순서로 각 set에서 거꾸러 출력된다.
            sum += e[1] # sum에 추가
    return sum // 2 # 모두 2번씩 중복 되므로 2로 나눈다.

# 인접행렬에서 모든 간선을 중복을 허용하며 출력하는 함수
def printAllEdges(graph):
    for v in graph: # 그래프의 모든 정점 v에 대해
        for e in graph[v]: # v의 모든 간선 e에 대해
            print("(%s, %s, %d)"%(v, e[0], e[1]), end=' ') # end=' '은 자동 줄바꿈을 띄어쓰기로 바꿔줌

 # 인접행렬에서 모든 간선을 중복을 허용하지 않으며 출력하는 함수
def printOneEdges(graph):
    list= []
    for v in graph:
        for e in graph[v]:
            if (v, e[0]) not in list:
                print("(%s, %s, %d)"%(v, e[0], e[1]), end=' ')
                list.append((v, e[0]))
                list.append((e[0], v))

if __name__ == "__main__":
    # 2. 인접 리스트를 이용한 표현
    # 딕셔너리, 집합, 리스트, 튜플 사용
    graphAL = { 'A' : set([('B',29),('F',10)]),
               'B' : set([('A',29),('C',16),('G',15)]),
               'C' : set([('B',16),('D',12)]),
               'D' : set([('C',12),('E',22),('G',18)]),
               'E' : set([('D',22),('F',27),('G',25)]),
               'F' : set([('A',10),('E',27)]),
               'G' : set([('B',15), ('D',18),('E',25)])}

    print('AL : weight sum = ', weightSum(graphAL))
    print()
    printAllEdges(graphAL)
    print("\n\n")
    printOneEdges(graphAL)

# 출력
# AL : weight sum =  174
#
# (A, F, 10) (A, B, 29) (B, G, 15) (B, A, 29) (B, C, 16) (C, D, 12) (C, B, 16) (줄바꿈)
# (D, C, 12) (D, E, 22) (D, G, 18) (E, G, 25) (E, F, 27) (E, D, 22) (F, E, 27) (줄바꿈)
# (F, A, 10) (G, E, 25) (G, B, 15) (G, D, 18)
#
#
# (A, F, 10) (A, B, 29) (B, G, 15) (B, C, 16) (C, D, 12) (D, E, 22) (D, G, 18) (E, G, 25) (E, F, 27)
```

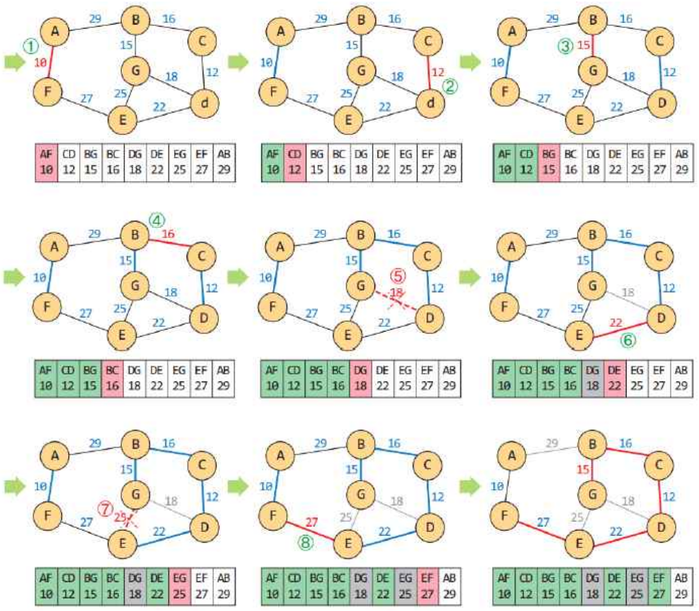
최소비용 신장트리(MST)
간선들의 가중치 합이 최소인 신장트리
반드시 (n-1)개의 간선만 사용, 사이클이 되면 안됨
사용 예) 도로, 통신, 배관 건설 : 모두 연결하면서 길이/비용을 최소화
사용 예) 전기 회로 : 단자를 모두 연결하면서 전선의 길이를 최소화

MST의 알고리즘으로 2가지가 있다.
1. Kruskal 알고리즘 (크루스칼)
2. Prim 알고리즘 (프림)

Kruskal MST(min 신장 tree) 알고리즘
탐욕적인 방법으로 그순간에 최적이라고 생각되는 것을 선택
각 단계에서 최선의 답을 선택하며 최종적인 해답에 도달
따라서 항상 최적의 해답을 주는지 검증이 필요하며, Kruskal MST 알고리즘은 증명이 됨

Kruskal MST(최소 비용 신장 트리) 알고리즘
1. 그래프의 모든 간선을 가중치에 따라 오름차순으로 정렬한다.
2. 가장 가중치가 작은 간선 e를 뽑는다.
3. e를 신장트리에 놓었을때, 사이클이 생기면 넣지 않고 2번으로 이동
4. 사이클이 생기지 않으면 최소 신장 트리에 삽입한다.
5. n-1개의 간선이 삽입될 때까지 2번으로 이동

알고리즘



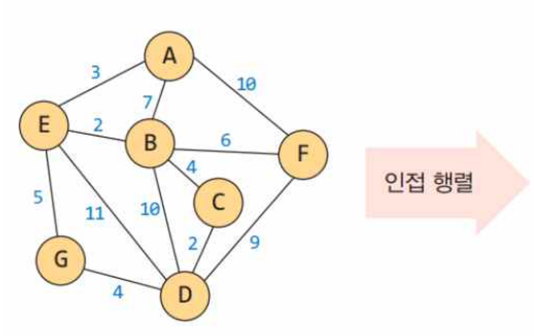
```python
parent = [] # 각노드의 부모노드 인덱스
set_size = 0 # 정점의 개수, 전역변수로 사용하기 위해 선언


def init_set(nSets): # 집합의 초기화 함수, nSets는 숫자로 받는다.
    global set_size, parent # 전역변수로 사용(변경)을 위함
    set_size = nSets; # 정점의 개수를 전역변수에 저장
    for i in range(nSets): # 모든 정점에 대해  #  range(A, B) : A부터  B-1까지 반복, A가 없으면 0부터 B-1까지 반복
        parent.append(-1) # 각각이 고유의 집합(부모가 -1)

# 원소(대입값)가 속한 트리의(집합의) 뿌리 노드를(제일 위에 있는) 찾는 연산
def find(id): # 정점 id가 속한 트리의 뿌리노드 찾기
    while (parent[id] >= 0): # id에 해당하는 부모노드가 -1이 아니면 반복,
        # union에서 s1의 부모에 s2를 삽입하므로 parent는 그 인덱스 값을 가지는 노드에 부모 노드를 나타낸다.
        id = parent[id] # id를 부모 id로 대입
    return id; # 최종 id 반환, 트리의 맨 위 노드의 id임

# S1, S2를 합치는 연산으로, S2의 뿌리노드를 S1의 부모노드로 하여 결합
def union(s1, s2): # s1, s2는 입력된 두 수가 각각 속하는 트리의 뿌리노드 값이다. 따라서 서로 다른 트리를 합치는 과정
    global set_size # 전역변수 사용(변경)을 위함
    parent[s1] = s2 # s1을 s2의 뿌리노드에 연결, parent[s1] = s2
    set_size = set_size -1 # 정점을 연결했으므로 떨어져있는 정점의 개수 감소

def MSTKruskal(vertex, adj): # 매개변수 : 정점 리스트, 인접행렬
    vsize = len(vertex) # 정점의 개수
    init_set(vsize) # 정점의 개수와 부모 리스트 전역변수로 생성
    eList = [] # 간선 리스트

    # 모든 간선을 리스트에 넣음
    for i in range(vsize-1):
        for j in range(i+1, vsize): # 상 감각행렬의 모든 요소 출력
            if adj[i][j] != None:
                eList.append( (i, j, adj[i][j]) ) # 간선 정보를 튜플로 변환하여 저장(숫자들로 이루어짐)

    # 간선 리스트를 가중치의 내림차순으로 정렬: 람다 함수 사용
    eList.sort(key=lambda e : e[2], reverse=True)

    adgeAccepted = 0 # 현재 이어진 간선 수를 저장
    while (adgeAccepted < vsize -1): # vsize(정점의 수)-1 = 간선의수
        e = eList.pop(-1) # 가장 작은 가중치를 가진 간선
        print('알고 싶은 부분 : ' , e, e[0], e[1], e[2], adj[e[0]])
        uset = find(e[0]) # e[0]가 속한 트리의 뿌리노드 값을 반환하는 과정
        vset = find(e[1]) # e[1]가 속한 트리의 뿌리노드 값을 반환하는 과정

        if uset != vset: # 두 뿌리노드값이 다르다면, 즉 서로 다른 트리라면
            print('간선 추가 : (%s, %s, %d)'%(vertex[e[0]], vertex[e[1]], e[2])) # 간선추가 출력
            union(uset, vset) # 두 집합을 합함, uset, vset 둘다 숫자이다.
            adgeAccepted += 1 # 간선이 하나 추가됨
```

```python
if __name__ == "__main__":
    vertex = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
    # 이차원 배열에 인접 행렬을 표시
    weight = [[None, 29, None, None, None, 10, None],
              [29, None, 16, None, None, None, 15],
              [None, 16, None, 12, None, None, None],
              [None, None, 12, None, 22, None, 18],
              [None, None, None, 22, None, 27, 25],
              [10, None, None, None, 27, None, None],
              [None, 15, None, 18, 25, None, None]]

    print("MST By Kruskal's Algorithm")
    MSTKruskal(vertex, weight)

'''
MST By Kruskal's Algorithm
알고 싶은 부분 :  (0, 5, 10) 0 5 10 [None, 29, None, None, None, 10, None]
간선 추가 : (A, F, 10)
알고 싶은 부분 :  (2, 3, 12) 2 3 12 [None, 16, None, 12, None, None, None]
간선 추가 : (C, D, 12)
알고 싶은 부분 :  (1, 6, 15) 1 6 15 [29, None, 16, None, None, None, 15]
간선 추가 : (B, G, 15)
알고 싶은 부분 :  (1, 2, 16) 1 2 16 [29, None, 16, None, None, None, 15]
간선 추가 : (B, C, 16)
알고 싶은 부분 :  (3, 6, 18) 3 6 18 [None, None, 12, None, 22, None, 18]
알고 싶은 부분 :  (3, 4, 22) 3 4 22 [None, None, 12, None, 22, None, 18]
간선 추가 : (D, E, 22)
알고 싶은 부분 :  (4, 6, 25) 4 6 25 [None, None, None, 22, None, 27, 25]
알고 싶은 부분 :  (4, 5, 27) 4 5 27 [None, None, None, 22, None, 27, 25]
간선 추가 : (E, F, 27)
'''
```

```
# 최소비용 신장트리(MST)
# 간선들의 가중치 합이 최소인 신장트리
# 반드시 (n-1)개의 간선만 사용, 사이클이 되면 안됨
# 사용 예) 도로, 통신, 배관 건설 : 모두 연결하면서 길이/비용을 최소화
# 사용 예) 전기 회로 : 단자를 모두 연결하면서 전선의 길이를 최소화

# MST의 알고리즘으로 2가지가 있다.
# 1. Kruskal 알고리즘
# 2. Prim 알고리즘

# 2. Prim MST 알고리즘
# 하나의 정점에서부터 시작하여 트리를 단계적으로 확장
# 현재의 신장 트리 집합에 인접한 정점 중 최저 간선으로 연결된 정점을 선택하여 신장 트리 집합에 추가
# 이과정을 n-1개의 간선을 가질 때까지 반복

# Kruskal MST(최소 비용 신장 트리) 알고리즘
# 1. 그래프에서 시작 정점을 선택하여 초기 트리를 만든다.
# 2. 현재 트리의 정점들과 인접한 정점들 중에서 간선의 가중치가 가장 작은 정점 v를 선택한다.
# 3. 이 정점 v와 이때의 간선을 트리에 추가한다.
# 4. 모든 정점이 삽입될 때 까지 2번으로 이동한다.

# 알고리즘
# A -> F -> E -> D(25보다 22가 작으므로) -> C(18보단 12가 작으므로) -> B -> G(29보단 15가 작으므로)

# Kruskal MST 알고리즘과 Prim MST 알고리즘비교
# Kruskal MST 알고리즘 : O(e log e)
# 대부분의 간선들을 정렬하는 시간에 좌우됨(가중치에 따라 역순으로 나열했으므로)
# 간선 e개를 정렬하는 시간, 간선이 적으면 유리
# 희박한 그래프가 유리

# 2. Prim MST 알고리즘 : O(n^2)
# 주 반복문이 n번, 내부 반복문이 n번 반복
# 밀집한 그래츠가 유리, 정점이 적으면 유리

def getMinVertex(dist, selected):
    minv = 0 # 가중치의 최소 값을 가지는 인덱스를 저장할 변수
    mindist = INF # # 가중치의 최소 값을 저장할 변수
    for v in range(len(dist)): # len(dist) = vsize로 정덤의 수이다.
        # 한번 들리지 않은 정점이고, 해당 dist 중 가장 작은 인덱스를 가장 가중치를 찾기 위한 값 비교
        if selected[v] == False and dist[v] < mindist :
            mindist = dist[v] # 가중치 저장
            minv = v # 인덱스 저장
    return minv # 인덱스 반환

def MSTPrim(vertex, adj):
    vsize = len(vertex)# 정점의 수 저장
    # 정해진 시작점을 기준으로 정점이 이어져있엇다면, 그 정점에 해당하는 인덱스에 간선의 가중치 저장
    # 전에 사용된 내용을 리셋시키지 않고 사용,
    # != None으로 이어지지 않은 간선은 들르지 않았기 때문에 이어진 간선의 가중치만 재정의 함
    dist = [INF]*vsize # dist: [INF, INF, ... , INF]
    selected = [False]*vsize # selected: [False, ... , False]의 형태로 한 번 지나간 정점을 표시하는 리스트, 사이클 막는 변수
    dist[0] = 0 # dist: [0, INF, ... , INF] 첫번째를 0으로 삽입하여 A에서 출발하는 것을 표현

    for i in range(vsize): # 정점의 수만큼 반복, 0~5
        print()
        for x in range(vsize):
```

```
            print(dist[x],end=' ')
        print()

        u = getMinVertex(dist, selected) # 가장 작은 인덱스 반환
        selected[u] = True # 다시 뒤로 가지 않게 F -> T로 설정, 한번 지나간 점을 T로 표현하여 사이클을 막는다.
        print(vertex[u], end=' ') # getMinVertex로 찾은 가중치가 작은 정점을 지나기 위해 출력
        for v in range(vsize): # 간선의 수만큼 반복
            if (adj[u][v] != None): # 출발점 u를 기준으로 이어진 간선이 있다면
                # V를 목표지점이라고 할 때, V정점을 들르지 않고, 현재 dist에 저장되어있는 가중치보다 현재 dist에 자장되어있는
                if selected[v] == False and adj[u][v] < dist[v]: # 가중치보다 현재 (U,V)간선 가중치가 작으면
                    dist[v] = adj[u][v] # 가중치 재정의
    print()

if __name__ == "__main__":
    INF = 9999

    vertex = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
    # 이차원 배열에 인접 행렬을 표시
    weight = [[None, 29, None, None, None, 10, None],
              [29, None, 16, None, None, None, 15],
              [None, 16, None, 12, None, None, None],
              [None, None, 12, None, 22, None, 18],
              [None, None, None, 22, None, 27, 25],
              [10, None, None, None, 27, None, None],
              [None, 15, None, 18, 25, None, None]]

    print("MST By Prim's Algorithm")
    MSTPrim(vertex, weight)

'''
MST By Prim's Algorithm

0 9999 9999 9999 9999 9999 9999
A
0 29 9999 9999 9999 10 9999
F
0 29 9999 9999 27 10 9999
E
0 29 9999 22 27 10 25
D
0 29 12 22 27 10 18
C
0 16 12 22 27 10 18
B
0 16 12 22 27 10 15
G
'''
```

```
# 최단경로 알고리즘이란
# 정점 U와 정점 V를 연결하는 경로 중에서 간선들의 가중치 합이 최소가 되는 경로
# 간선의 가중치는 비용, 거리, 시간 등이 있다.
# 알고리즘으로 Dijkstra와 Floyd 알고리즘이 있다.
# 간선이 없으면 가중치를 무한대로 처리
```

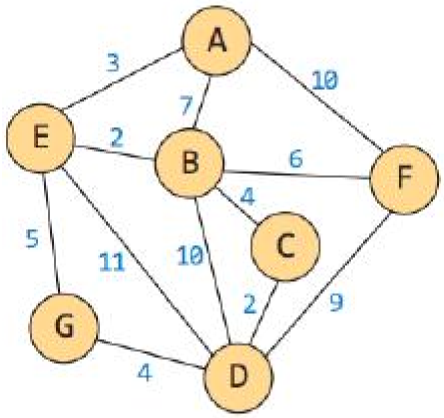


	A	B	C	D	E	F	G
A	0	7	∞	∞	3	10	∞
B	7	0	4	10	2	6	∞
C	∞	4	0	2	∞	∞	∞
D	∞	10	2	0	11	9	4
E	3	2	∞	11	0	∞	5
F	10	6	∞	9	∞	0	∞
G	∞	∞	∞	4	5	∞	0

간선이 없으면 무한대(∞)

```
# Dijkstra의 최단 경로 알고리즘(데이크스트라)
# 시작 정점 v에서 모든 다른 정점까지의 최단 경로 찾음
#   시작 정점 V : 최단 경로 탐색의 시작 정점
#   집합 S : 시작 정점 V로부터 최단경로가 이미 발견된 정점들의 집합
#   dist배열 : S에 있는 정점만을 거쳐서 다른 정점으로 가는 최단거리를 기록하는 배열
# 매 단계에서 최소 거리인 정점을 s에 추가
# 새로운 정점이 S에 추가되면 dist갱신

# Dijkstra의 최단 경로 알고리즘
def choose_vertex(dist, selected):
    minv = 0 # 가중치의 최소 값을 가지는 인덱스를 저장할 변수
    mindist = INF # 가중치의 최소 값을 저장할 변수

    for v in range(len(dist)): # len(dist) = 정점의 수
        if selected[v] == False and dist[v] < mindist : # 가보지 않은 정점과 저장된 가중치보다 작은 가중치라면
            mindist = dist[v] # 가중치 저장
            minv = v # 인덱스 저장
    return minv # 인덱스 반환

def shortest_path_dijkstra(vtx, adj, start):
    vsize = len(vtx) # 정점 수
    dist = list(adj[start]) # start와 연결된 간선과 가중치 정보들을 리스트로 형변환하여 dist 저장
    path = [start] * vsize # 시작점을 정점의 수로 곱하여 다음을 뜻하는 Path 변수 생성
    found = [False] * vsize # 사이클을 막기위해 갔던 정점들을 표기하는 found 변수 생성
    found[start] = True # 시작점을 Ture 표시
    dist[start] = 0 # 시작점까지의 거리를 0으로 표시

    for i in range(vsize): # 정점의 수만큼 순환
        print("step%2d: "%(i+1), dist) # 단계별 dist[] 출력용, 현재 단계에서 각 목적지까지의 거리를 출력
        u = choose_vertex(dist, found) # 현재 거리와 지나간 정점정보 전달
        found[u] = True # 가중치가 가장 작은 U로 가야하기 때문에 U를 T로 표시

        for w in range(vsize): # 정점의 수만큼 순환
            if not found[w]: # 즉 가보지 않은 곳이라고 했다.
                if dist[u] + adj[u][w] < dist[w]: # U의 가중치 + U에서 W까지의 가중치 < 새로운 Q가지의 가중치
                    dist[w] = dist[u] + adj[u][w] # U를 걸쳐 두단계로 가는 것이 더 가중치가 작으면 갱신된다..
                    path[w] = u # 이전 정점 갱신
    return path # 찾아진 최단 경로 반환
```

```
if __name__ == "__main__":

    INF = 9999 # 최대값 상수로 지정

    vertex = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
    # 이차원 배열에 인접 행렬을 표시
    weight = [[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]]
```




```
    print("Shortest Path By Dijkstra Algorithm")
    start = 0 # 시작점 설정
    path = shortest_path_dijkstra(vertex, weight, start) # 정점 그래프 시작점

    # 최종 경로를 출력하기 위한 코드
    for end in range(len(vertex)) : # 정점의 수만큼 순서대로 순환
        if end != start: # 도착점이 시작점이 아니면, 제자리 끝점으로 가는게 아니면 거꾸로 출력함
            print("[최단경로: %s -> %s] %s"%(vertex[start], vertex[end], vertex[end]), end='')
            while (path[end] != start):
                print(" <- %s"%vertex[path[end]], end=' ')
                end = path[end]
            print(" <- %s"% vertex[path[end]])

# 출력
# Shortest Path By Dijkstra Algorithm
# step 1:  [0, 7, 9999, 9999, 3, 10, 9999]
# step 2:  [0, 5, 9999, 14, 3, 10, 8]
# step 3:  [0, 5, 9, 14, 3, 10, 8]
# step 4:  [0, 5, 9, 12, 3, 10, 8]
# step 5:  [0, 5, 9, 11, 3, 10, 8]
# step 6:  [0, 5, 9, 11, 3, 10, 8]
# step 7:  [0, 5, 9, 11, 3, 10, 8]
# [최단경로: A -> B] B <- E  <- A
# [최단경로: A -> C] C <- B  <- E  <- A
# [최단경로: A -> D] D <- C  <- B  <- E  <- A
# [최단경로: A -> E] E <- A
# [최단경로: A -> F] F <- A
# [최단경로: A -> G] G <- E  <- A
```

```python
# 최단경로 알고리즘이란
# 정점 U와 정점 V를 연결하는 경로 중에서 간선들의 가중치 합이 최소가 되는 경로
# 간선의 가중치는 비용, 거리, 시간 등이 있다.
# 알고리즘으로 Dijkstra와 Floyd 알고리즘이 있다.

# Floyd의 최단경로 알고리즘
# 모든 정점 사이의 최단경로를 찾는다.
# 2차원 배열 A를 이용하여 3중 반복을 하는 루프로 구성
# 배열 A의 초기 값은 인접 행렬의 가중치

# A[i][j]^k : 0~k까지의 정점만을 이용한 정점i에서 j까지의 최단 경로 길이
# A^-1 -> A^0 -> A^1 -> ... -> A^n-1 순으로 최단경로 길이를 구함

# Dijkstra : O(n^2)
# 주 반복문을 n번 반복
# 내부 반복문을 2n번 반복
# 모든 정점 쌍의 최단 경로를 구한려면 n번 반복 -> O(n^3)

# Floyd-Warshall : O(n^3)
# 모든 정점 쌍의 최단 경로 거리를 구함
# 3중 반복문을 실행
# Floyd의 알고리즘은 매우 간결한 반복 구문을 사용

def shortest_path_floyd(vertex, adj):
    vsize = len(vertex) # 정점의 개수
    A = list(adj) # 두의: 2차원 배열(리스트의 리스트)의 복사
    for i in range(vsize): # 각각의 열에 대해
        A[i] = list(adj[i]) # 열(리스트)을 복사

    for k in range(vsize): # 정점 k를 추가할 때
        for i in range(vsize):
            for j in range(vsize): # 모든 A[i][j]
                if (A[i][k] + A[k][j] < A[i][j]): # i에서 j로 가는데(모든간선) K를 (0~6차례로) 거쳐가는 게 빠르면 정정
                    A[i][j] = A[i][k] + A[k][j]
        print("==============================") # 변경한 배열 출력
        for x in range(vsize):
            for y in range(vsize):
                print("%3d"%A[x][y], end=' ')
            print()
if __name__ == "__main__":
    INF = 999

    vertex = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
    # 이차원 배열에 인접 행렬을 표시
    weight = [[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]]

    # Dijkstra 알고리즘의 결과와 동일
    print("Shortest Path By Floyd's Algorithm")
    path = shortest_path_floyd(vertex, weight)
```

출력

```
# Shortest Path By Floyd's Algorithm
# ==============================
#   0   7 999 999   3  10 999
#   7   0   4  10   2   6 999
# 999   4   0   2 999 999 999
# 999  10   2   0  11   9   4
#   3   2 999  11   0  13   5
#  10   6 999   9  13   0 999
# 999 999 999   4   5 999   0
# ==============================
#   0   7  11  17   3  10 999
#   7   0   4  10   2   6 999
#  11   4   0   2   6  10 999
#  17  10   2   0  11   9   4
#   3   2   6  11   0   8   5
#  10   6  10   9   8   0 999
# 999 999 999   4   5 999   0
# ==============================
#   0   7  11  13   3  10 999
#   7   0   4   6   2   6 999
#  11   4   0   2   6  10 999
#  13   6   2   0   8   9   4
#   3   2   6   8   0   8   5
#  10   6  10   9   8   0 999
# 999 999 999   4   5 999   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   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   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   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
```