그래프

자료구조와 알고리즘
10주차 강의

강의 계획표 / C# 프로그래밍 2판

주	주제
1	자료구조와 알고리즘 소개
2	파이썬 기초 문법과 데이터 형식
3	선형 리스트
4	단순 연결 리스트
5	원형 연결 리스트
6	스택
7	큐
8	중간고사
9	이진 트리
10	**그래프**
11	재귀 호출
12	정렬 기본
13	정렬 고급
14	검색
15	동적 계획법
16	기말고사

그래프 기본

그래프 기본 / 그래프 구조란?

그래프 구조란?

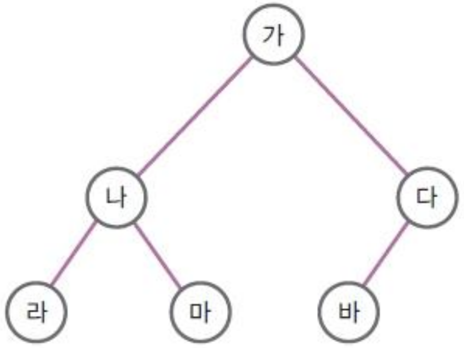
-> 버스 정류장과 여러 노선이 함께 포함된 형태 또는
 링크드인(Linked in)과 같은 사회 관계망 서비스의 연결 등의 형태

그래프 기본 / 그래프 개념

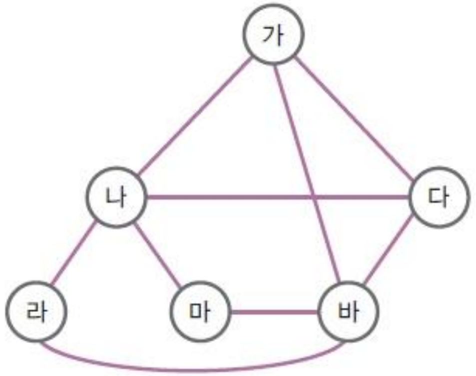
그래프의 개념

-> 여러 노드가 서로 연결된 자료구조

- 트리는 하위 노드로만 연결



(a) 트리 예

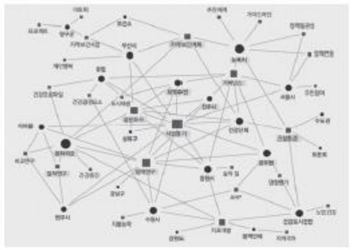


(b) 그래프 예

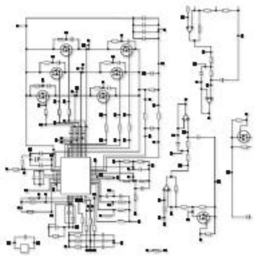
-> 그래프를 활용한 예

지하철 노선도(부산)

KTX 노선도



도시 도로망



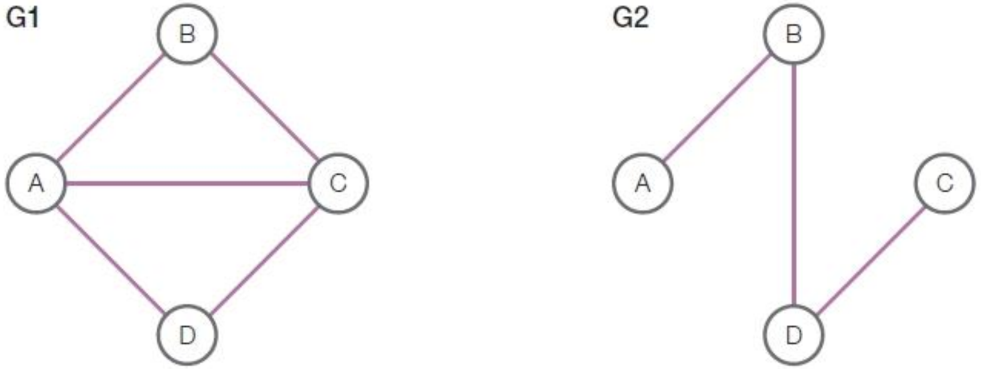
전기 회로도

인맥 관계도

그래프 기본 / 그래프 종류

그래프의 종류

-> 무방향 그래프 : 간선에 방향성이 없는 그래프



-> G1, G2의 정점(Vertex) 집합 표현

V(G1) = { A, B, C, D }
V(G2) = { A, B, C, D }

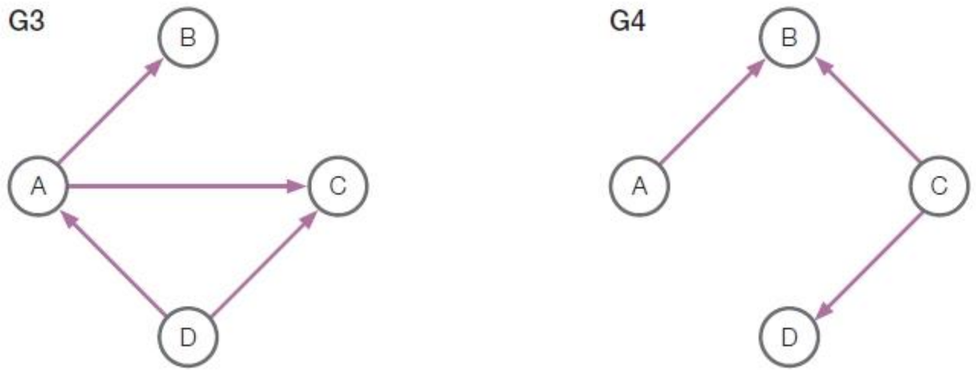
-> G1, G2의 간선(Edge) 집합 표현 - 정점 간 연결, (A, B)와 (B, A)는 같은 간선

E(G1) = { (A, B), (A, C), (A, D), (B, C), (C, D) }
E(G2) = { (A, B), (B, D), (D, C) }

그래프 기본 / 그래프 종류

그래프의 종류

-> 방향 그래프 : 화살표로 간선 방향을 표기하고, 그래프의 정점 집합이 무방향 그래프와 같음



-> G3, G4의 정점 집합 표현(무방향 그래프와 같음)

V(G3) = { A, B, C, D }
V(G4) = { A, B, C, D }

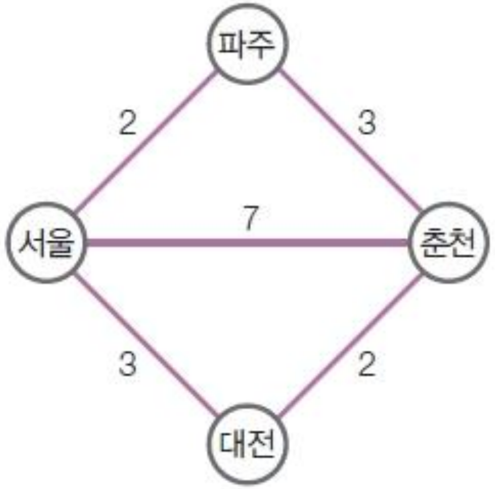
-> G3, G4의 간선 집합 표현 - 방향 그래프는 괄호 표기가 다름, <A, B>와 <B, A>는 다른 간선

E(G3) = { <A, B>, <A, C>, <D, A>, <D, C> }
E(G4) = { <A, B>, <C, B>, <C, D> }

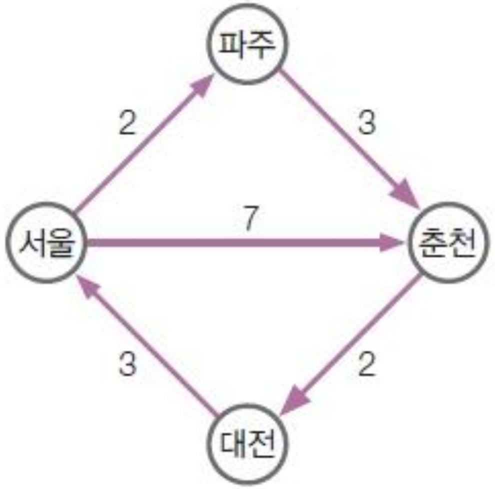
그래프 기본 / 그래프 종류

그래프의 종류

-> 가중치(Weight) 그래프 : 간선마다 가중치가 다르게 부여된 그래프
-> 무방향, 방향 그래프 모두 가중치 부여 가능



(a) 무방향 가중치 그래프



(b) 방향 가중치 그래프

무방향 그래프와 방향 그래프에 각각 가중치를 부여한 경우 예

서울에서 대전으로 가는 경우

\> 무방향 그래프 : 3

\> 방향 그래프 : 7 (서울, 파주, 춘천, 대전 - 최소비용)

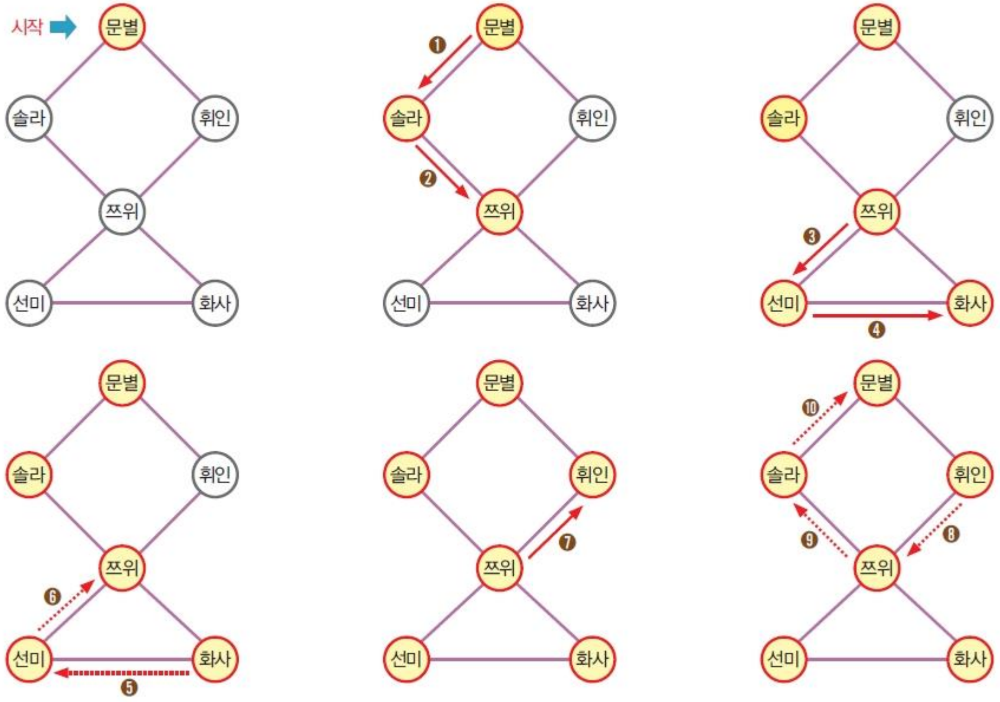
그래프 기본 / 그래프 탐색

깊이 우선 탐색의 작동

-> 그래프의 모든 정점을 한 번씩 방문하는 것을 그래프 순회(Graph Traversal)라고 함

-> 그래프 순회 방식은 깊이 우선 탐색(Depth First Search, DFS)
너비 우선 탐색(Breadth First Search, BFS)이 대표적

-> 그래프는 트리와 달리 부모 자식 개념이 없으므로 어떤 정점에서 시작해도 결과는 같음



1) [문별]에서 시작해서 가나다 순에 따라 [솔라] 방문
2) [솔라]에서 방문 가능한 [쯔위]로 이동
3) [쯔위]에서 가나다 순에 따라 [선미] 방문
4) [선미]에서 방문 가능한 [화사] 방문
5) [화사]와 연결된 노드는 모두 방문했으므로 되돌아 감
6) [선미] 와 연결된 노드는 모두 방문했으므로 되돌아 감
7) [쯔위]에서 방문 가능한 [휘인] 방문
8,9,10) 시작 정점까지 되돌아가며 방문 가능한 노드를 확인 후 종료

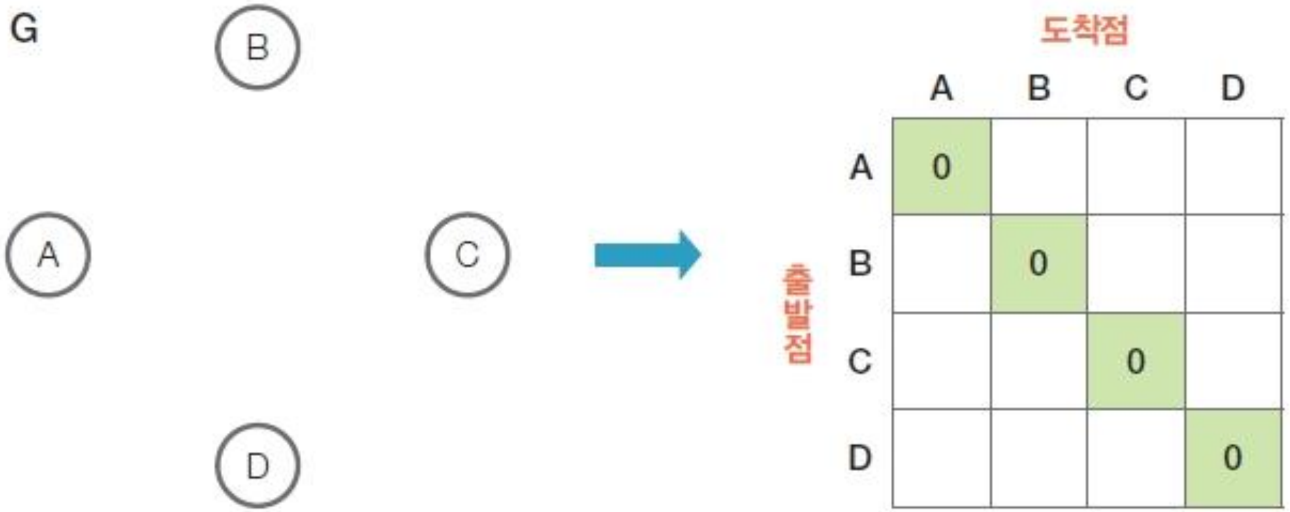
너비 우선 탐색은 시작 노드에서 방문 가능한
모든 노드를 방문한 후에 이후 노드들을 방문

그래프 기본 / 그래프 행렬 표현

그래프의 인접 행렬 표현

-> 그래프를 코드로 구현할 때는 인접 행렬을 사용

-> 인접 행렬은 정방형으로 구성된 행렬로 정점이 4개인 그래프는 4×4로 표현



도착점

출발점

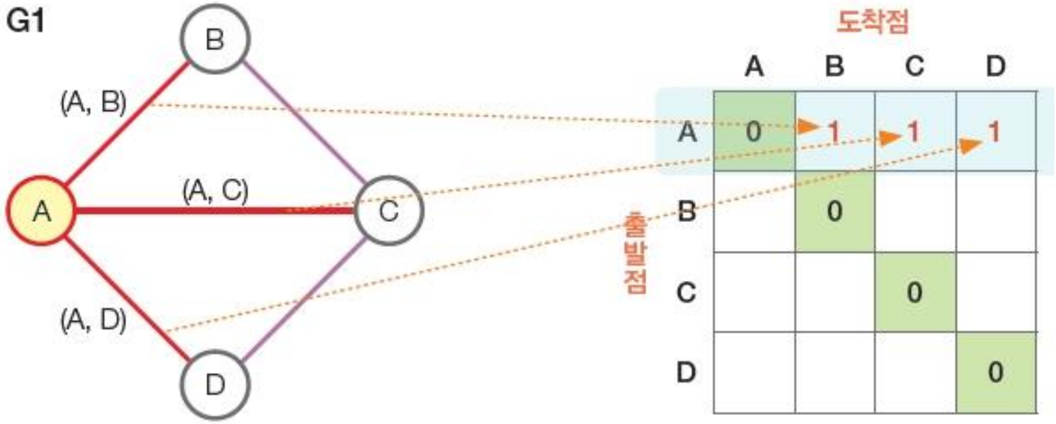
	A	B	C	D
A	0			
B		0		
C			0	
D				0

정점 4개로 된 그래프의 인접 행렬 초기 상태

그래프 기본 / 그래프 행렬 표현

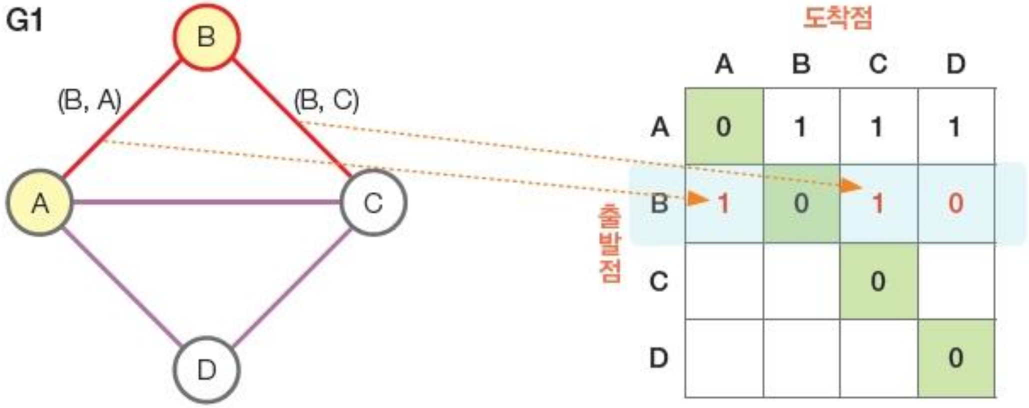
무방향 그래프의 인접 행렬

1) 출발점 A와 연결된 도착점 B, C, D의 칸을 1로 설정한다.



	A	B	C	D
A	0	1	1	1
B		0		
C			0	
D				0

2) 출발점 B와 연결된 도착점 A와 C의 칸을 1로 설정하고, 연결되지 않은 도착점 D는 0으로 설정한다.

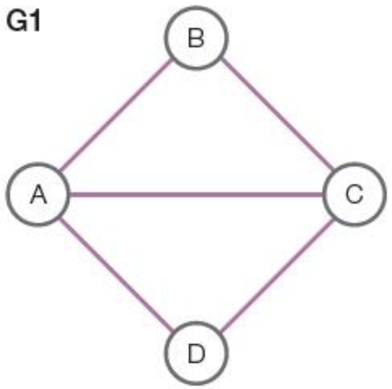


	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C			0	
D				0

그래프 기본 / 그래프 행렬 표현

무방향 그래프의 인접 행렬

3) 같은 방식으로 출발점 C와 D를 인접 행렬로 추가한다.




도착점

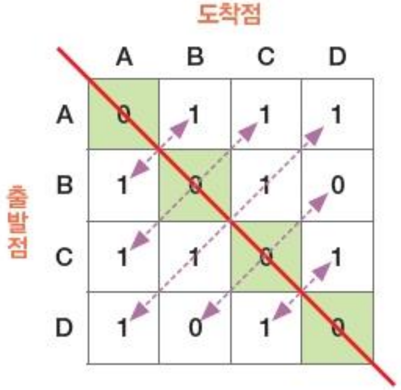
출발점

	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	1
D	1	0	1	0


도착점

출발점

	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	1
D	1	0	1	0


도착점

출발점

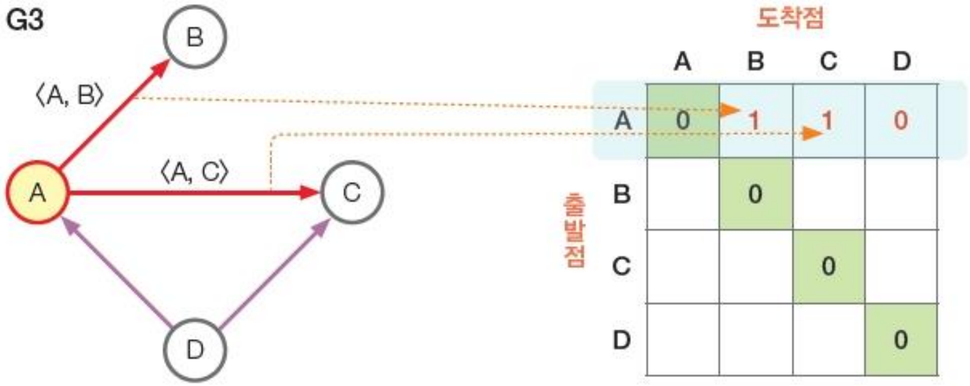
	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	1
D	1	0	1	0

무방향 그래프의 인접 행렬은 대각선을 기준으로 서로 대칭

그래프 기본 / 그래프 행렬 표현

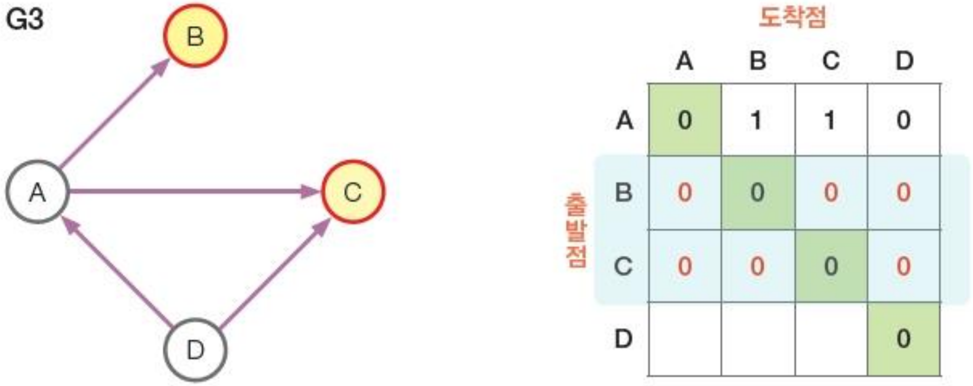
방향 그래프의 인접 행렬

1) 출발점 A에서 나가 도착점이 B, C의 칸만 1로 설정하고 나머지는 0으로 채운다.



	A	B	C	D
A	0	1	1	0
B		0		
C			0	
D				0

2) 출발점 B와 C는 나가는 곳이 없으므로 모두 0으로 채운다.

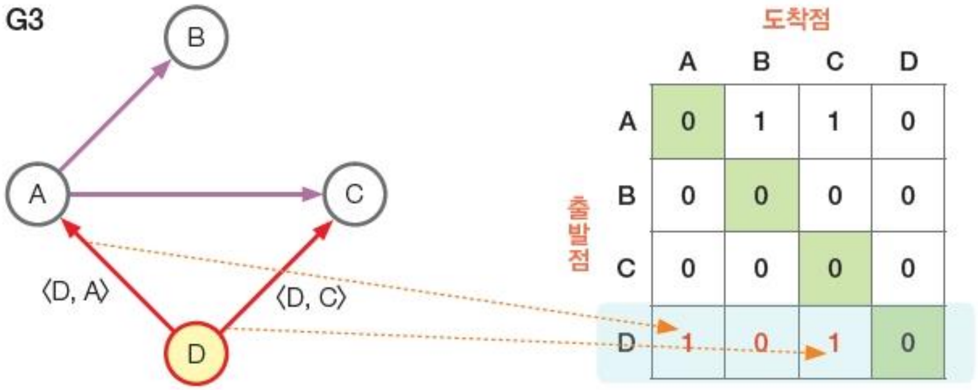


	A	B	C	D
A	0	1	1	0
B	0	0	0	0
C	0	0	0	0
D				0

그래프 기본 / 그래프 행렬 표현

방향 그래프의 인접 행렬

1) 출발점 D는 도착점 A와 C만 1로 설정하고 나머지는 0으로 채운다.



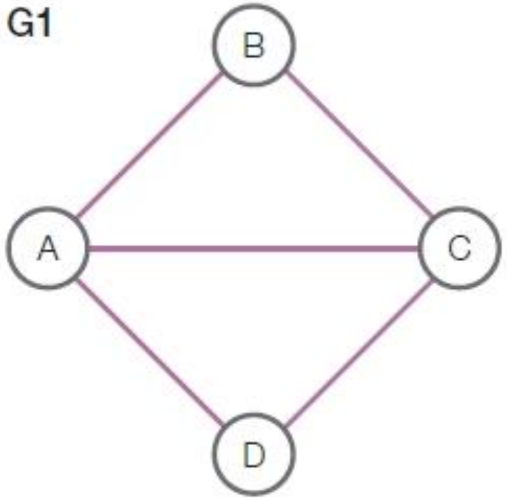
	A	B	C	D
A	0	1	1	0
B	0	0	0	0
C	0	0	0	0
D	1	0	1	0

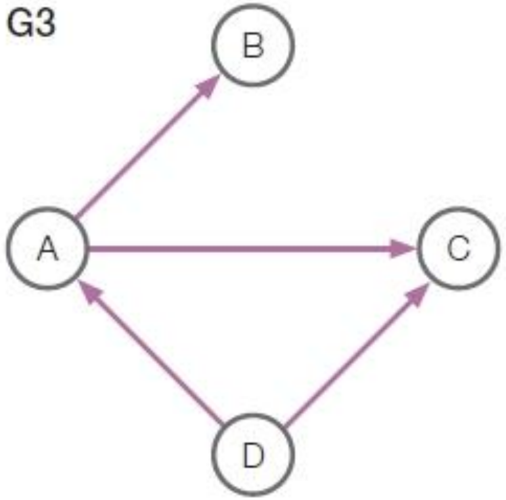
그래프 구현

그래프 구현 / 그래프 구현

무방향성 G1 그래프와 방향성 G3 그래프 구현 예

-> 그래프 G1과 G3를 각각 구현한다.





그래프 구현 / 그래프 구현

그래프의 정점 생성

-> 행과 열이 같은 2차원 배열(정점이 4개 이므로 4 x 4)을 생성하는 클래스로 작성

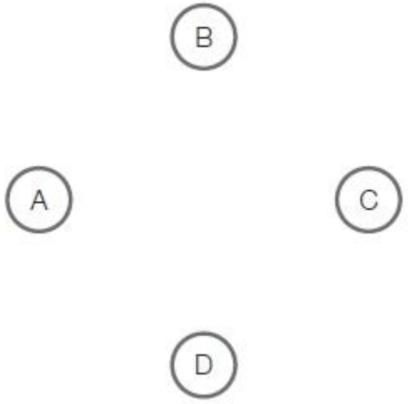
```
class Graph() :
   def __init__ (self, size) :
     self.SIZE = size
     self.graph = [[0 for _ in range(size)] for _ in range(size)]

G1 = Graph(4)
```

도착점

출발점

	A=0	B=1	C=2	D=3
A=0	0	0	0	0
B=1	0	0	0	0
C=2	0	0	0	0
D=3	0	0	0	0

G1(size=4)



G1(size=4)

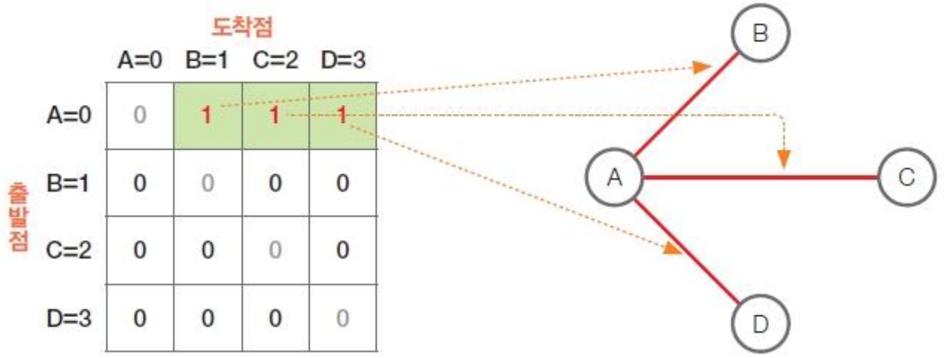
정점 4개를 가진 그래프의 초기 상태(인접 행렬)와 그래프

그래프 구현 / 그래프 구현

그래프의 정점 연결

-> 정점 A에 연결된 간선 구현

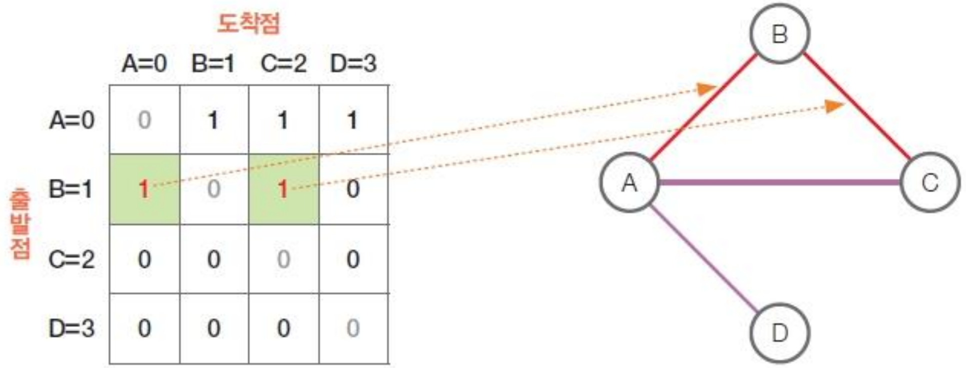
```
G1.graph[0][1] = 1    # (A, B) 간선
G1.graph[0][2] = 1    # (A, C) 간선
G1.graph[0][3] = 1    # (A, D) 간선
```



	A=0	B=1	C=2	D=3
A=0	0	1	1	1
B=1	0	0	0	0
C=2	0	0	0	0
D=3	0	0	0	0

-> 정점 B에 연결된 간선 구현

```
G1.graph[1][0] = 1    # (B, A) 간선
G1.graph[1][2] = 1    # (B, C) 간선
```



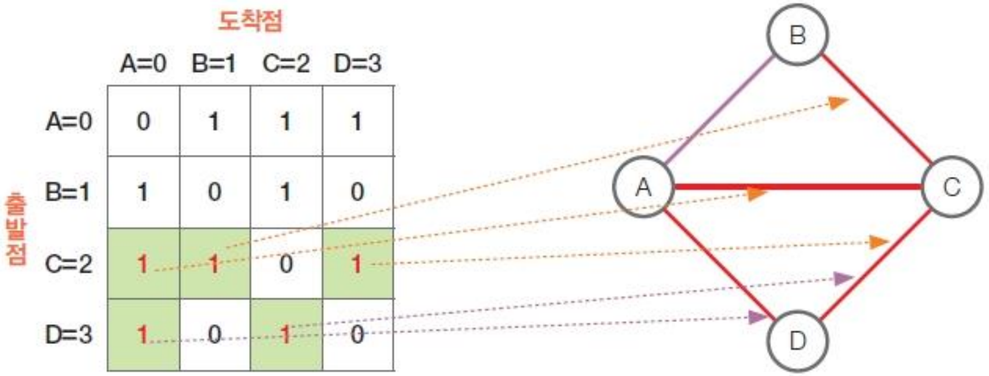
	A=0	B=1	C=2	D=3
A=0	0	1	1	1
B=1	1	0	1	0
C=2	0	0	0	0
D=3	0	0	0	0

그래프 구현 / 그래프 구현

그래프의 정점 연결

-> 같은 방식으로 출발점 C와 D를 다음과 같이 연결

```
G1.graph[2][0] = 1    # (C, A) 간선
G1.graph[2][1] = 1    # (C, B) 간선
G1.graph[2][3] = 1    # (C, D) 간선

G1.graph[3][0] = 1    # (D, A) 간선
G1.graph[3][2] = 1    # (D, C) 간선
```

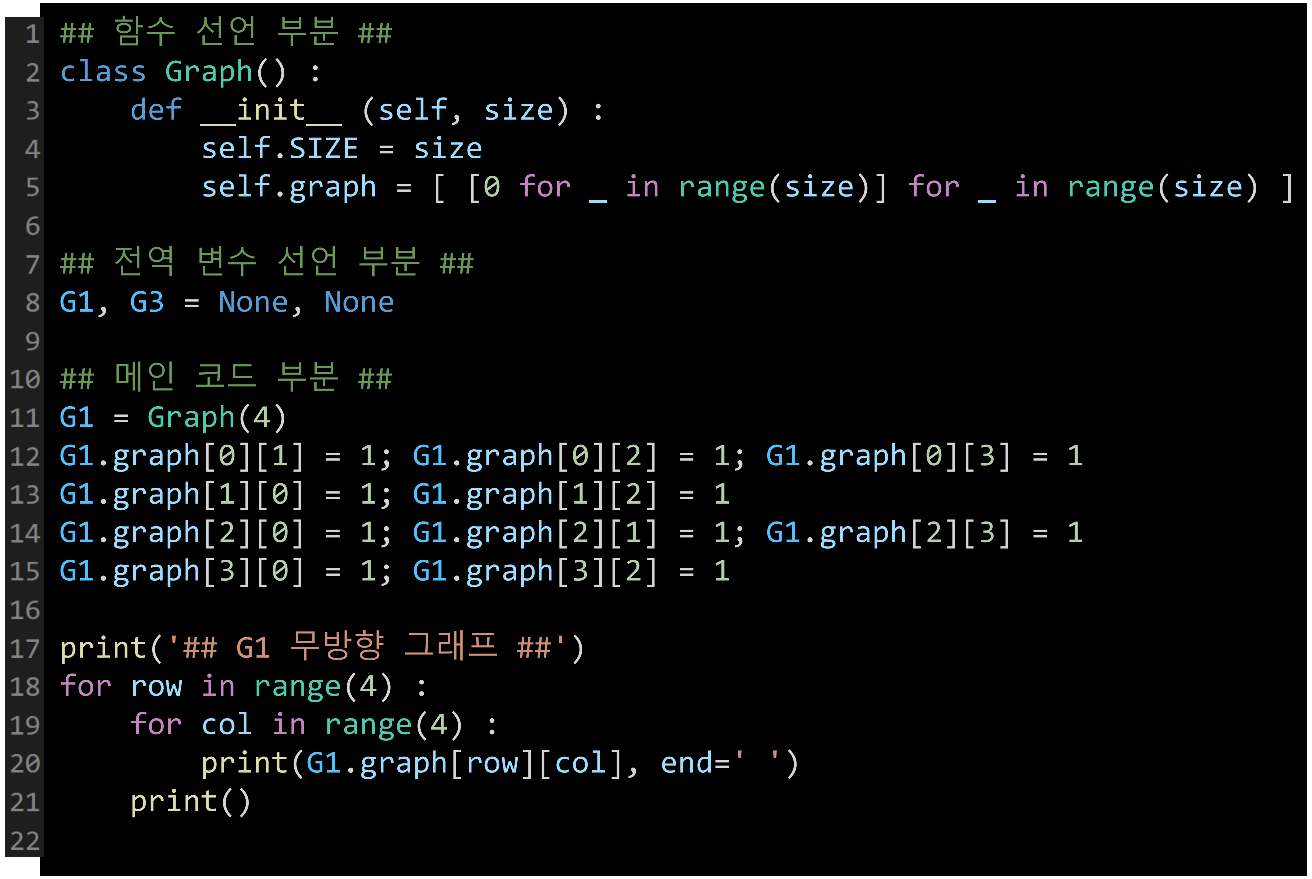


	A=0	B=1	C=2	D=3
A=0	0	1	1	1
B=1	1	0	1	0
C=2	1	1	0	1
D=3	1	0	1	0

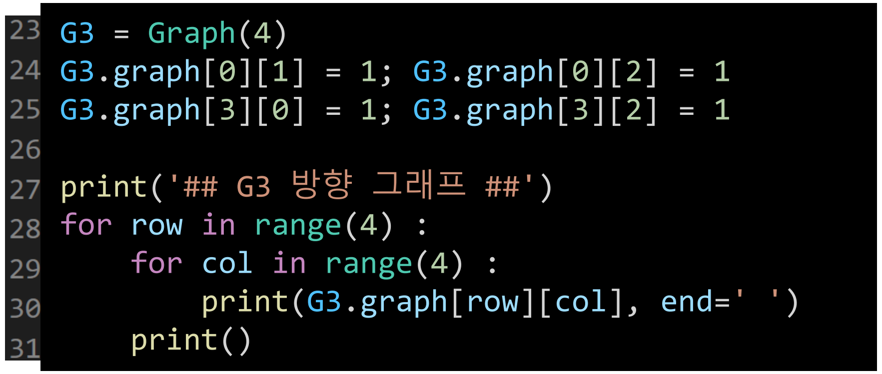
그래프 구현 / 그래프 구현

그래프의 정점 연결

-> 소스 코드



```python
## 함수 선언 부분 ##
class Graph() :
    def __init__ (self, size) :
        self.SIZE = size
        self.graph = [ [0 for _ in range(size)] for _ in range(size) ]

## 전역 변수 선언 부분 ##
G1, G3 = None, None

## 메인 코드 부분 ##
G1 = Graph(4)
G1.graph[0][1] = 1; G1.graph[0][2] = 1; G1.graph[0][3] = 1
G1.graph[1][0] = 1; G1.graph[1][2] = 1
G1.graph[2][0] = 1; G1.graph[2][1] = 1; G1.graph[2][3] = 1
G1.graph[3][0] = 1; G1.graph[3][2] = 1

print('## G1 무방향 그래프 ##')
for row in range(4) :
    for col in range(4) :
        print(G1.graph[row][col], end=' ')
    print()

```



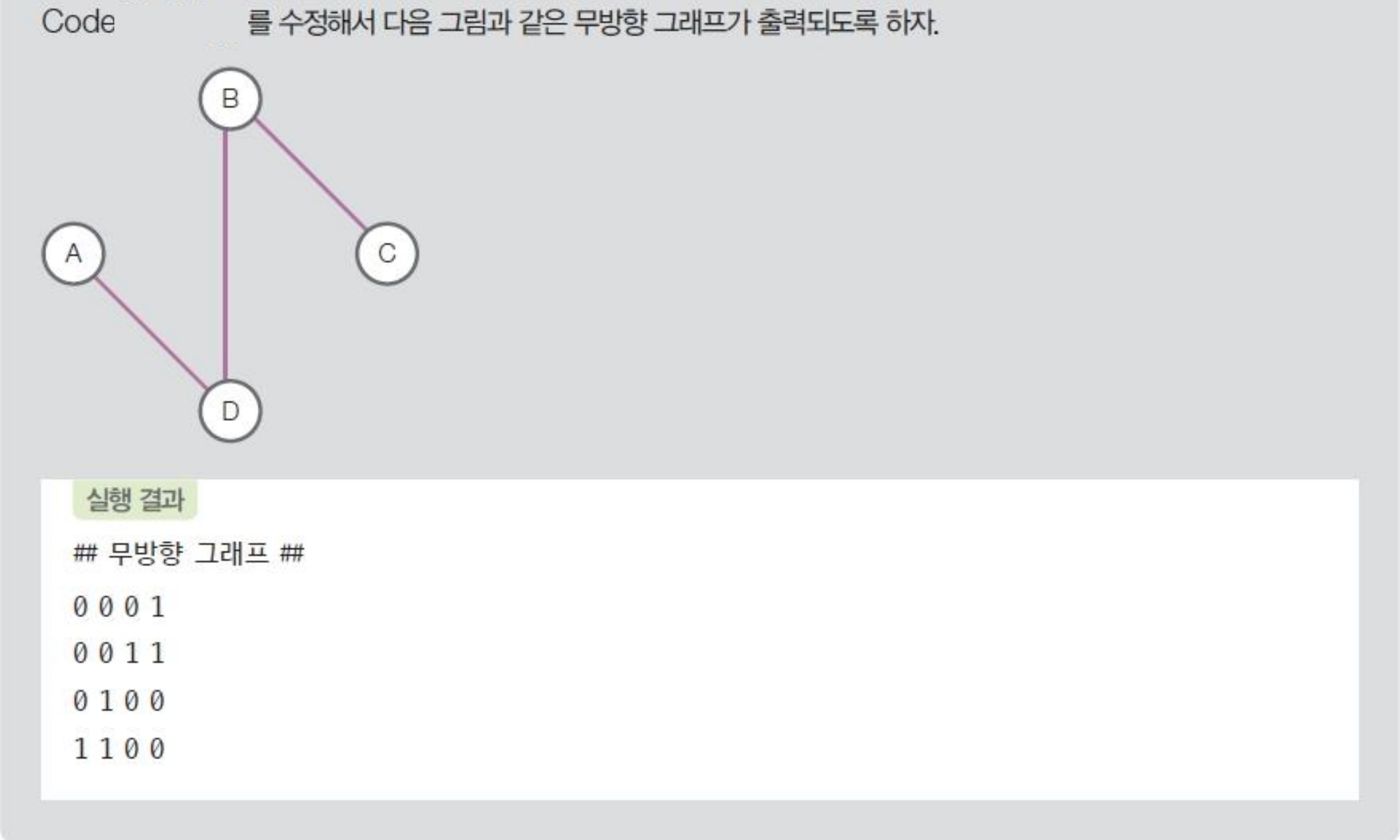
```python
G3 = Graph(4)
G3.graph[0][1] = 1; G3.graph[0][2] = 1
G3.graph[3][0] = 1; G3.graph[3][2] = 1

print('## G3 방향 그래프 ##')
for row in range(4) :
    for col in range(4) :
        print(G3.graph[row][col], end=' ')
    print()
```

실행 결과

```
## G1 무방향 그래프 ##
0 1 1 1
1 0 1 0
1 1 0 1
1 0 1 0
## G3 방향 그래프 ##
0 1 1 0
0 0 0 0
0 0 0 0
1 0 1 0
```

그래프 구현 / 그래프 구현

연습 문제



Code 를 수정해서 다음 그림과 같은 무방향 그래프가 출력되도록 하자.

실행 결과

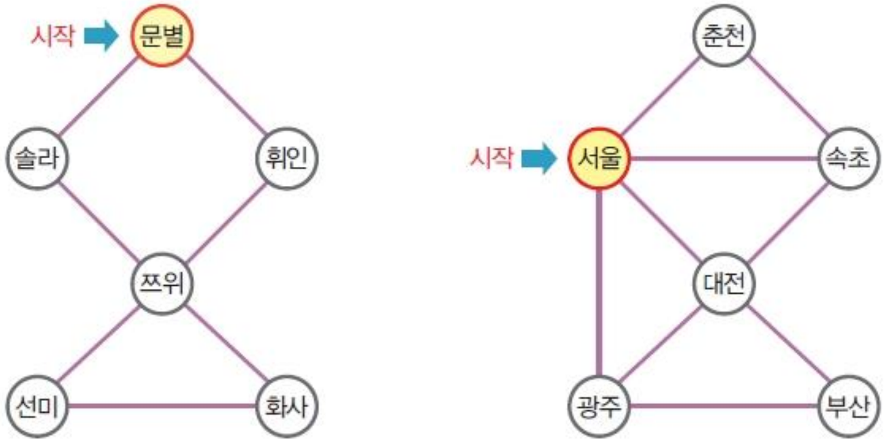
```
## 무방향 그래프 ##
0 0 0 1
0 0 1 1
0 1 0 0
1 1 0 0
```

그래프 구현 / 그래프 개선

그래프 개선

-> 그래프 정점은 숫자가 아닌 문자로 구성하기도 하므로 코드 수정

-> 무방향 그래프를 인접 행렬로 구성할 때 사람 이름, 도시 이름으로 구성 예



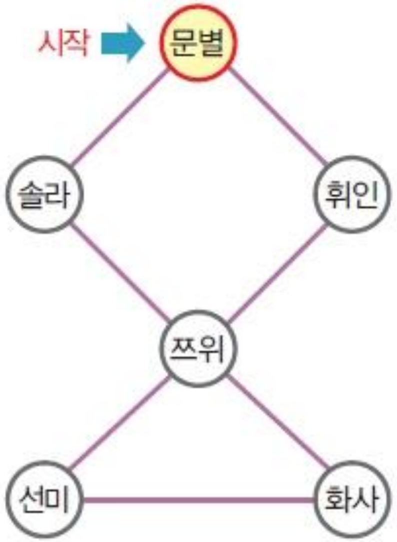
```
G1.graph[0][1] = 1; G1.graph[0][2] = 1
G1.graph[1][0] = 1; G1.graph[1][3] = 1
```

```
문별, 솔라, 휘인, 쯔위 = 0, 1, 2, 3
G1.graph[문별][솔라] = 1; G1.graph[문별][휘인] = 1
G1.graph[솔라][문별] = 1; G1.graph[솔라][쯔위] = 1
```

변수 이름을 정점 번호로 지정하면 더 직관적임

그래프 구현 / 그래프 개선

인접 행렬 출력 시 주석 없이 출력하는 예와 주석을 추가하여 출력하는 예



(a) 구현할 그래프

0	1	1	0	0	0
1	0	0	1	0	0
1	0	0	1	0	0
0	1	1	0	1	1
0	0	0	1	0	1
0	0	0	1	1	0

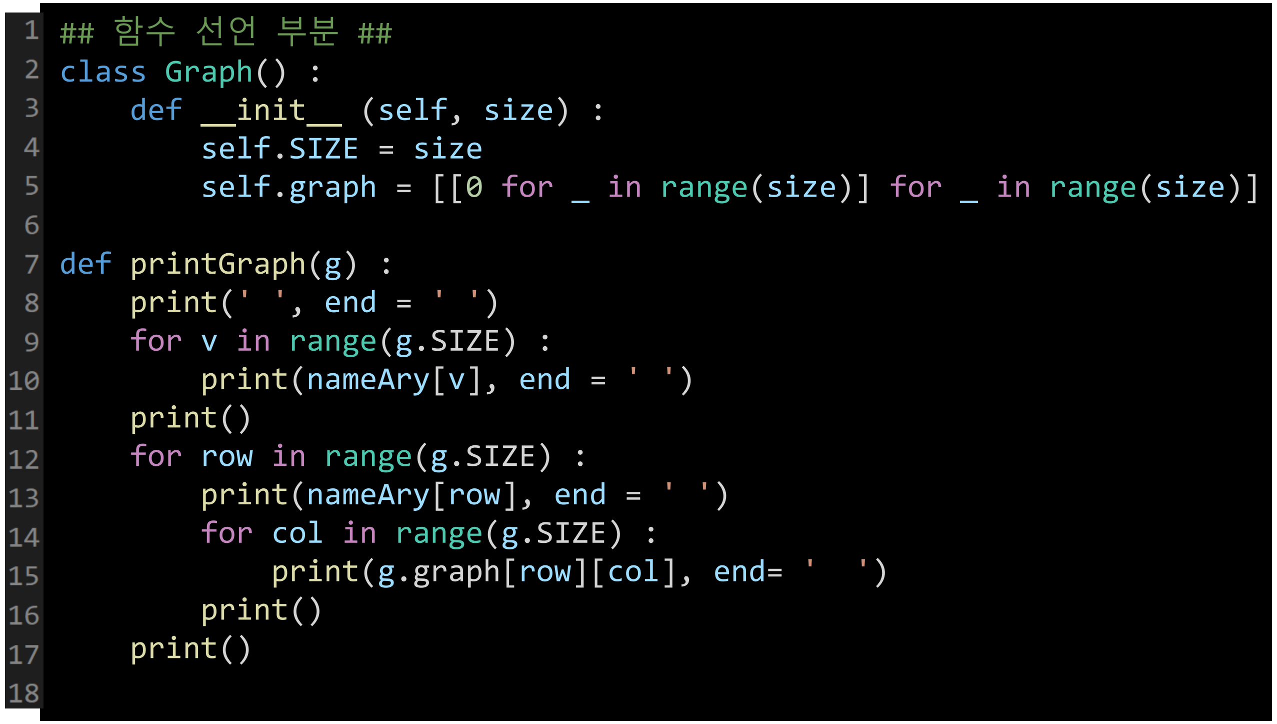
(b) 행과 열의 주석이 없는 인접 행렬

	문별	솔라	휘인	쯔위	선미	화사
문별	0	1	1	0	0	0
솔라	1	0	0	1	0	0
휘인	1	0	0	1	0	0
쯔위	0	1	1	0	1	1
선미	0	0	0	1	0	1
화사	0	0	0	1	1	0

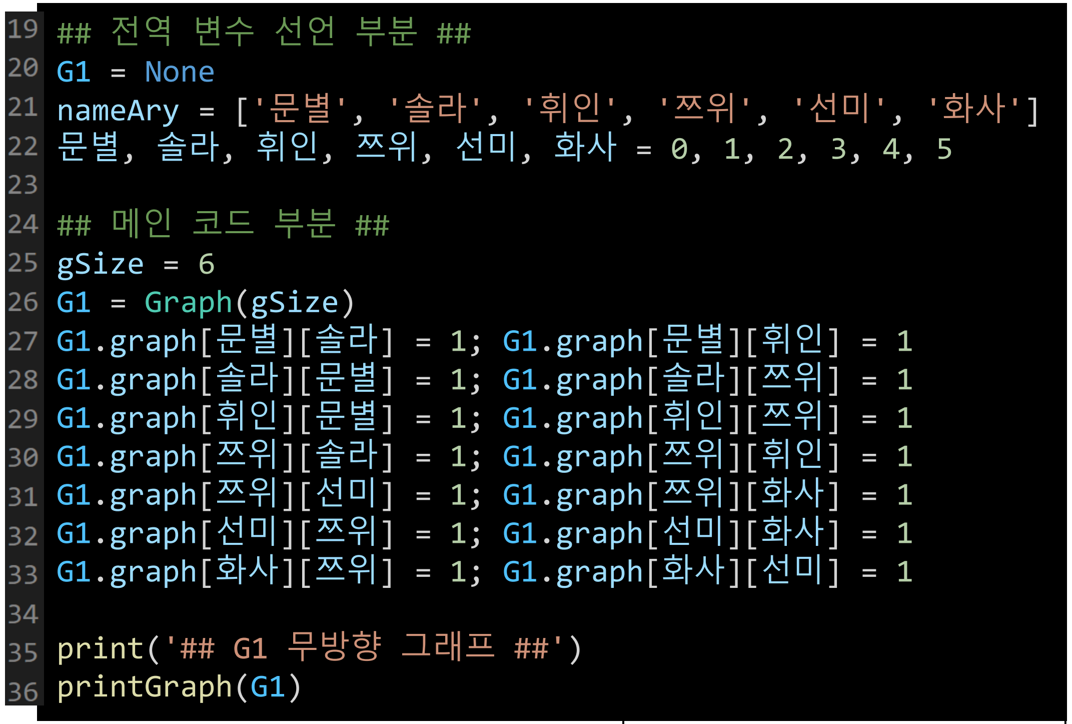
(c) 행과 열의 주석이 있는 인접 행렬

그래프 구현 / 그래프 개선

개선된 그래프 코드



```python
## 함수 선언 부분 ##
class Graph() :
    def __init__ (self, size) :
        self.SIZE = size
        self.graph = [[0 for _ in range(size)] for _ in range(size)]

def printGraph(g) :
    print(' ', end = ' ')
    for v in range(g.SIZE) :
        print(nameAry[v], end = ' ')
    print()
    for row in range(g.SIZE) :
        print(nameAry[row], end = ' ')
        for col in range(g.SIZE) :
            print(g.graph[row][col], end= '  ')
        print()
    print()

```



```python
## 전역 변수 선언 부분 ##
G1 = None
nameAry = ['문별', '솔라', '휘인', '쯔위', '선미', '화사']
문별, 솔라, 휘인, 쯔위, 선미, 화사 = 0, 1, 2, 3, 4, 5

## 메인 코드 부분 ##
gSize = 6
G1 = Graph(gSize)
G1.graph[문별][솔라] = 1; G1.graph[문별][휘인] = 1
G1.graph[솔라][문별] = 1; G1.graph[솔라][쯔위] = 1
G1.graph[휘인][문별] = 1; G1.graph[휘인][쯔위] = 1
G1.graph[쯔위][솔라] = 1; G1.graph[쯔위][휘인] = 1
G1.graph[쯔위][선미] = 1; G1.graph[쯔위][화사] = 1
G1.graph[선미][쯔위] = 1; G1.graph[선미][화사] = 1
G1.graph[화사][쯔위] = 1; G1.graph[화사][선미] = 1

print('## G1 무방향 그래프 ##')
printGraph(G1)
```

실행 결과

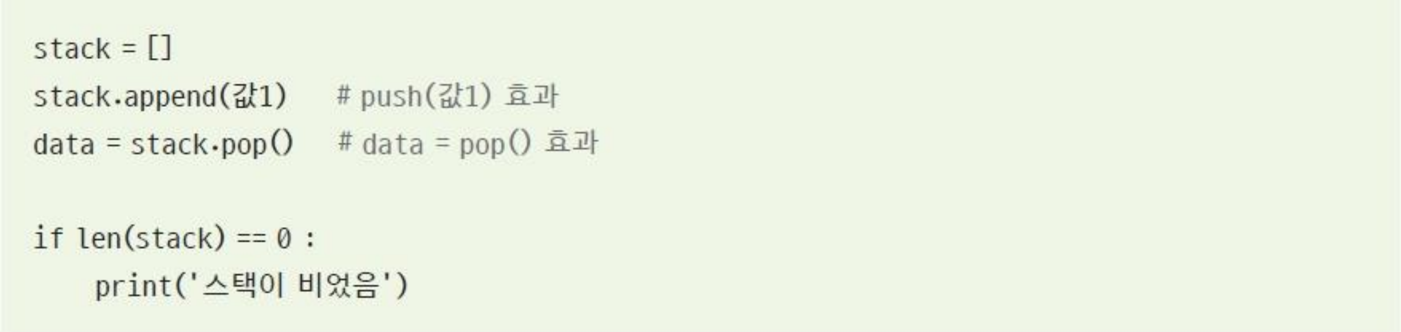
```
## G1 무방향 그래프 ##
  문별 솔라 휘인 쯔위 선미 화사
문별 0  1  1  0  0  0
솔라 1  0  0  1  0  0
휘인 1  0  0  1  0  0
쯔위 0  1  1  0  1  1
선미 0  0  0  1  0  1
화사 0  0  0  1  1  0
```

그래프 구현 / 깊이 우선 탐색의 구현

깊이 우선 탐색의 구현

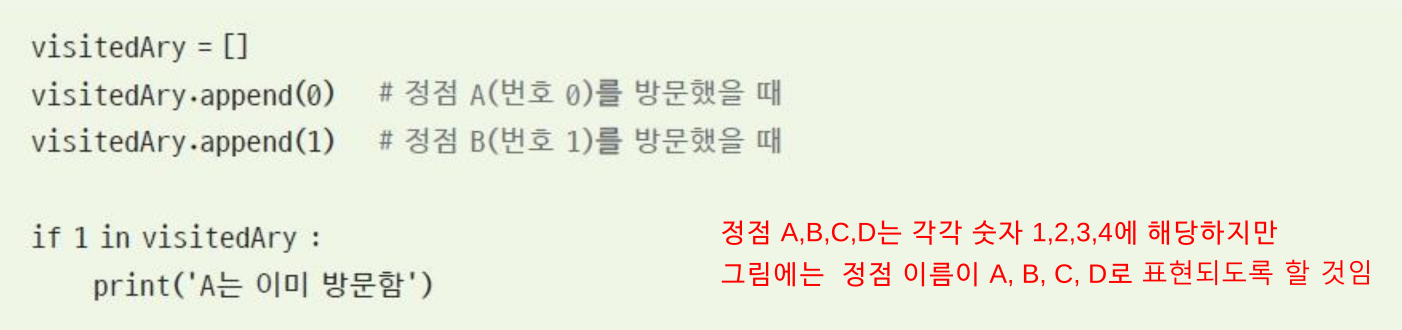
-> 깊이 우선 탐색을 구현하려면 **스택**을 사용해야 함

-> 코드를 좀 더 간략히 하고자 별도의 top을 사용하지 않고 append( )로 푸시를, pop( )으로 팝하는 스택을 사용



```
stack = []
stack.append(값1)    # push(값1) 효과
data = stack.pop()   # data = pop() 효과

if len(stack) == 0 :
    print('스택이 비었음')
```

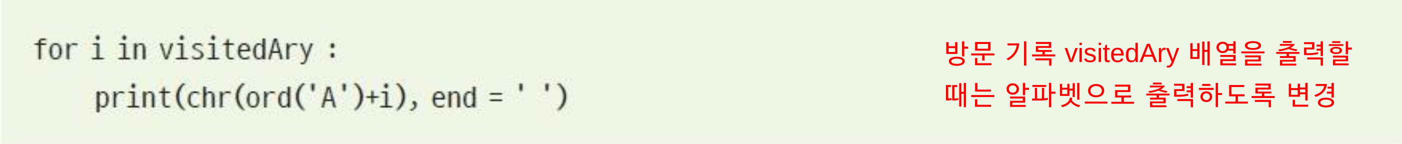
-> visitedAry 배열에 방문 정점을 저장해서 visitedAry 배열에 해당 정점이 있다면 방문한 적이 있는 것으로 처리



```
visitedAry = []
visitedAry.append(0)   # 정점 A(번호 0)를 방문했을 때
visitedAry.append(1)   # 정점 B(번호 1)를 방문했을 때

if 1 in visitedAry :
    print('A는 이미 방문함')
```

정점 A,B,C,D는 각각 숫자 1,2,3,4에 해당하지만
그림에는 정점 이름이 A, B, C, D로 표현되도록 할 것임

ord() : 아스키 코드 변환
chr() : 아스키 코드를 문자로 변환



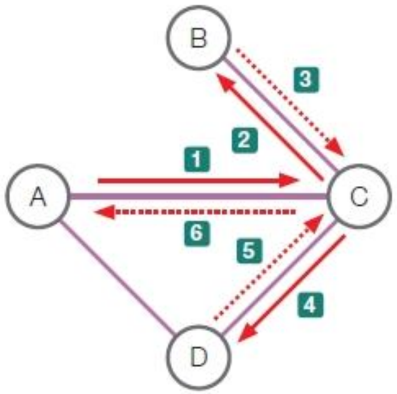
```
for i in visitedAry :
    print(chr(ord('A')+i), end = ' ')
```

방문 기록 visitedAry 배열을 출력할
때는 알파벳으로 출력하도록 변경

그래프 구현 / 깊이 우선 탐색의 구현

깊이 우선 탐색의 구현

-> 간단한 그래프를 깊이 우선 탐색하는 과정 구현 예



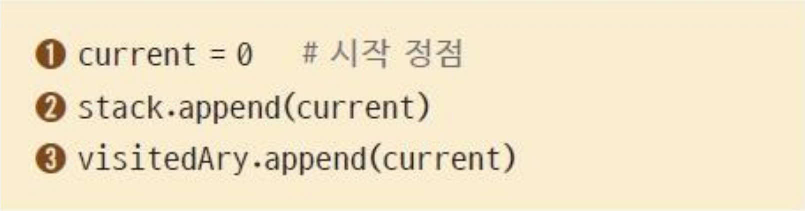

도착점

출발점

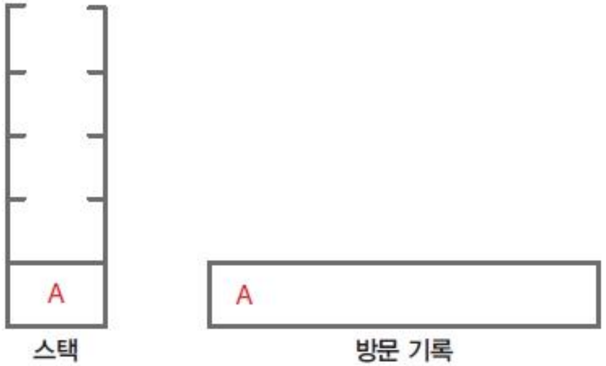
	A=0	B=1	C=2	D=3
A=0	0	0	1	1
B=1	0	0	1	0
C=2	1	1	0	1
D=3	1	0	1	0

-> 첫 번째 정점을 방문하는 것부터 시작하여 1~6 이동을 단계별로 구현

0 첫 번째 정점 방문



```
❶ current = 0    # 시작 정점
❷ stack.append(current)
❸ visitedAry.append(current)
```

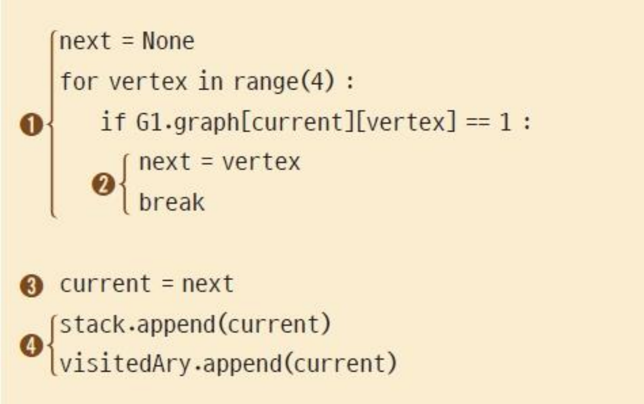


그래프 구현 / 깊이 우선 탐색의 구현

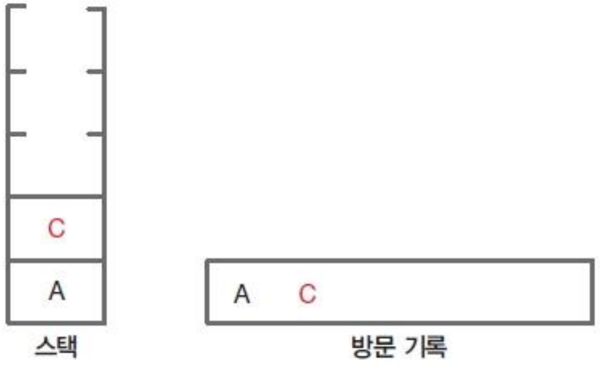
깊이 우선 탐색의 구현

-> 간단한 그래프를 깊이 우선 탐색하는 과정 구현 예

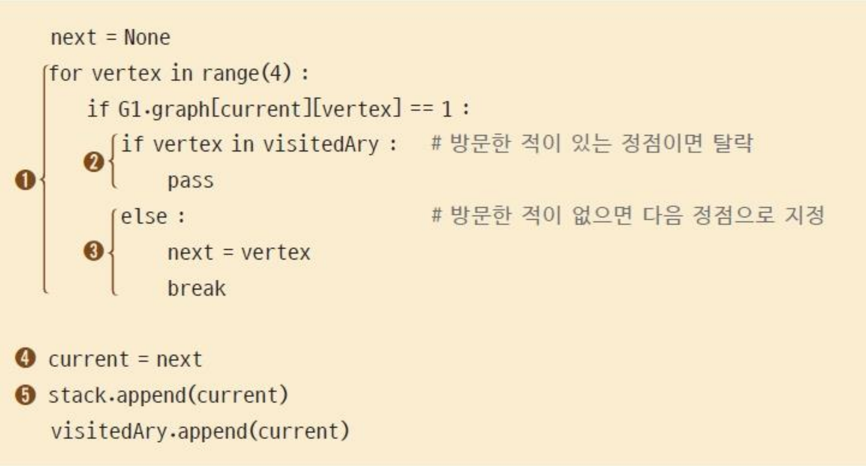
1 과정(정점 C 방문)



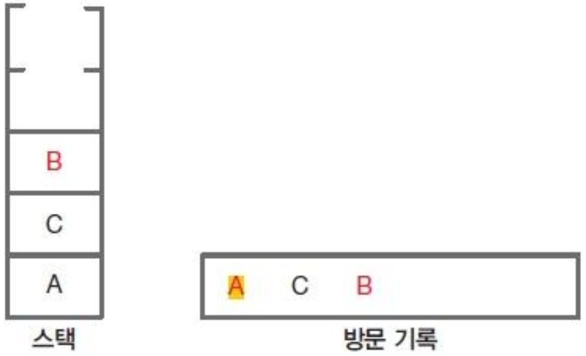
```
   next = None
   for vertex in range(4) :
❶      if G1.graph[current][vertex] == 1 :
         ❷ next = vertex
            break

❸  current = next
❹  stack.append(current)
   visitedAry.append(current)
```



2 과정(정점 B 방문)



```
   next = None
   for vertex in range(4) :
       if G1.graph[current][vertex] == 1 :
        ❷ if vertex in visitedAry :   # 방문한 적이 있는 정점이면 탈락
❶               pass
           else :                      # 방문한 적이 없으면 다음 정점으로 지정
        ❸       next = vertex
                break

❹  current = next
❺  stack.append(current)
   visitedAry.append(current)
```

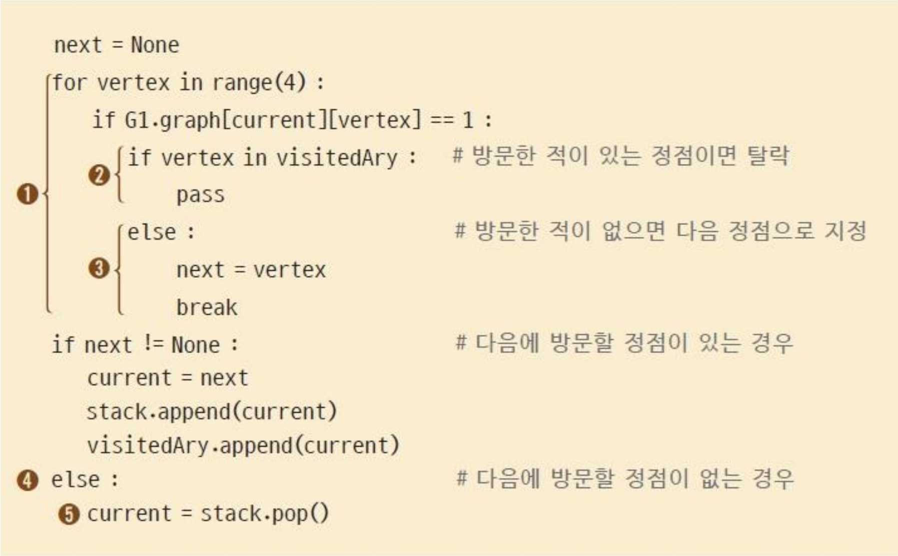


그래프 구현 / 깊이 우선 탐색의 구현

깊이 우선 탐색의 구현

-> 간단한 그래프를 깊이 우선 탐색하는 과정 구현 예

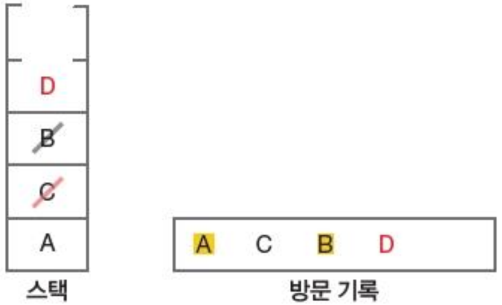
3 과정(되돌아오는 이동)



```
    next = None
   ┌for vertex in range(4) :
   │    if G1.graph[current][vertex] == 1 :
   │      ┌if vertex in visitedAry :      # 방문한 적이 있는 정점이면 탈락
❶ ┤    ❷┤      pass
   │      ┌else :                         # 방문한 적이 없으면 다음 정점으로 지정
   │    ❸┤      next = vertex
   └      └     break
    if next != None :                     # 다음에 방문할 정점이 있는 경우
        current = next
        stack.append(current)
        visitedAry.append(current)
❹  else :                                 # 다음에 방문할 정점이 없는 경우
     ❺ current = stack.pop()
```

② 방문기록에서 C는 이미 방문한 적이 있음을 확인
⑤ B에서 다음에 방문할 정점이 없으므로 pop

4 과정(정점 D 방문)



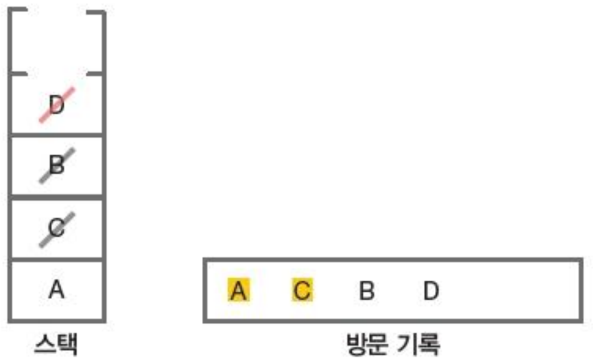
정점 C에서 A와 B는 방문한 적이 있으므로 pass하고
정점 D를 방문함

그래프 구현 / 깊이 우선 탐색의 구현

깊이 우선 탐색의 구현

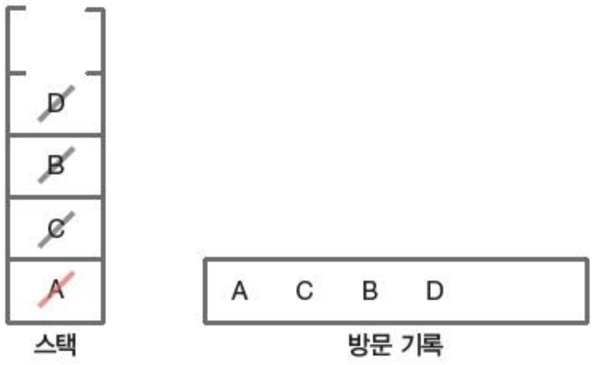
-> 간단한 그래프를 깊이 우선 탐색하는 과정 구현 예

5 과정(되돌아오는 이동)



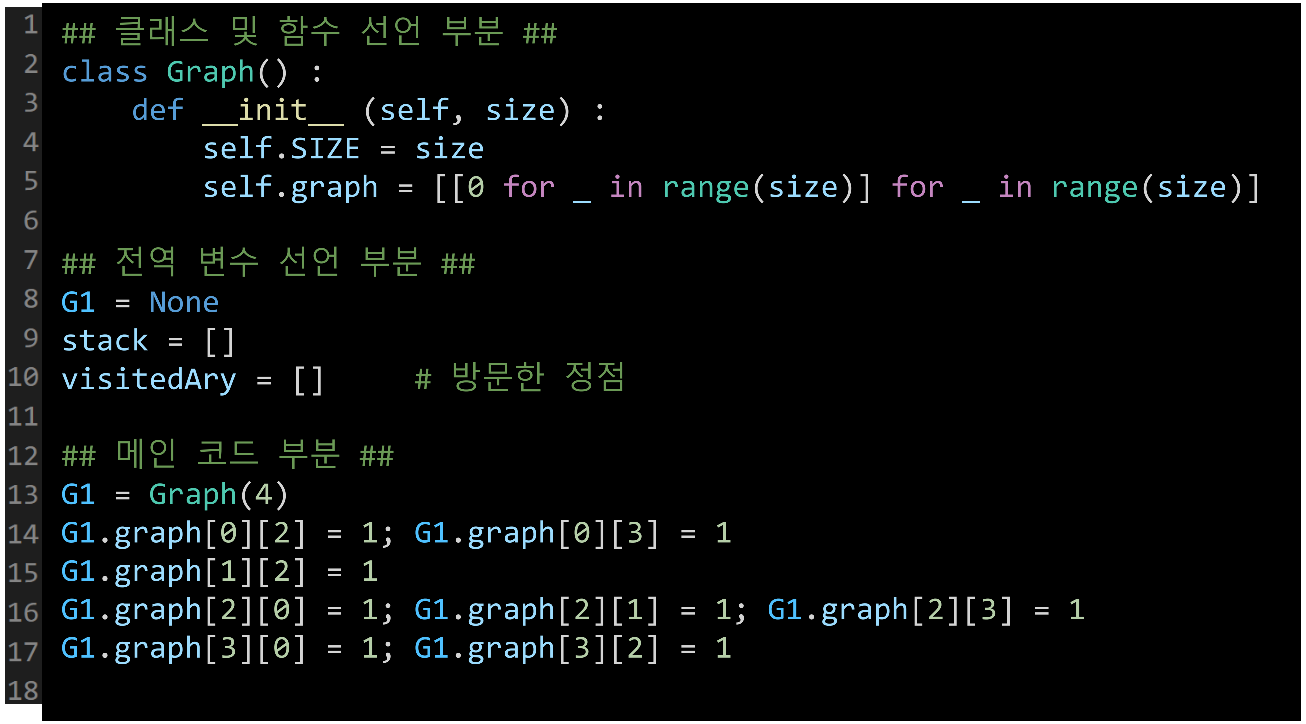
정점 D에서 방문 가능한 곳이 없음

6 과정(되돌아오는 이동)

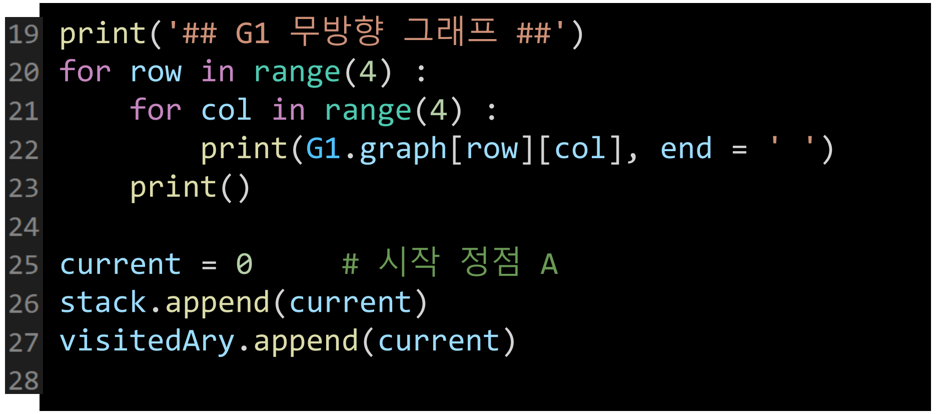


정점 A에서 방문 가능한 곳이 없음

깊이 우선 탐색의 구현 코드 1



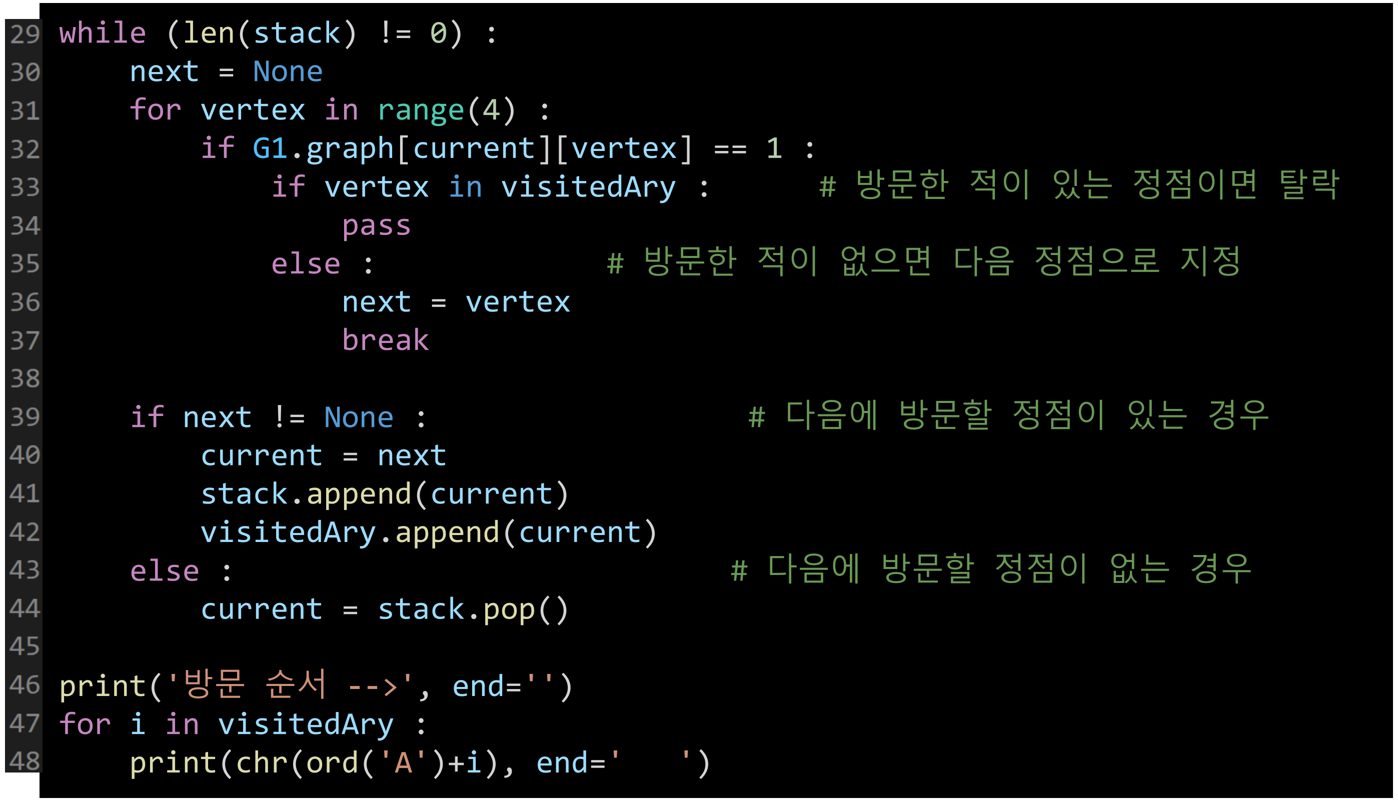
```
## 클래스 및 함수 선언 부분 ##
class Graph() :
    def __init__ (self, size) :
        self.SIZE = size
        self.graph = [[0 for _ in range(size)] for _ in range(size)]

## 전역 변수 선언 부분 ##
G1 = None
stack = []
visitedAry = []     # 방문한 정점

## 메인 코드 부분 ##
G1 = Graph(4)
G1.graph[0][2] = 1; G1.graph[0][3] = 1
G1.graph[1][2] = 1
G1.graph[2][0] = 1; G1.graph[2][1] = 1; G1.graph[2][3] = 1
G1.graph[3][0] = 1; G1.graph[3][2] = 1

```



```
print('## G1 무방향 그래프 ##')
for row in range(4) :
    for col in range(4) :
        print(G1.graph[row][col], end = ' ')
    print()

current = 0     # 시작 정점 A
stack.append(current)
visitedAry.append(current)

```

그래프 구현 / 깊이 우선 탐색의 구현

깊이 우선 탐색의 구현 코드 2



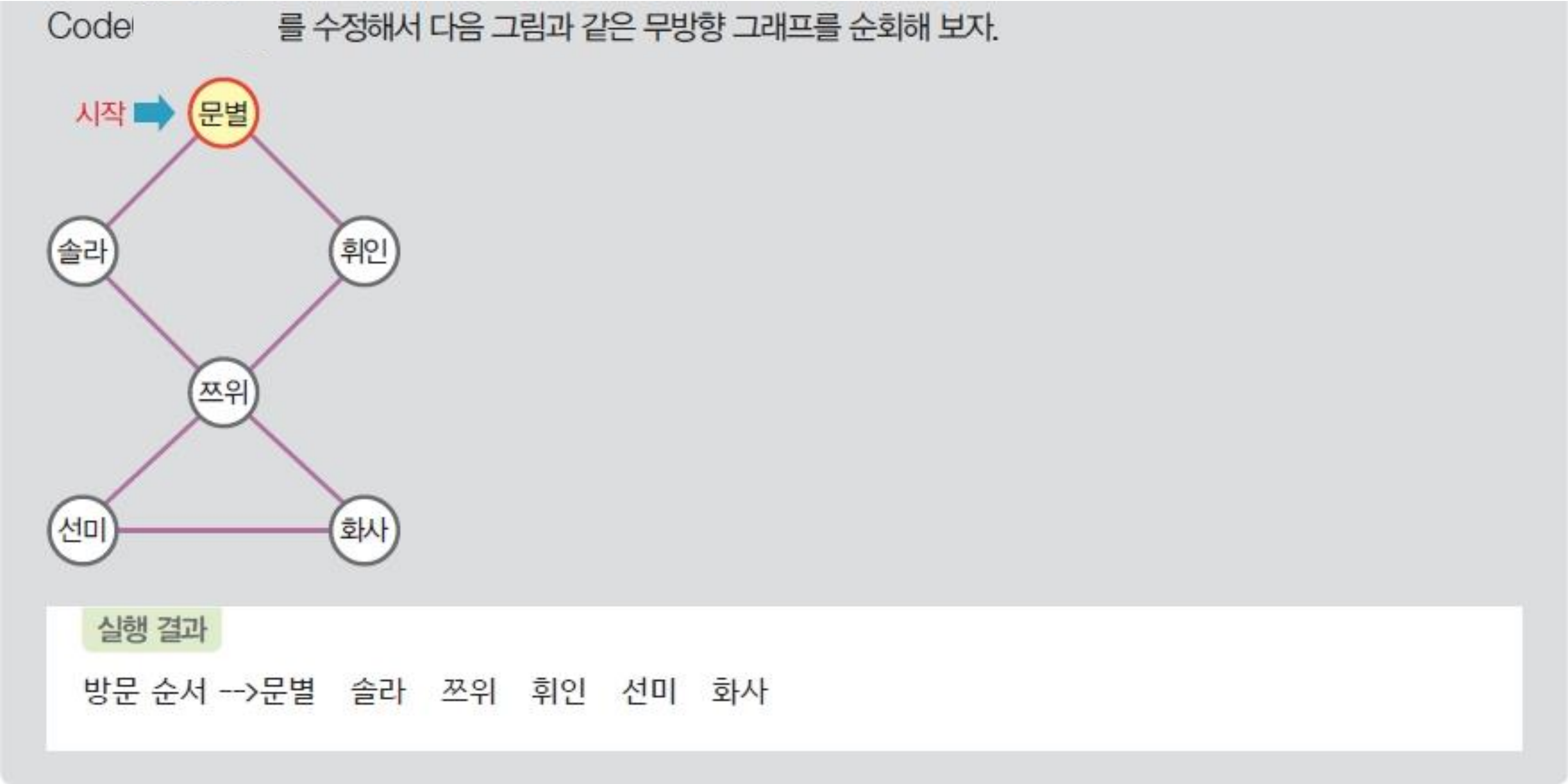
```python
while (len(stack) != 0) :
    next = None
    for vertex in range(4) :
        if G1.graph[current][vertex] == 1 :
            if vertex in visitedAry :      # 방문한 적이 있는 정점이면 탈락
                pass
            else :             # 방문한 적이 없으면 다음 정점으로 지정
                next = vertex
                break

    if next != None :                  # 다음에 방문할 정점이 있는 경우
        current = next
        stack.append(current)
        visitedAry.append(current)
    else :                            # 다음에 방문할 정점이 없는 경우
        current = stack.pop()

print('방문 순서 -->', end='')
for i in visitedAry :
    print(chr(ord('A')+i), end='   ')
```

실행 결과

```
## G1 무방향 그래프 ##
0 0 1 1
0 0 1 0
1 1 0 1
1 0 1 0
방문 순서 -->A   C   B   D
```

그래프 구현 / 깊이 우선 탐색의 구현

연습문제



Code 를 수정해서 다음 그림과 같은 무방향 그래프를 순회해 보자.

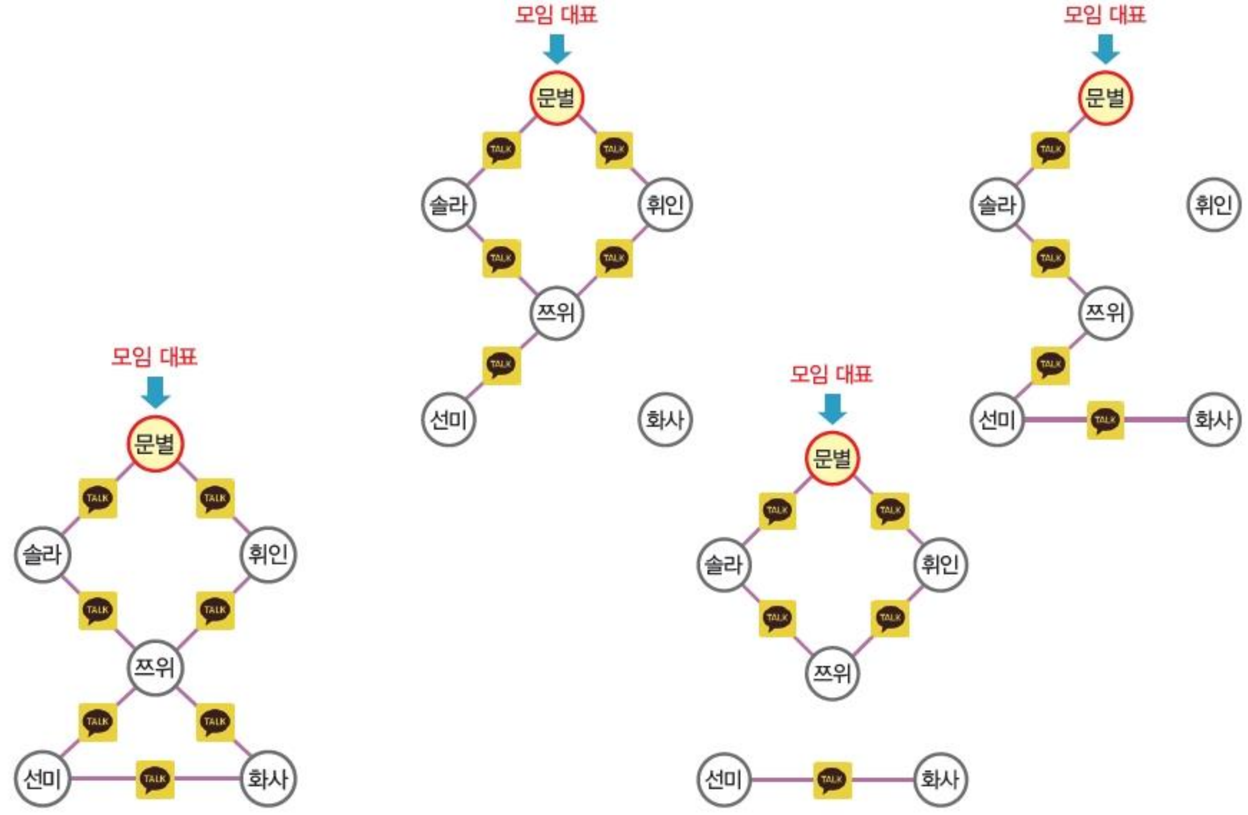
실행 결과

```
방문 순서 -->문별   솔라   쯔위   휘인   선미   화사
```

그래프 응용

그래프 응용 / 친구가 연락되는지 확인

- **친구들의 비상연락망에서 특정 친구가 연락되는지 확인하는 방법**
  -> 비상 연락망의 예



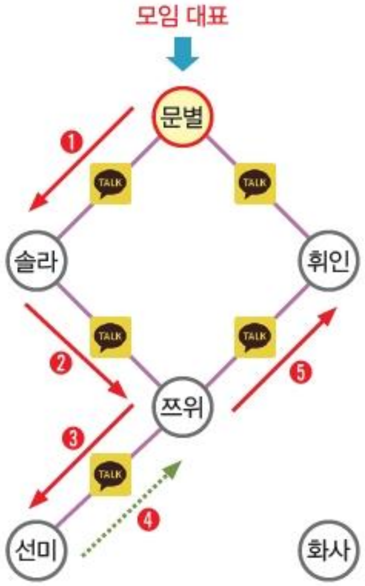
(a) 비상 연락망의 기본 형태

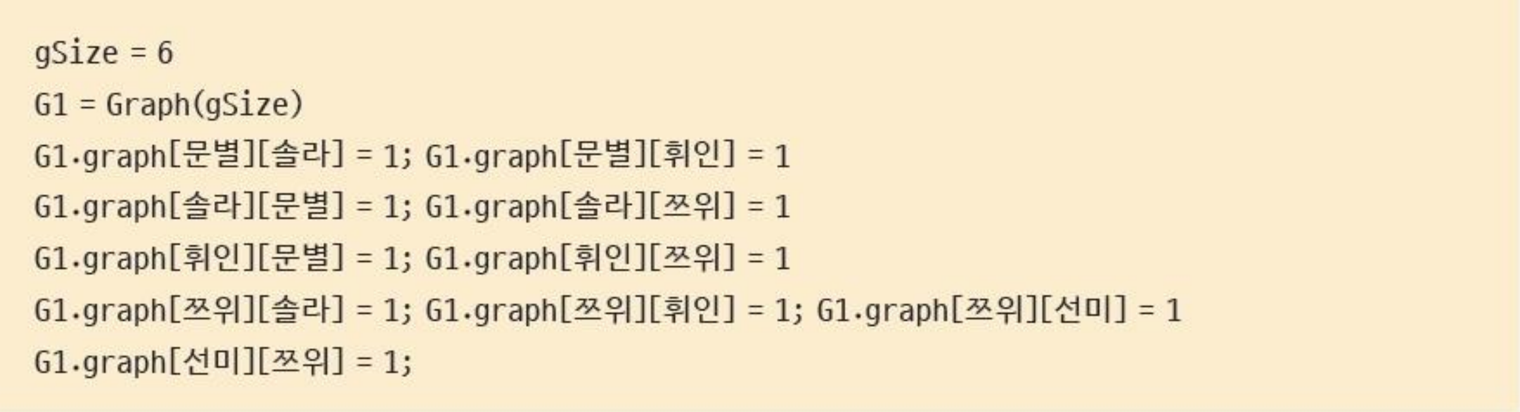
(b) 비상 연락망의 연결이 끊긴 경우

그래프 응용 / 친구가 연락되는지 확인

친구들의 비상연락망에서 특정 친구가 연락되는지 확인하는 방법

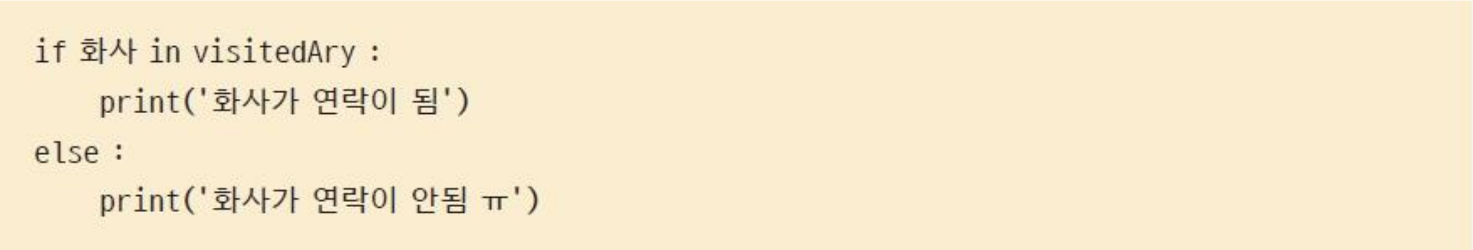
-> 비상연락망 연결이 끊긴 경우(‘화사’ 만 동떨어진 예)





```
gSize = 6
G1 = Graph(gSize)
G1.graph[문별][솔라] = 1; G1.graph[문별][휘인] = 1
G1.graph[솔라][문별] = 1; G1.graph[솔라][쯔위] = 1
G1.graph[휘인][문별] = 1; G1.graph[휘인][쯔위] = 1
G1.graph[쯔위][솔라] = 1; G1.graph[쯔위][휘인] = 1; G1.graph[쯔위][선미] = 1
G1.graph[선미][쯔위] = 1;
```

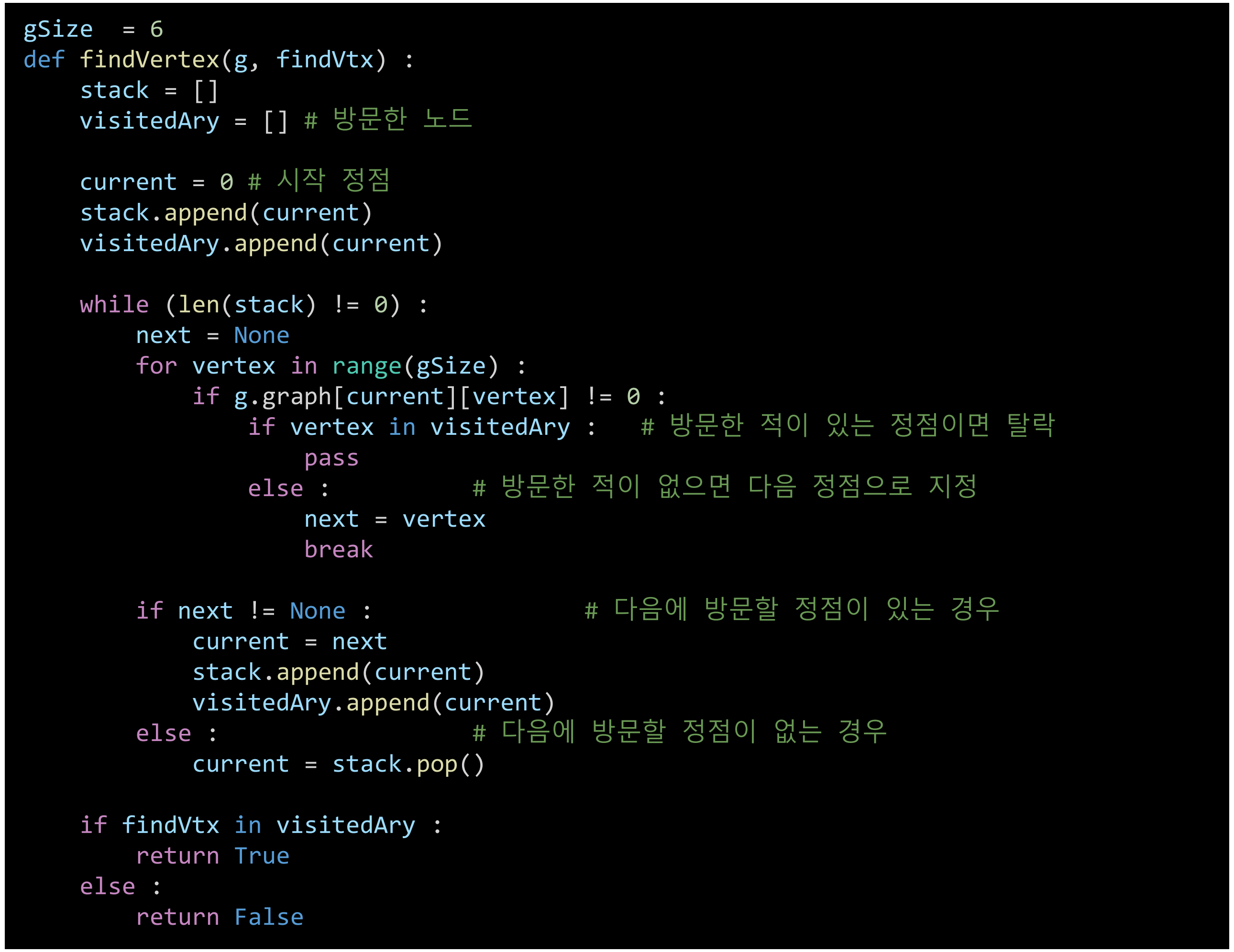
앞의 그래프 깊이 탐색 코드를 활용하여 탐색한 후
순회 결과인 visitedAry에 ‘화사’가 들어 있는지 확인하면 그래프에 연결된 상태인지 알 수 있음



```
if 화사 in visitedAry :
    print('화사가 연락이 됨')
else :
    print('화사가 연락이 안됨 ㅠ')
```

그래프 응용 / 친구가 연락되는지 확인

특정 정점이 그래프에 연결되어 있는지 확인하는 함수 (이후 활용될 코드이므로 입력해둠)

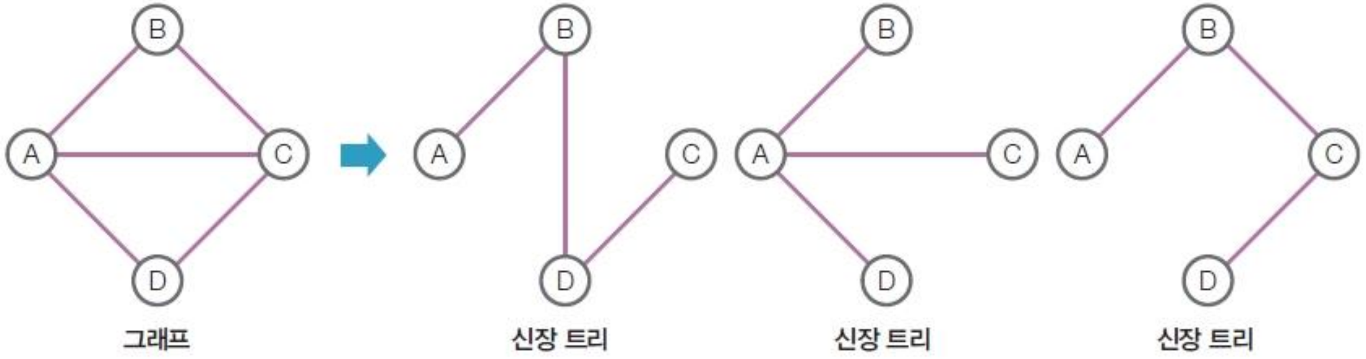


```python
gSize  = 6
def findVertex(g, findVtx) :
    stack = []
    visitedAry = [] # 방문한 노드

    current = 0 # 시작 정점
    stack.append(current)
    visitedAry.append(current)

    while (len(stack) != 0) :
        next = None
        for vertex in range(gSize) :
            if g.graph[current][vertex] != 0 :
                if vertex in visitedAry :   # 방문한 적이 있는 정점이면 탈락
                    pass
                else :          # 방문한 적이 없으면 다음 정점으로 지정
                    next = vertex
                    break

        if next != None :               # 다음에 방문할 정점이 있는 경우
            current = next
            stack.append(current)
            visitedAry.append(current)
        else :                  # 다음에 방문할 정점이 없는 경우
            current = stack.pop()

    if findVtx in visitedAry :
        return True
    else :
        return False
```

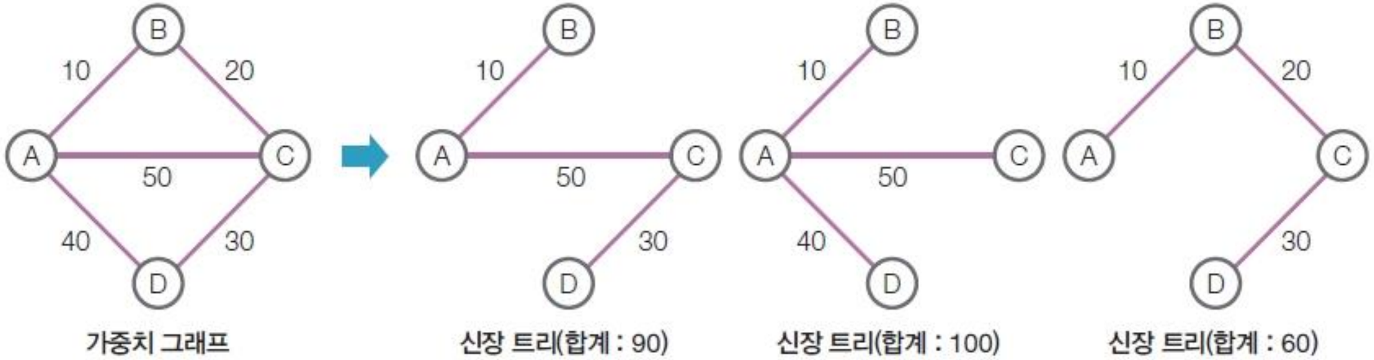
그래프 응용 / 최소 신장 트리

최소 비용으로 자전거 도로 연결

-> 신장 트리(Spanning Tree)는 최소 간선으로 그래프의 모든 정점이 연결되는 그래프



-> 가중치 그래프와 신장 트리



↑ 최소 비용 신장 트리

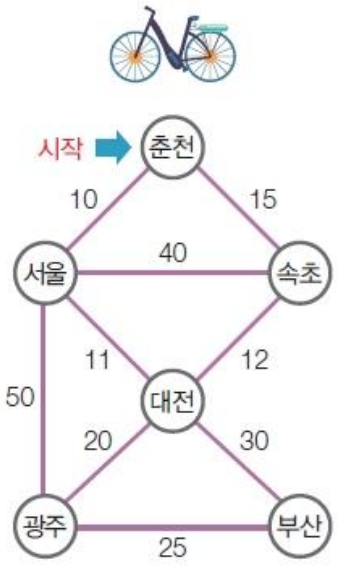
그래프 응용 / 최소 신장 트리

최소 비용으로 자전거 도로 연결

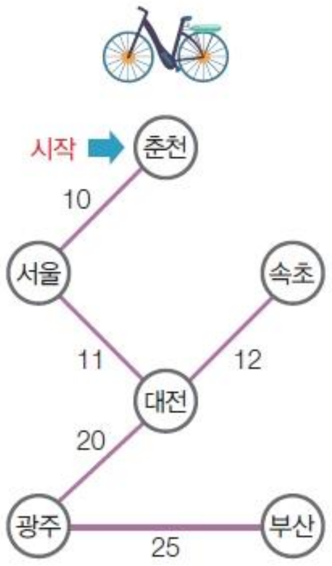
-> 최소 비용 신장 트리는 가중치 그래프에서 만들 수 있는 신장 트리 중 합계가 최소인 것

-> 구현하는 방법은 프림(Prim) 알고리즘, 크루스컬(Kruskal) 알고리즘 등이 있음

- 최소 비용 신장 트리를 활용하여 자전거 도로를 최소 비용으로 연결하는 예(크루스컬 알고리즘을 활용)



↑ 그래프에 가중치를 추가한 형태

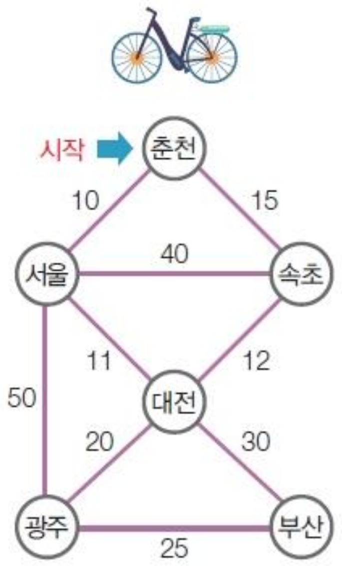


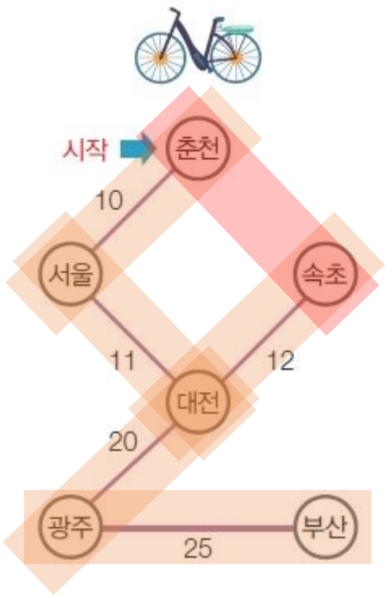
↑ 최소 비용 신장 트리로 구성한 상태

그래프 응용 / 최소 신장 트리

최소 비용으로 자전거 도로 연결

-> 크루스컬 알고리즘 : 가중치를 정렬한 후 간선을 선택해 나가는 방법





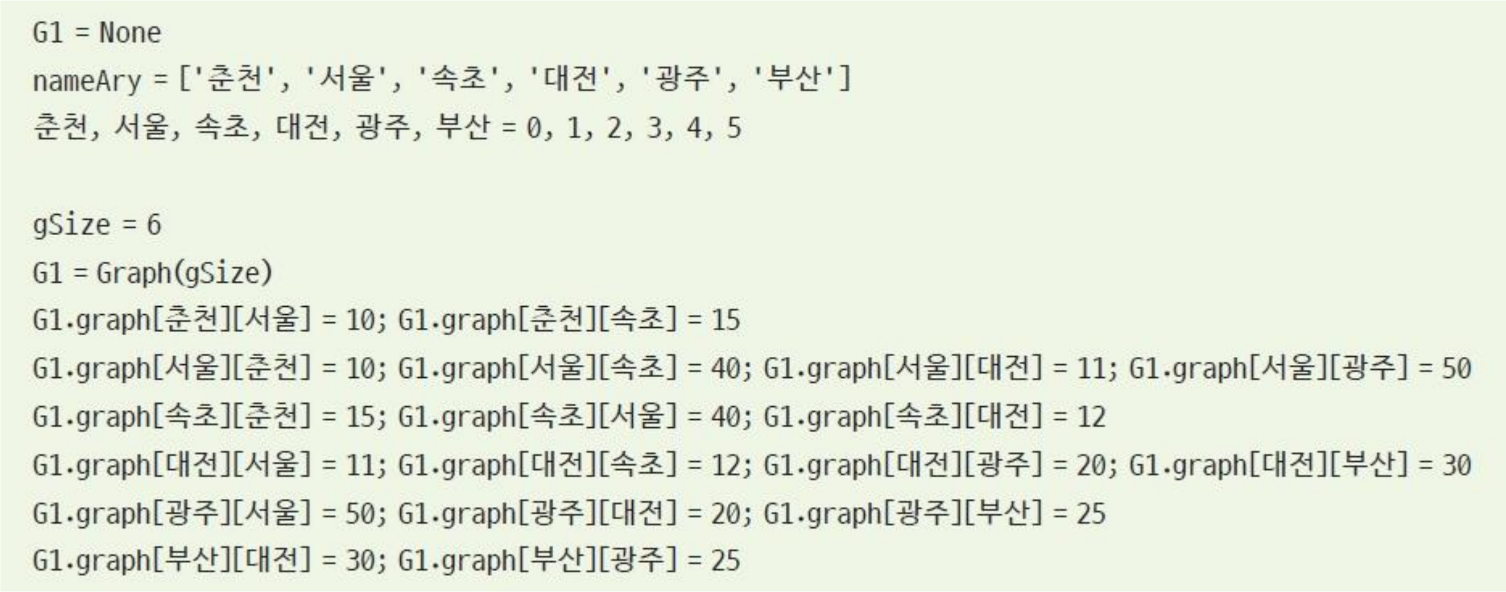
10 : 서울, 춘천
11 : 서울, 대전
12 : 대전, 속초
15 : 춘천, 속초 사이클 발생, 포함하지 않음
20 : 대전, 광주
25 : 광주, 부산
30 : 대전, 부산 신장트리가 만들어 졌으므로 아래 생략
40 : 서울, 속초
50 : 서울, 광주

그래프 응용 / 최소 신장 트리

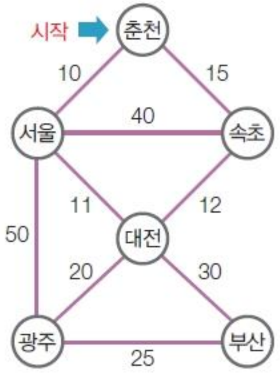
최소 비용으로 자전거 도로 연결

-> **구현 실습은 정점 간 연결이 끊어지지 않는 상태를 유지하며 최대 비용의 간선들을 제거하는 방법** 사용

-> 전체 비용이 나와 있는 가중치 그래프 구현 예



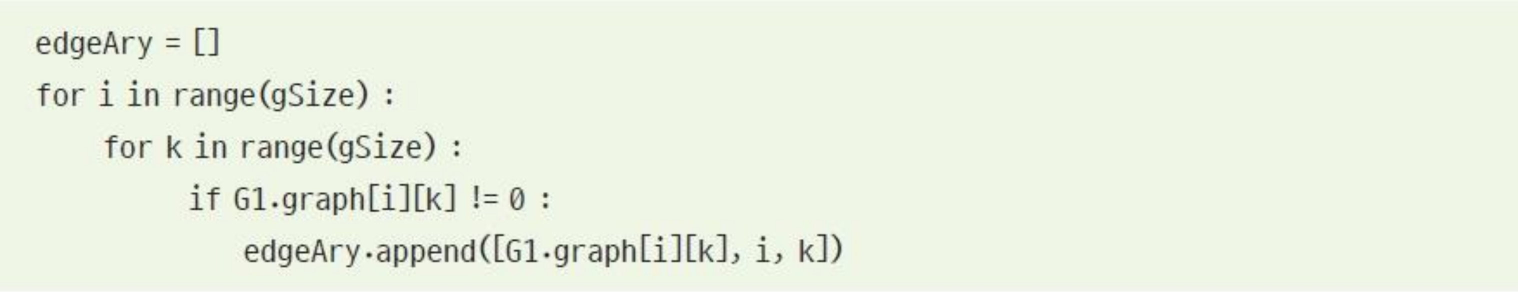
```
G1 = None
nameAry = ['춘천', '서울', '속초', '대전', '광주', '부산']
춘천, 서울, 속초, 대전, 광주, 부산 = 0, 1, 2, 3, 4, 5

gSize = 6
G1 = Graph(gSize)
G1.graph[춘천][서울] = 10; G1.graph[춘천][속초] = 15
G1.graph[서울][춘천] = 10; G1.graph[서울][속초] = 40; G1.graph[서울][대전] = 11; G1.graph[서울][광주] = 50
G1.graph[속초][춘천] = 15; G1.graph[속초][서울] = 40; G1.graph[속초][대전] = 12
G1.graph[대전][서울] = 11; G1.graph[대전][속초] = 12; G1.graph[대전][광주] = 20; G1.graph[대전][부산] = 30
G1.graph[광주][서울] = 50; G1.graph[광주][대전] = 20; G1.graph[광주][부산] = 25
G1.graph[부산][대전] = 30; G1.graph[부산][광주] = 25
```



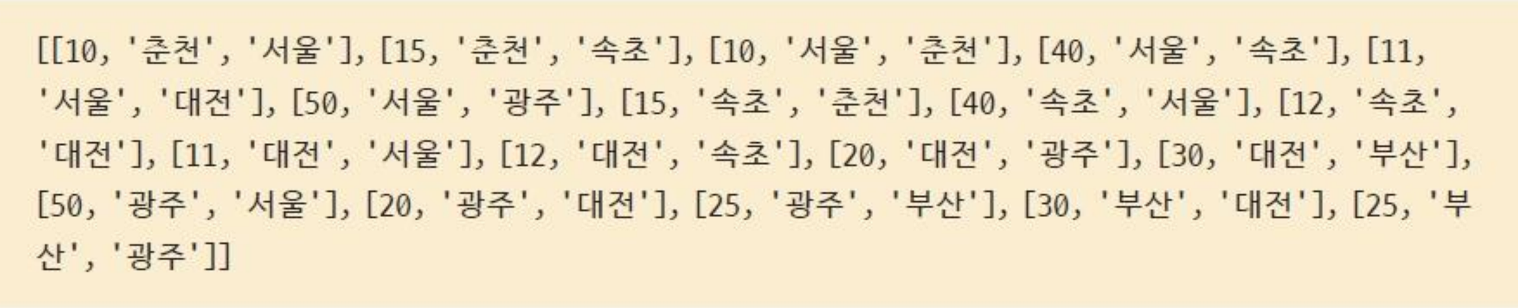
	춘천	서울	속초	대전	광주	부산
춘천	0	10	15	0	0	0
서울	10	0	40	11	50	0
속초	15	40	0	12	0	0
대전	0	11	12	0	20	30
광주	0	50	0	20	0	25
부산	0	0	0	30	25	0

최소 비용으로 자전거 도로 연결

-> 가중치와 간선 목록 생성



```
edgeAry = []
for i in range(gSize) :
    for k in range(gSize) :
        if G1.graph[i][k] != 0 :
            edgeAry.append([G1.graph[i][k], i, k])
```



```
[[10, '춘천', '서울'], [15, '춘천', '속초'], [10, '서울', '춘천'], [40, '서울', '속초'], [11,
'서울', '대전'], [50, '서울', '광주'], [15, '속초', '춘천'], [40, '속초', '서울'], [12, '속초',
'대전'], [11, '대전', '서울'], [12, '대전', '속초'], [20, '대전', '광주'], [30, '대전', '부산'],
[50, '광주', '서울'], [20, '광주', '대전'], [25, '광주', '부산'], [30, '부산', '대전'], [25, '부
산', '광주']]
```

생성한 6 x 6 행렬에서 가중치가 있는 정점들에 대한 데이터 모두 생성

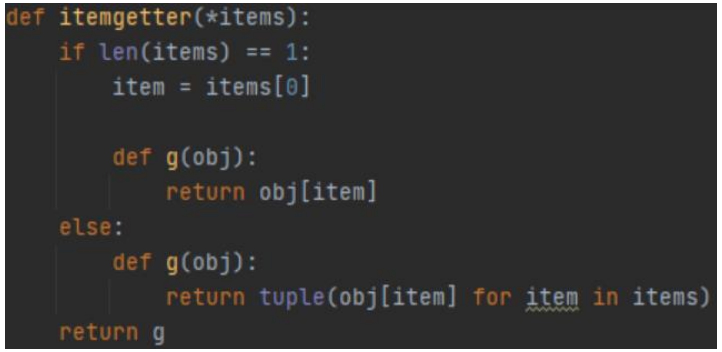
그래프 응용 / 최소 신장 트리

최소 비용으로 자전거 도로 연결

-> 가중치를 기준으로 내림차순으로 간선 정렬

```
from operator import itemgetter
edgeAry = sorted(edgeAry, key=itemgetter(0), reverse=True)
```

```
[[50, '서울', '광주'], [50, '광주', '서울'], [40, '서울', '속초'], [40, '속초', '서울'], [12, '속
초', '대전'], [12, '대전', '속초'], [25, '광주', '부산'], [25, '부산', '광주'], [15, '춘천', '속
초'], [15, '속초', '춘천'], [20, '대전', '광주'], [20, '광주', '대전'], [30, '대전', '부산'],
[30, '부산', '대전'], [11, '서울', '대전'], [11, '대전', '서울'], [10, '춘천', '서울'], [10, '서
울', '춘천']]
```



```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]

        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

itemgetter() 함수는 정렬하고자 하는 기준 데이터를 설정함
reverse=True 는 내림차순 정렬
itemgetter로 정렬 기준을 여러 개 지정 가능

그래프 응용 / 최소 신장 트리

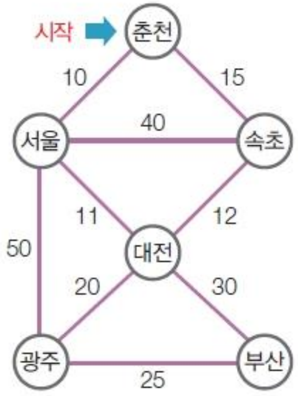
최소 비용으로 자전거 도로 연결

-> 중복 간선 제거 (중복된 데이터가 연속으로 있으므로, 인덱스를 2씩 건너뛰며 데이터 추가)

```
newAry = []
for i in range(0, len(edgeAry), 2) :
    newAry.append(edgeAry[i])
```

```
[[50, '서울', '광주'], [40, '서울', '속초'], [30, '대전', '부산'], [25, '광주', '부산'], [20,
'대전', '광주'], [15, '춘천', '속초'], [12, '속초', '대전'], [11, '서울', '대전'], [10, '춘천',
'서울']]
```

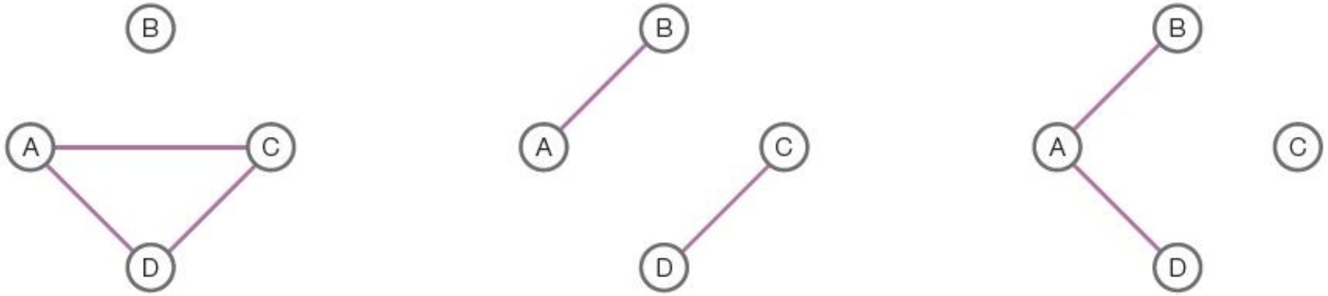
가중치	간선
50	서울 – 광주
40	서울 – 속초
30	대전 – 부산
25	광주 – 부산
20	대전 – 광주
15	춘천 – 속초
12	속초 – 대전
11	서울 – 대전
10	춘천 – 서울



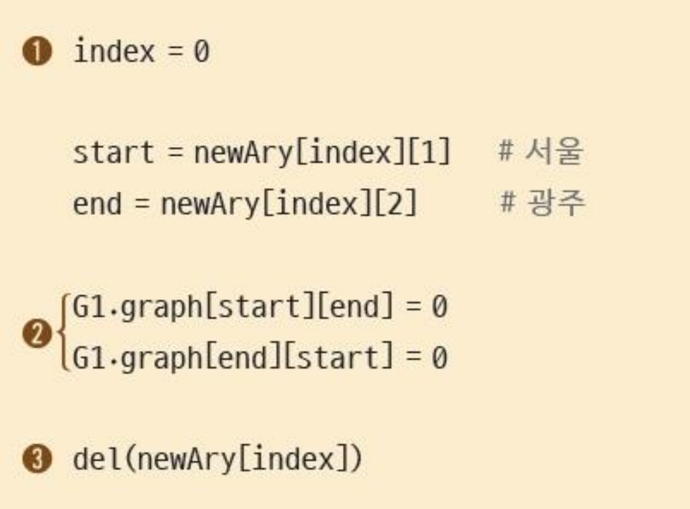
그래프 응용 / 최소 신장 트리

최소 비용으로 자전거 도로 연결

-> 가중치가 높은 간선부터 제거하며 도시가 연결되지 않는 경우는 허용하지 않음



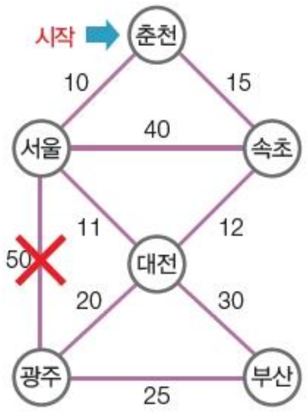
1) 서울-광주 간선 제거 - 최소 신장 트리의 간선 개수는 [정점 개수 - 1] (=5개) 임



```
❶ index = 0

   start = newAry[index][1]    # 서울
   end = newAry[index][2]      # 광주

❷ {G1.graph[start][end] = 0
   G1.graph[end][start] = 0

❸ del(newAry[index])
```

가중치 배열(newAry)

	가중치	간선
0	~~50~~	~~서울 – 광주~~
1	40	서울 – 속초
2	30	대전 – 부산
3	25	광주 – 부산
4	20	대전 – 광주
5	15	춘천 – 속초
6	12	속초 – 대전
7	11	서울 – 대전
8	10	춘천 – 서울



그래프 응용 / 최소 신장 트리

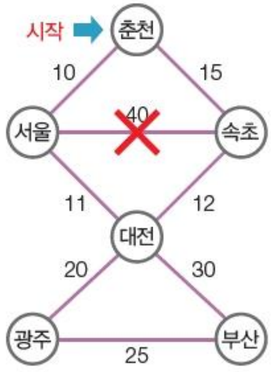
최소 비용으로 자전거 도로 연결

2) 서울-속초 간선 제거

```
❶ index = 0

   start = newAry[index][1]    # 서울
   end = newAry[index][2]      # 속초

❷ {G1.graph[start][end] = 0
   G1.graph[end][start] = 0

❸ del(newAry[index])
```

가중치 배열(newAry)

	가중치	간선
	~~50~~	~~서울 – 광주~~
index ➡ 0	~~40~~	~~서울 – 속초~~
1	30	대전 – 부산
2	25	광주 – 부산
3	20	대전 – 광주
4	15	춘천 – 속초
5	12	속초 – 대전
6	11	서울 – 대전
7	10	춘천 – 서울

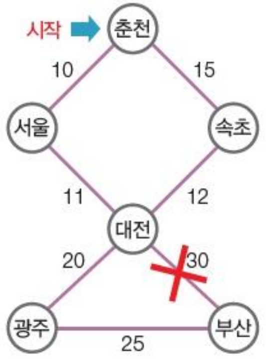


3) 대전-부산 간선 제거

```
❶ index = 0

   start = newAry[index][1]    # 대전
   end = newAry[index][2]      # 부산

❷ {G1.graph[start][end] = 0
   G1.graph[end][start] = 0

❸ del(newAry[index])
```

가중치 배열(newAry)

	가중치	간선
	~~50~~	~~서울 – 광주~~
	~~40~~	~~서울 – 속초~~
index ➡ 0	~~30~~	~~대전 – 부산~~
1	25	광주 – 부산
2	20	대전 – 광주
3	15	춘천 – 속초
4	12	속초 – 대전
5	11	서울 – 대전
6	10	춘천 – 서울



그래프 응용 / 최소 신장 트리

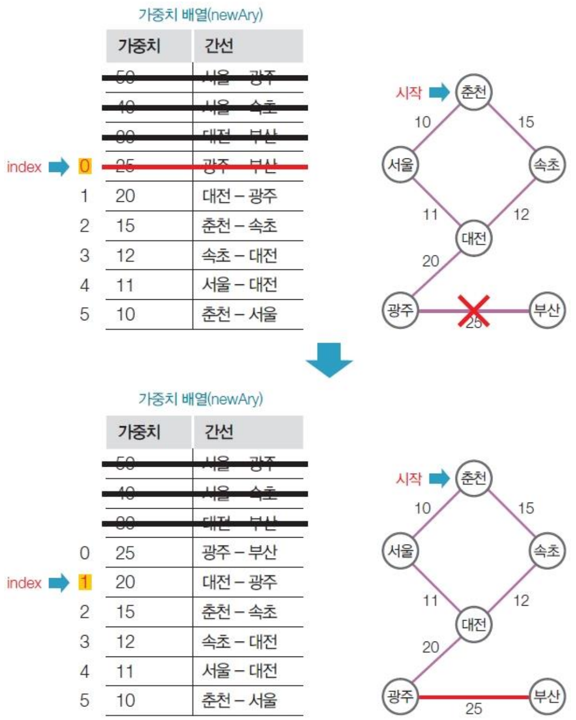
최소 비용으로 자전거 도로 연결

4) 광주-부산 간선의 제거는 고립된 정점을 만들어냄

```
❶ index = 0

  start = newAry[index][1]        # 광주
  end = newAry[index][2] # 부산
❷ saveCost = newAry[index][0]

❸ G1.graph[start][end] = 0
  G1.graph[end][start] = 0

❹ startYN = findVertex(G1, start)
  endYN = findVertex(G1, end)

  if startYN and endYN : # 두 정점 모두 그래프와 연결되어 있다면
      del(newAry[index]) # 가중치 배열에서 완전히 제거
❺ else :
      G1.graph[start][end] = saveCost
      G1.graph[end][start] = saveCost
❻ index += 1
```

이 간선은 유지해야 하므로 index를 증가시켜줌

앞서 제거한 간선들도
고립된 정점이 만들어지는지
검사하며 제거를 해야함



가중치 배열(newAry)

	가중치	간선
	~~50~~	~~서울 – 광주~~
	~~40~~	~~서울 – 속초~~
	~~30~~	~~대전 – 부산~~
index ➡ 0	~~25~~	~~광주 – 부산~~
1	20	대전 – 광주
2	15	춘천 – 속초
3	12	속초 – 대전
4	11	서울 – 대전
5	10	춘천 – 서울

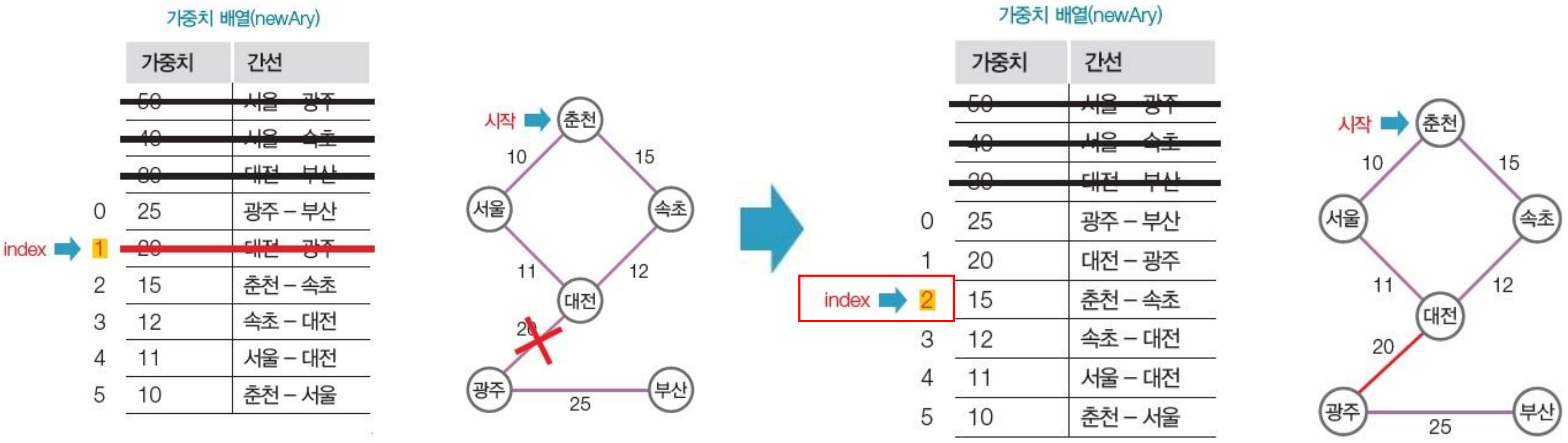
가중치 배열(newAry)

	가중치	간선
	~~50~~	~~서울 – 광주~~
	~~40~~	~~서울 – 속초~~
	~~30~~	~~대전 – 부산~~
0	25	광주 – 부산
index ➡ 1	20	대전 – 광주
2	15	춘천 – 속초
3	12	속초 – 대전
4	11	서울 – 대전
5	10	춘천 – 서울

그래프 응용 / 최소 신장 트리

최소 비용으로 자전거 도로 연결

5) 대전-광주 간선의 제거 시도와 원상 복구 : 대전 광주 간선을 제거하면 고립된 정점이 발생



가중치 배열(newAry)

	가중치	간선
	~~50~~	~~서울 – 광주~~
	~~40~~	~~서울 – 속초~~
	~~30~~	~~대전 – 부산~~
0	25	광주 – 부산
index 1	~~20~~	~~대전 – 광주~~
2	15	춘천 – 속초
3	12	속초 – 대전
4	11	서울 – 대전
5	10	춘천 – 서울

가중치 배열(newAry)

	가중치	간선
	~~50~~	~~서울 – 광주~~
	~~40~~	~~서울 – 속초~~
	~~30~~	~~대전 – 부산~~
0	25	광주 – 부산
1	20	대전 – 광주
index 2	15	춘천 – 속초
3	12	속초 – 대전
4	11	서울 – 대전
5	10	춘천 – 서울

그래프 응용 / 최소 신장 트리

최소 비용으로 자전거 도로 연결

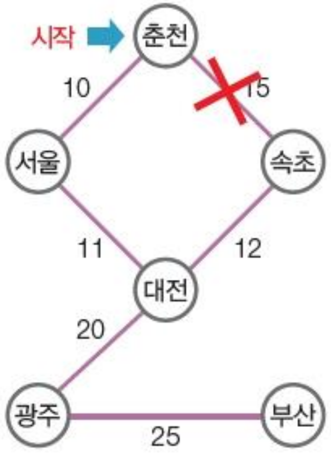
6) 춘천-속초 간선 제거 : 제거가 가능하므로 제거해 줌

-> 이 간선을 제거하는 순간 간선의 수가 5개가 되므로 신장 트리가 완성되었음

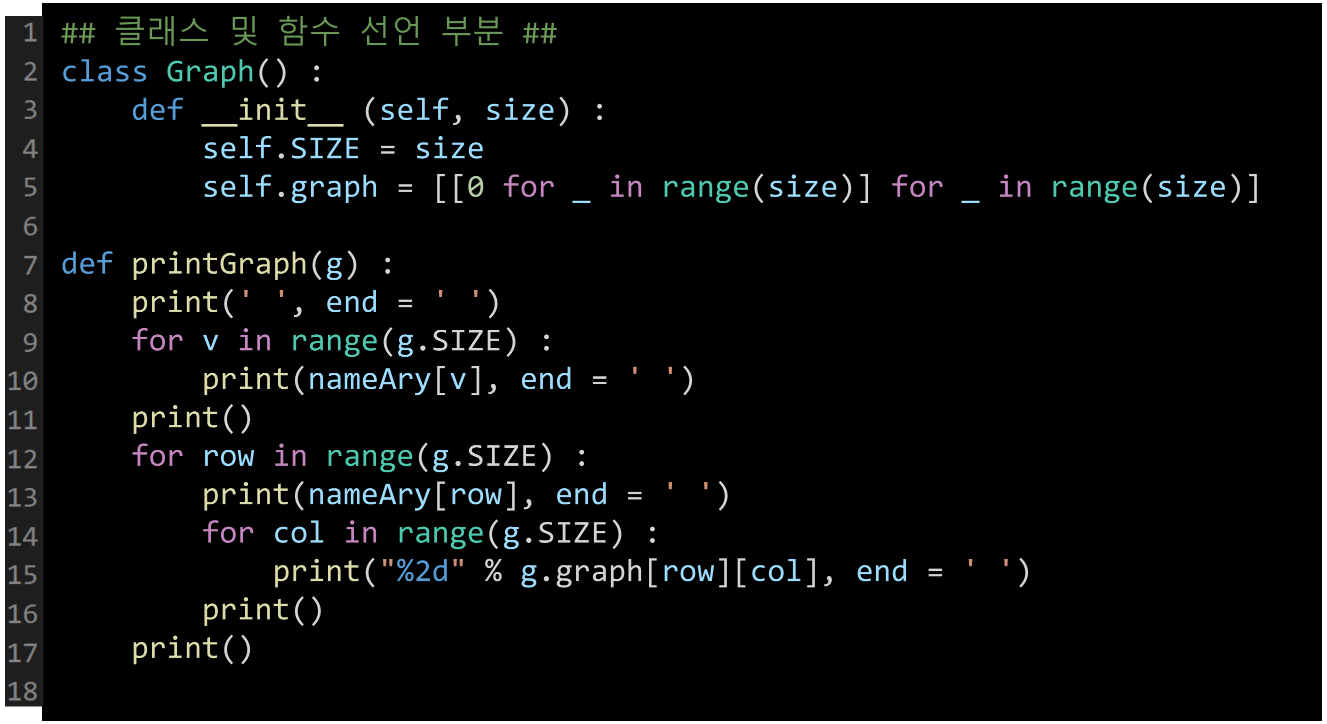
-> 나머지 간선은 검사하지 않음

가중치 배열(newAry)

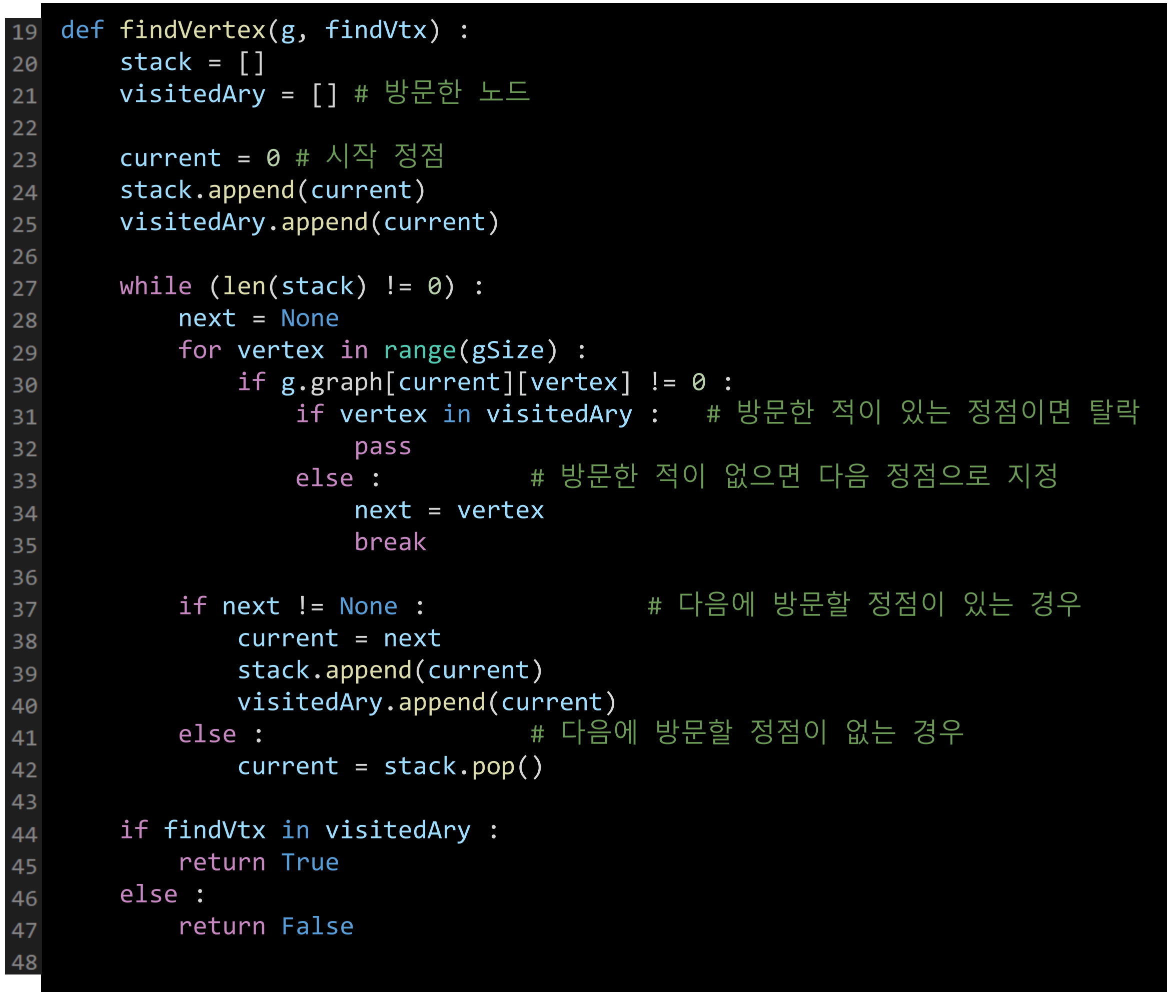
	가중치	간선
	~~50~~	~~서울 – 광주~~
	~~40~~	~~서울 – 속초~~
	~~30~~	~~대전 – 부산~~
0	25	광주 – 부산
1	20	대전 – 광주
index ➡ 2	~~15~~	~~춘천 – 속초~~
3	12	속초 – 대전
4	11	서울 – 대전
5	10	춘천 – 서울



최소 비용으로 자전거 도로 연결 코드 1 (기존 코드 동일)



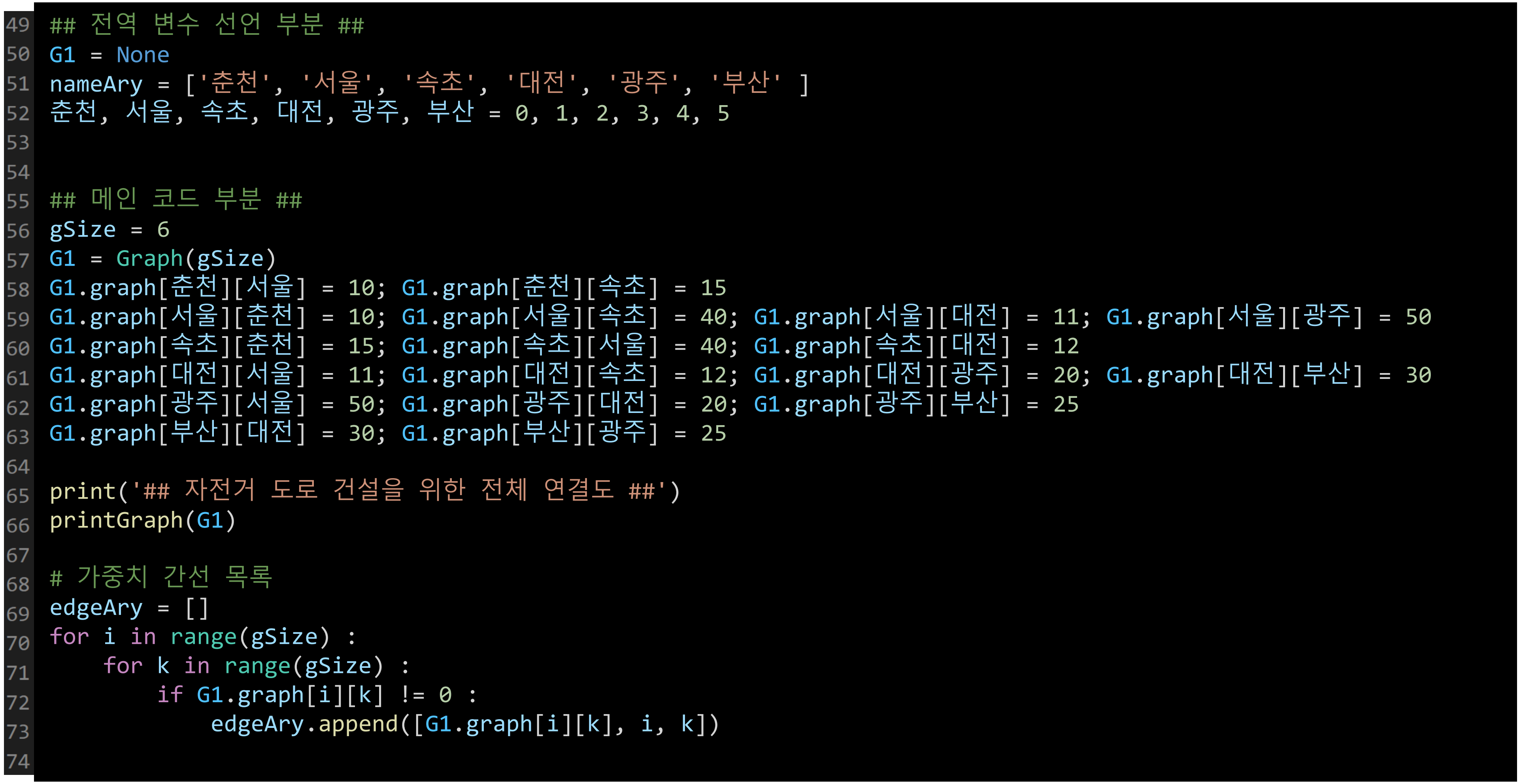
```python
## 클래스 및 함수 선언 부분 ##
class Graph() :
    def __init__ (self, size) :
        self.SIZE = size
        self.graph = [[0 for _ in range(size)] for _ in range(size)]

def printGraph(g) :
    print(' ', end = ' ')
    for v in range(g.SIZE) :
        print(nameAry[v], end = ' ')
    print()
    for row in range(g.SIZE) :
        print(nameAry[row], end = ' ')
        for col in range(g.SIZE) :
            print("%2d" % g.graph[row][col], end = ' ')
        print()
    print()

```



```python
def findVertex(g, findVtx) :
    stack = []
    visitedAry = [] # 방문한 노드

    current = 0 # 시작 정점
    stack.append(current)
    visitedAry.append(current)

    while (len(stack) != 0) :
        next = None
        for vertex in range(gSize) :
            if g.graph[current][vertex] != 0 :
                if vertex in visitedAry :   # 방문한 적이 있는 정점이면 탈락
                    pass
                else :          # 방문한 적이 없으면 다음 정점으로 지정
                    next = vertex
                    break

        if next != None :               # 다음에 방문할 정점이 있는 경우
            current = next
            stack.append(current)
            visitedAry.append(current)
        else :                  # 다음에 방문할 정점이 없는 경우
            current = stack.pop()

    if findVtx in visitedAry :
        return True
    else :
        return False

```

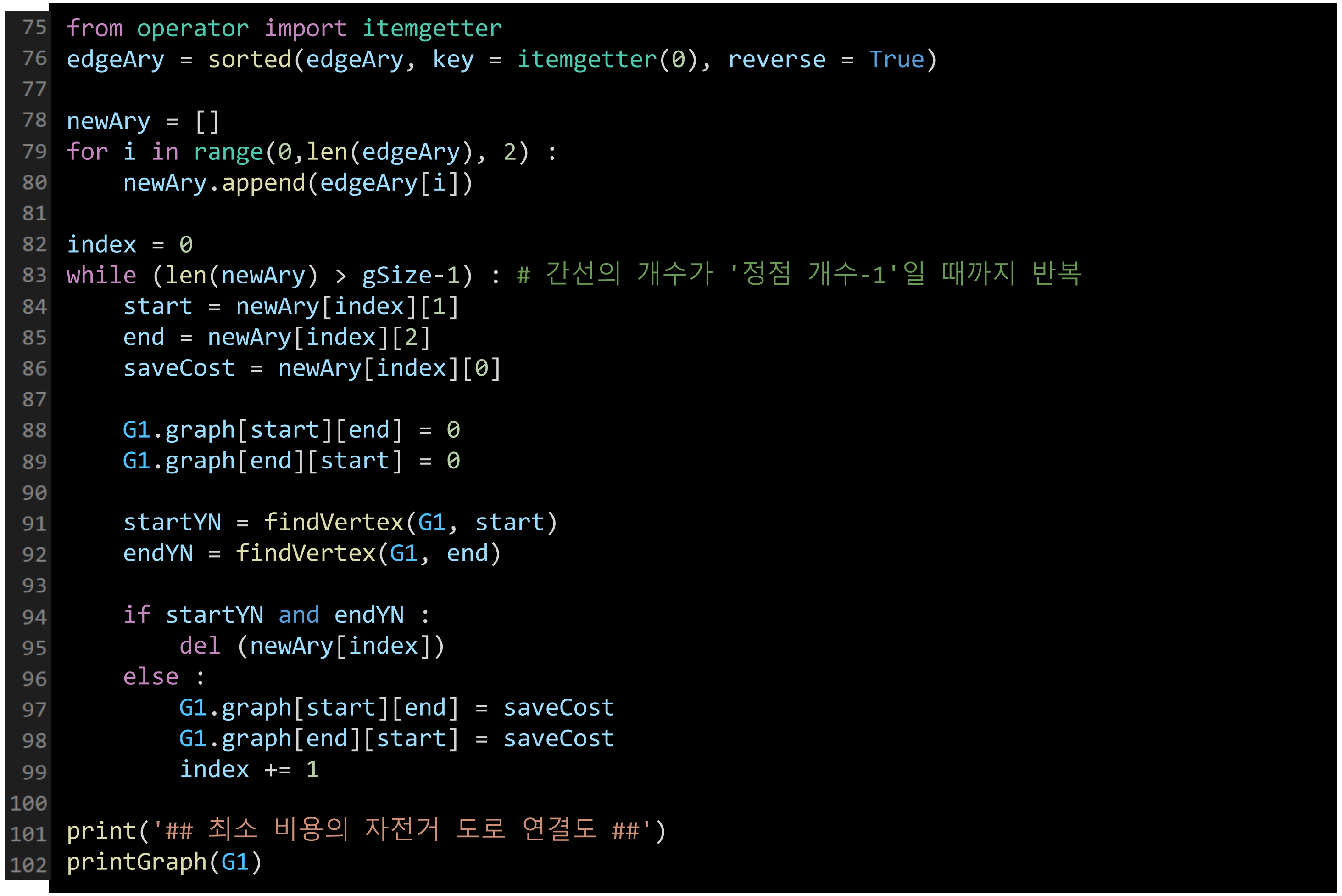
그래프 응용 / 최소 신장 트리

최소 비용으로 자전거 도로 연결 코드 2



```python
## 전역 변수 선언 부분 ##
G1 = None
nameAry = ['춘천', '서울', '속초', '대전', '광주', '부산' ]
춘천, 서울, 속초, 대전, 광주, 부산 = 0, 1, 2, 3, 4, 5


## 메인 코드 부분 ##
gSize = 6
G1 = Graph(gSize)
G1.graph[춘천][서울] = 10; G1.graph[춘천][속초] = 15
G1.graph[서울][춘천] = 10; G1.graph[서울][속초] = 40; G1.graph[서울][대전] = 11; G1.graph[서울][광주] = 50
G1.graph[속초][춘천] = 15; G1.graph[속초][서울] = 40; G1.graph[속초][대전] = 12
G1.graph[대전][서울] = 11; G1.graph[대전][속초] = 12; G1.graph[대전][광주] = 20; G1.graph[대전][부산] = 30
G1.graph[광주][서울] = 50; G1.graph[광주][대전] = 20; G1.graph[광주][부산] = 25
G1.graph[부산][대전] = 30; G1.graph[부산][광주] = 25

print('## 자전거 도로 건설을 위한 전체 연결도 ##')
printGraph(G1)

# 가중치 간선 목록
edgeAry = []
for i in range(gSize) :
    for k in range(gSize) :
        if G1.graph[i][k] != 0 :
            edgeAry.append([G1.graph[i][k], i, k])

```

최소 비용으로 자전거 도로 연결 코드 3



```python
from operator import itemgetter
edgeAry = sorted(edgeAry, key = itemgetter(0), reverse = True)

newAry = []
for i in range(0,len(edgeAry), 2) :
    newAry.append(edgeAry[i])

index = 0
while (len(newAry) > gSize-1) : # 간선의 개수가 '정점 개수-1'일 때까지 반복
    start = newAry[index][1]
    end = newAry[index][2]
    saveCost = newAry[index][0]

    G1.graph[start][end] = 0
    G1.graph[end][start] = 0

    startYN = findVertex(G1, start)
    endYN = findVertex(G1, end)

    if startYN and endYN :
        del (newAry[index])
    else :
        G1.graph[start][end] = saveCost
        G1.graph[end][start] = saveCost
        index += 1

print('## 최소 비용의 자전거 도로 연결도 ##')
printGraph(G1)
```

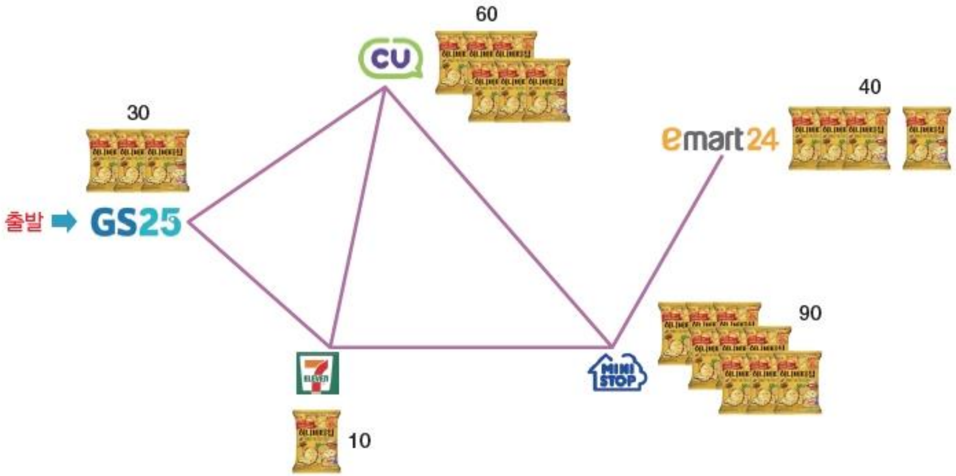
실행 결과

```
## 자전거 도로 건설을 위한 전체 연결도 ##
  춘천 서울 속초 대전 광주 부산
춘천  0 10 15  0  0  0
서울 10  0 40 11 50  0
속초 15 40  0 12  0  0
대전  0 11 12  0 20 30
광주  0 50  0 20  0 25
부산  0  0  0 30 25  0

## 최소 비용의 자전거 도로 연결도 ##
  춘천 서울 속초 대전 광주 부산
춘천  0 10  0  0  0  0
서울 10  0  0 11  0  0
속초  0  0  0 12  0  0
대전  0 11 12  0 20  0
광주  0  0  0 20  0 25
부산  0  0  0  0 25  0
```

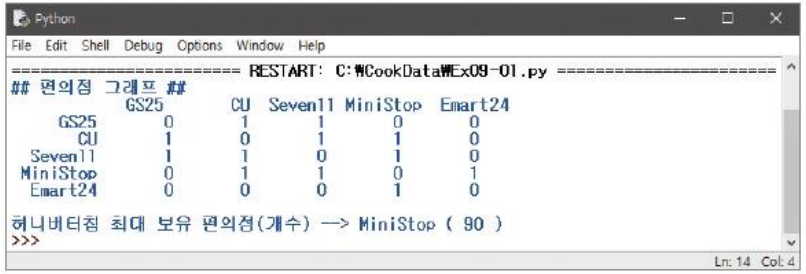
그래프 응용 / 응용 예제 1

허니버터칩이 가장 많이 남은 편의점 찾기

예제 설명 2014년에 출시한 허니버터칩은 한동안 상당한 인기로 구하기가 하늘의 별 따기만큼 어려울 정도였다. 우리 동네에서 허니버터칩 재고가 가장 많은 편의점을 알아내려고 한다. 편의점끼리는 서로 그래프 형태로 이어져 있다고 가정하자.



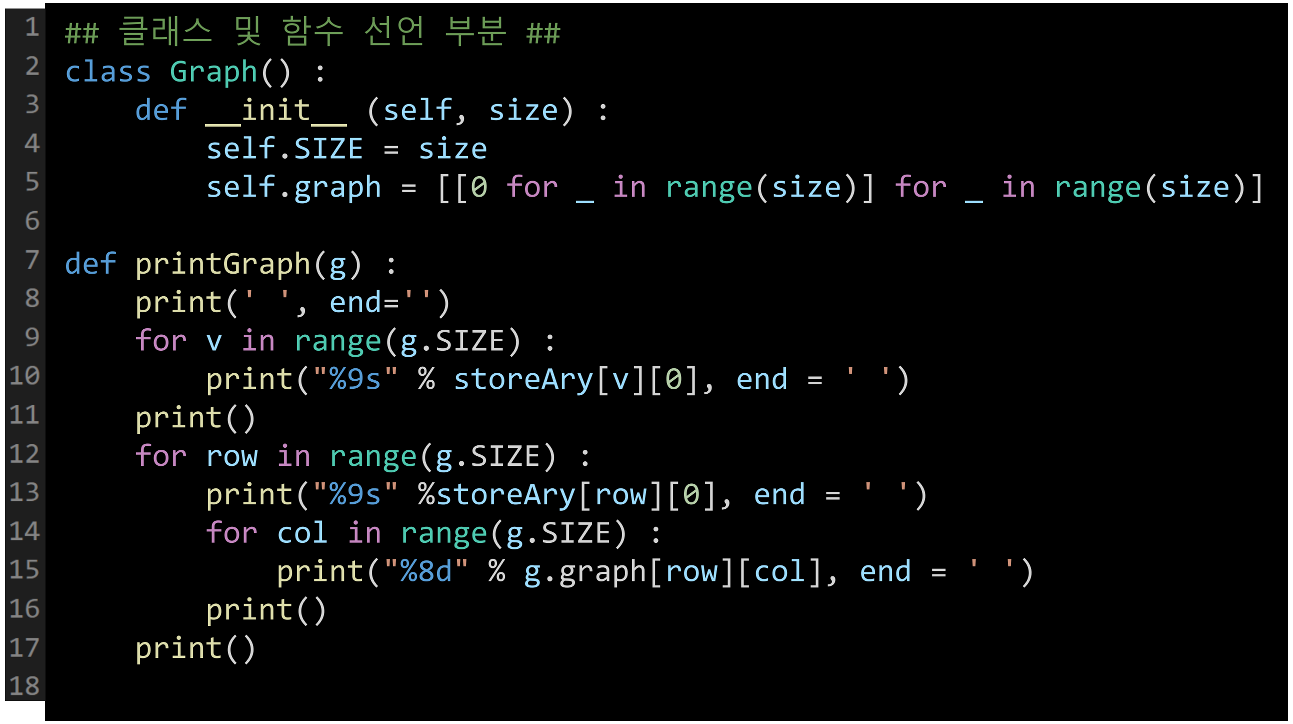
실행 결과



```
======================== RESTART: C:\CookData\Ex09-01.py ========================
## 편의점 그래프 ##
              GS25        CU  Seven11 MiniStop  Emart24
     GS25        0         1        1        0        0
       CU        1         0        1        1        0
  Seven11        1         1        0        1        0
 MiniStop        0         1        1        0        1
  Emart24        0         0        0        1        0

허니버터칩 최대 보유 편의점(개수) --> MiniStop ( 90 )
>>>
```

그래프 응용 / 응용 예제 1

허니버터칩이 가장 많이 남은 편의점 찾기 코드 1



```
## 클래스 및 함수 선언 부분 ##
class Graph() :
    def __init__ (self, size) :
        self.SIZE = size
        self.graph = [[0 for _ in range(size)] for _ in range(size)]

def printGraph(g) :
    print(' ', end='')
    for v in range(g.SIZE) :
        print("%9s" % storeAry[v][0], end = ' ')
    print()
    for row in range(g.SIZE) :
        print("%9s" %storeAry[row][0], end = ' ')
        for col in range(g.SIZE) :
            print("%8d" % g.graph[row][col], end = ' ')
        print()
    print()

```

그래프 응용 / 응용 예제 1

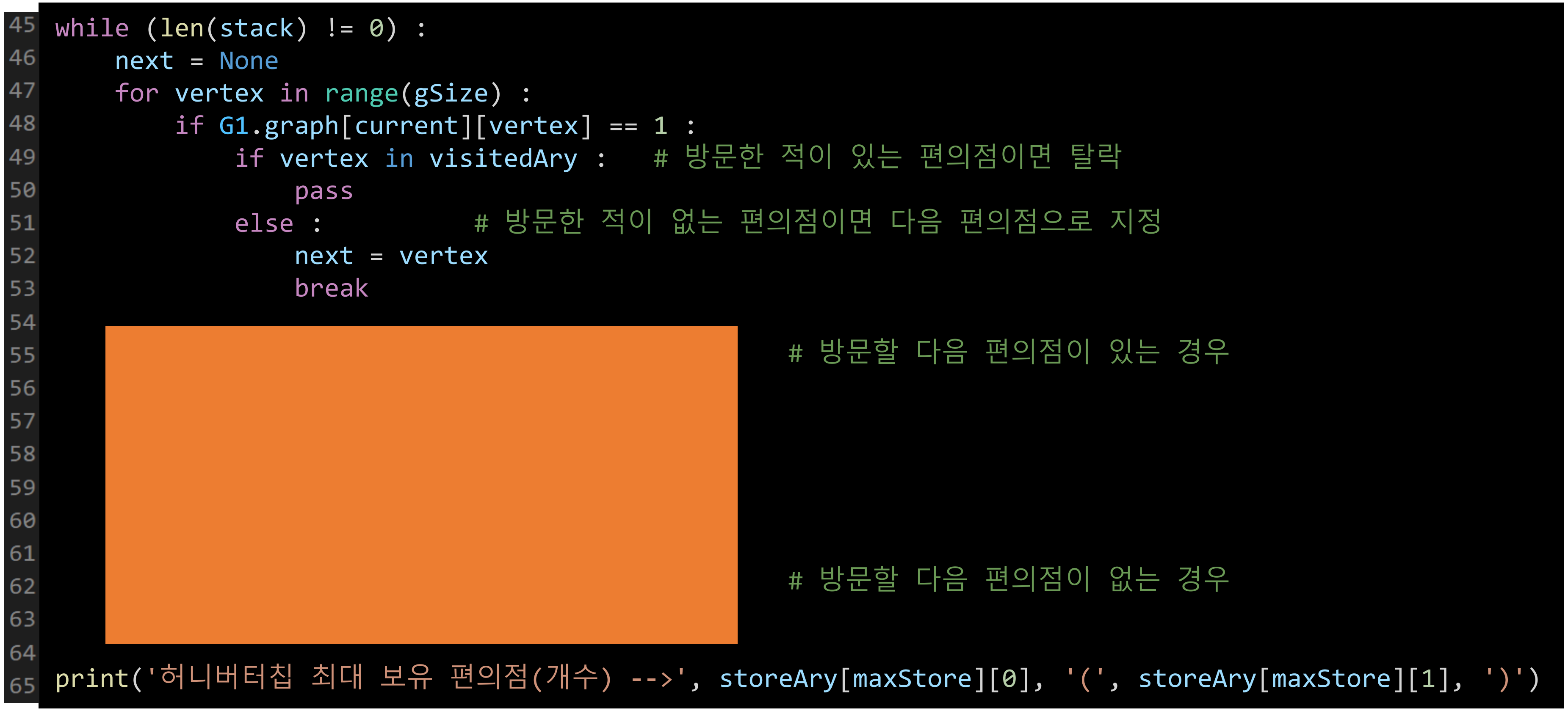
허니버터칩이 가장 많이 남은 편의점 찾기 코드 2

```
## 전역 변수 선언 부분 ##
G1 = None
storeAry = [['GS25', 30], ['CU', 60], ['Seven11', 10], ['MiniStop', 90],
['Emart24', 40]]
GS25, CU, Seven11, MiniStop, Emart24 = 0, 1, 2, 3, 4

## 메인 코드 부분 ##
gSize = 5
G1 = Graph(gSize)
[redacted]

print('## 편의점 그래프 ##')
printGraph(G1)

stack = []
visitedAry = []         # 방문한 편의점

current = 0         # 시작 편의점
maxStore = current      # 최대 개수 편의점 번호(GS25)
maxCount = storeAry[current][1] # 편의점의 허니버터 숫자
stack.append(current)
visitedAry.append(current)
```

그래프 데이터를 만드는 부분

그래프 응용 / 응용 예제 1

허니버터칩이 가장 많이 남은 편의점 찾기 코드 3



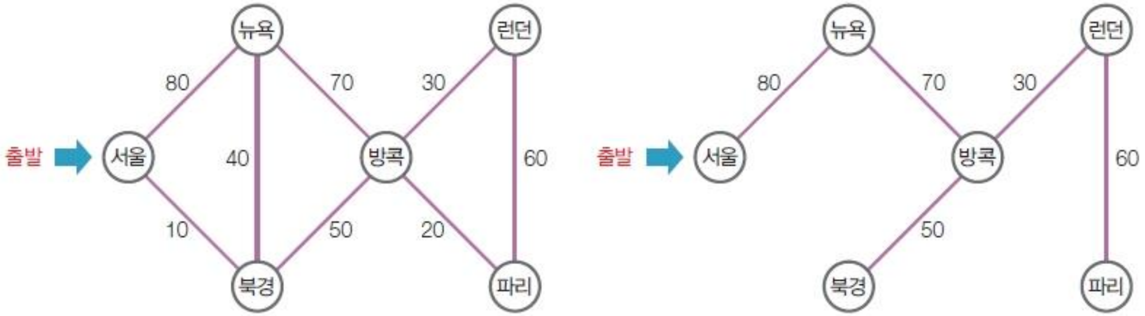
```
while (len(stack) != 0) :
    next = None
    for vertex in range(gSize) :
        if G1.graph[current][vertex] == 1 :
            if vertex in visitedAry :   # 방문한 적이 있는 편의점이면 탈락
                pass
            else :          # 방문한 적이 없는 편의점이면 다음 편의점으로 지정
                next = vertex
                break

    [redacted]                                          # 방문할 다음 편의점이 있는 경우






                                                        # 방문할 다음 편의점이 없는 경우


print('허니버터칩 최대 보유 편의점(개수) -->', storeAry[maxStore][0], '(', storeAry[maxStore][1], ')')
```

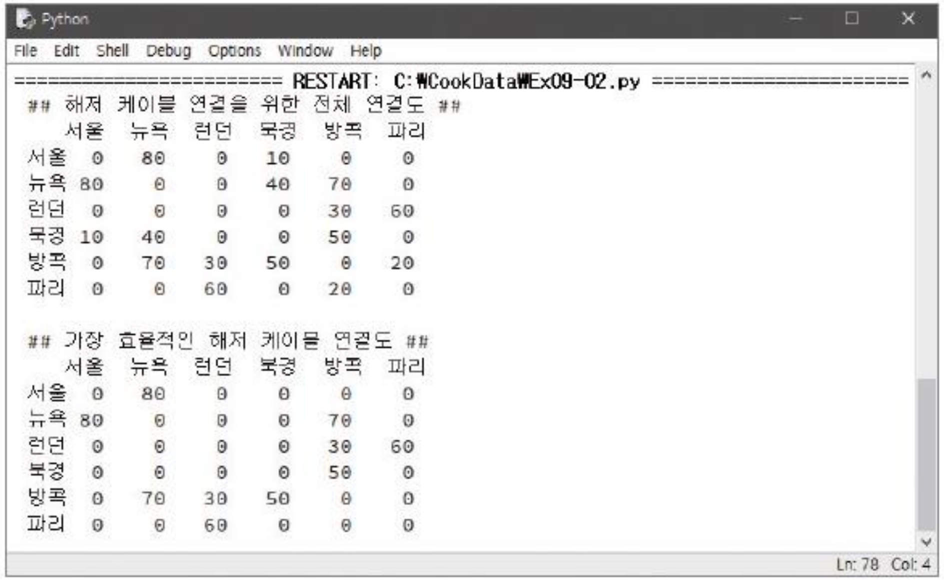
방문 가능한 노드 확인
그리고
허니버터칩 최대 개수
확인 코드

그래프 응용 / 응용 예제 2

가장 효율적인 해저 케이블망 구성하기

예제 설명 전 세계를 연결하는 해저 케이블망을 신규로 구성하고자 한다. 왼쪽 그림은 해저 케이블망 구성 전 속도를 계획한 지도다. 숫자는 네트워크 속도다. 가장 효율적인 비용으로 해저 케이블망을 구성하고자 속도가 가장 높은 연결은 남기고, 모든 도시가 연결되도록 가장 적은 개수의 연결도 남겨 놓는다. 결과는 오른쪽 그림과 같다.





```
========================= RESTART: C:₩CookData₩Ex09-02.py ========================
 ## 해저 케이블 연결을 위한 전체 연결도 ##
    서울  뉴욕  런던  북경  방콕  파리
 서울  0   80    0   10    0    0
 뉴욕 80    0    0   40   70    0
 런던  0    0    0    0   30   60
 북경 10   40    0    0   50    0
 방콕  0   70   30   50    0   20
 파리  0    0   60    0   20    0

 ## 가장 효율적인 해저 케이블 연결도 ##
    서울  뉴욕  런던  북경  방콕  파리
 서울  0   80    0    0    0    0
 뉴욕 80    0    0    0   70    0
 런던  0    0    0    0   30   60
 북경  0    0    0    0   50    0
 방콕  0   70   30   50    0    0
 파리  0    0   60    0    0    0
```

그래프 응용 / 응용 예제 2

가장 효율적인 해저 케이블망 구성하기 코드

최소 비용 자전거 도로 연결 코드를 활용하면 쉽게 해결되므로 직접 코딩

다음 강의 예고

- 재귀 호출
  - 재귀 호출 기본
  - 재귀 호출 작동 방식의 이해
  - 재귀 호출 연습
  - 재귀 호출 응용

Q&A

감사합니다.