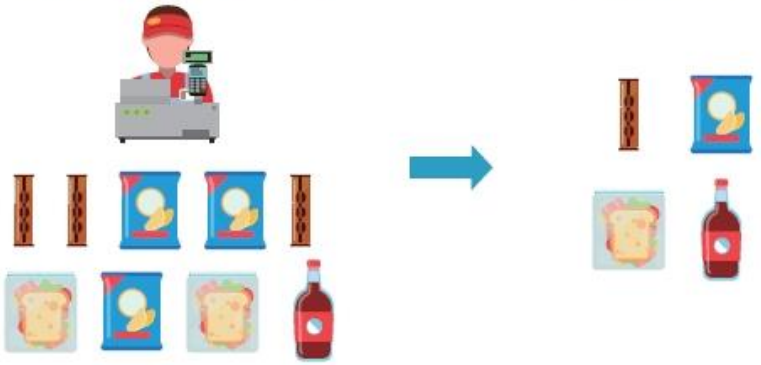
응용예제 01 편의점에서 판매된 물건 목록 출력하기

난이도★★☆☆☆

예제 설명

편의점에서는 매일 다양한 물품을 판매한다. 하루에 판매하는 물건은 당연히 중복해서 여러 개 판매한다. 마감 시간에 오늘 판매된 물건 종류를 살펴볼 때는 중복된 것은 하나만 남기도록 한다. 이진 탐색 트리를 활용해서 중복된 물품은 하나만 남기자.

편의점에서 하루 판매된 물건(중복○) 오늘 판매된 물건(중복×)



실행 결과

```
Python
File  Edit  Shell  Debug  Options  Window  Help
========== RESTART: C:\CookData\Ex08-01.py ==========
오늘 판매된 물건(중복O) --> ['레쓰비캔커피', '레쓰비캔커피', '레쓰비캔커피', '도시락', '도시락', '삼각김밥', '레쓰비캔커피', '도시락', '코카콜라',
'삼다수', '레쓰비캔커피', '레쓰비캔커피', '레쓰비캔커피', '츄파춥스', '츄파춥스', '레쓰비캔커피', '코카콜라', '츄파춥스', '삼각김밥', '코카콜라']

이진 탐색 트리 구성 완료!

오늘 판매된 종류(중복X)--> 레쓰비캔커피 도시락 삼각김밥 코카콜라 삼다수 츄파춥스
>>>
                                                                    Ln: 22  Col: 4
```

```
import random
## 함수 선언 부분 ##
class TreeNode() :
        def __init__ (self) :
                self.left = None
                self.data = None
                self.right = None

## 전역 변수 선언 부분 ##
memory = []
root = None
dataAry = ['바나나맛우유', '레쓰비캔커피', '츄파춥스', '도시락',  '삼다수', '코카콜라' ,'삼각김밥']
sellAry = [ random.choice(dataAry) for _ in  range(20)]

print('오늘 판매된 물건(중복O) -->', sellAry)

## 메인 코드 부분 ##
node = TreeNode()
node.data = sellAry[0]
root = node
memory.append(node)

for name in sellAry[1:] :

        node = TreeNode()
        node.data = name

        current = root
```

```
        while True :
                if name == current.data :
                        break
                if name < current.data :
                        if current.left == None :
                                current.left = node
                                memory.append(node)
                                break
                        current = current.left
                else :
                        if current.right == None :
                                current.right = node
                                memory.append(node)
                                break
                        current = current.right

print("이진 탐색 트리 구성 완료!")

def preorder(node) :
        if node == None :
                return
        print(node.data, end = ' ')
        preorder(node.left)
        preorder(node.right)

print('오늘 판매된 종류(중복X)--> ', end = ' ')
preorder(root)
```

응용예제 02 폴더 및 하위 폴더에 중복된 파일 이름 찾기

난이도★★★☆☆

예제 설명 특정 폴더를 지정해서 해당 폴더 및 그 하위 폴더에 모든 파일을 조회한다. 그리고 이름이 동일한 파일이 있으면 그 이름을 출력한다. 예로 C:/Program Files/Common Files/ 폴더 및 그 하위 폴더 아래에는 이름이 동일한 파일이 몇 개 있다. 단 여러 번 중복되더라도 한 번만 출력하자.

실행 결과

```
import os
## 함수 선언 부분 ##
class TreeNode() :
        def __init__ (self) :
                self.left = None
                self.data = None
                self.right = None

## 전역 변수 선언 부분 ##
memory = []
root = None
fnameAry = []

## 메인 코드 부분 ##
folderName = 'C:/Program Files/Common Files/'
for dirName, subDirList, fnames in os.walk(folderName) :
        for fname in fnames :
                fnameAry.append(fname)

node = TreeNode()
node.data = fnameAry[0]
root = node
memory.append(node)

dupNameAry = []

for name in fnameAry[1:] :

        node = TreeNode()
        node.data = name
```

```
                current = root
                while True :
                                if name == current.data :
                                                dupNameAry.append(name)
                                                break
                                if name < current.data :
                                                if current.left == None :
                                                                current.left = node
                                                                memory.append(node)
                                                                break
                                                current = current.left

                                else :
                                                if current.right == None :
                                                                current.right = node
                                                                memory.append(node)
                                                                break
                                                current = current.right

dupNameAry = list(set(dupNameAry))

print(folderName, '및 그 하위 디렉터리의 중복된 파일 목록 -->')
print(dupNameAry)
```

CHAPTER 09

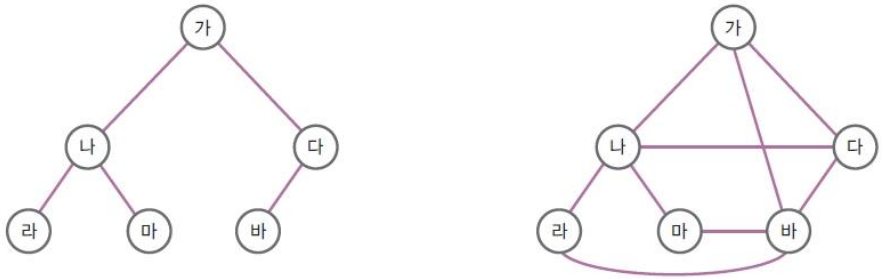
그래프

학습목표

- 그래프의 개념을 파악한다.
- 그래프를 구성하는 파이썬 코드를 작성한다.
- 그래프로 활용되는 응용 프로그램을 작성한다.

- **그래프 구조란?**
  - 버스 정류장과 여러 노선이 함께 포함된 형태 또는, 링크드인(Linked in)과 같은 사회 관계망 서비스의 연결 등의 형태

- **그래프의 개념**
  - 여러 노드가 서로 연결된 자료구조



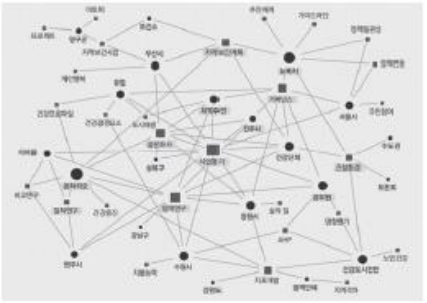
(a) 트리 예 (b) 그래프 예

그림 9-1 트리와 그래프의 차이

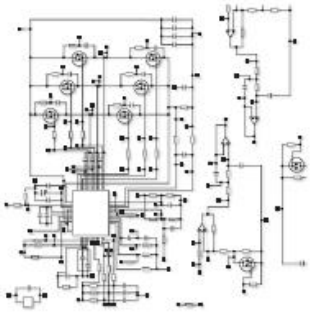
  - 그래프를 활용한 예

지하철 노선도(부산)

KTX 노선도



도시 도로망



전기 회로도

인맥 관계도

그림 9-2 그래프를 활용한 다양한 예

- **그래프의 종류 :** 무방향, 방향, 가중치
  - 무방향 그래프
    - 간선에 방향성이 없는 그래프

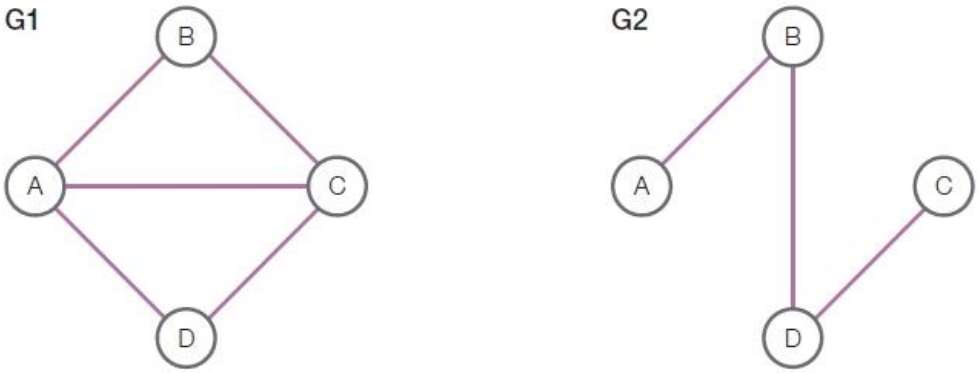


그림 9-3 무방향 그래프의 형태

- G1, G2의 정점 집합 표현

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

- G1, G2의 간선 집합 표현

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

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

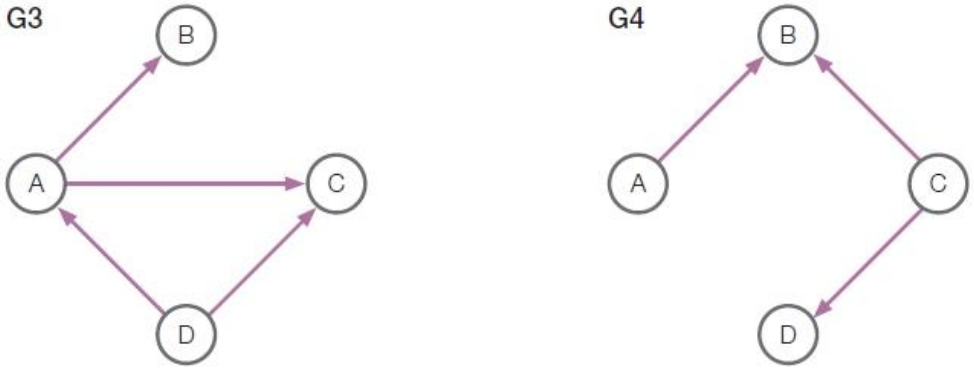


그림 9-4 방향 그래프의 형태

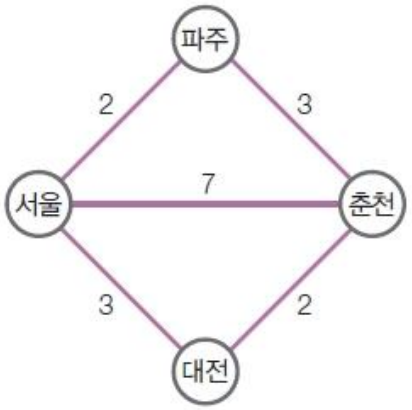
  - G3, G4의 정점 집합 표현(무방향 그래프와 같음)

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

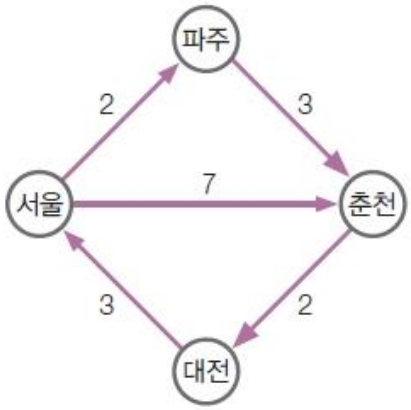
  - G3, G4의 간선 집합 표현

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

- 가중치 그래프
  - 간선마다 가중치가 다르게 부여된 그래프
  - 무방향 그래프와 방향 그래프에 각각 가중치를 부여한 경우 예



(a) 무방향 가중치 그래프



(b) 방향 가중치 그래프

그림 9-5 **가중치 그래프 예**

- **깊이 우선 탐색의 작동**
  - 그래프의 모든 정점을 한 번씩 방문하는 것을 그래프 순회(Graph Traversal)라고 함
  - 그래프 순회 방식은 깊이 우선 탐색, 너비 우선 탐색이 대표적
    - 깊이 우선 탐색 과정 예

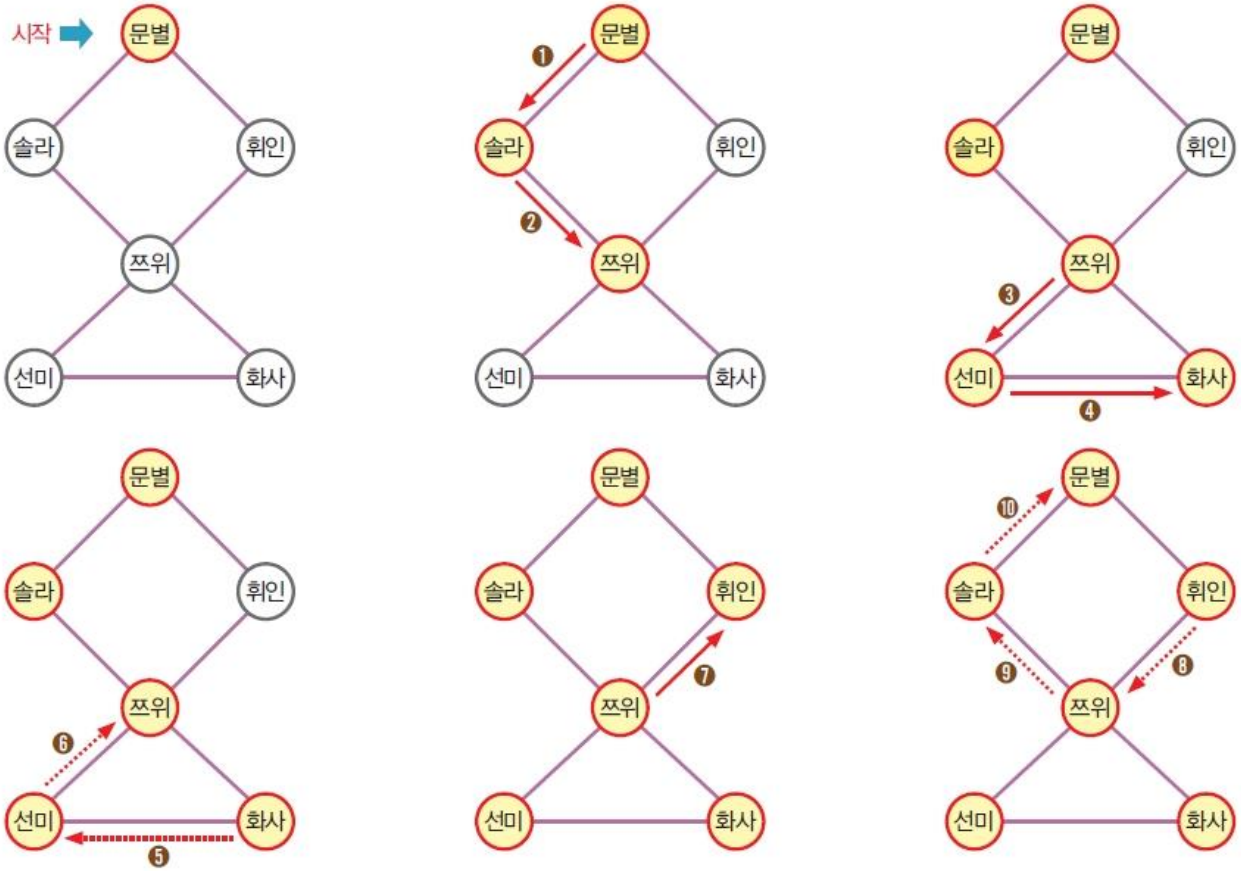


그림 9-6 깊이 우선 탐색 그래프 예 : 문별부터 시작

- **그래프의 인접 행렬 표현**
  - 그래프를 코드로 구현할 때는 인접 행렬을 사용
  - 인접 행렬은 정방형으로 구성된 행렬로 정점이 4개인 그래프는 4×4로 표현

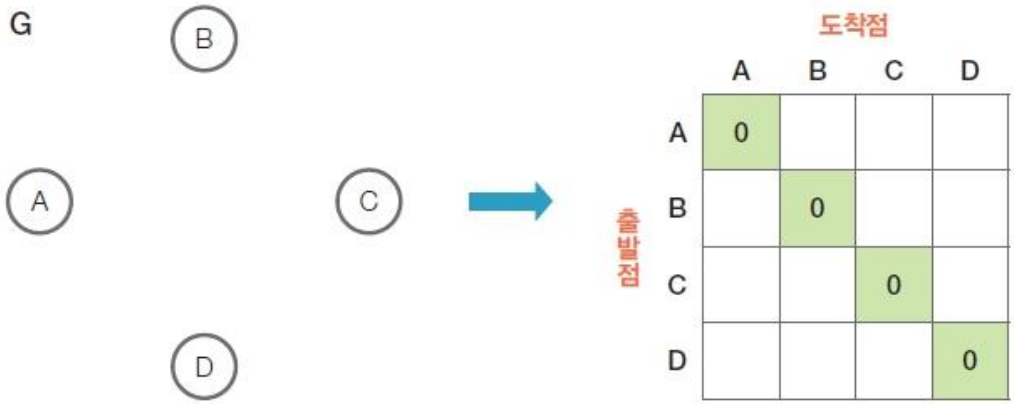
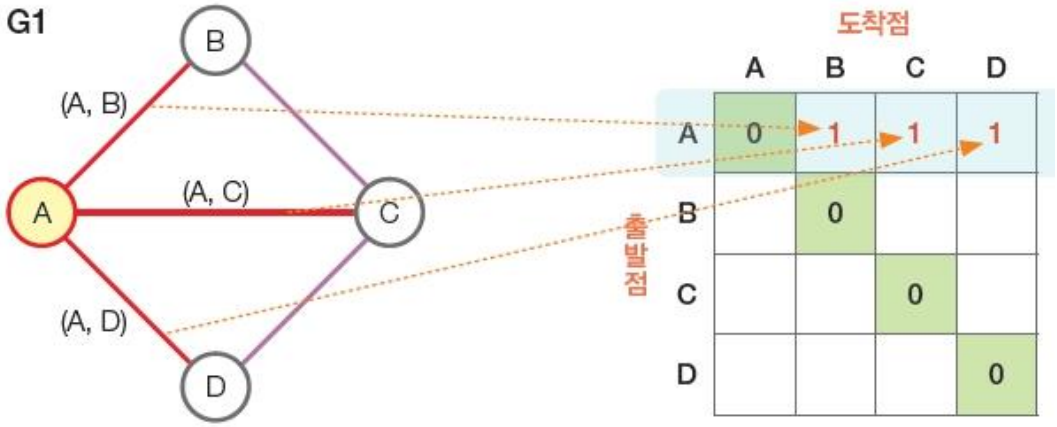


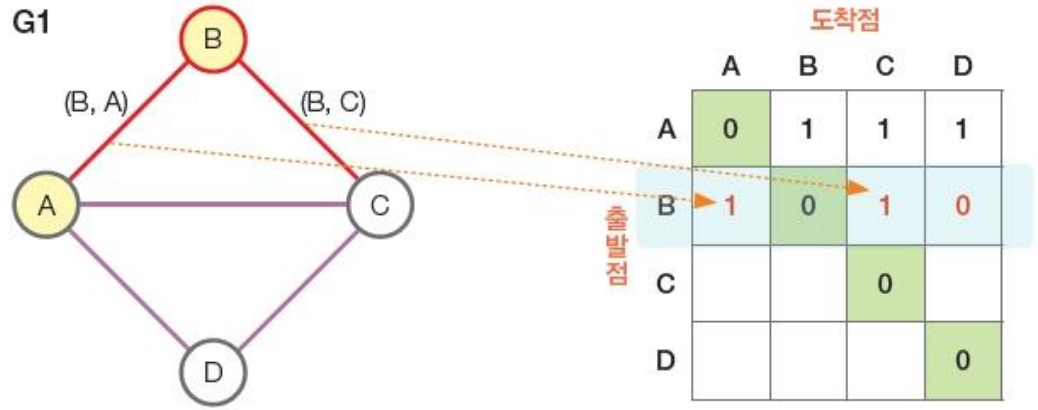
그림 9-7 정점 4개로 된 그래프의 인접 행렬 초기 상태

- 1. 무방향 그래프의 인접 행렬

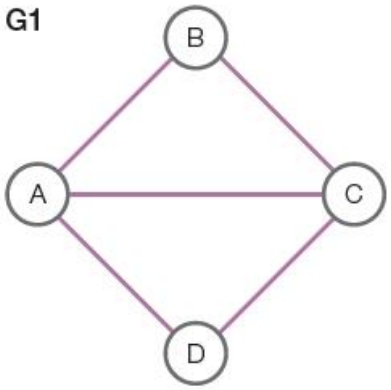
1 출발점 A와 연결된 도착점 B, C, D의 칸을 1로 설정한다.



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



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

- 무방향 그래프의 인접 행렬은 대각선을 기준으로 서로 대칭된다.

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

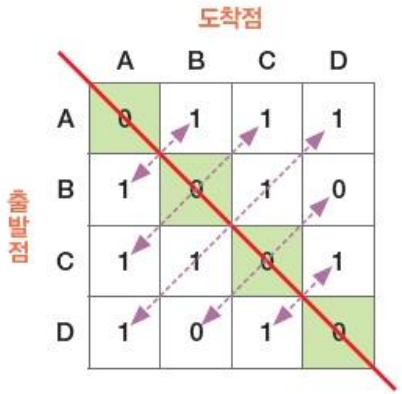
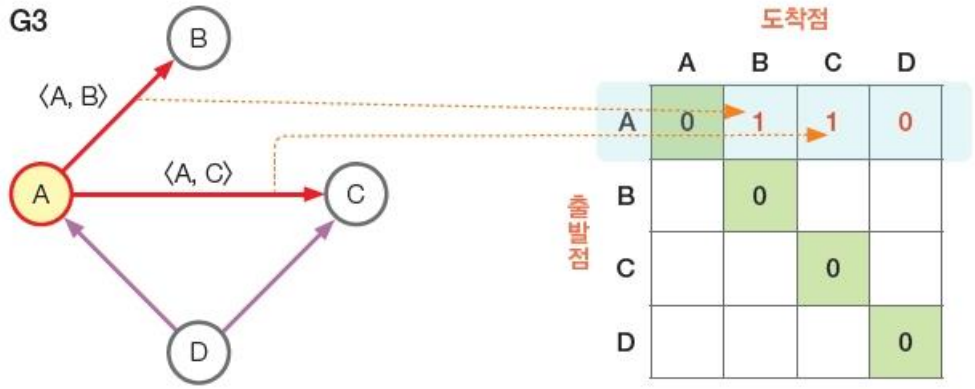


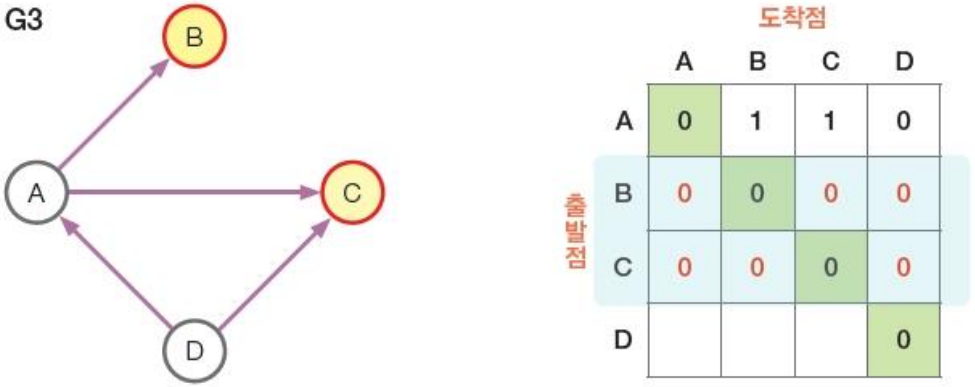
그림 9-8 무방향 그래프의 대칭 특성

- 2. 방향 그래프의 인접 행렬

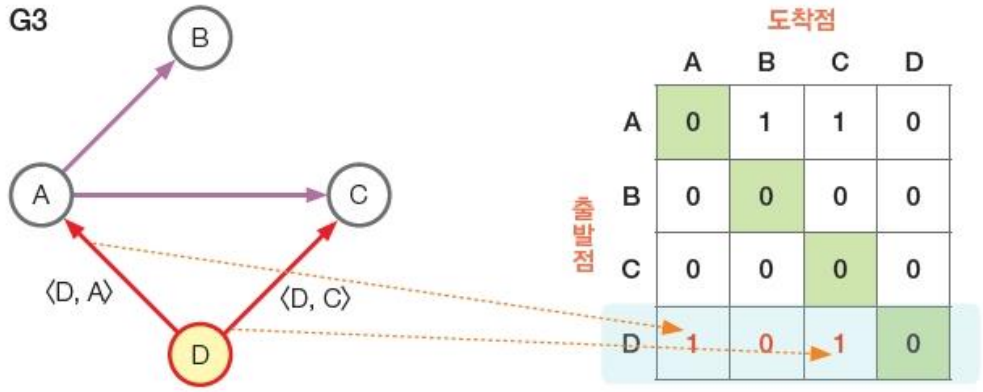
1 출발점 A에서 나가 도착점이 B, C의 칸만 1로 설정하고 나머지는 0으로 채운다.



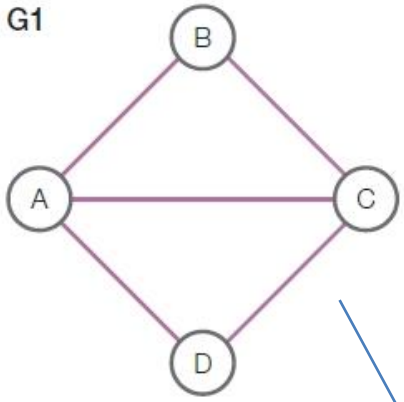
2 출발점 B와 C는 나가는 곳이 없으므로 모두 0으로 채운다.



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



- **무방향성 G1 그래프와 방향성 G3 그래프 구현 예**



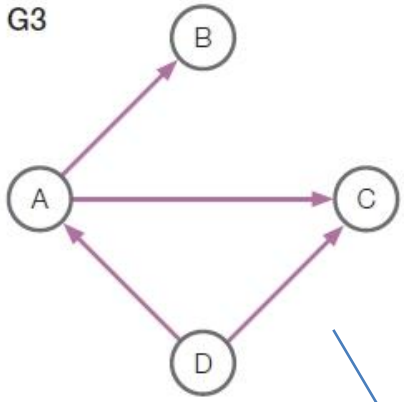


그림 9-9 구현할 무방향 그래프 G1과 방향 그래프 G3

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

▪ 그래프의 정점 생성

- 행과 열이 같은 2차원 배열을 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)
```

- 4×4 크기의 초기화된 그래프를 생성

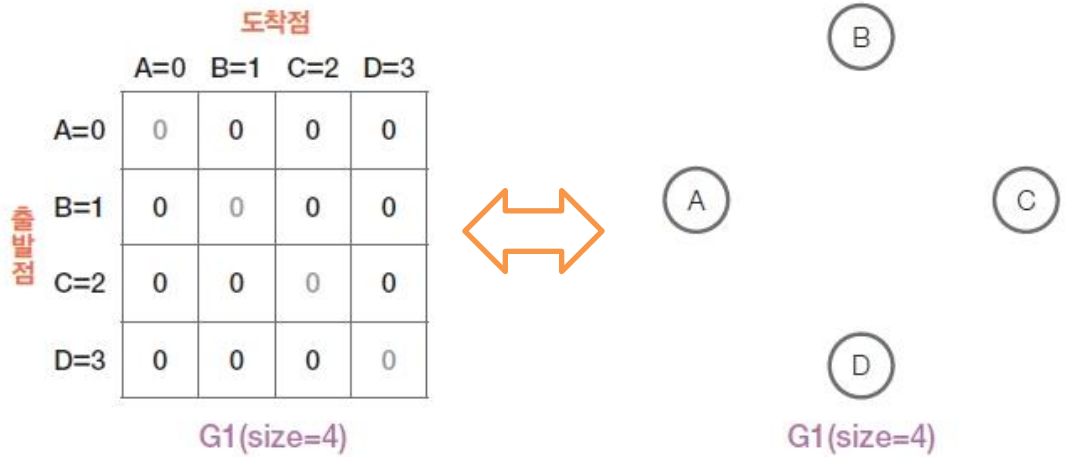


그림 9-10 정점 4개를 가진 그래프의 초기 상태(인접 행렬)와 그래프

- **그래프의 정점 연결**
  - 정점 A와 B에 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
```

출발점 도착점

출발점 \ 도착점	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

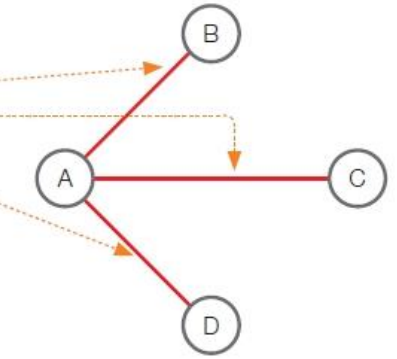


그림 9-11 정점 A와 연결된 간선 구현 코드와 결과

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

출발점 \ 도착점	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

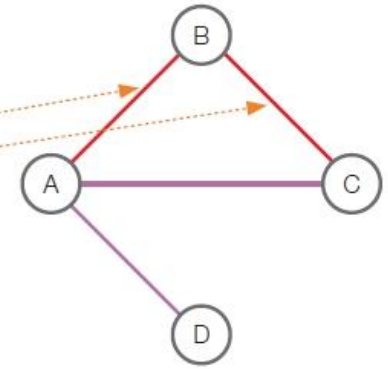


그림 9-12 정점 B와 연결된 간선 구현 코드와 결과

- 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
```

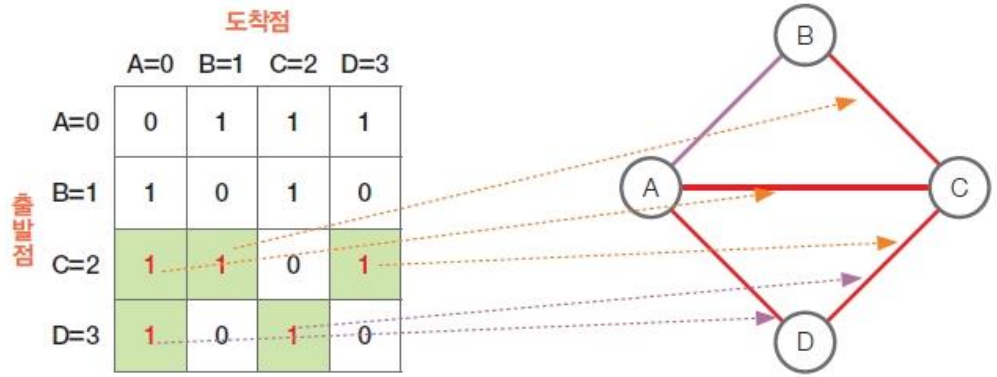


그림 9-13 정점 C 및 D와 연결된 간선 구현 코드와 결과

Code09-01.py 무방향 그래프 G1과 방향 그래프 G3의 구현

```
1  ## 함수 선언 부분 ##
2  class Graph() :
3      def __init__ (self, size) :
4          self.SIZE = size
5          self.graph = [[0 for _ in range(size)] for _ in range(size)]
6
7  ## 전역 변수 선언 부분 ##
8  G1, G3 = None, None
9
```

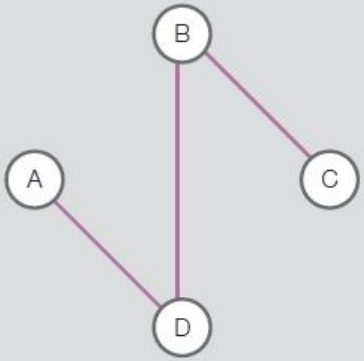
```
10 ## 메인 코드 부분 ##
11 G1 = Graph(4)
12 G1.graph[0][1] = 1; G1.graph[0][2] = 1; G1.graph[0][3] = 1
13 G1.graph[1][0] = 1; G1.graph[1][2] = 1
14 G1.graph[2][0] = 1; G1.graph[2][1] = 1; G1.graph[2][3] = 1
15 G1.graph[3][0] = 1; G1.graph[3][2] = 1
16
17 print('## G1 무방향 그래프 ##')
18 for row in range(4) :
19     for col in range(4) :
20         print(G1.graph[row][col], end = ' ')
21     print()
22
23 G3 = Graph(4)
24 G3.graph[0][1] = 1; G3.graph[0][2] = 1
25 G3.graph[3][0] = 1; G3.graph[3][2] = 1
26
27 print('## G3 방향 그래프 ##')
28 for row in range(4) :
29     for col in range(4) :
30         print(G3.graph[row][col], end = ' ')
31     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
```

SELF STUDY 9-1

Code09-01.py를 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 개선**
  - 무방향 그래프를 인접 행렬로 구성할 때 사람 이름, 도시 이름으로 구성 예

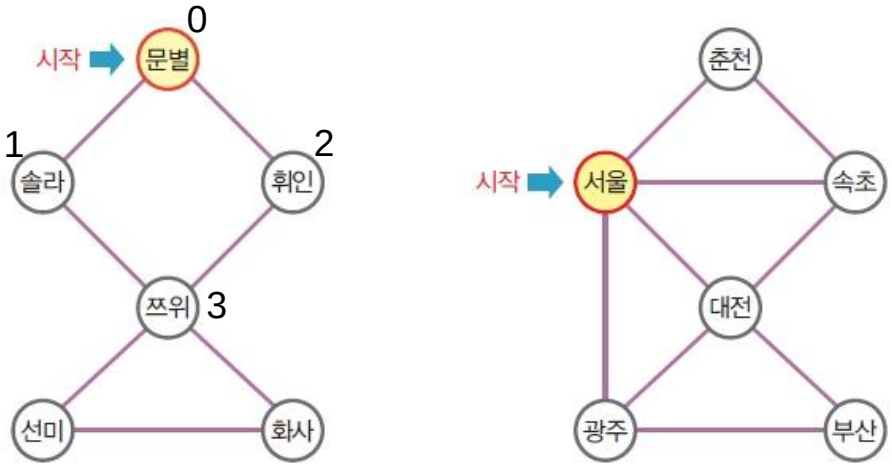


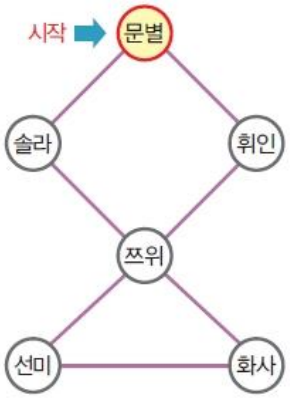
그림 9-14 그래프의 실제 형태

```
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

- 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

그림 9-15 행과 열의 주석이 없는 인접 행렬

- [그림 9-15]의 (c)와 같이 출력하기 위한 코드

```
nameAry = ['문별', '솔라', '휘인', '쯔위', '선미', '화사']
print(' ', end = ' ')
for v in range(G1.SIZE) :
    print(nameAry[v], end = ' ')
print()
for row in range(G1.SIZE) :
    print(nameAry[row], end = ' ')
    for col in range(G1.SIZE) :
        print(G1.graph[row][col], end = ' ')
    print()
print()
```

Section 02 그래프의 구현

- Code09-01.py에서 무방향 그래프만 [그림 9-15]의 (c)와 같이 행과 열의 주석이 있는 인접 행렬 형태로 구현

Code09-02.py 개선된 무방향 그래프 G1

```
## 클래스와 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()


## 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
```

```
24
25
26 ## 메인 코드 부분 ##
27 gSize = 6
28 G1 = Graph(gSize)
29 G1.graph[문별][솔라] = 1; G1.graph[문별][휘인] = 1
30 G1.graph[솔라][문별] = 1; G1.graph[솔라][쯔위] = 1
31 G1.graph[휘인][문별] = 1; G1.graph[휘인][쯔위] = 1
32 G1.graph[쯔위][솔라] = 1; G1.graph[쯔위][휘인] = 1; G1.graph[쯔위][선미] = 1; G1.graph[쯔위][화사] = 1
33 G1.graph[선미][쯔위] = 1; G1.graph[선미][화사] = 1
34 G1.graph[화사][쯔위] = 1; G1.graph[화사][선미] = 1
35
36 print('## G1 무방향 그래프 ##')
37 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
```

깊이 우선 탐색의 구현

깊이 우선 탐색 구현을 위한 준비

- 깊이 우선 탐색을 구현하려면 스택을 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')
```

그림에는 정점 이름이 A, B, C, D로 표현되도록 할 것임

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

방문 기록 visitedAry 배열을 출력할 때는 알파벳으로 출력하도록 변경

- 깊이 우선 탐색의 단계별 구현
  - 간단한 그래프를 깊이 우선 탐색하는 과정 구현 예

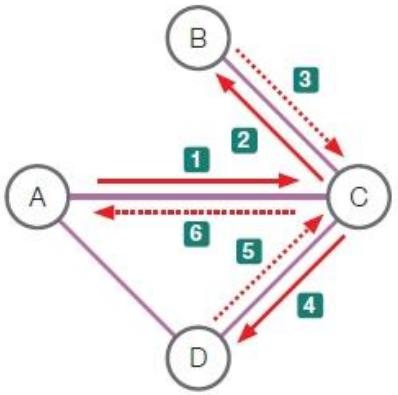


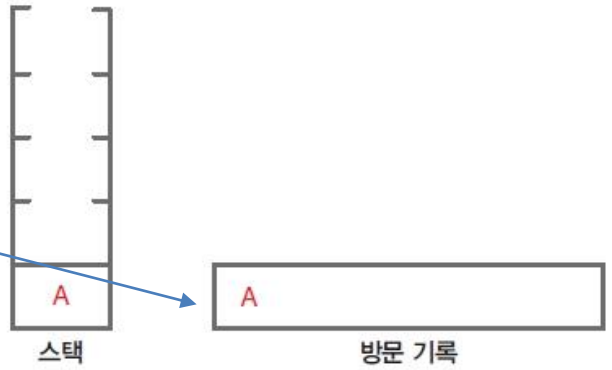
그림 9-16 깊이 우선 탐색으로 탐색할 그래프

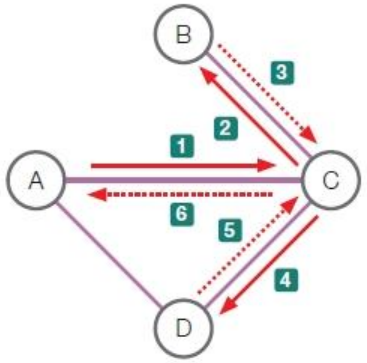
출발점 \ 도착점	A=0	B=1	C=2	D=3
A=0	0	0	1	1
B=1	0	0	1	0
C=2	1	1	0	1
D=3	1	0	1	0

  - 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)
```





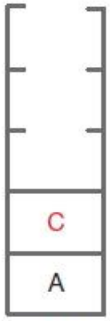
도착점

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

그림 9-16 깊이 우선 탐색으로 탐색할 그래프

1 과정(정점 C 방문)

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

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

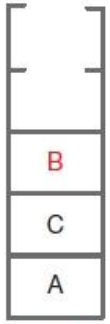


스택

방문 기록

2 과정(정점 B 방문)

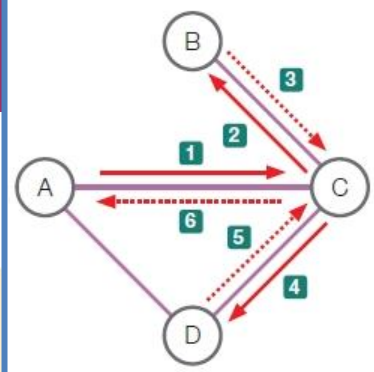
```
   next = None
   for vertex in range(4) :
       if G1.graph[current][vertex] == 1 :
❷          if vertex in visitedAry :    # 방문한 적이 있는 정점이면 탈락
❶              pass
❸          else :                       # 방문한 적이 없으면 다음 정점으로 지정
               next = vertex
               break

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



스택

방문 기록

Section 02 그래프의 구현



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

그림 9-16 깊이 우선 탐색으로 탐색할 그래프

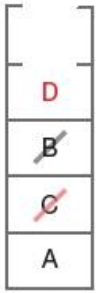
3 과정(되돌아오는 이동)

```
next = None
❶ for vertex in range(4) :
      if G1.graph[current][vertex] == 1 :
          ❷ if vertex in visitedAry :    # 방문한 적이 있는 정점이면 탈락
                pass
          ❸ 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)
❹ else :                                 # 다음에 방문할 정점이 없는 경우
    ❺ current = stack.pop()
```

스택

방문 기록

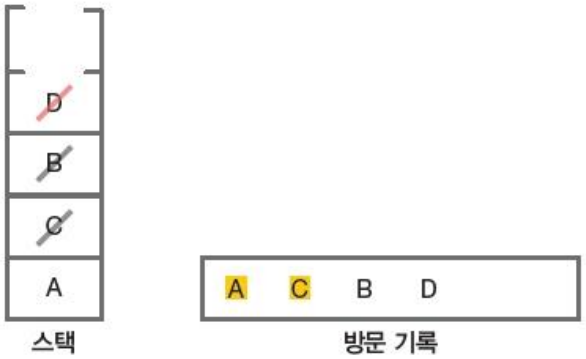
4 과정(정점 D 방문)



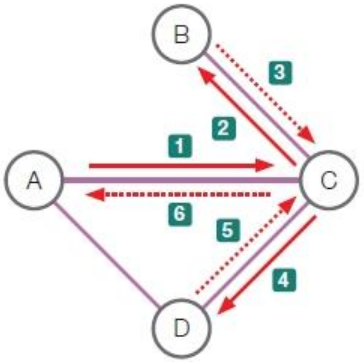
스택

방문 기록

5 과정(되돌아오는 이동)



6 과정(되돌아오는 이동)



출발점 \ 도착점	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

그림 9-16 깊이 우선 탐색으로 탐색할 그래프

Code09-03.py 깊이 우선 탐색의 구현

```
## 클래스와 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()
```

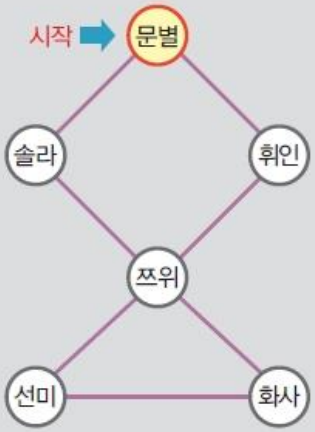
```
24
25 current = 0       # 시작 정점
26 stack.append(current)
27 visitedAry.append(current)
28
29 while (len(stack) != 0) :
30     next = None
31     for vertex in range(4) :
32         if G1.graph[current][vertex] == 1 :
33             if vertex in visitedAry :  # 방문한 적이 있는 정점이면 탈락
34                 pass
35             else :                     # 방문한 적이 없으면 다음 정점으로 지정
36                 next = vertex
37                 break
38
39     if next != None :                  # 다음에 방문할 정점이 있는 경우
40         current = next
41         stack.append(current)
42         visitedAry.append(current)
43     else :                             # 다음에 방문할 정점이 없는 경우
44         current = stack.pop()
45
46
47 print('방문 순서 -->', end = ' ')
48 for i in visitedAry :
49     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
```

SELF STUDY 9-2

Code09-03.py를 수정해서 다음 그림과 같은 무방향 그래프를 순회해 보자.



실행 결과

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

- **친구가 연락되는지 확인**
  - 친구들의 비상연락망에서 특정 친구가 연락되는지 확인하는 방법

(a) 비상 연락망의 기본 형태

(b) 비상 연락망의 연결이 끊긴 경우

그림 9-17 **비상 연락망 예**

- 비상연락망 연결이 끊긴 경우('화사'만 동떨어진 예)

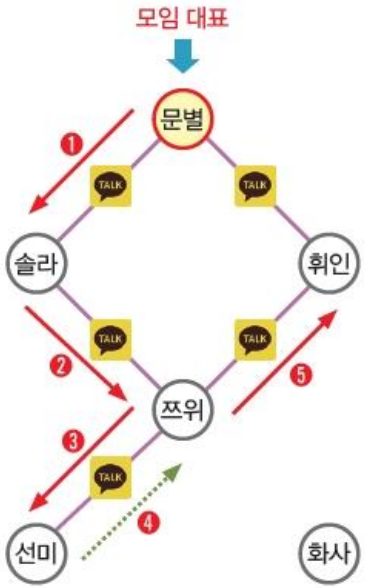


그림 9-18 비상연락망 연결이 끊긴 경우

```
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;
```

- 시작 정점인 문별부터 방문을 시작하여 연결된 모든 친구에게 연락하고 그 결과를 visitedAry에 저장

```
stack = []
visitedAry = []    # 방문한 정점

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

while (len(stack) != 0) :
    next = None
    for vertex in range(gSize) :
        if G1.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()
```

- 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')
```

Code09-04.py 특정 정점이 그래프에 연결되어 있는지 확인하는 함수

```
1  gSize = 6
2  def findVertex(g, findVtx) :
3      stack = []
4      visitedAry = []    # 방문한 정점
5
6      current = 0        # 시작 정점
7      stack.append(current)
8      visitedAry.append(current)
9
10     while (len(stack) != 0) :
11         next = None
12         for vertex in range(gSize) :
13             if g.graph[current][vertex] != 0 :
14                 if vertex in visitedAry :    # 방문한 적이 있는 정점이면 탈락
```

```
15                      pass
16                  else :                          # 방문한 적이 없으면 다음 정점으로 지정
17                      next = vertex
18                      break
19
20          if next != None :                       # 다음에 방문할 정점이 있는 경우
21              current = next
22              stack.append(current)
23              visitedAry.append(current)
24          else :                                  # 다음에 방문할 정점이 없는 경우
25              current = stack.pop()
26
27      if findVtx in visitedAry :
28          return True
29      else :
30          return False
```

실행 결과

없음

- **최소 비용으로 자전거 도로 연결**
  - 최소 신장 트리 개념
    - 신장 트리(Spanning Tree) : 최소 간선으로 그래프의 모든 정점이 연결되는 그래프

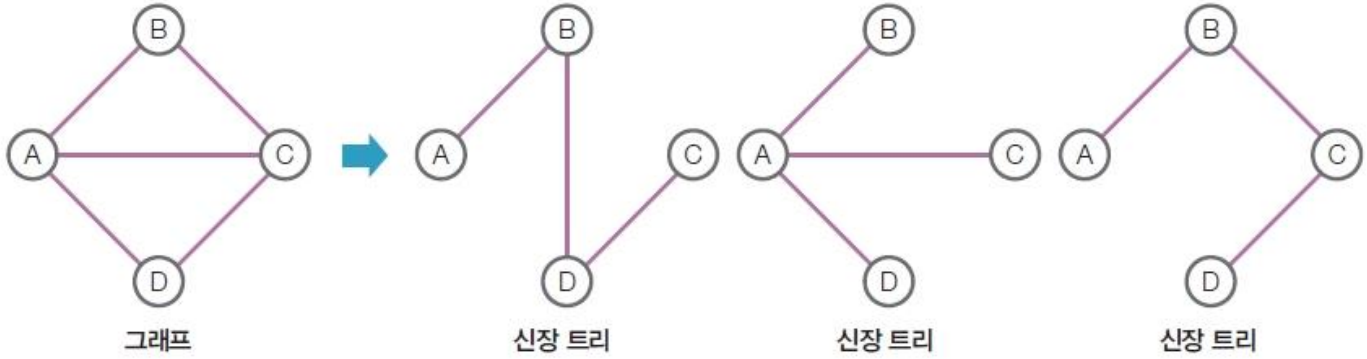


그림 9-19 그래프와 다양한 신장 트리

    - 가중치 그래프와 신장 트리

Minimum Cost Spanning Tree

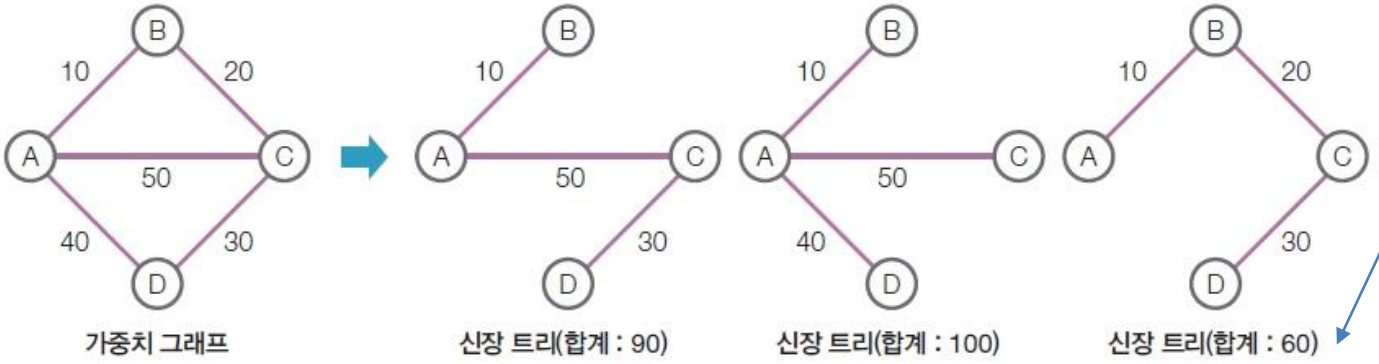


그림 9-20 가중치 그래프와 다양한 신장 트리

Section 03 그래프의 응용

- 최소 비용 신장 트리(Minimum Cost Spanning Tree)는 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)

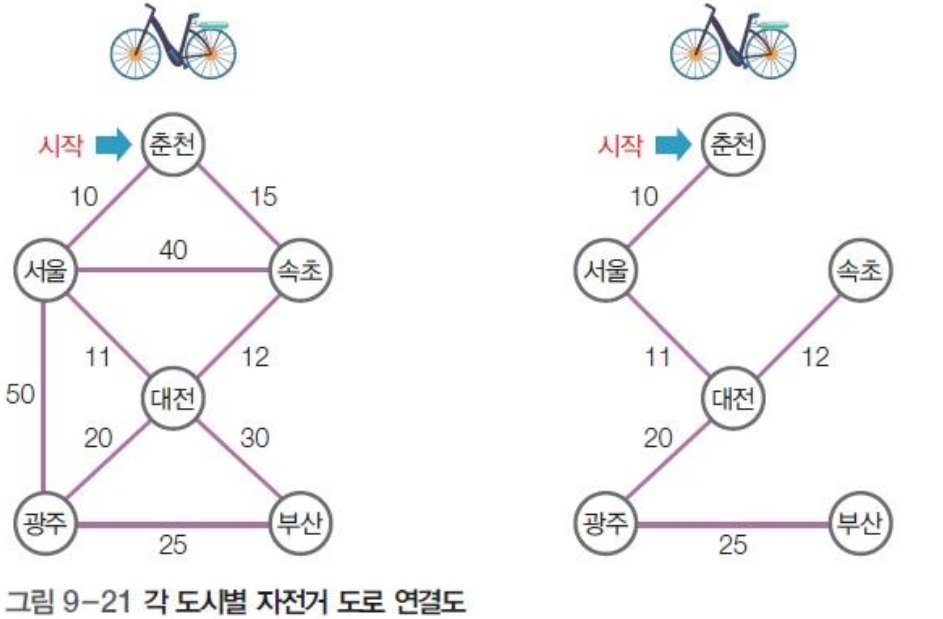


그림 9-21 각 도시별 자전거 도로 연결도

↑ 그래프에 가중치를 추가한 형태

↑ 최소 비용 신장 트리로 구성한 상태

- 전체 자전거 도로를 위한 가중치 그래프 구현
  - 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
```

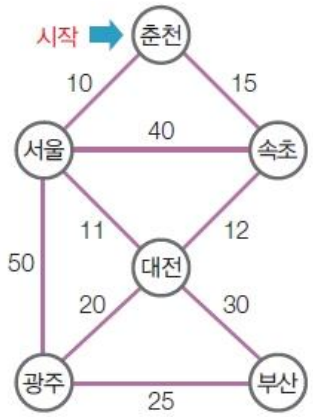


그림 9-22 각 도시별 자전거 도로 연결 계획도

	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

- 가중치와 간선 목록 생성
  - 가중치와 간선을 별도 배열로 만드는 예

```
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']]
```

그림 9-23 생성된 가중치와 간선 목록

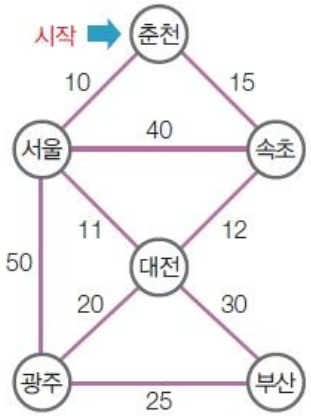


그림 9-22 각 도시별 자전거 도로 연결 계획도

	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

- 간선 정렬
  - 가중치를 기준으로 내림차순으로 간선 정렬

파이썬 operator 패키지의 itemgetter로 정렬

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

정렬 기준을 0번째 부터

내림차순

```
[[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']]
```

그림 9-24 정렬된 가중치와 간선 목록

- 중복 간선 제거

```
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']]
```

그림 9-25 중복을 제거한 가중치와 간선 목록

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

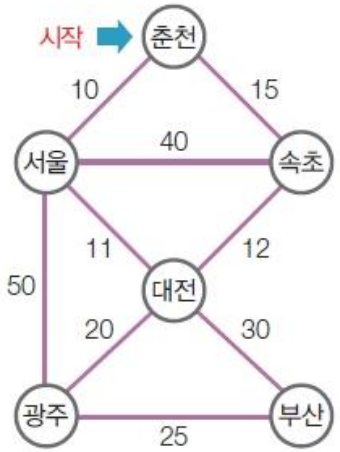


그림 9-26 가중치표와 자전거 도로 연결망

- 가중치가 높은 간선부터 제거

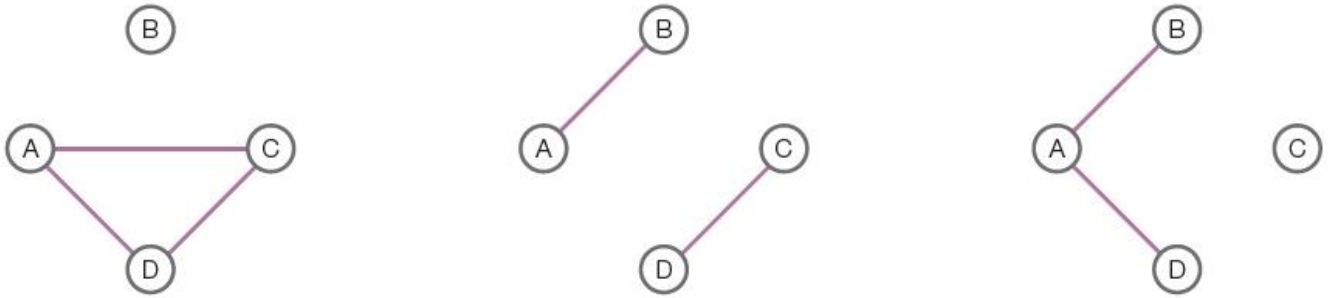


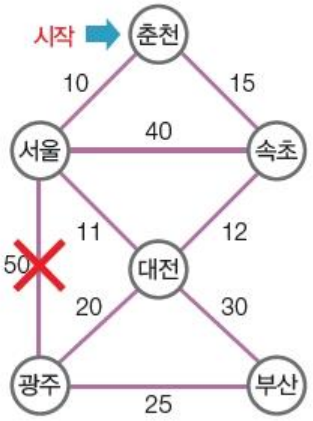
그림 9-27 **도시가 연결되지 않는 것은 허용하지 않음**

1 서울-광주 간선 제거

```
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
index → 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

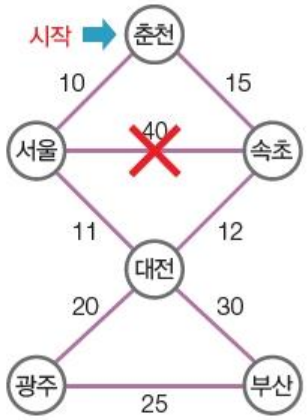


2 서울–속초 간선 제거

```
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

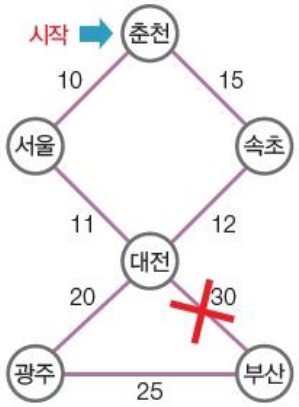


3 대전–부산 간선 제거

```
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

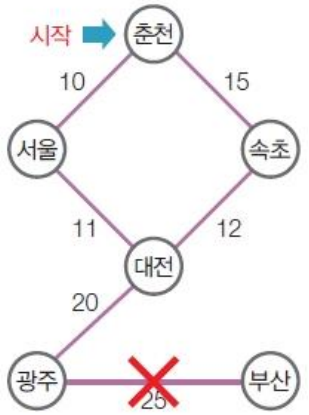


4 광주-부산 간선의 제거 시도와 원상 복구

```
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
   else :
❺      G1.graph[start][end] = saveCost
       G1.graph[end][start] = saveCost
❻   index += 1
```

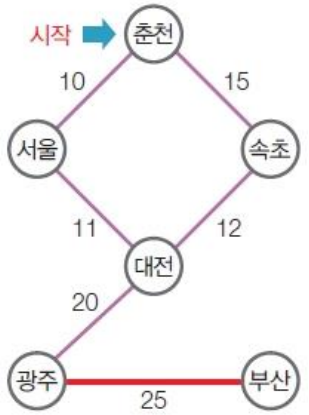
가중치 배열(newAry)

	가중치	간선
index → 0	~~50~~	~~서울 – 광주~~
	~~10~~	~~서울 – 속초~~
	~~00~~	~~대전 – 부산~~
	~~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~~
	~~10~~	~~서울 – 속초~~
	~~00~~	~~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

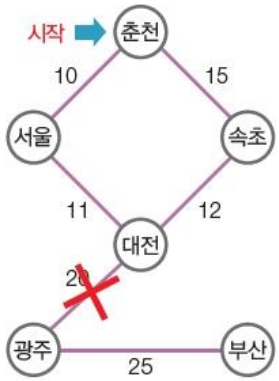


5 대전–광주 간선의 제거 시도와 원상 복구

```
index = 1    # 배열의 1번째 위치(대전-광주). 앞 단계에서 index를 1 증가시켰음

## 나머지는 앞 코드와 동일
```

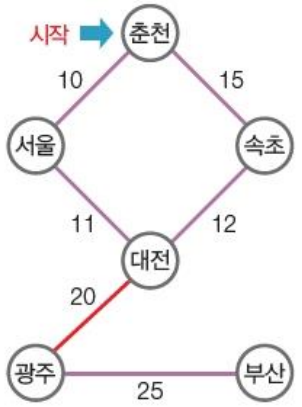
가중치 배열(newAry)

	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)

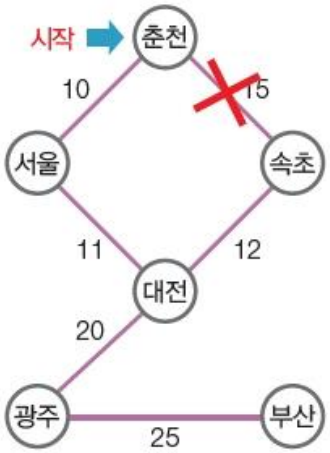
	가중치	간선
	~~50~~	~~서울 – 광주~~
	~~40~~	~~서울 – 속초~~
	~~30~~	~~대전 – 부산~~
0	25	광주 – 부산
1	20	대전 – 광주
index → 2	15	춘천 – 속초
3	12	속초 – 대전
4	11	서울 – 대전
5	10	춘천 – 서울



6 춘천–속초 간선 제거

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



- 자전거 도로 건설을 위한 최소 비용 신장 트리의 전체 코드

```
## 클래스와 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) :
    # 생략(Code09-02.py의 8~17행과 동일)

def findVertex(g, findVtx) :
    # 생략(Code09-04.py의 3~30행과 동일)


## 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

```

```
57 ## 메인 코드 부분 ##
58 gSize = 6
59 G1 = Graph(gSize)
60 G1.graph[춘천][서울] = 10; G1.graph[춘천][속초] = 15
61 G1.graph[서울][춘천] = 10; G1.graph[서울][속초] = 40; G1.graph[서울][대전] = 11;
   G1.graph[서울][광주] = 50
62 G1.graph[속초][춘천] = 15; G1.graph[속초][서울] = 40; G1.graph[속초][대전] = 12
63 G1.graph[대전][서울] = 11; G1.graph[대전][속초] = 12; G1.graph[대전][광주] = 20;
   G1.graph[대전][부산] = 30
64 G1.graph[광주][서울] = 50; G1.graph[광주][대전] = 20; G1.graph[광주][부산] = 25
65 G1.graph[부산][대전] = 30; G1.graph[부산][광주] = 25
66
67 print('## 자전거 도로 건설을 위한 전체 연결도 ##')
68 printGraph(G1)
69
70 # 가중치 간선 목록
71 edgeAry = []
72 for i in range(gSize) :
73     for k in range(gSize) :
74         if G1.graph[i][k] != 0 :
75             edgeAry.append([G1.graph[i][k], i, k])
76
77 from operator import itemgetter
```

```
78 edgeAry = sorted(edgeAry, key=itemgetter(0), reverse=True)
79
80 newAry = []
81 for i in range(0, len(edgeAry), 2) :
82     newAry.append(edgeAry[i])
83
84 index = 0
85 while (len(newAry) > gSize-1) :   # 간선 개수가 '정점 개수-1'일 때까지 반복
86     start = newAry[index][1]
87     end = newAry[index][2]
88     saveCost = newAry[index][0]
89
90     G1.graph[start][end] = 0
91     G1.graph[end][start] = 0
92
93     startYN = findVertex(G1, start)
94     endYN = findVertex(G1, end)
95
96     if startYN and endYN :
97         del(newAry[index])
98     else :
99         G1.graph[start][end] = saveCost
100        G1.graph[end][start] = saveCost
101        index += 1
102
103 print('## 최소 비용의 자전거 도로 연결도 ##')
104 printGraph(G1)
```

실행 결과

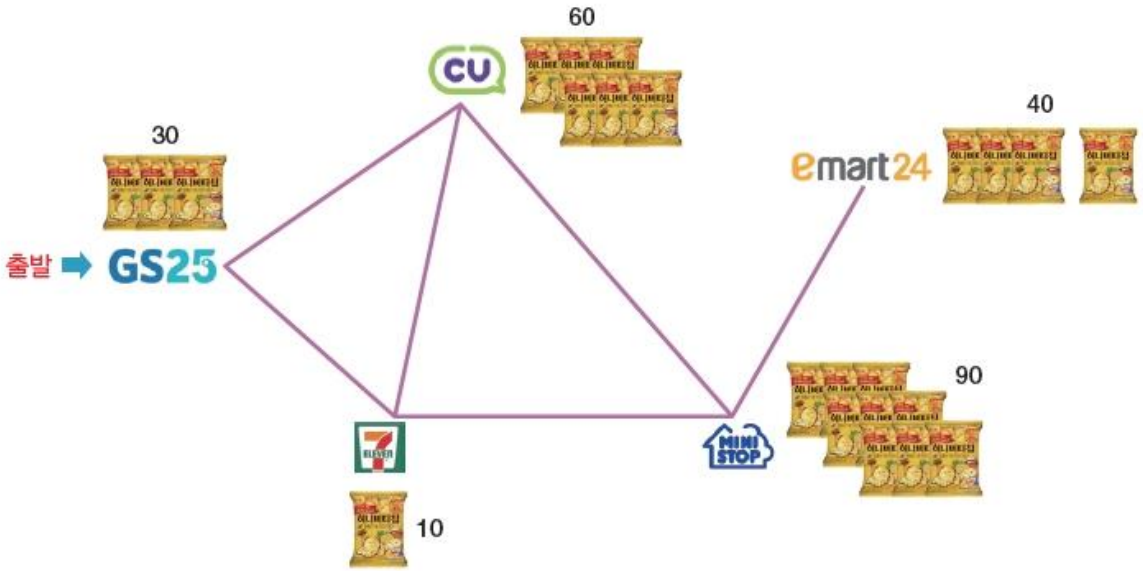
```
## 자전거 도로 건설을 위한 전체 연결도 ##
      춘천  서울  속초  대전  광주  부산
춘천    0    10    15     0     0     0
서울   10     0    40    11    50     0
속초   15    40     0    12     0     0
대전    0    11    12     0    20    30
광주    0    50     0    20     0    25
부산    0     0     0    30    25     0

## 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
```

응용예제 01 허니버터칩이 가장 많이 남은 편의점 찾기

난이도★★☆☆☆

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.



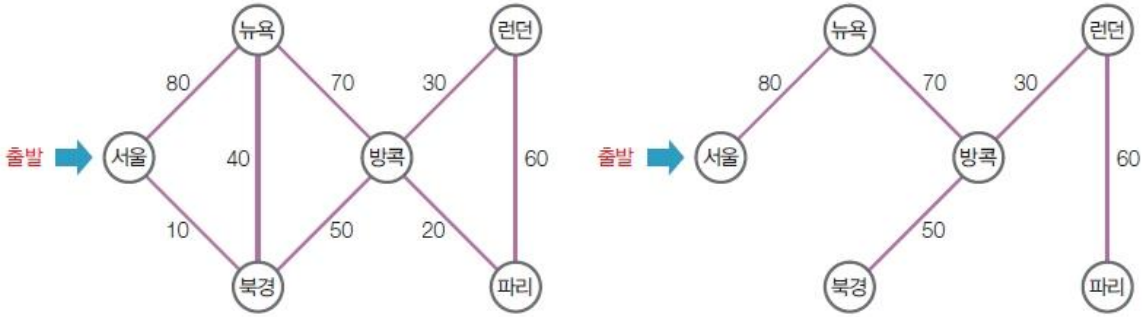
실행 결과

```
Python
File  Edit  Shell  Debug  Options  Window  Help
========== 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 )
>>>
                                                Ln: 14  Col: 4
```

응용예제 02 가장 효율적인 해저 케이블망 구성하기

난이도★★☆☆☆

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.



실행 결과

Python

File Edit Shell Debug Options Window Help

```
========== 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
```

Ln: 78 Col: 4

Thank You