알고리즘과 코딩테스트 준비

2022/1/13~1/19

윤형기 (hky@openwith.net)

다이나믹 프로그래밍 (2)

- 개념
  - Optimization 문제를 다수의 subproblem으로 분해하여 처리.

- 3가지 DP 코딩 패턴
  - Recursion

```
# A naive recursive solution
def fib(n):
    if n == 1 or n == 2:
        result = 1
    else:
        result = fib(n-1) + fib(n-2)
    return result
```

fibonaci.ipynb

– Store (memoization)

```
# A memoized solution
def fib_2(n, memo):
    if memo[n] is not None:
        return memo[n]
    if n == 1 or n == 2:
        result = 1
    else:
        result = fib_2(n-1, memo) + fib_2(n-2, memo)
    memo[n] = result
    return result

def fib_memo(n):
    memo = [None] * (n + 1)
    return fib_2(n, memo)
```

– Bottom-up

```
# A bottom-up solution
def fib_bottom_up(n):
    if n == 1 or n == 2:
        return 1
    bottom_up = [None] * (n+1)
    bottom_up[1] = 1
    bottom_up[2] = 1
    for i in range(3, n+1):
        bottom_up[i] = bottom_up[i-1] + bottom_up[i-2]
    return bottom_up[n]
```

- 주된 방식
  - Tabulation: Bottom Up
  - Memoization: Top Down

- Memoization을 이용한 Top-down 방식
  - sub-problem을 다룰 때 그 결과를 cache에 저장한 후 사용.

- non-DP recursive solution for finding nth Fib no:

```
def calculateFibonacci(n):
  if n < 2:
     return n

  return calculateFibonacci(n - 1) + calculateFibonacci(n - 2)

def main():
  print("5th Fibonacci is ---> " + str(calculateFibonacci(5)))
  print("6th Fibonacci is ---> " + str(calculateFibonacci(6)))
  print("7th Fibonacci is ---> " + str(calculateFibonacci(7)))

main()
```

```
using namespace std;

#include <iostream>

class Fibonacci {

public:
  virtual int CalculateFibonacci(int n) {
    if (n < 2) {
      return n;
    }
    return CalculateFibonacci(n - 1) + CalculateFibonacci(n - 2);
  }
};

int main(int argc, char *argv[]) {
  Fibonacci *fib = new Fibonacci();
  cout << "5th Fibonacci is ---> " << fib->CalculateFibonacci(5) << endl;
  cout << "6th Fibonacci is ---> " << fib->CalculateFibonacci(6) << endl;
  cout << "7th Fibonacci is ---> " << fib->CalculateFibonacci(7) << endl;

  delete fib;
}
```

- memoization

```
def calculateFibonacci(n):
  memoize = [-1 for x in range(n+1)]
  return calculateFibonacciRecur(memoize, n)

def calculateFibonacciRecur(memoize, n):
    if n < 2:
       return n

  # if we have already solved this subproblem, simply return the result from cache
  if memoize[n] >= 0:
     return memoize[n]

  memoize[n] = calculateFibonacciRecur(
    memoize, n - 1) + calculateFibonacciRecur(memoize, n - 2)
  return memoize[n]

def main():
  print("5th Fibonacci is ---> " + str(calculateFibonacci(5)))
  print("6th Fibonacci is ---> " + str(calculateFibonacci(6)))
  print("7th Fibonacci is ---> " + str(calculateFibonacci(7)))

main()
```

```
using namespace std;

#include <iostream>

class Fibonacci {

public:
  virtual int CalculateFibonacci(int n) {
    if (n < 2) {
      return n;
    }
    return CalculateFibonacci(n - 1) + CalculateFibonacci(n - 2);
  }
};

int main(int argc, char *argv[]) {
  Fibonacci *fib = new Fibonacci();
  cout << "5th Fibonacci is ---> " << fib->CalculateFibonacci(5) << endl;
  cout << "6th Fibonacci is ---> " << fib->CalculateFibonacci(6) << endl;
  cout << "7th Fibonacci is ---> " << fib->CalculateFibonacci(7) << endl;

  delete fib;
}
```

- Bottom-up with Tabulation
  - Recursion을 회피 하고 solve the problem “bottom-up”
  - (= 즉, 관련된 모든 sub-problem들을 먼저 해결).
  - n-D 테이블을 채워 넣고 나서 이에 의거하여 top/original 문제에 대한 솔루션을 도출
  - Tabulation 은 Memoization의 반대 형태
    - cf. Memoization
      - solve problem and maintain a map of already solved sub-problems.
      - 즉, top-down in the sense that we solve the top problem first (which typically recurses down to solve the sub-problems).

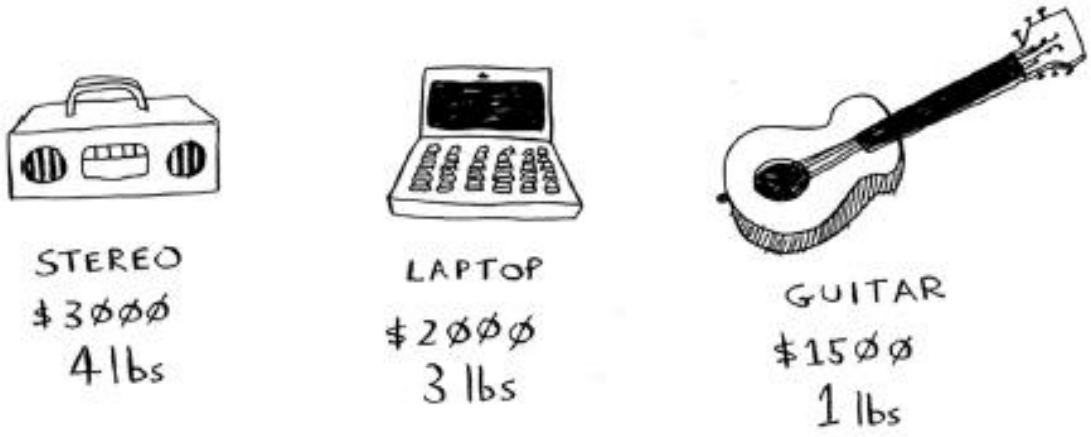
- Tabulation

```
def calculateFibonacci(n):
  dp = [0, 1]
  for i in range(2, n + 1):
    dp.append(dp[i - 1] + dp[i - 2])

  return dp[n]

def main():
  print("5th Fibonacci is ---> " + str(calculateFibonacci(5)))
  print("6th Fibonacci is ---> " + str(calculateFibonacci(6)))
  print("7th Fibonacci is ---> " + str(calculateFibonacci(7)))


main()
```

```
using namespace std;

#include <iostream>
#include <vector>

class Fibonacci {

public:
  virtual int CalculateFibonacci(int n) {
    if (n==0) return 0;
    vector<int> dp(n + 1);
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
      dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
  }
};

int main(int argc, char *argv[]) {
  Fibonacci *fib = new Fibonacci();
  cout << "5th Fibonacci is ---> " << fib->CalculateFibonacci(5) << endl;
  cout << "6th Fibonacci is ---> " << fib->CalculateFibonacci(6) << endl;
  cout << "7th Fibonacci is ---> " << fib->CalculateFibonacci(7) << endl;

  delete fib;
}
```

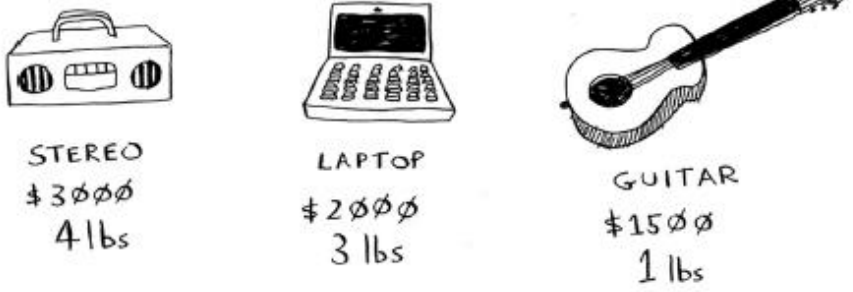
knapsack 문제

- knapsack can carry 4 lb of goods.



- 솔루션
  - Simple solution: 가능한 모든 대안 탐색
    - 2^n의 경우의 수

- DP



knapsack 크기

상품 선택

	1	2	3	4
Guitar				
Stereo				
Laptop				

cell[i][j] = max of

(1) previous max (value at cell[i-1][j])

vs.

(2) value of current item + value of the remaining space

Cell[i-1][j – item's weight]

Longest common substring

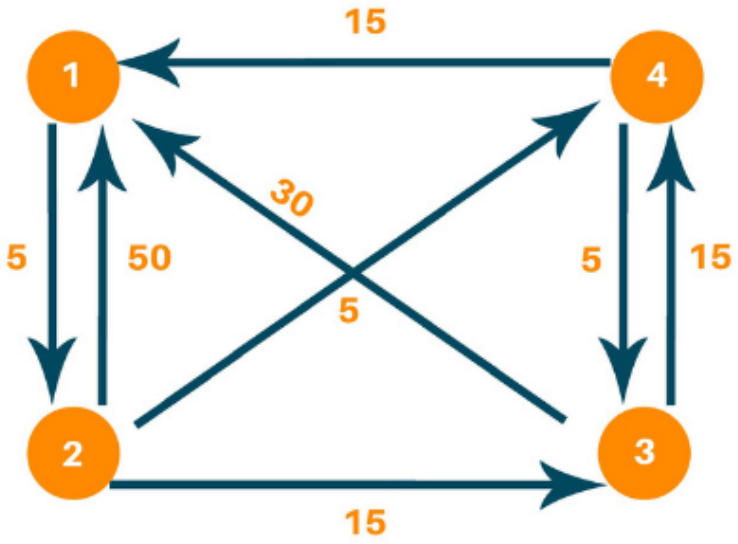
- LCS(Longest Common Subsequence: 최장 공통 부분 수열)
- 예:
  - HISH vs. FISH
  - FOSH vs. FISH

```
if word_a[i] == word_b[j]:
    cell[i][j] = cell[i-1][j-1] +1
else
    cell[i][j] = max(cell[i-1][j], cell[i][j-1] )
```

- DP Tip
  - Every DP solution involves a grid
  - 통상 optimization
  - Each cell is a subproblem

그래프 알고리즘에의 적용

- Floyd Warshall 알고리즘
  - All Pairs Shortest Path
    - 모든 지점에서 다른 모든 지점까지의 최단경로 탐색
    - cf. Dijkstra's algorithm: "한 지점에서 다른 지점까지의 최단경로"



- Bellman-Ford 알고리즘
  - Dijkstra 알고리즘의 negative edge 문제를 해결
  - DP 알고리즘으로서 try out all possible solution and then pick up the best solution
    - - relax all (n-1) edges (예: |V|=7일 때 7-1=6번 relax (6 edges)

실습과제

Q 31 금광

난이도 ●◐○ | 풀이 시간 30분 | 시간 제한 1초 | 메모리 제한 128MB | 기출 Flipkart 인터뷰

$n \times m$ 크기의 금광이 있습니다. 금광은 1×1 크기의 칸으로 나누어져 있으며, 각 칸은 특정한 크기의 금이 들어 있습니다. 채굴자는 첫 번째 열부터 출발하여 금을 캐기 시작합니다. 맨 처음에는 첫 번째 열의 어느 행에서든 출발할 수 있습니다. 이후에 m번에 걸쳐서 매번 오른쪽 위, 오른쪽, 오른쪽 아래 3가지 중 하나의 위치로 이동해야 합니다. 결과적으로 채굴자가 얻을 수 있는 금의 최대 크기를 출력하는 프로그램을 작성하세요.

만약 다음과 같이 3×4 크기의 금광이 존재한다고 가정합시다.

1	3	3	2
2	1	4	1
0	6	4	7

가장 왼쪽 위의 위치를 (1, 1), 가장 오른쪽 아래의 위치를 (n, m)이라고 할 때, 위 예시에서는 (2, 1) → (3, 2) → (3, 3) → (3, 4)의 위치로 이동하면 총 19만큼의 금을 채굴할 수 있으며, 이때의 값이 최댓값입니다.

입력 조건
- 첫째 줄에 테스트 케이스 T가 입력됩니다. ($1 \le T \le 1000$)
- 매 테스트 케이스 첫째 줄에 n과 m이 공백으로 구분되어 입력됩니다. ($1 \le n, m \le 20$) 둘째 줄에 $n \times m$개의 위치에 매장된 금의 개수가 공백으로 구분되어 입력됩니다. (0 ≤ 각 위치에 매장된 금의 개수 ≤ 100)

출력 조건
- 테스트 케이스마다 채굴자가 얻을 수 있는 금의 최대 크기를 출력합니다. 각 테스트 케이스는 줄 바꿈을 이용해 구분합니다.

입력 예시

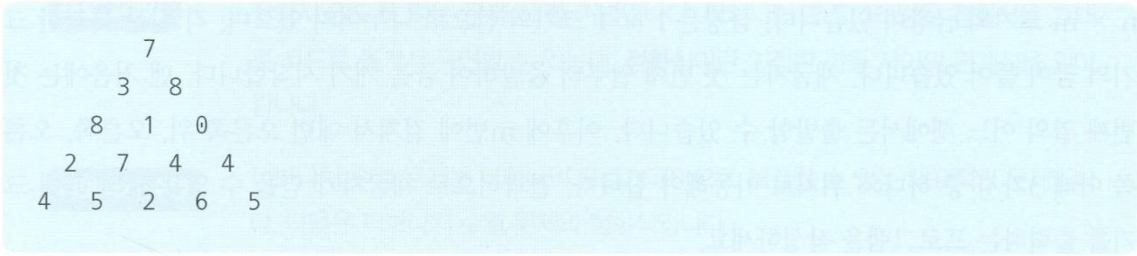
```
2
3 4
1 3 3 2 2 1 4 1 0 6 4 7
4 4
1 3 1 5 2 2 4 1 5 0 2 3 0 6 1 2
```

출력 예시

```
19
16
```

Q 32 정수 삼각형

난이도 ●◐○ | 풀이 시간 30분 | 시간 제한 2초 | 메모리 제한 128MB | 기출 IOI 1994년도
링크 https://www.acmicpc.net/problem/1932



위 그림은 크기가 5인 정수 삼각형의 한 모습입니다.

맨 위층 7부터 시작해서 아래에 있는 수 중 하나를 선택하여 아래층으로 내려올 때, 이제까지 선택된 수의 합이 최대가 되는 경로를 구하는 프로그램을 작성하세요. 아래층에 있는 수는 현재 층에서 선택된 수의 대각선 왼쪽 또는 대각선 오른쪽에 있는 것 중에서만 선택할 수 있습니다.

삼각형의 크기는 1 이상 500 이하입니다. 삼각형을 이루고 있는 각 수는 모두 정수이며, 그 값의 범위는 0 이상 9999 이하입니다.

입력 조건
- 첫째 줄에 삼각형의 크기 n(1 ≤ n ≤ 500)이 주어지고, 둘째 줄부터 n + 1번째 줄까지 정수 삼각형이 주어집니다.

출력 조건
- 첫째 줄에 합이 최대가 되는 경로에 있는 수의 합을 출력합니다.

입력 예시

```
5
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

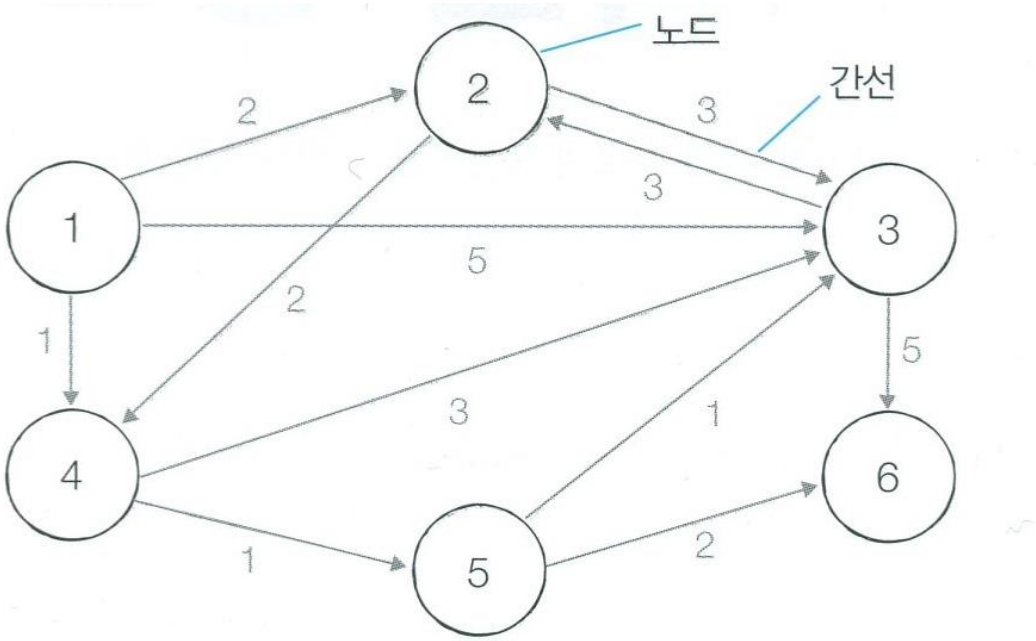
출력 예시

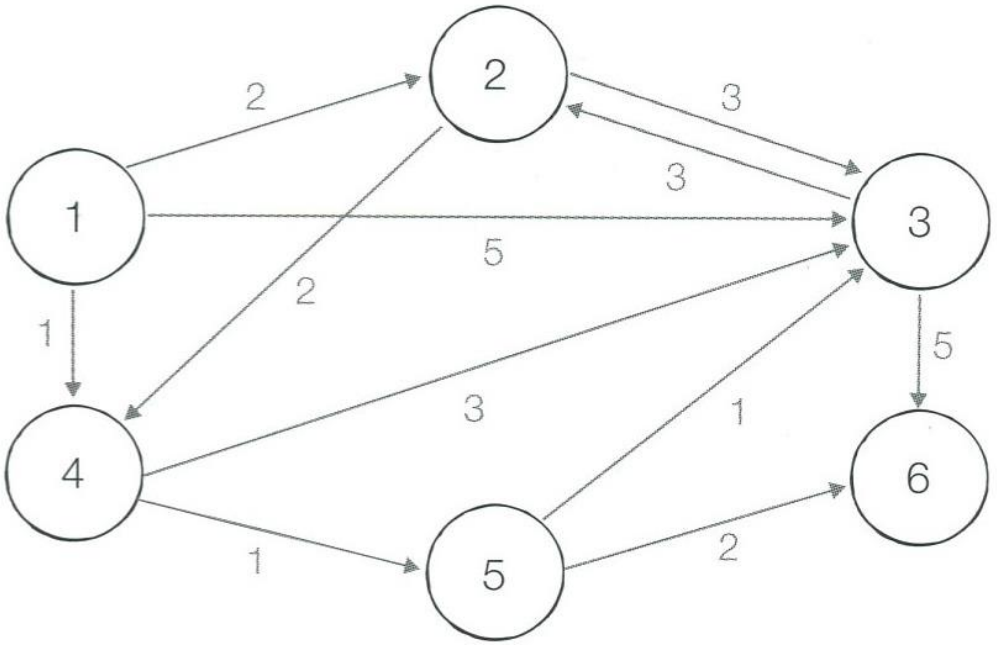
```
30
```

그래프 (2) Chapter 09 최단경로

1 가장 빠른 길 찾기

- 가장 빠르게 도달하는 방법
  - 최단 경로 (Shortest Path) 알고리즘



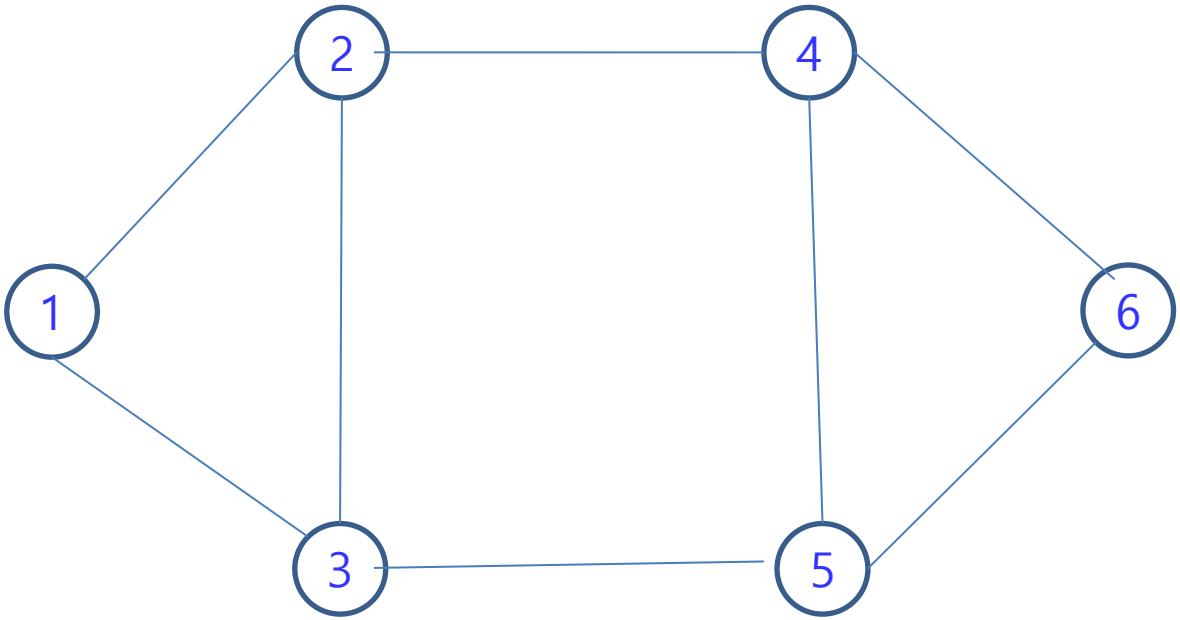

1
2
3
4
5
6
2
3
3
5
2
1
5
3
1
1
2

- 다익스트라 (Dijkstra) 최단 경로 알고리즘
  - 특정 노드에서 출발하여 다른 모든 노드로 가는 최단경로를 계산
  - Greedy 알고리즘의 일종 - 매 상황에서 가장 비용이 적은 노드를 선택하고 그 과정을 반복

    - Whenever we select the shortest path, we try to relax other vertices

      relaxation

      if (d[u] + c[u,v] < d[v]) then d[v] = d[u] + c[u,v]


2
4
1
6
3
5

- 9-1.py 간단한 다익스트라 알고리즘 소스코드

```
import sys
input = sys.stdin.readline
INF = int(1e9) # 무한을 의미하는 값으로 10억을 설정

# 노드의 개수, 간선의 개수를 입력받기
n, m = map(int, input().split())
# 시작 노드 번호를 입력받기
start = int(input())
# 각 노드에 연결되어 있는 노드에 대한 정보를 담는 리스트를 만들기
graph = [[] for i in range(n + 1)]
# 방문한 적이 있는지 체크하는 목적의 리스트를 만들기
visited = [False] * (n + 1)
# 최단 거리 테이블을 모두 무한으로 초기화
distance = [INF] * (n + 1)

# 모든 간선 정보를 입력받기
for _ in range(m):
    a, b, c = map(int, input().split())
    # a번 노드에서 b번 노드로 가는 비용이 c라는 의미
    graph[a].append((b, c))

# 방문하지 않은 노드 중에서, 가장 최단 거리가 짧은 노드의 번호를 반환
def get_smallest_node():
    min_value = INF
    index = 0 # 가장 최단 거리가 짧은 노드(인덱스)
    for i in range(1, n + 1):
        if distance[i] < min_value and not visited[i]:
            min_value = distance[i]
            index = i
    return index
```

```
def dijkstra(start):
    # 시작 노드에 대해서 초기화
    distance[start] = 0
    visited[start] = True
    for j in graph[start]:
        distance[j[0]] = j[1]
    # 시작 노드를 제외한 전체 n - 1개의 노드에 대해 반복
    for i in range(n - 1):
        # 현재 최단 거리가 가장 짧은 노드를 꺼내서, 방문 처리
        now = get_smallest_node()
        visited[now] = True
        # 현재 노드와 연결된 다른 노드를 확인
        for j in graph[now]:
            cost = distance[now] + j[1]
            # 현재 노드를 거쳐서 다른 노드로 이동하는 거리가 더 짧은 경우
            if cost < distance[j[0]]:
                distance[j[0]] = cost

# 다익스트라 알고리즘을 수행
dijkstra(start)

# 모든 노드로 가기 위한 최단 거리를 출력
for i in range(1, n + 1):
    # 도달할 수 없는 경우, 무한(INFINITY)이라고 출력
    if distance[i] == INF:
        print("INFINITY")
    # 도달할 수 있는 경우 거리를 출력
    else:
        print(distance[i])
```

- 9-1.cpp

```
#include <bits/stdc++.h>
#define INF 1e9 // 무한을 의미하는 값으로 10억을 설정

using namespace std;

// 노드의 개수(N), 간선의 개수(M), 시작 노드 번호(Start)
// 노드의 개수는 최대 100,000개라고 가정
int n, m, start;
// 각 노드에 연결되어 있는 노드에 대한 정보를 담는 배열
vector<pair<int, int> > graph[100001];
// 방문한 적이 있는지 체크하는 목적의 배열 만들기
bool visited[100001];
// 최단 거리 테이블 만들기
int d[100001];

// 방문하지 않은 노드 중에서, 가장 최단 거리가 짧은 노드의 번호를 반환
int getSmallestNode() {
    int min_value = INF;
    int index = 0; // 가장 최단 거리가 짧은 노드(인덱스)
    for (int i = 1; i <= n; i++) {
        if (d[i] < min_value && !visited[i]) {
            min_value = d[i];
            index = i;
        }
    }
    return index;
}
```

```
void dijkstra(int start) {
    // 시작 노드에 대해서 초기화
    d[start] = 0;
    visited[start] = true;
    for (int j = 0; j < graph[start].size(); j++) {
        d[graph[start][j].first] = graph[start][j].second;
    }
    // 시작 노드를 제외한 전체 n - 1개의 노드에 대해 반복
    for (int i = 0; i < n - 1; i++) {
        // 현재 최단 거리가 가장 짧은 노드를 꺼내서, 방문 처리
        int now = getSmallestNode();
        visited[now] = true;
        // 현재 노드와 연결된 다른 노드를 확인
        for (int j = 0; j < graph[now].size(); j++) {
            int cost = d[now] + graph[now][j].second;
            // 현재 노드를 거쳐서 다른 노드로 이동하는 거리가 더 짧은 경우
            if (cost < d[graph[now][j].first]) {
                d[graph[now][j].first] = cost;
            }
        }
    }
}
```

```
int main(void) {
    cin >> n >> m >> start;

    // 모든 간선 정보를 입력받기
    for (int i = 0; i < m; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        // a번 노드에서 b번 노드로 가는 비용이 c라는 의미
        graph[a].push_back({b, c});
    }

    // 최단 거리 테이블을 모두 무한으로 초기화
    fill_n(d, 100001, INF);

    // 다익스트라 알고리즘을 수행
    dijkstra(start);

    // 모든 노드로 가기 위한 최단 거리를 출력
    for (int i = 1; i <= n; i++) {
        // 도달할 수 없는 경우, 무한(INFINITY)이라고 출력
        if (d[i] == INF) {
            cout << "INFINITY" << '₩n';
        }
        // 도달할 수 있는 경우 거리를 출력
        else {
            cout << d[i] << '₩n';
        }
    }
}
```

- 개선된 다익스트라 알고리즘 소스코드
  - Heap (즉, Priority Queue)이용하여 가장 거리가 짧은 노드를 찾음
  - O(V2) → O(E log V)

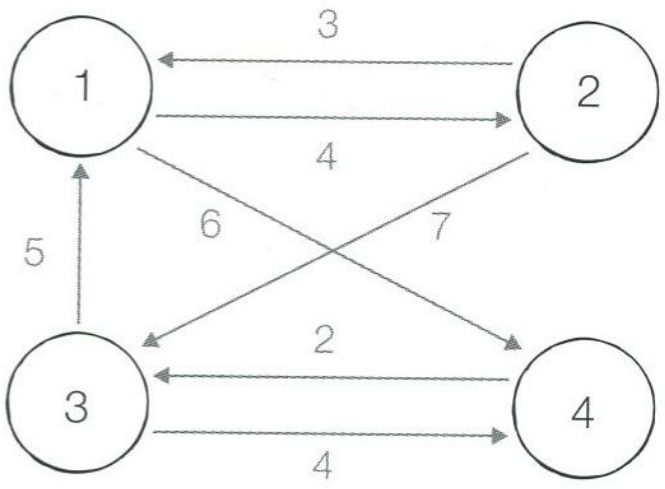
- 9-2.py

- 9-2.cpp

- 플로이드 워셜 (Floyd-Warshall) 알고리즘
  - 모든 노드에서 다른 모든 노드까지의 최단 경로를 모두 계산
  - Floyd-Warshll 알고리즘은 다익스트라 알고리즘과 마찬가지로 단계별로 거쳐가는 노드를 기준으로 알고리즘 수행
    - A → b의 최단거리보다 a → k → b로 가는 거리가 더 짧은지 검사

$$D_{ab} = min(D_{ab}, D_{ak} + D_{kb})$$

- DP 유형



step 0

출발 \ 도착	1번	2번	3번	4번
1번	0	4	무한	6
2번	3	0	7	무한
3번	5	무한	0	4
4번	무한	무한	2	0

step 1 [step 1]에서는 단순히 1번 노드를 거쳐 가는 경우를 고려한다. 이때는 정확히 다음과 같이 $6 = {}_3P_2$가지 경우에 대해서만 고민하면 된다. 2차원 테이블에서는 다른 색으로 칠해 놓았는데, 계산해야 할 값들은 구체적으로 다음과 같다.

$$D_{23} = min(D_{23}, D_{21} + D_{13})$$
$$D_{24} = min(D_{24}, D_{21} + D_{14})$$
$$D_{32} = min(D_{32}, D_{31} + D_{12})$$
$$D_{34} = min(D_{34}, D_{31} + D_{14})$$
$$D_{42} = min(D_{42}, D_{41} + D_{12})$$
$$D_{43} = min(D_{43}, D_{41} + D_{13})$$

0	4	무한	6
3	0		
5		0	
무한			0

- Dijkstra's
  - shortest path from one node to all nodes
- Bellman-Ford
  - shortest path from one node to all nodes, negative edges allowed
- Floyd-Warshall
  - shortest path from all pairs of vertices, negative edges allowed

- 9-3.py
- 9-3.cpp

2 [실전 문제] 미래 도시

난이도 ●●○ | 풀이 시간 40분 | 시간 제한 1초 | 메모리 제한 128MB | 기출 M 기업 코딩 테스트

방문 판매원 A는 많은 회사가 모여 있는 공중 미래 도시에 있다. 공중 미래 도시에는 1번부터 N번까지의 회사가 있는데 특정 회사끼리는 서로 도로를 통해 연결되어 있다. 방문 판매원 A는 현재 1번 회사에 위치해 있으며, X번 회사에 방문해 물건을 판매하고자 한다.

공중 미래 도시에서 특정 회사에 도착하기 위한 방법은 회사끼리 연결되어 있는 도로를 이용하는 방법이 유일하다. 또한 연결된 2개의 회사는 양방향으로 이동할 수 있다. 공중 미래 도시에서의 도로는 마하의 속도로 사람을 이동시켜주기 때문에 특정 회사와 다른 회사가 도로로 연결되어 있다면, 정확히 1만큼의 시간으로 이동할 수 있다.

또한 오늘 방문 판매원 A는 기대하던 소개팅에도 참석하고자 한다. 소개팅의 상대는 K번 회사에 존재한다. 방문 판매원 A는 X번 회사에 가서 물건을 판매하기 전에 먼저 소개팅 상대의 회사에 찾아가서 함께 커피를 마실 예정이다. 따라서 방문 판매원 A는 1번 회사에서 출발하여 K번 회사를 방문한 뒤에 X번 회사로 가는 것이 목표다. 이때 방문 판매원 A는 가능한 한 빠르게 이동하고자 한다. 방문 판매원이 회사 사이를 이동하게 되는 최소 시간을 계산하는 프로그램을 작성하시오. 이때 소개팅의 상대방과 커피를 마시는 시간 등은 고려하지 않는다고 가정한다. 예를 들어 N = 5, X = 4, K = 5이고 회사 간 도로가 7개면서 각 도로가 다음과 같이 연결되어 있을 때를 가정할 수 있다.

(1번, 2번), (1번, 3번), (1번, 4번), (2번, 4번), (3번, 4번), (3번, 5번), (4번, 5번)

이때 방문 판매원 A가 최종적으로 4번 회사에 가는 경로를 (1번 – 3번 – 5번 – 4번)으로 설정하면, 소개팅에도 참석할 수 있으면서 총 3만큼의 시간으로 이동할 수 있다. 따라서 이 경우 최소 이동 시간은 3이다.

입력 조건

- 첫째 줄에 전체 회사의 개수 N과 경로의 개수 M이 공백으로 구분되어 차례대로 주어진다. ($1 \le N, M \le 100$)
- 둘째 줄부터 M + 1번째 줄에는 연결된 두 회사의 번호가 공백으로 구분되어 주어진다.
- M + 2번째 줄에는 X와 K가 공백으로 구분되어 차례대로 주어진다. ($1 \le K \le 100$)

출력 조건

- 첫째 줄에 방문 판매원 A가 K번 회사를 거쳐 X번 회사로 가는 최소 이동 시간을 출력한다.
- 만약 X번 회사에 도달할 수 없다면 −1을 출력한다.

입력 예시 1

```
5 7
1 2
1 3
1 4
2 4
3 4
3 5
4 5
4 5
```

출력 예시 1

```
3
```

입력 예시 2

```
4 2
1 3
2 4
3 4
```

출력 예시 2

```
-1
```

3 [실전 문제] 전보

난이도 ●●● | 풀이 시간 60분 | 시간 제한 1초 | 메모리 제한 128MB | 기출 유명 알고리즘 대회

어떤 나라에는 N개의 도시가 있다. 그리고 각 도시는 보내고자 하는 메시지가 있는 경우, 다른 도시로 전보를 보내서 다른 도시로 해당 메시지를 전송할 수 있다. 하지만 X라는 도시에서 Y라는 도시로 전보를 보내고자 한다면, 도시 X에서 Y로 향하는 통로가 설치되어 있어야 한다. 예를 들어 X에서 Y로 향하는 통로는 있지만, Y에서 X로 향하는 통로가 없다면 Y는 X로 메시지를 보낼 수 없다. 또한 통로를 거쳐 메시지를 보낼 때는 일정 시간이 소요된다.

어느 날 C라는 도시에서 위급 상황이 발생했다. 그래서 최대한 많은 도시로 메시지를 보내고자 한다. 메시지는 도시 C에서 출발하여 각 도시 사이에 설치된 통로를 거쳐, 최대한 많이 퍼져나갈 것이다. 각 도시의 번호와 통로가 설치되어 있는 정보가 주어졌을 때, 도시 C에서 보낸 메시지를 받게 되는 도시의 개수는 총 몇 개이며 도시들이 모두 메시지를 받는 데까지 걸리는 시간은 얼마인지 계산하는 프로그램을 작성하시오.

입력 조건

- 첫째 줄에 도시의 개수 N, 통로의 개수 M, 메시지를 보내고자 하는 도시 C가 주어진다.
  (1 ≤ N ≤ 30,000, 1 ≤ M ≤ 200,000, 1 ≤ C ≤ N)
- 둘째 줄부터 M + 1번째 줄에 걸쳐서 통로에 대한 정보 X, Y, Z가 주어진다. 이는 특정 도시 X에서 다른 특정 도시 Y로 이어지는 통로가 있으며, 메시지가 전달되는 시간이 Z라는 의미다.
  (1 ≤ X, Y ≤ N, 1 ≤ Z ≤ 1,000)

출력 조건

- 첫째 줄에 도시 C에서 보낸 메시지를 받는 도시의 총 개수와 총 걸리는 시간을 공백으로 구분하여 출력한다.

입력 예시

```
3 2 1
1 2 4
1 3 2
```

출력 예시

```
2 4
```