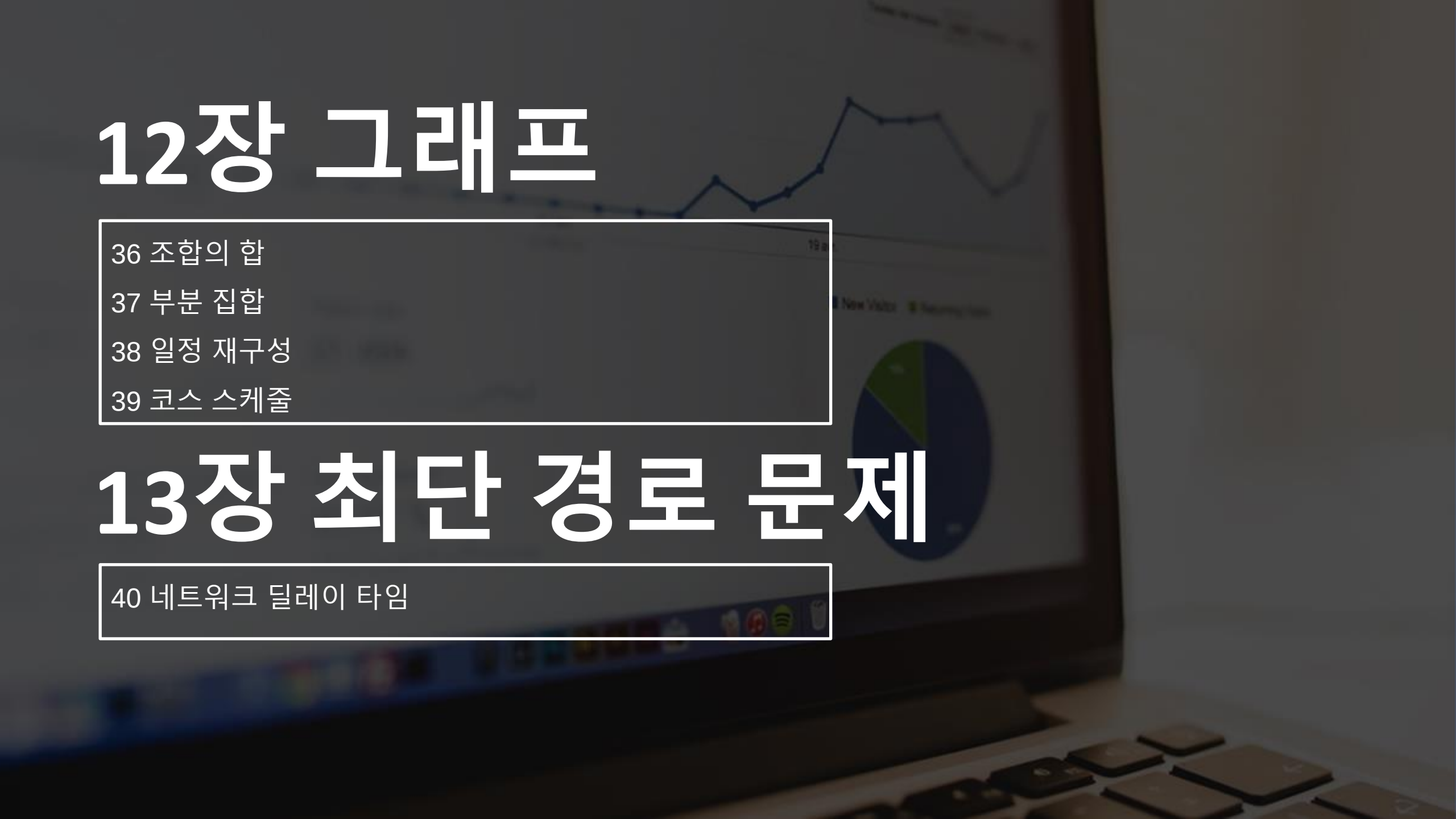
# 창의적 문제 해결
# 14주차

공주대학교 컴퓨터공학과

김영부



# 12장 그래프

36 조합의 합

37 부분 집합

38 일정 재구성

39 코스 스케줄

# 13장 최단 경로 문제

40 네트워크 딜레이 타임

36번

# 조합의 합

# 36 조합의 합

**39. Combination Sum (https://leetcode.com/problems/combination-sum/)**

## 39. Combination Sum

Medium 7996 190 Add to List Share

- DFS로 중복 조합 그래프 탐색

Given an array of **distinct** integers `candidates` and a target integer `target`, return *a list of all **unique combinations** of* `candidates` *where the chosen numbers sum to* `target`. You may return the combinations in **any order**.

The **same** number may be chosen from `candidates` an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

It is **guaranteed** that the number of unique combinations that sum up to `target` is less than `150` combinations for the given input.

**Example 1:**

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.
```

# 36 조합의 합

- DFS로 중복 조합 그래프 탐색

```python
from typing import List


class Solution:
    def combinationSum(self, candidates: List[int], target: int) \
            -> List[List[int]]:
        result = []

        def dfs(csum, index, path):
            # 종료 조건
            if csum < 0:
                return
            if csum == 0:
                result.append(path)
                return

            # 자신 부터 하위 원소 까지의 나열 재귀 호출
            for i in range(index, len(candidates)):
                dfs(csum - candidates[i], i, path + [candidates[i]])

        dfs(target, 0, [])
        return result
```

→ target을 만족하지 않음

→ target을 만족

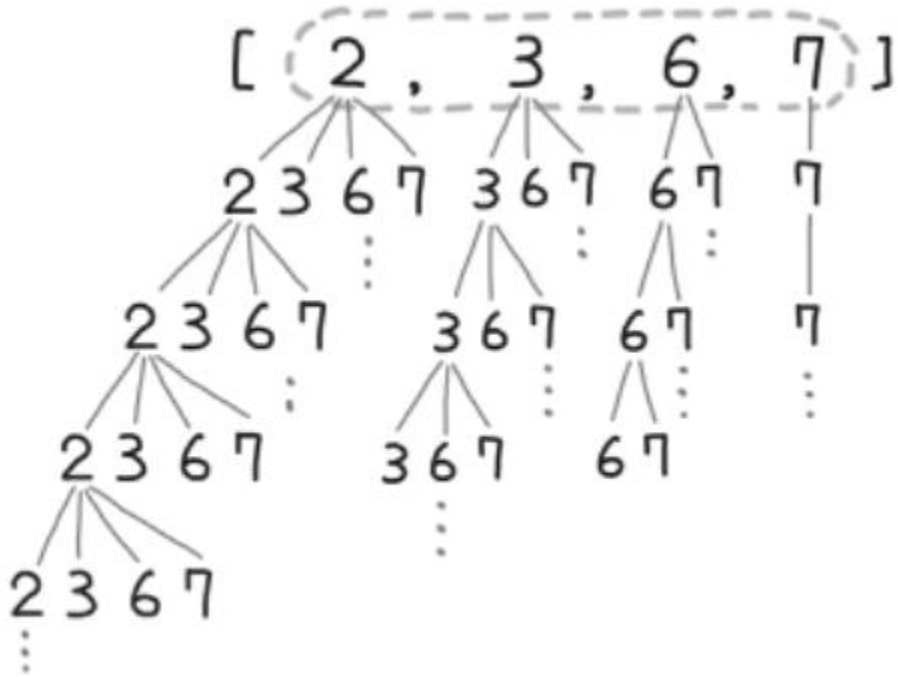



그림 12-15 입력값의 중복 조합을 그래프 형태로 나열

7-2-2-2-2=-1 X

7-2-2-3=0 O

# 36 조합의 합

- DFS로 중복 조합 그래프 탐색

```python
from typing import List


class Solution:
    def combinationSum(self, candidates: List[int], target: int) \
            -> List[List[int]]:
        result = []

        def dfs(csum, index, path):
            # 종료 조건
            if csum < 0:
                return
            if csum == 0:
                result.append(path)
                return

            # 자신 부터 하위 원소 까지의 나열 재귀 호출
            for i in range(index, len(candidates)):
                dfs(csum - candidates[i], i, path + [candidates[i]])

        dfs(target, 0, [])
        return result
```

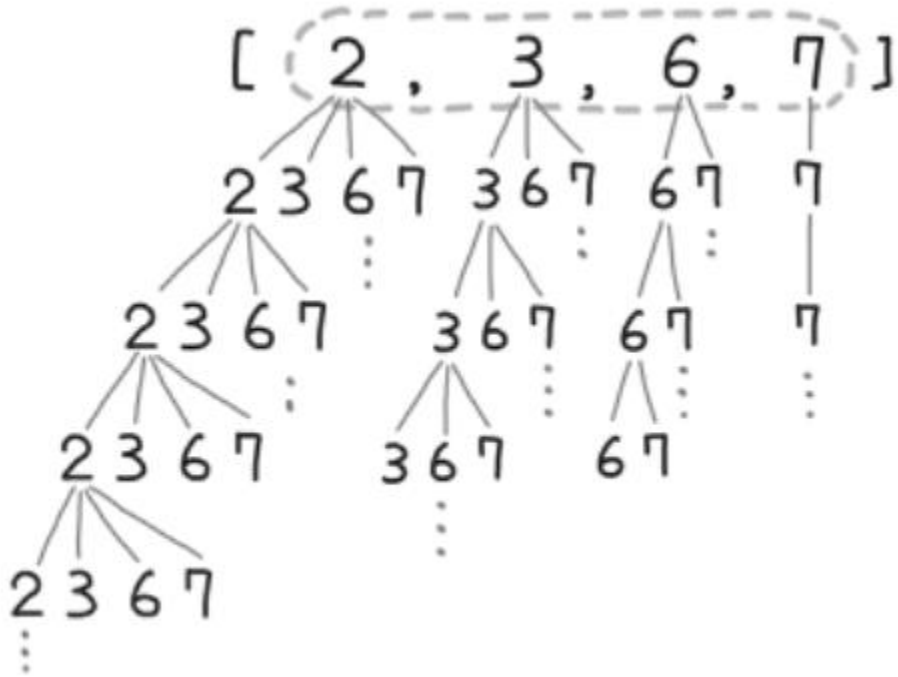



그림 12-15 입력값의 중복 조합을 그래프 형태로 나열

37번

# 부분 집합

# 37 부분 집합

**78. Subsets (https://leetcode.com/problems/subsets/)**

- 트리의 모든 DFS 결과

## 78. Subsets

Medium 7559 126 Add to List Share

Given an integer array `nums` of **unique** elements, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

**Example 1:**

```
Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

이것도 출력해야 함

# 37 부분 집합

- 트리의 모든 DFS 결과

```python
from typing import List


class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        result = []

        def dfs(index, path):
            # 매 번 결과 추가
            result.append(path)

            # 경로를 만들면서 DFS
            for i in range(index, len(nums)):
                dfs(i + 1, path + [nums[i]])

        dfs(0, [])
        return result
```

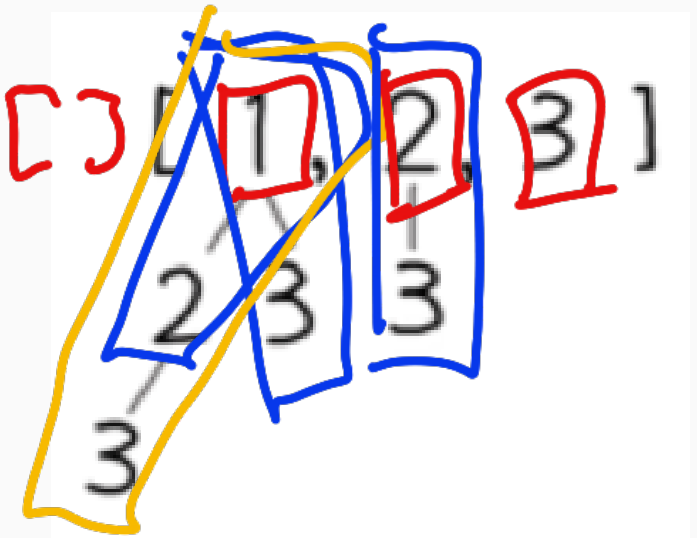



그림 12-16 부분 집합 트리

# 37 부분 집합

- 트리의 모든 DFS 결과

```python
from typing import List


class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        result = []

        def dfs(index, path):
            # 매 번 결과 추가
            result.append(path)

            # 경로를 만들면서 DFS
            for i in range(index, len(nums)):
                dfs(i + 1, path + [nums[i]])

        dfs(0, [])
        return result
```

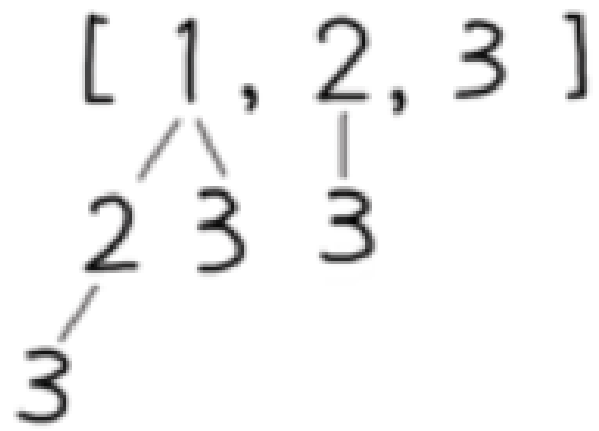



그림 12-16 부분 집합 트리

38번

# 일정 재구성

# 38 일정 재구성

**332. Reconstruct Itinerary (**https://leetcode.com/problems/reconstruct-itinerary/**)**

## 332. Reconstruct Itinerary

Hard 3398 1504 Add to List Share


You are given a list of airline `tickets` where `tickets[i] = [from_i, to_i]` represent the departure and the arrival airports of one flight. Reconstruct the itinerary in order and return it.

All of the tickets belong to a man who departs from `"JFK"`, thus, the itinerary must begin with `"JFK"`. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string.

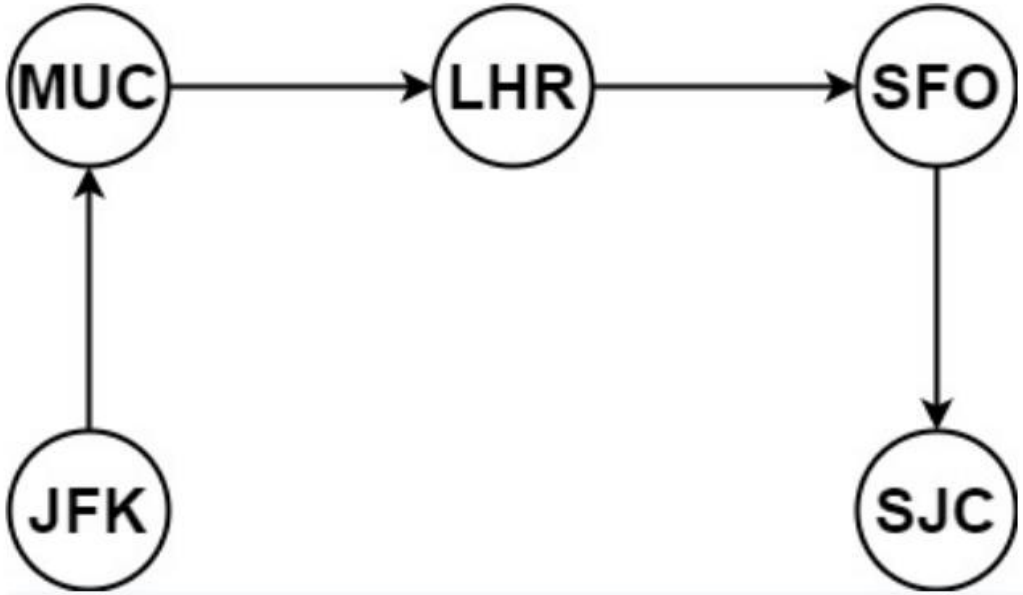
- For example, the itinerary `["JFK", "LGA"]` has a smaller lexical order than `["JFK", "LGB"]`.

You may assume all tickets form at least one valid itinerary. You must use all the tickets once and only once.

# 38 일정 재구성

**332. Reconstruct Itinerary (**https://leetcode.com/problems/reconstruct-itinerary/**)**

**Example 1:**




```
Input: tickets = [["MUC","LHR"],["JFK","MUC"],["SFO","SJC"],["LHR","SFO"]]
Output: ["JFK","MUC","LHR","SFO","SJC"]
```

M → L 2

J → M 1

SF → SJ 4

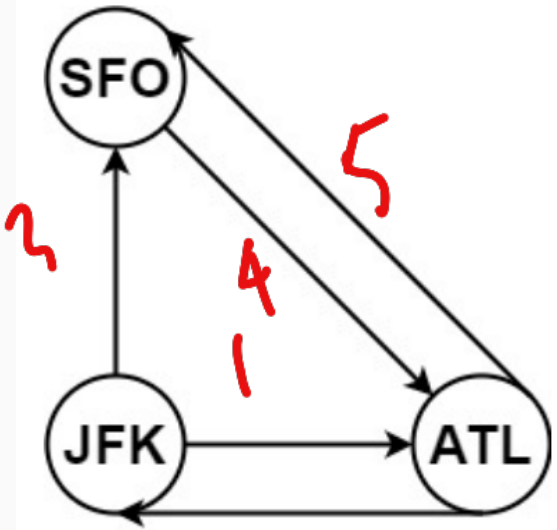
L → SF 3

# 38 일정 재구성

**332. Reconstruct Itinerary (**https://leetcode.com/problems/reconstruct-itinerary/**)**

- DFS로 일정 그래프 구성
- 스택 연산으로 큐 연산 최적화 시도
- 일정 그래프 반복

**Example 2:**




```
Input: tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]
Output: ["JFK","ATL","JFK","SFO","ATL","SFO"]
Explanation: Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SFO"] but it is larger in lexical
order.
```

# 38 일정 재구성

- DFS로 일정 그래프 구성

[["MUC","LHR"],["JFK","MUC"],["SFO","SJC"],["LHR","SFO"]]

```python
import collections
from typing import List


class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        graph = collections.defaultdict(list)
        # 그래프 순서대로 구성
        for a, b in sorted(tickets):
            graph[a].append(b)

        route = []

        def dfs(a):
            # 첫 번째 값을 읽어 어휘순 방문
            while graph[a]:
                dfs(graph[a].pop(0))
            route.append(a)

        dfs('JFK')
        # 다시 뒤집어 어휘순 결과로
        return route[::-1]
```

[["JFK","MUC"],["LHR","SFO"],["MUC","LHR"],["SFO","SJC"]]

에러 발생방지 디폴트딕셔너리

오름차순 J L M S 순

[["JFK" : "MUC"]
["LHR" : "SFO"]
["MUC" : "LHR"]
["SFO" : "SJC"]]

↳ 딕셔너리 형태로 변환

→ 결과값 리턴 변수

→ + reverse=True

→ pop(0) 로 바꾸면 정방향 출력됨

→ 시작지점은 JFK

# 38 일정 재구성

- DFS로 일정 그래프 구성

[["MUC","LHR"],["JFK","MUC"],["SFO","SJC"],["LHR","SFO"]]

```python
import collections
from typing import List


class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        graph = collections.defaultdict(list)
        # 그래프 순서대로 구성
        for a, b in sorted(tickets):
            graph[a].append(b)

        route = []

        def dfs(a):
            # 첫 번째 값을 읽어 어휘순 방문
            while graph[a]:
                dfs(graph[a].pop(0))
            route.append(a)

        dfs('JFK')
        # 다시 뒤집어 어휘순 결과로
        return route[::-1]
```

[["JFK“ : "MUC"]
["LHR“ : "SFO"]
["MUC“ : "LHR"]
["SFO“ : "SJC"]]

[["JFK","MUC"],["LHR","SFO"],["MUC","LHR"],["SFO","SJC"]]

# 38 일정 재구성

- DFS로 일정 그래프 구성

[["MUC","LHR"],["JFK","MUC"],["SFO","SJC"],["LHR","SFO"]]

```python
import collections
from typing import List


class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        graph = collections.defaultdict(list)
        # 그래프 순서대로 구성
        for a, b in sorted(tickets):
            graph[a].append(b)

        route = []

        def dfs(a):
            # 첫 번째 값을 읽어 어휘순 방문
            while graph[a]:
                dfs(graph[a].pop(0))
            route.append(a)

        dfs('JFK')
        # 다시 뒤집어 어휘순 결과로
        return route[::-1]
```

[["JFK“ : "MUC"]
["LHR“ : "SFO"]
["MUC“ : "LHR"]
["SFO“ : "SJC"]]

[["JFK","MUC"],["LHR","SFO"],["MUC","LHR"],["SFO","SJC"]]

# 38 일정 재구성

- 스택 연산으로 큐 연산 최적화 시도

```python
import collections
from typing import List


class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List
        graph = collections.defaultdict(list)
        # 그래프 순서대로 구성
        for a, b in sorted(tickets):
            graph[a].append(b)

        route = []

        def dfs(a):
            # 첫 번째 값을 읽어 어휘순 방문
            while graph[a]:
                dfs(graph[a].pop(0))
            route.append(a)

        dfs('JFK')
        # 다시 뒤집어 어휘순 결과로
        return route[::-1]
```

```python
import collections
from typing import List


class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        graph = collections.defaultdict(list)
        # 그래프 뒤집어서 구성
        for a, b in sorted(tickets, reverse=True):
            graph[a].append(b)

        route = []

        def dfs(a):
            # 마지막 값을 읽어 어휘순 방문
            while graph[a]:
                dfs(graph[a].pop())
            route.append(a)

        dfs('JFK')
        # 다시 뒤집어 어휘순 결과로
        return route[::-1]
```

# 38 일정 재구성

- 일정 그래프 반복

[["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]

```python
import collections
from typing import List


class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        graph = collections.defaultdict(list)
        # 그래프 순서대로 구성
        for a, b in sorted(tickets):
            graph[a].append(b)

        route, stack = [], ['JFK']
        while stack:
            # 반복으로 스택을 구성하되 막히는 부분에서 풀어내는 처리
            while graph[stack[-1]]:
                stack.append(graph[stack[-1]].pop(0))
            route.append(stack.pop())

        # 다시 뒤집어 어휘순 결과로
        return route[::-1]
```

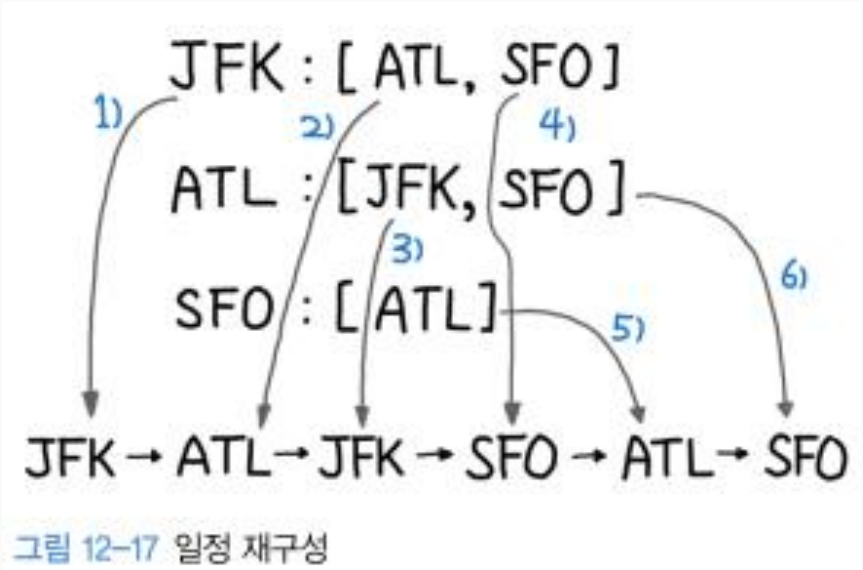



그림 12-17 일정 재구성

39번

# 코스 스케줄

# 39 코스 스케줄

**207. Course Schedule (https://leetcode.com/problems/course-schedule/)**

## 207. Course Schedule

Medium 7641 308 Add to List Share

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [a_i, b_i]` indicates that you **must** take course `b_i` first if you want to take course `a_i`.

이걸하려면 → 이걸 먼저 해야 한다

- For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.

Return `true` if you can finish all courses. Otherwise, return `false`.

# 39 코스 스케줄

**207. Course Schedule (https://leetcode.com/problems/course-schedule/)**

- DFS로 순환 구조 판별
- 가지치기를 이용한 최적화

**Example 1:**

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0. So it is possible.
```

0 -> 1

**Example 2:**

```
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0, and to take course 0 you should also
have finished course 1. So it is impossible.
```

1 -> 0

0 -> 1

X

# 39 코스 스케줄

- DFS로 순환 구조 판별

```python
import collections
from typing import List


class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        graph = collections.defaultdict(list)
        # 그래프 구성
        for x, y in prerequisites:
            graph[x].append(y)

        traced = set()

        def dfs(i):↔

        # 순환 구조 판별
        for x in list(graph):
            if not dfs(x):
                return False

        return True
```

```python
        def dfs(i):
            # 순환 구조이면 False
            if i in traced:
                return False

            traced.add(i)
            for y in graph[i]:
                if not dfs(y):
                    return False
            # 탐색 종료 후 순환 노드 삭제
            traced.remove(i)

            return True
```

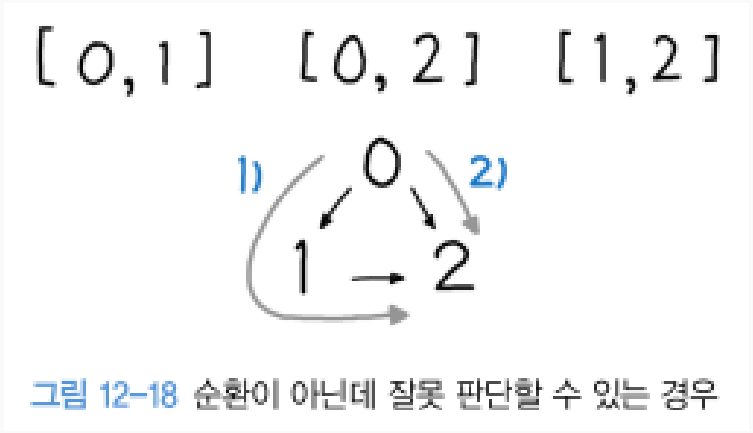



그림 12-18 순환이 아닌데 잘못 판단할 수 있는 경우

# 39 코스 스케줄

- DFS로 순환 구조 판별

```python
import collections
from typing import List


class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        graph = collections.defaultdict(list)
        # 그래프 구성
        for x, y in prerequisites:
            graph[x].append(y)

        traced = set()

        def dfs(i):
            # 순환 구조이면 False
            if i in traced:
                return False

            traced.add(i)
            for y in graph[i]:
                if not dfs(y):
                    return False
            # 탐색 종료 후 순환 노드 삭제
            traced.remove(i)

            return True

        # 순환 구조 판별
        for x in list(graph):
            if not dfs(x):
                return False

        return True
```

[0,1] [0,2] [1,2]

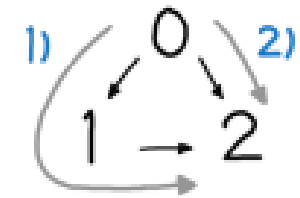



그림 12-18 순환이 아닌데 잘못 판단할 수 있는 경우

이경우에 같을 판단할 수 없도록 방문노드들삭제

# 39 코스 스케줄

- 가지치기를 이용한 최적화

→ 이게 시간 복잡도가 더 낮음

```python
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        graph = collections.defaultdict(list)
        # 그래프 구성
        for x, y in prerequisites:
            graph[x].append(y)

        traced = set()
        visited = set()

        def dfs(i):↔

        # 순환 구조 판별
        for x in list(graph):
            if not dfs(x):
                return False

        return True
```

```python
        def dfs(i):
            # 순환 구조이면 False
            if i in traced:
                return False
            # 이미 방문했던 노드이면 True
            if i in visited:
                return True

            traced.add(i)
            for y in graph[i]:
                if not dfs(y):
                    return False
            # 탐색 종료 후 순환 노드 삭제
            traced.remove(i)
            # 탐색 종료 후 방문 노드 추가
            visited.add(i)

            return True
```

# 13장
# 최단 경로 문제

40 네트워크 딜레이 타임

41 K 경유지 내 가장 저렴한 항공권

# 최단 경로 문제

# 최단 경로 문제

"최단 경로 문제는 각 간선의 가중치 합이 최소가 되는

두 정점 (또는 노드) 사이의 경로를 찾는 문제다"

# 최단 경로 문제

최단 경로는 지도 상의 한 지점에서 다른 지점으로 갈 때 가장 빠른 길을 찾는 것과 비슷한 문제다.

쉽게 말해 내비게이션에서 목적지로 이동할 때, 경로 탐색을 하면 나오는 최적의 경로 문제가 바로 최소 비용이 되는 최단 경로 문제다.

정점Vertex은 교차로에 해당하고 간선Edge은 길에 해당한다. 가중치Weight는 거리나 시간과 같은 이동 비용에 해당한다
.

# 최단 경로 문제

↳ 단순한 알고리즘

다익스트라 알고리즘은 임의의 정점을 출발 집합에 더할 때, 그 정점까지의 최단거리는 계산이 끝났다는 확신을 갖고 더한다.

최단거리의 집합

만일 이후에 더 짧은 경로가 존재한다면 다익스트라 알고리즘의 논리적 기반이 무너진다.

이때는 모두 값을 더해서 양수로 변환하는 방법이 있으며, 이마저도 어렵다면 벨만-포드Bellman-Ford 알고리즘 같은, 음수 가중치를계산할 수 있는 다른 알고리즘을 사용해야 한다.

같은 이유로 최장 거리를 구하는 데에는 다익스트라 알고리즘을 사용할 수 없다.

**참고** 오컴의 면도날

과학자들이 쓰는 용어 중에 '오컴의 면도날'이라는 표현이 있다. 14세기 영국의 논리학자였던 오컴(Ockham)의 이름에서 탄생한 이 용어는 어떤 현상을 설명할 때 필요 이상의 가정과 개념들은 면도날로 베어낼 필요가 있다는 권고로 쓰인다. 사고의 절약을 요구하는 이 원리는 과학 분야에서 널리 응용되는 일반적인 지침이다.[5] 대개 과학 분야에서는, 같은 것을 설명할 때 복잡한 것보다는 단순한 과학적 설명을 선호하고, 적은 과정으로 자연 현상을 많이 설명할 수 있는 이론을 복잡한 이론보다 더 선호한다. 최소 비용, 최대 만족이라는 경제적 원리가 여기에도 적용된다.[6]

# 최단 경로 문제

다익스트라의 최초 구현에서는 시간 복잡도가 $O(V^2)$였으나

현재는 너비 우선 탐색(BFS) 시 가장 가까운 순서를 찾을 때

우선순위 큐를 적용하여 이 경우 시간 복잡도는 O((V+E)log V),

모든 정점이 출발지에서 도달이 가능하다면 최종적으로 O(E log V)가 된다.

이제 최단 경로를 찾는 문제를 직접 풀어보면서 다익스트라 알고리즘의 구체적인 구현과 응용하는 방법을 한번 살펴보자.

40번

# 네트워크 딜레이 타임

# 40 네트워크 딜레이 타임

**743. Network Delay Time (https://leetcode.com/problems/network-delay-time/)**

## 743. Network Delay Time

Medium 3351 263 Add to List Share

- 다익스트라 알고리즘 구현

You are given a network of `n` nodes, labeled from `1` to `n`. You are also given `times`, a list of travel times as directed edges `times[i] = (u_i, v_i, w_i)`, where `u_i` is the source node, `v_i` is the target node, and `w_i` is the time it takes for a signal to travel from source to target.

We will send a signal from a given node `k`. Return the time it takes for all the `n` nodes to receive the signal. If it is impossible for all the `n` nodes to receive the signal, return `-1`.

u와 v사이의 w 시간

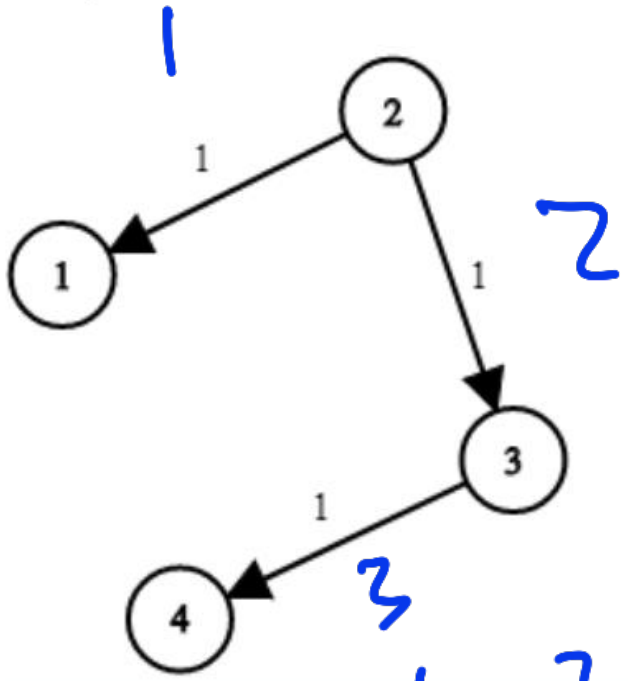
모든정점X

# 40 네트워크 딜레이 타임

**743. Network Delay Time (https://leetcode.com/problems/network-delay-time/)**

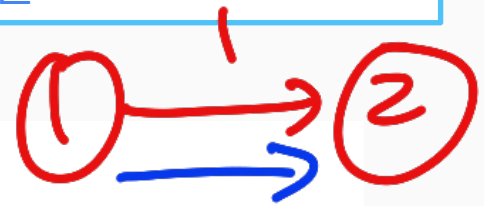
- 다익스트라 알고리즘 구현

**Example 1:**




```
Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
Output: 2
```

최대시간

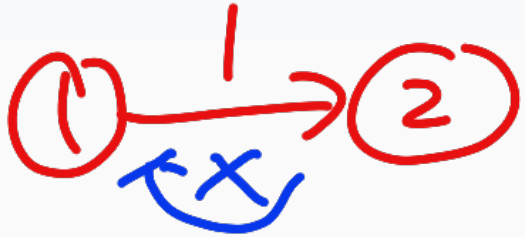



**Example 2:**

```
Input: times = [[1,2,1]], n = 2, k = 1
Output: 1
```

**Example 3:**

```
Input: times = [[1,2,1]], n = 2, k = 2
Output: -1
```



# 40 네트워크 딜레이 타임

- 다익스트라 알고리즘 구현

```python
import collections
import heapq
from typing import List


class Solution:
    def networkDelayTime(self, times: List[List[int]], N: int, K: int) -> int:
        graph = collections.defaultdict(list)
        # 그래프 인접 리스트 구성
        for u, v, w in times:
            graph[u].append((v, w))

        # 큐 변수: [(소요 시간, 정점)]
        Q = [(0, K)]
        dist = collections.defaultdict(int)

        # 우선 순위 큐 최소값 기준으로 정점까지 최단 경로 삽입
        while Q:
            time, node = heapq.heappop(Q)
            if node not in dist:
                dist[node] = time
                for v, w in graph[node]:
                    alt = time + w
                    heapq.heappush(Q, (alt, v))

        # 모든 노드 최단 경로 존재 여부 판별
        if len(dist) == N:
            return max(dist.values())
        return -1
```

연결노드, 시간 과정

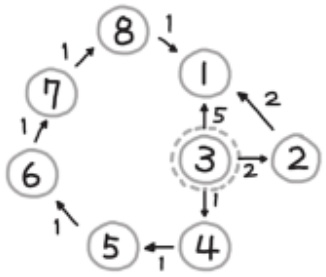
→ 개수 비교를 이용해 확인

[3 1 5], [3 2 2], [2 1 2], [3 4 1], [4 5 1]

[5 6 1], [6 7 1], [7 8 1], [8 1 1]

N=8, K=3

1)




2)

| Q time/node | dist node/time |
|---|---|
| 0 3 | 3 0 |
| ~~5 1~~ | 4 1 |
| 2 2 | 2 2 |
| 1 4 | 5 2 |
| 2 5 | 6 3 |
| 4 1 | 1 4 |
| 3 6 | 7 4 |
| 4 7 | 8 5 |
| 5 8 | |
| ~~6 1~~ | |

그림 13-1 최단 경로를 찾아 나가는 과정 도식화

# 40 네트워크 딜레이 타임

- 다익스트라 알고리즘 구현

```
13      # 큐 변수: [(소요 시간, 정점)]
14      Q = [(0, K)]
15      dist = collections.defaultdict(int)
16
17      # 우선 순위 큐 최소값 기준으로 정점까지 최단 경로 삽입
18      while Q:
19          time, node = heapq.heappop(Q)
20          if node not in dist:
21              dist[node] = time
22              for v, w in graph[node]:
23                  alt = time + w
24                  heapq.heappush(Q, (alt, v))
```

[3 1 5], [3 2 2], [2 1 2], [3 4 1], [4 5 1]

[5 6 1], [6 7 1], [7 8 1], [8 1 1]

N=8, K=3

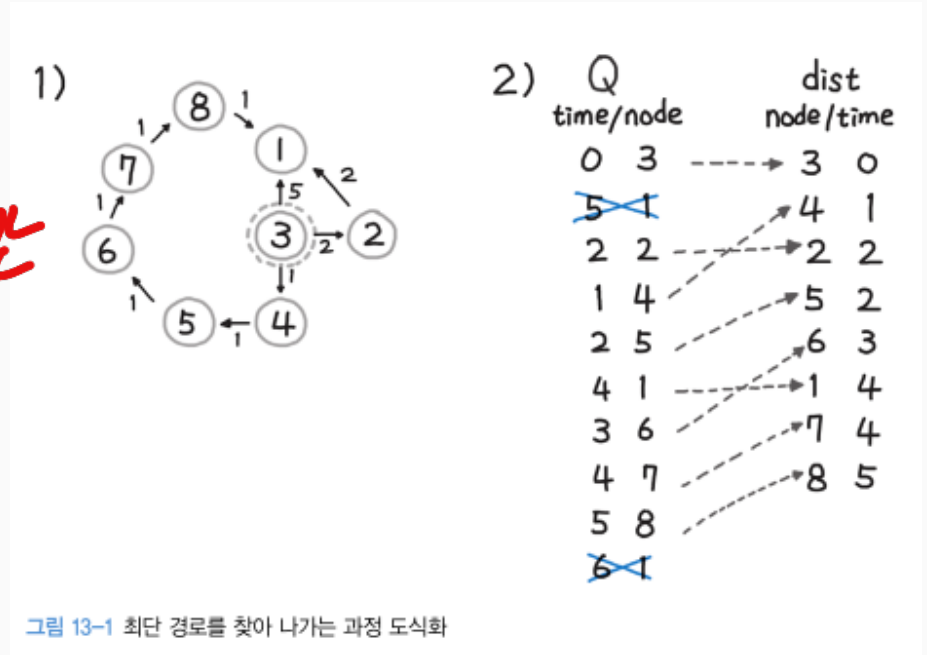



그림 13-1 최단 경로를 찾아 나가는 과정 도식화

# 40 네트워크 딜레이 타임

- 다익스트라 알고리즘 구현

```python
import collections
import heapq
from typing import List


class Solution:
    def networkDelayTime(self, times: List[List[int]], N: int, K: int) -> int:
        graph = collections.defaultdict(list)
        # 그래프 인접 리스트 구성
        for u, v, w in times:
            graph[u].append((v, w))

        # 큐 변수: [(소요 시간, 정점)]
        Q = [(0, K)]
        dist = collections.defaultdict(int)

        # 우선 순위 큐 최소값 기준으로 정점까지 최단 경로 삽입
        while Q:
            time, node = heapq.heappop(Q)
            if node not in dist:
                dist[node] = time
                for v, w in graph[node]:
                    alt = time + w
                    heapq.heappush(Q, (alt, v))

        # 모든 노드 최단 경로 존재 여부 판별
        if len(dist) == N:
            return max(dist.values())
        return -1
```

[3 1 5], [3 2 2], [2 1 2], [3 4 1], [4 5 1]

[5 6 1], [6 7 1], [7 8 1], [8 1 1]

N=8, K=3

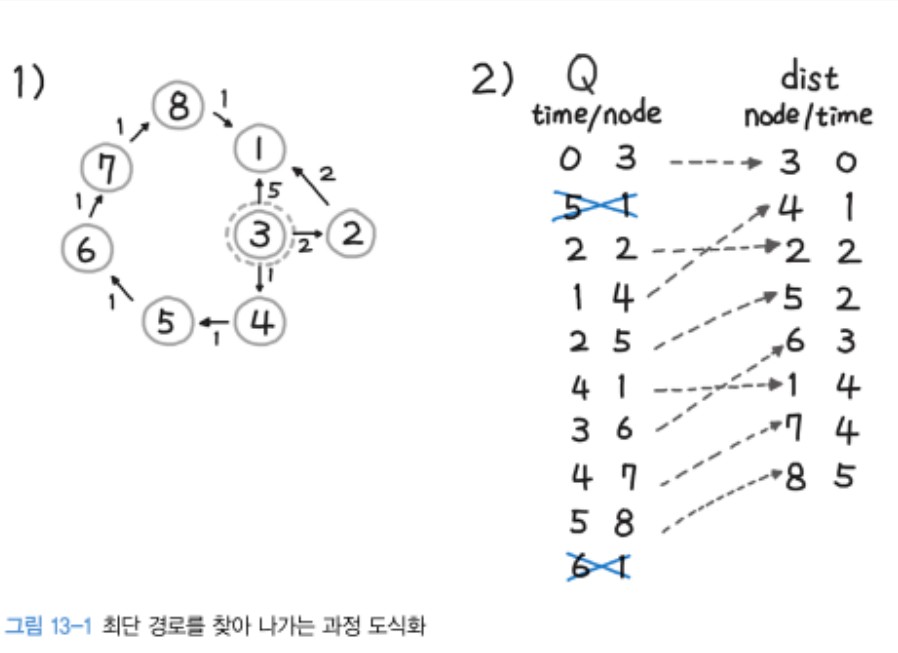
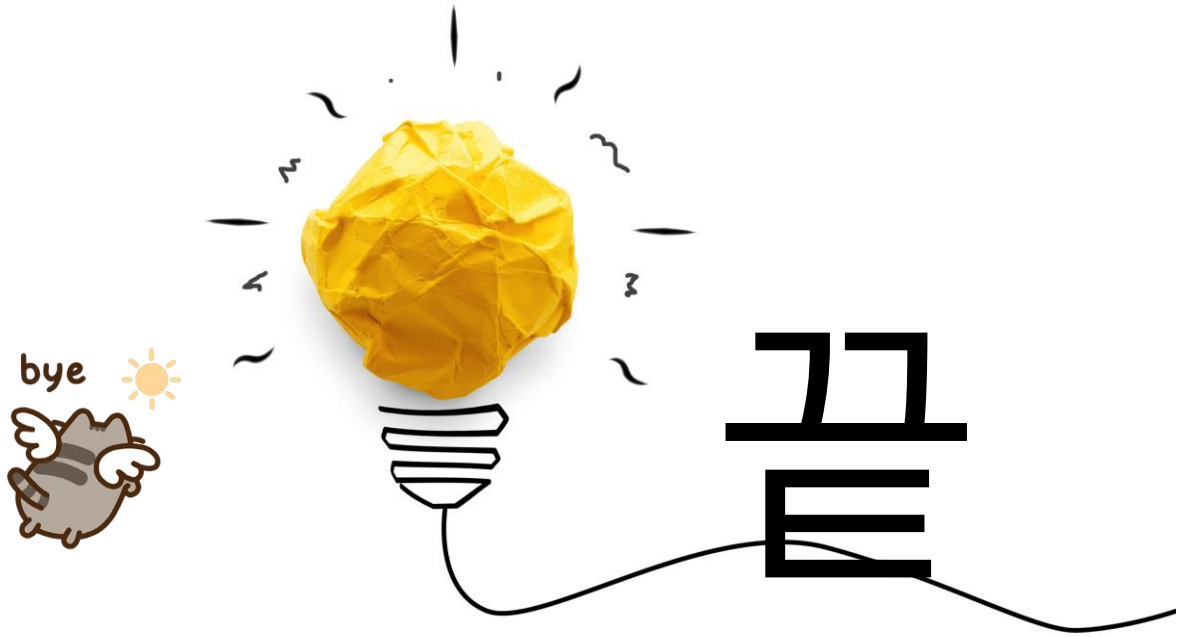



그림 13-1 최단 경로를 찾아 나가는 과정 도식화


bye
끝