알고리즘2 (2024-2)

8. 백트래킹과 탐색

국립금오공과대학교 컴퓨터공학과

김 경 수

학습 목표

- 백트래킹을 활용하여 문제를 해결할 때 주의해야 할 사항이 무엇인지에 대해서 이해하고, 이를 실제 문제 풀이에 활용할 수 있다.
- 백트래킹에서 활용되는 유망함수가 무엇인지 이해하고, 그 필요성과 설계 방법에 대하여 설명할 수 있다.
- 백트래킹이 활용되는 다양한 문제를 직접 풀어보고, 이에 대한 해결 방법을 논리적으로 설명할 수 있다.

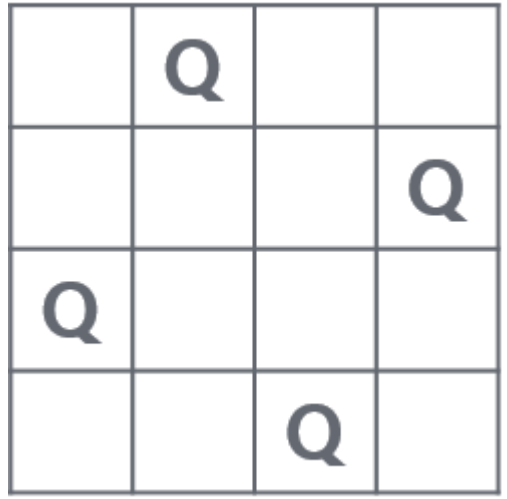
문제 1. N-Queens

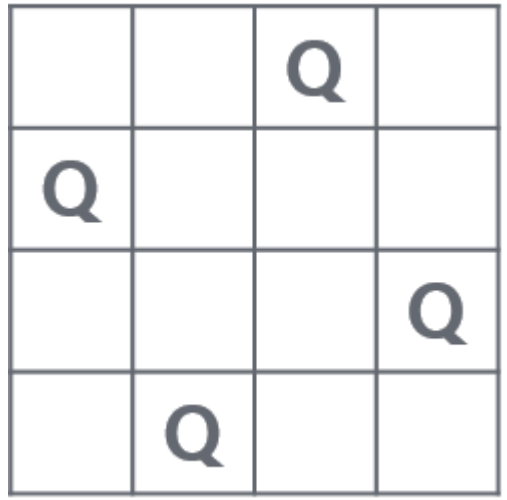
문제 1. N-Queens

제한시간: 40분

- 가로, 세로 길이가 n인 정사각형으로 된 체스판이 있습니다. 이때, 체스판 위의 n개의 퀸이 서로를 공격할 수 없도록 배치하고 싶습니다.
- 예를 들어서 n이 4인 경우 다음과 같이 퀸을 배치하면 n개의 퀸은 서로를 한번에 공격 할 수 없습니다. (※ 왼쪽 상단/하단 그림 참고)
- 체스판의 가로 및 세로의 길이 n이 매개변수로 주어질 때, n개의 퀸이 조건에 만족 하도록 배치할 수 있는 방법의 수를 return하는 **solution1** 함수를 완성해주세요.
- **제한사항**
  - 퀸(Queen)은 가로, 세로, 대각선으로 이동할 수 있습니다.
  - n은 12이하의 자연수 입니다.
- **입출력 예제 및 설명**: 위 예시와 같습니다.

n	result
4	2

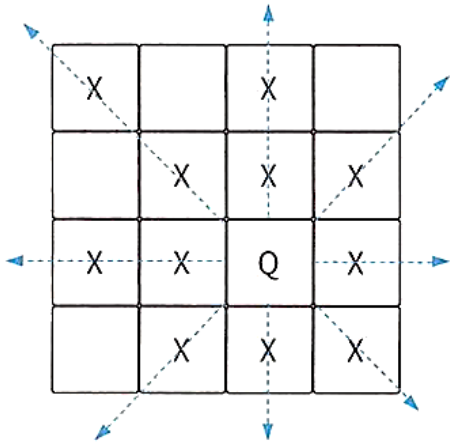




★ 문제 출처: https://school.programmers.co.kr/learn/courses/30/lessons/12952

문제 1. 문제 분석

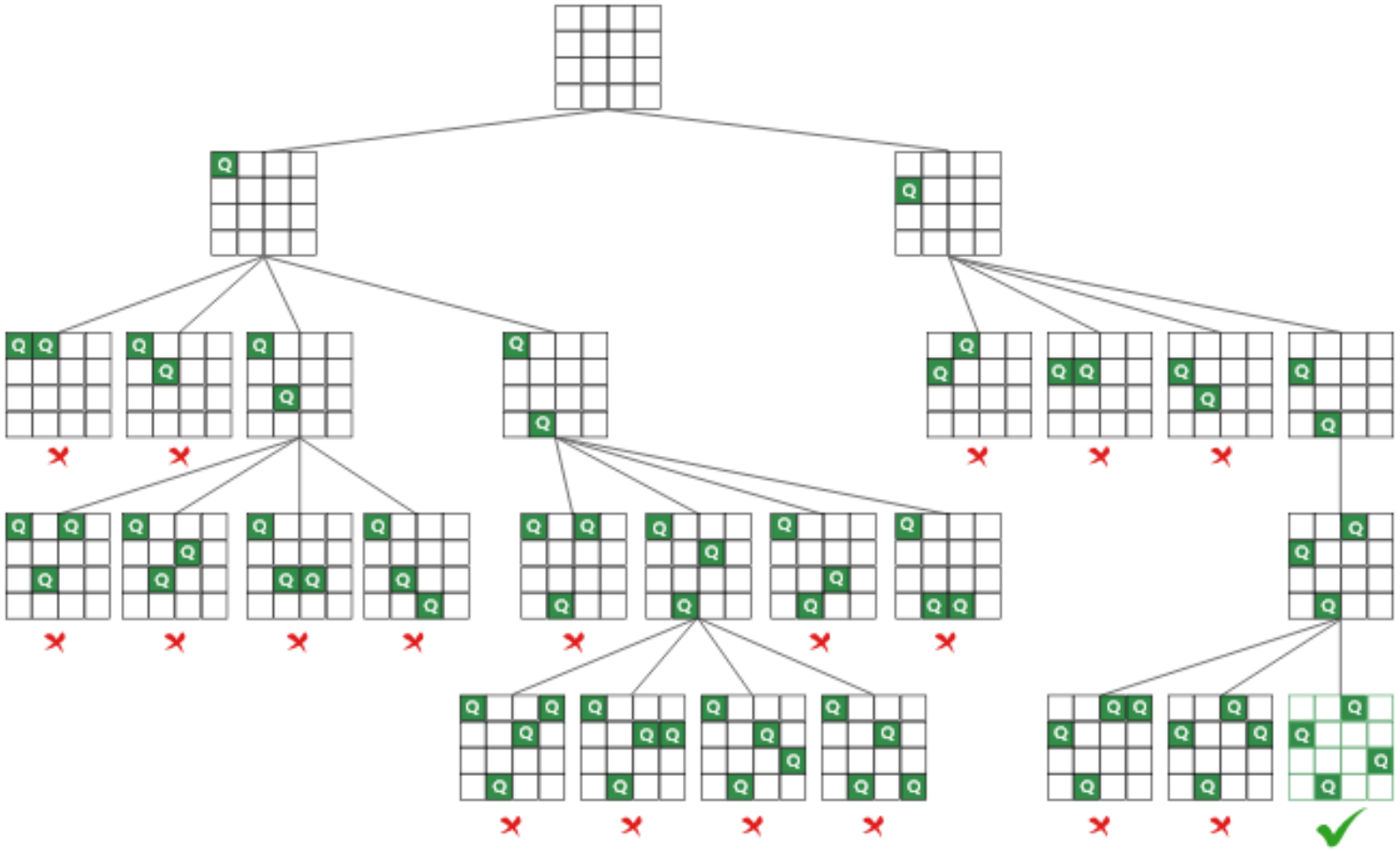
- **주어진 문제의 목표**
  - ➢ $N \times N$ 체스판에 퀸을 N개 배치했을 때, 서로를 공격할 수 없는 위치에 놓을 수 있는 방법의 개수를 구한다.



- **문제의 특성**
  - ✓ 퀸의 이동 방향: 상, 하, 좌, 우, 대각선 방향
  - ✓ 체스판의 각 지점을 행과 열로 구성된 좌표로 생각할 수 있다.
  - ✓ 이 경우, 퀸이 배치되는 좌표를 이용하여 모든 가능한 경우를 생각해 볼 수 있다.

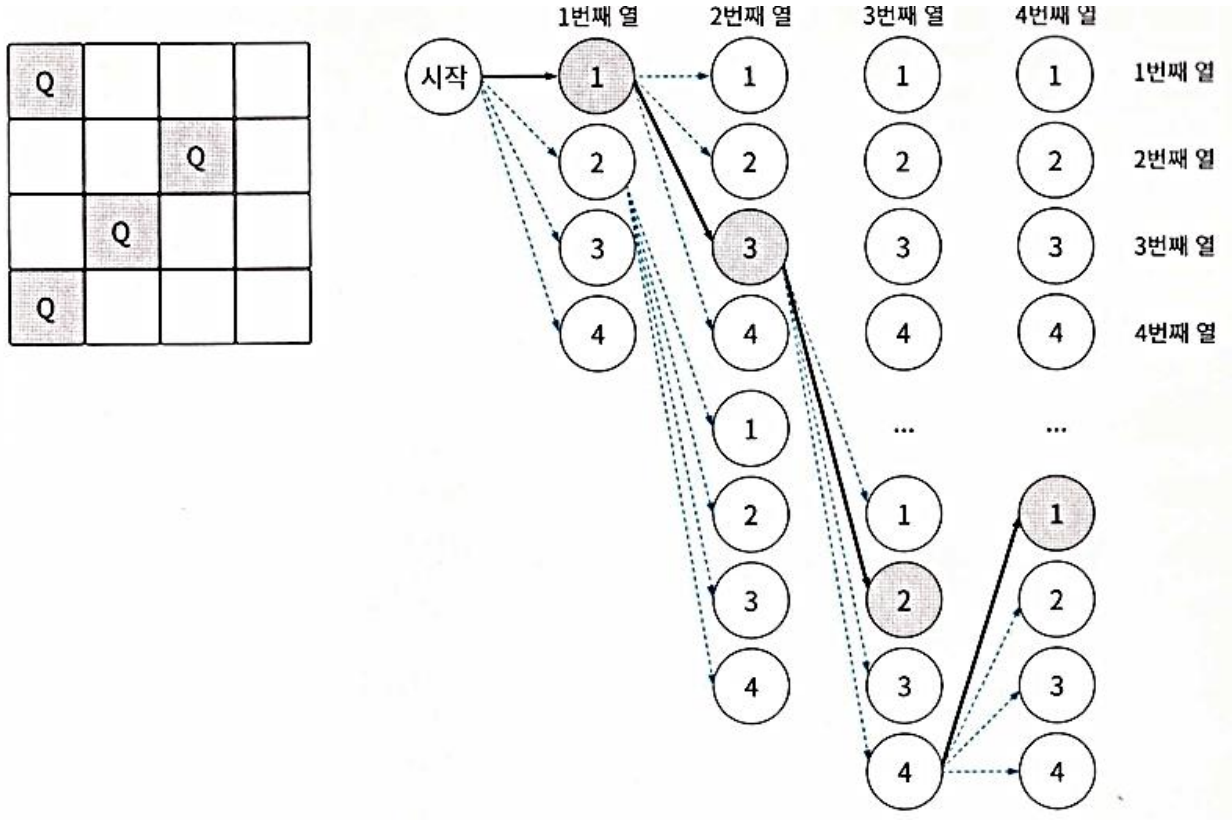
문제 1. 문제 분석

- N-queens problem을 상태 공간 트리로 표현한 결과



문제 1. 문제 해결 전략 수립

- **완전 탐색으로 해결하는 방법**
  - ✓ $N \times N$ 체스판에 퀸을 배치하는 모든 경우를 탐색한다.
  - ✓ 이때의 시간 복잡도는 $O(N^N)$ → 매우 비효율적



문제 1. 문제 해결 전략 수립

- **백트래킹으로 해결하기**
  - 백트래킹으로 해결할 때 필요한 것 → 유망 함수 (Promising function)
  - 유망 함수(Promising function)란?
    - 특정 탐색 조건을 정의한 함수
    - 탐색 과정의 각 상태에서 유망 함수 값을 계산하고, 그 결과를 토대로 계속 탐색을 진행할 것인지, 이전 상태로 되돌아갈 것인지(=backtracking) 결정한다.
    - 백트래킹 문제의 핵심 → 적절한 유망 함수를 설계하는 것
  - 이와 같이, 상태 공간 탐색에서 유망함수를 이용하여 불필요한 탐색을 수행하지 않는 방법을 "**가지치기(pruning)**"라 한다.
  - 즉, 유망함수는 **"가지치기"를 수행하기 위한 조건을 확인**하는 함수이다.

문제 1. 문제 해결 전략 수립

- **N-Queens 문제에서 가능한 탐색 조건은?**
  - ➢ 특정 좌표에 여왕이 추가하는 경우, 같은 열 또는 대각선 방향에 겹치는 여왕이 존재하면 해당 좌표에는 여왕을 둘 수 없다.
  - ➢ (예) 아래와 같이 (0, 0)과 (1, 3) 지점에 여왕이 존재하는 경우를 생각해 보자.
    - (2, 1)에 여왕을 놓을 수 있는가? → **No**
    - (2, 2)에 여왕을 놓을 수 있는가? → **Yes**
    - (2, 3)에 여왕을 놓을 수 있는가? → **No**
    - (2, 4)에 여왕을 놓을 수 있는가? → **No**

	0	1	2	3
0	Q			
1				Q
2				
3				

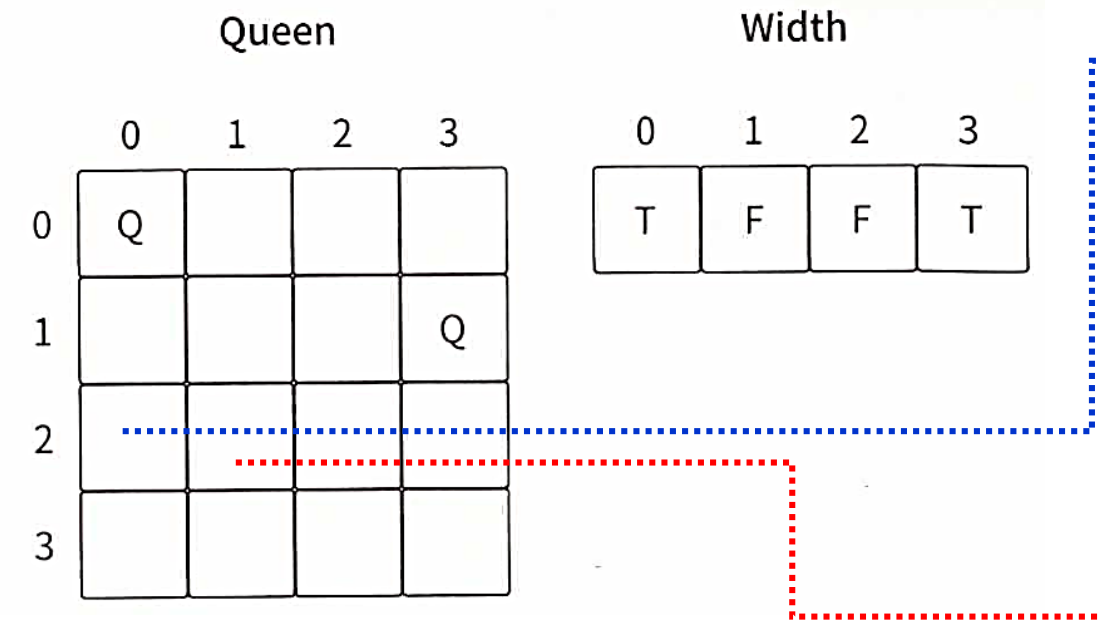
문제 1. 문제 해결 전략 수립

- **N-Queens 문제에서 가능한 탐색 조건은?**
  - ➢ 그렇다면 N-Queens 문제를 위한 유망 함수는 어떻게 설계해야 하는가?
    - ➢ 퀸이 추가될 때 마다, **같은 열**이나 **대각선 경로**에 겹치는 여왕이 있는지 확인한다.
    - ➢ 즉, 현재 좌표 (x, y)를 기준으로…
      - ① 같은 열(= y번째 열)에 퀸이 존재하는지 확인
      - ② "오른쪽 위 → 왼쪽 아래 대각선" 경로에 퀸이 존재하는지 확인
      - ③ "왼쪽 위 → 오른쪽 아래 대각선" 경로에 퀸이 존재하는지 확인

문제 1. 문제 해결 전략 수립

① 현재 위치를 기준으로 같은 열에 퀸이 존재하는 지 확인하는 방법

- **n** : 체스판의 크기
- **y** : 현재 퀸을 놓을 위치(= 열)
- **width[i]**: 특정 열에 퀸이 위치하는지 표시하는 배열



[예제 1]

- 체스판의 (2,0) 지점에 퀸을 배치하고자 한다면, 해당 지점의 0번째 열에 대한 width[0]을 확인한다.
- width[0]=T 이므로, 해당 열에는 퀸을 놓을 수 없다.

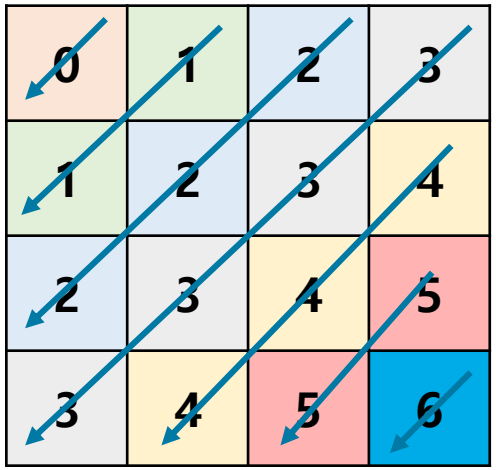
[예제 2]

- 체스판의 (2,1) 지점의 경우 width[1]=F 이므로, 해당 열에는 퀸을 놓을 수 있다.

문제 1. 문제 해결 전략 수립

② 현재 위치를 기준으로 대각선 방향에 퀸이 존재하는 지 확인하는 방법 (1)

- **`diagonal1[]`**: 오른쪽 위 → 왼쪽 아래 방향 대각선 방향 퀸 중복 체크용 배열
- **원리 설명**



diagonal1:

0	1	2	3	4	5	6

- 배열 "diagonal1"의 각 요소는 체스판에서 "오른쪽 위 → 왼쪽 아래" 방향 대각선 경로 7가지를 각각 나타낸다.

diagonal1:

T/F	T/F	T/F	T/F	T/F	T/F	T/F

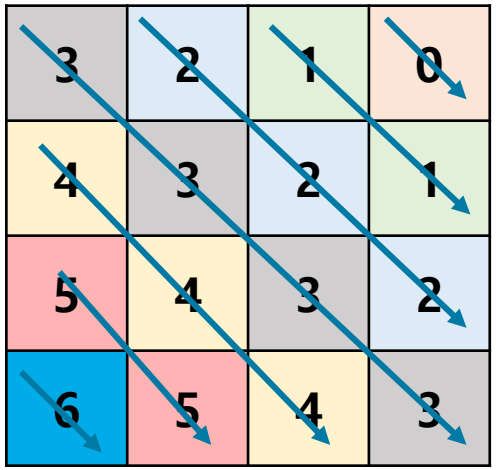
- 배열 "diagonal1"의 각 요소에는 해당되는 대각선 경로 상에 퀸이 존재하면 True, 없으면 False가 저장된다.

※ 참고: $n \times n$ matrix에서 주 방향 대각선의 개수는 $2n-1$개 이다.

문제 1. 문제 해결 전략 수립

③ 현재 위치를 기준으로 대각선 방향에 퀸이 존재하는 지 확인하는 방법 (2)

- **`diagonal2[]`**: 왼쪽 위 → 오른쪽 아래 방향 대각선 방향 퀸 중복 체크용 배열
- **원리 설명**



diagonal2:

0	1	2	3	4	5	6

- 배열 "diagonal2"의 각 요소는 체스판에서 "왼쪽 위 → 오른쪽 아래" 방향 대각선 경로 7가지를 각각 나타낸다.

diagonal2:

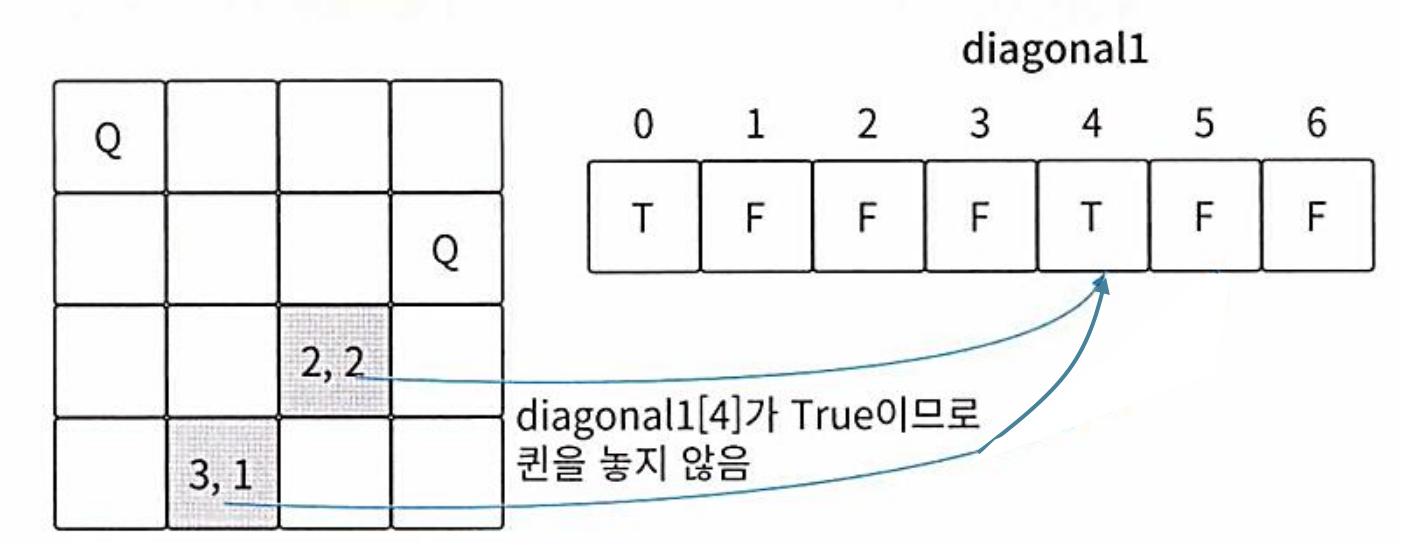
T/F	T/F	T/F	T/F	T/F	T/F	T/F

- 배열 "diagonal2"의 각 요소에는 해당되는 대각선 경로 상에 퀸이 존재하면 True, 없으면 False가 저장된다.

※ 참고: $n \times n$ matrix에서 주 방향 대각선의 개수는 $2n-1$개 이다.

문제 1. 문제 해결 전략 수립

- 대각선 방향에 퀸이 존재하는 지 확인하는 예
  - ➢ 대각선 경로상의 퀸의 배치 여부를 diagonal1, diagonal2 배열을 이용하여 확인하는 방법의 예



- (2, 2) 지점에 퀸을 배치하는 경우, 4번째 대각선에 해당하는 위치이므로, "diagonal[4]"가 True인지 False인지 확인한다.
- (3, 1) 지점에 퀸을 배치하는 경우도 위와 동일하다.

문제 1. 구현 및 검증

```
# 퀸이 서로 공격할 수 없는 위치에 놓이는 경우의 수를 구하는 함수
def get_ans(n, y, width, diagonal1, diagonal2):
    ans = 0
    # 모든 행에 대해서 퀸의 위치가 결정된 경우
    if y == n:
        # 해결 가능한 경우의 수를 1 증가시킴
        ans += 1
    else:
        # 현재 행에서 퀸이 놓일 수 있는 모든 위치를 시도
        for i in range(n):
            # 해당 위치에 이미 퀸이 있는 경우, 대각선상에 퀸이 있는 경우 스킵
            if width[i] or diagonal1[i + y] or diagonal2[i - y + n]:
                continue
            # 해당 위치에 퀸을 놓음
            width[i] = diagonal1[i + y] = diagonal2[i - y + n] = True
            # 다음 행으로 이동하여 재귀적으로 해결 가능한 경우의 수 찾기
            ans += get_ans(n, y + 1, width, diagonal1, diagonal2)
            # 해당 위치에 놓인 퀸을 제거함
            width[i] = diagonal1[i + y] = diagonal2[i - y + n] = False
    return ans
```

문제 1. 구현 및 검증

```
def solution1(n):
    # get_ans() 함수를 호출하여 해결 가능한 경우의 수 찾기
    ans = get_ans(n, 0, [False]*n, [False]*(n*2), [False]*(n*2))
    return ans
```

문제 1. 구현 및 검증

테스트 케이스 #1

```
print( solution1(4) )  # 출력: 2
```

테스트 케이스 #2

```
print( solution1(8) )  # 출력: 92
```

테스트 케이스 #3

```
print( solution1(10) ) # 출력: 724
```

N	가능한 경우의 수
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724

문제 2. 피로도

문제 2. 피로도 (1/4)

제한시간: 60분

- XX게임에는 피로도 시스템(0 이상의 정수로 표현합니다)이 있으며, 일정 피로도를 사용해서 던전을 탐험할 수 있습니다.
- 이때, 각 던전마다 탐험을 시작하기 위해 필요한 "최소 필요 피로도"와 던전 탐험을 마쳤을 때 소모되는 "소모 피로도"가 있습니다.
- "최소 필요 피로도"는 해당 던전을 탐험하기 위해 가지고 있어야 하는 최소한의 피로도를 나타내며, "소모 피로도"는 던전을 탐험한 후 소모되는 피로도를 나타냅니다.
- 예를 들어 "최소 필요 피로도"가 80, "소모 피로도"가 20인 던전을 탐험하기 위해서는 유저의 현재 남은 피로도는 80 이상 이어야 하며, 던전을 탐험한 후에는 피로도 20이 소모됩니다.

★ 문제 출처: https://school.programmers.co.kr/learn/courses/30/lessons/87946

문제 2. 피로도 (2/4)

제한시간: 60분

- 이 게임에는 하루에 한 번씩 탐험할 수 있는 던전이 여러 개 있는데, 한 유저가 오늘 이 던전들을 최대한 많이 탐험하려 합니다.
- 유저의 현재 피로도 k와 각 던전 별 "최소 필요 피로도", "소모 피로도"가 담긴 2차원 배열 dungeons 가 매개변수로 주어질 때, 유저가 탐험할 수 있는 최대 던전 수를 return 하도록 **solution2** 함수를 완성해주세요.
- **제한사항**
  - k는 1 이상 5,000 이하인 자연수입니다.
  - dungeons의 세로(행) 길이(즉, 던전의 개수)는 1 이상 8 이하입니다.
    - dungeons의 가로(열) 길이는 2 입니다.
    - dungeons의 각 행은 각 던전의 ["최소 필요 피로도", "소모 피로도"] 입니다.
    - "최소 필요 피로도"는 항상 "소모 피로도"보다 크거나 같습니다.
    - "최소 필요 피로도"와 "소모 피로도"는 1 이상 1,000 이하인 자연수입니다.
    - 서로 다른 던전의 ["최소 필요 피로도", "소모 피로도"]가 서로 같을 수 있습니다.

★ 문제 출처: https://school.programmers.co.kr/learn/courses/30/lessons/87946

문제 2. 피로도 (3/4)

제한시간: 60분

- **입출력 예제 및 설명**

k	dungeons	result
80	[[80,20],[50,40],[30,10]]	3

  - 현재 피로도는 80입니다.
  - 만약, 첫 번째 → 두 번째 → 세 번째 던전 순서로 탐험한다면
    - 현재 피로도는 80이며, 첫 번째 던전을 돌기위해 필요한 "최소 필요 피로도" 또한 80이므로, 첫 번째 던전을 탐험할 수 있습니다. 첫 번째 던전의 "소모 피로도"는 20이므로, 던전을 탐험한 후 남은 피로도는 60입니다.
    - 남은 피로도는 60이며, 두 번째 던전을 돌기위해 필요한 "최소 필요 피로도"는 50이므로, 두 번째 던전을 탐험할 수 있습니다. 두 번째 던전의 "소모 피로도"는 40이므로, 던전을 탐험한 후 남은 피로도는 20입니다.
    - 남은 피로도는 20이며, 세 번째 던전을 돌기위해 필요한 "최소 필요 피로도"는 30입니다. 따라서 세 번째 던전은 탐험할 수 없습니다.

★ 문제 출처: https://school.programmers.co.kr/learn/courses/30/lessons/87946

문제 2. 피로도 (4/4)

제한시간: 60분

- 입출력 예제 및 설명

k	dungeons	result
80	[[80,20],[50,40],[30,10]]	3

  - 만약, 첫 번째 → 세 번째 → 두 번째 던전 순서로 탐험한다면
    - 현재 피로도는 80이며, 첫 번째 던전을 돌기위해 필요한 "최소 필요 피로도" 또한 80이므로, 첫 번째 던전을 탐험할 수 있습니다. 첫 번째 던전의 "소모 피로도"는 20이므로, 던전을 탐험한 후 남은 피로도는 60입니다.
    - 남은 피로도는 60이며, 세 번째 던전을 돌기위해 필요한 "최소 필요 피로도"는 30이므로, 세 번째 던전을 탐험할 수 있습니다. 세 번째 던전의 "소모 피로도"는 10이므로, 던전을 탐험한 후 남은 피로도는 50입니다.
    - 남은 피로도는 50이며, 두 번째 던전을 돌기위해 필요한 "최소 필요 피로도"는 50이므로, 두 번째 던전을 탐험할 수 있습니다. 두 번째 던전의 "소모 피로도"는 40이므로, 던전을 탐험한 후 남은 피로도는 10입니다.
  - 따라서 이 경우 세 던전을 모두 탐험할 수 있으며, 유저가 탐험할 수 있는 최대 던전 수는 3입니다.

★ 문제 출처: https://school.programmers.co.kr/learn/courses/30/lessons/87946

문제 2. 문제 분석

- **주어진 문제의 목표**
  - ➢ 유저의 현재 피로도 k와 각 던전 별 최소 필요 피로도, 소모 피로도가 주어질 때, 유저가 탐험할 수 있는 최대 던전 수를 계산하여 반환한다.

- **문제의 특성**
  - ✓ 탐험할 수 있는 던전의 수
  - ✓ 던전 탐험의 조건
    - ✓ 각 던전 탐험에 요구되는 최소 피로도
    - ✓ 각 던전 탐험에 요구되는 소모 피로도
    - ✓ 현재 유저의 남은 피로도

해당 던전을 탐험하기 위해서는...

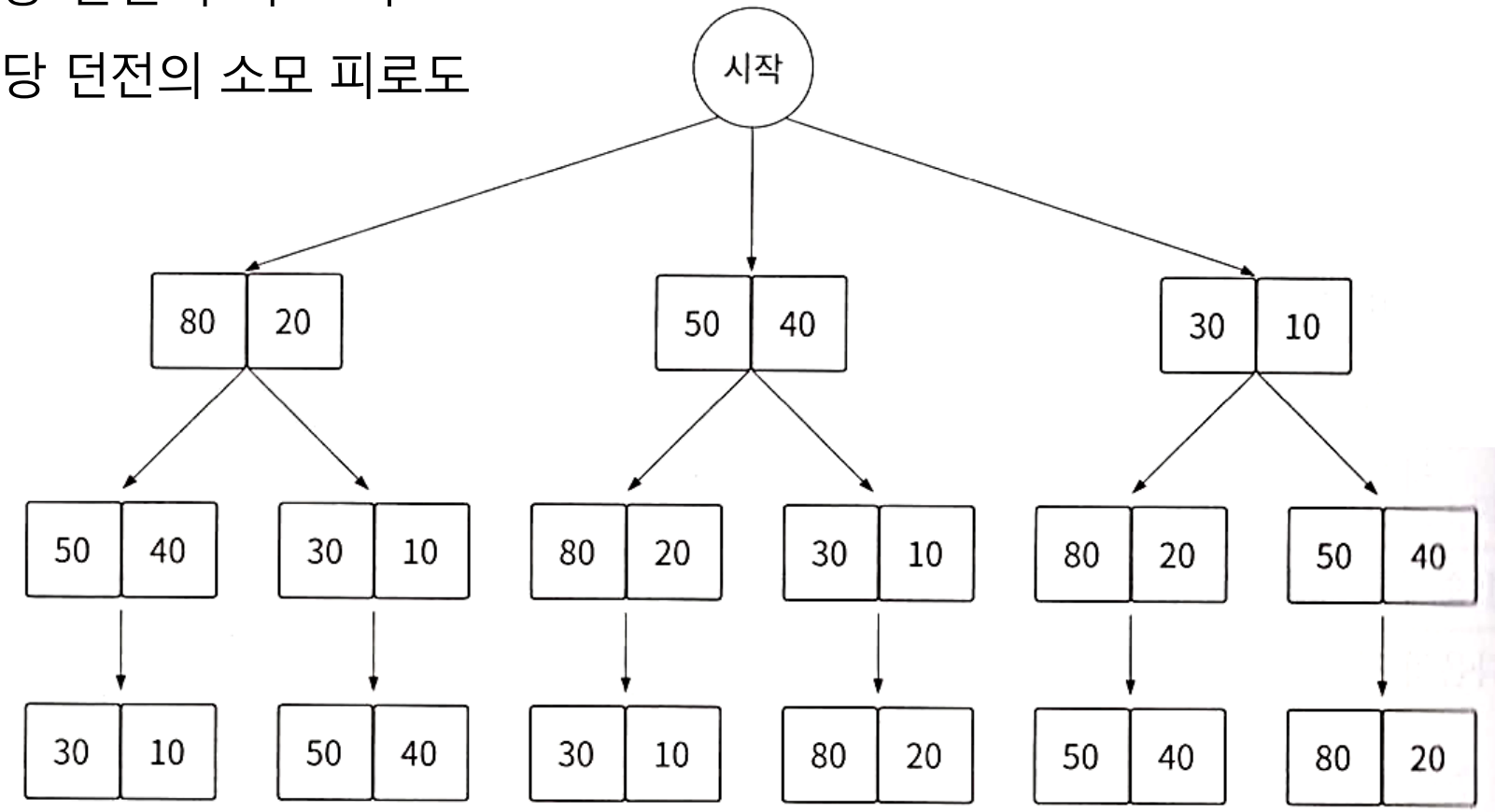
현재 남은 피로도 > 최소 필요 피로도

문제 2. 문제 해결 전략 수립

- **탐험 가능한 던전의 개수를 카운트하는 방법**
  - 모든 탐색 가능한 경우를 표현하는 방법 → 상태 공간 트리를 그려서 해결
  - 상태 공간 트리를 바탕으로, 루트에서 리프 레벨까지 탐색을 한 번 진행하면, 이것이 곧 하나의 가능한 경우(a possible case)가 된다.
  - 이때, 모든 경우를 탐색하는 경우 시간 복잡도가 $O(N^N)$이므로, 각 상태 노드를 기준으로 더 이상 탐색이 불필요하다고 판단되면 바로 탐색을 중단하고 되돌아가는(backtracking) 방법이 필수적으로 요구된다.
- **효율적인 탐색을 위한 가지치기(pruning) 방법**
  - 유망 함수(promising function)를 구현하여 현재 상태에서 가지치기 여부를 판단
  - 본 문제에서 유망 함수는 #.

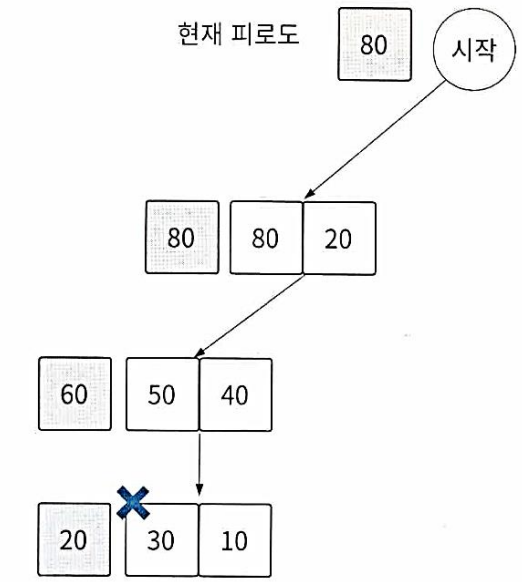
문제 2. 문제 해결 방법 기술

- **던전 탐색을 위한 상태 공간 트리**
  - 각 노드의 왼쪽 칸 = 해당 던전의 최소 피로도
  - " 오른쪽 칸 = 해당 던전의 소모 피로도

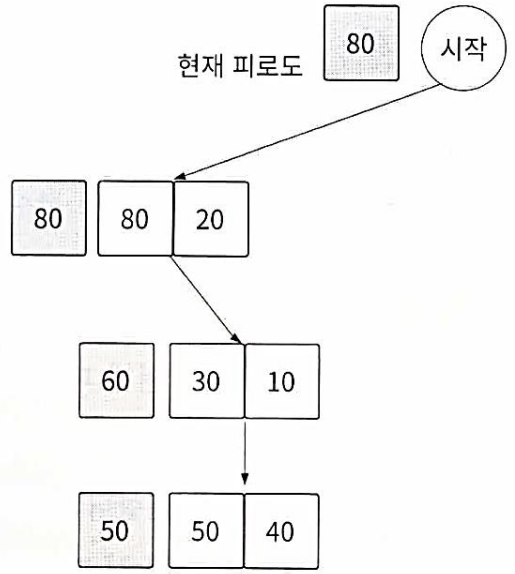


문제 2. 문제 해결 방법 기술

- **우리가 고민해야 할 사항**
  - ➢ 현재 피로도가 들어가려는 던전의 최소 프로도보다 높아야 던전에 들어갈 수 있음.
    - 현재 어떤 던전을 들어갈 수 있는지 선택할 때 영향을 미침.
  - ➢ 해당 던전을 빠져나올 때 현재 피로도가 소모 피로도만큼 줄어듦.
    - 현재 던전을 통과한 다음에 어떤 던전을 갈 수 있는지 선택할 때 영향을 미침.



- **유망 함수에 의해서 백트래킹이 발생하는 경우**
  - ➢ 현재 들어가려는 던전의 최소 필요도 <= 남아있는 피로도



문제 2. 구현 및 검증

```
# 백트래킹을 위한 DFS
def dfs(cur_k, cnt, dungeons, visited):
    answer_max = cnt
    for i in range(len(dungeons)):
        # 현재 피로도(cur_k)가 i번째 던전의 최소 필요도보다 크거나 같고, i번째 던전을 방문한 적이 없다면...
        if cur_k >= dungeons[i][0] and visited[i] == 0:
            visited[i] = 1

            # 현재까지의 최대 탐험 가능 던전 수와 i번째 던전에서 이동할 수 있는 최대 탐험 가능
            # 던전 수 중에서 큰 값을 선택하여 업데이트
            answer_max = max(
                answer_max, dfs(cur_k - dungeons[i][1], cnt+1, dungeons, visited)
            )
            visited[i] = 0

    return answer_max
```

문제 2. 구현 및 검증

```python
# 최대 탐험 가능 던전 수를 계산하는 함수
def solution2(k, dungeons):
    visited = [0] * len(dungeons) # 던전 방문 여부를 저장할 로컬 배열
    answer_max = dfs(k, 0, dungeons, visited)
    return answer_max
```

테스트 케이스 #1

```python
k = 80
dungeons = [[80,20],[50,40],[30,10]]
print( solution2(k, dungeons) )
```

문제 3. 외벽 점검

문제 3. 외벽 점검 (1/5)

제한시간: 80분

- 레스토랑을 운영하고 있는 "스카피"는 레스토랑 내부가 너무 낡아 친구들과 함께 직접 리모델링 하기로 했습니다. 레스토랑이 있는 곳은 스노우타운으로 매우 추운 지역이어서 내부 공사를 하는 도중에 주기적으로 외벽의 상태를 점검해야 할 필요가 있습니다.
- 레스토랑의 구조는 완전히 동그란 모양이고 외벽의 총 둘레는 n미터이며, 외벽의 몇몇 지점은 추위가 심할 경우 손상될 수도 있는 취약한 지점들이 있습니다.
- 따라서 내부 공사 도중에도 외벽의 취약 지점들이 손상되지 않았는 지, 주기적으로 친구들을 보내서 점검을 하기로 했습니다. 다만, 빠른 공사 진행을 위해 점검 시간을 1시간으로 제한했습니다.

★ 문제 출처: https://school.programmers.co.kr/learn/courses/30/lessons/60062

문제 3. 외벽 점검 (2/5)

제한시간: 80분

- 친구들이 1시간 동안 이동할 수 있는 거리는 제각각이기 때문에, 최소한의 친구들을 투입해 취약 지점을 점검하고 나머지 친구들은 내부 공사를 돕도록 하려고 합니다.
- 편의 상 레스토랑의 정북 방향 지점을 0으로 나타내며, 취약 지점의 위치는 정북 방향 지점으로부터 시계 방향으로 떨어진 거리로 나타냅니다. 또, 친구들은 출발 지점부터 시계, 혹은 반시계 방향으로 외벽을 따라서만 이동합니다.
- 외벽의 길이 n, 취약 지점의 위치가 담긴 배열 weak, 각 친구가 1시간 동안 이동할 수 있는 거리가 담긴 배열 dist가 매개변수로 주어질 때, 취약 지점을 점검하기 위해 보내야 하는 친구 수의 최소값을 return 하도록 **solution3** 함수를 완성해주세요.

★ 문제 출처: https://school.programmers.co.kr/learn/courses/30/lessons/60062

문제 3. 외벽 점검 (3/5)

제한시간: 80분

- **제한사항**
  - n은 1 이상 200 이하인 자연수입니다.
  - weak의 길이는 1 이상 15 이하입니다.
    - 서로 다른 두 취약점의 위치가 같은 경우는 주어지지 않습니다.
    - 취약 지점의 위치는 오름차순으로 정렬되어 주어집니다.
    - weak의 원소는 0 이상 n - 1 이하인 정수입니다.
  - dist의 길이는 1 이상 8 이하입니다.
    - dist의 원소는 1 이상 100 이하인 자연수입니다.
  - 친구들을 모두 투입해도 취약 지점을 전부 점검할 수 없는 경우에는 -1을 return 해주세요.

★ 문제 출처: https://school.programmers.co.kr/learn/courses/30/lessons/60062

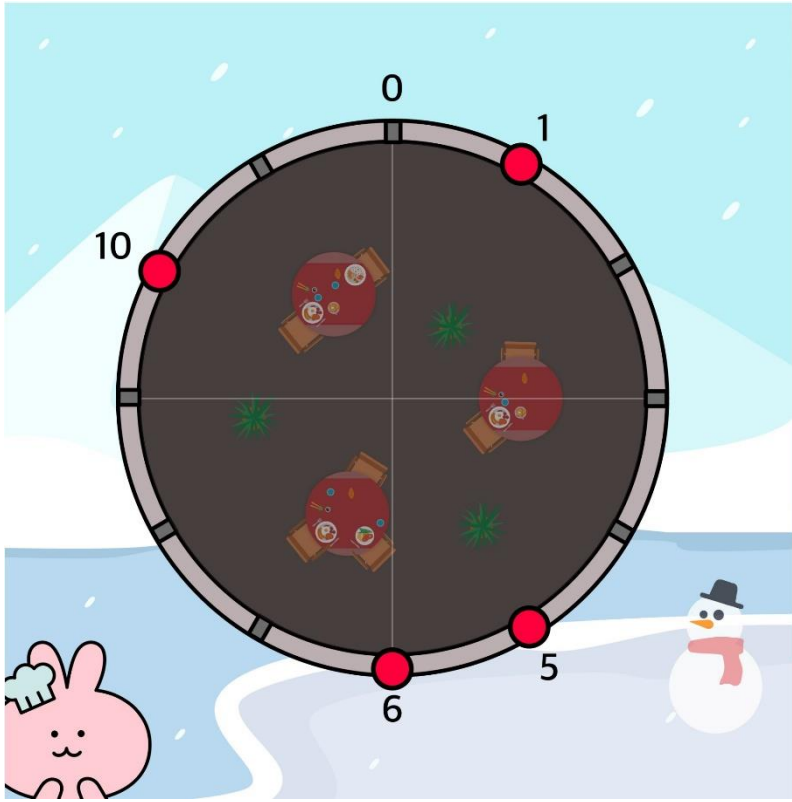
문제 3. 외벽 점검 (4/5)

제한시간: 80분

- 입출력 예제 설명
  - **입출력 예제 #1**

n	weak	dist	result
12	[1, 5, 6, 10]	[1, 2, 3, 4]	2
12	[1, 3, 4, 9, 10]	[3, 5, 7]	1

➢ 원형 레스토랑에서 외벽의 취약 지점의 위치는 다음과 같습니다.



➢ 친구들을 투입하는 예시 중 하나는 다음과 같습니다.

- 4m를 이동할 수 있는 친구는 10m 지점에서 출발해 시계방향으로 돌아 1m 위치에 있는 취약 지점에서 외벽 점검을 마칩니다.
- 2m를 이동할 수 있는 친구는 4.5m 지점에서 출발해 6.5m 지점에서 외벽 점검을 마칩니다.

➢ 그 외에 여러 방법들이 있지만, 두 명보다 적은 친구를 투입하는 방법은 없습니다.

➢ 따라서 친구를 최소 두 명 투입해야 합니다.

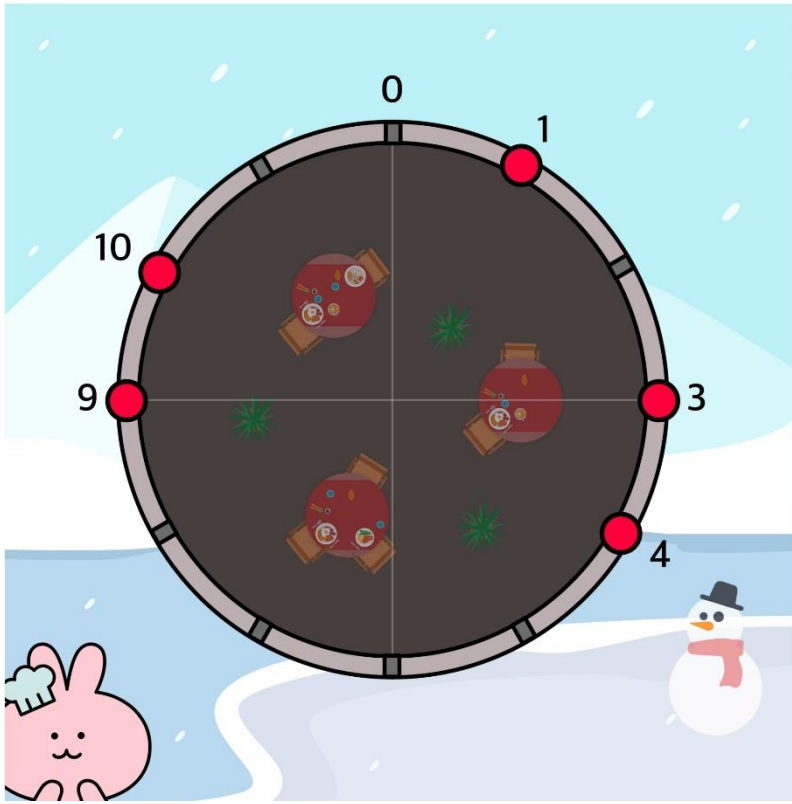
★ 문제 출처: https://school.programmers.co.kr/learn/courses/30/lessons/60062

문제 3. 외벽 점검 (5/5)

제한시간: 80분

- 입출력 예제 설명
  - **입출력 예제 #2**
    - 원형 레스토랑에서 외벽의 취약 지점의 위치는 다음과 같습니다.
    - 7m를 이동할 수 있는 친구가 4m 지점에서 출발해 반시계 방향으로 점검을 돌면 모든 취약 지점을 점검할 수 있습니다.
    - 따라서 친구를 최소 한 명 투입하면 됩니다.

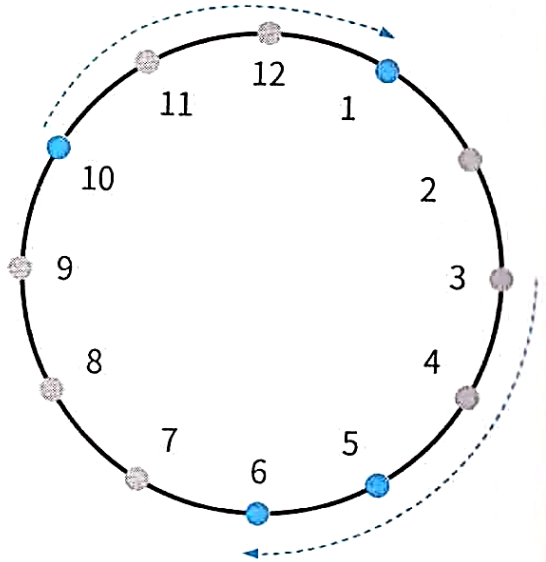
n	weak	dist	result
12	[1, 5, 6, 10]	[1, 2, 3, 4]	2
12	**[1, 3, 4, 9, 10]**	**[3, 5, 7]**	**1**



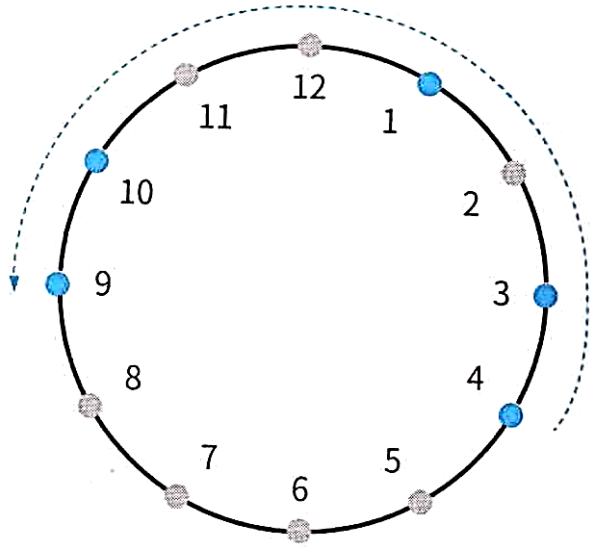
★ 문제 출처: https://school.programmers.co.kr/learn/courses/30/lessons/60062

문제 3. 문제 분석

- **주어진 문제의 목표**
  - ➢ 외벽의 길이와 취약 지점의 위치, 각 친구가 1시간 동안 이동할 수 있는 거리가 주어질 때, 취약 지점으로 보내야 하는 친구 수의 최소값을 계산하여 반환한다.

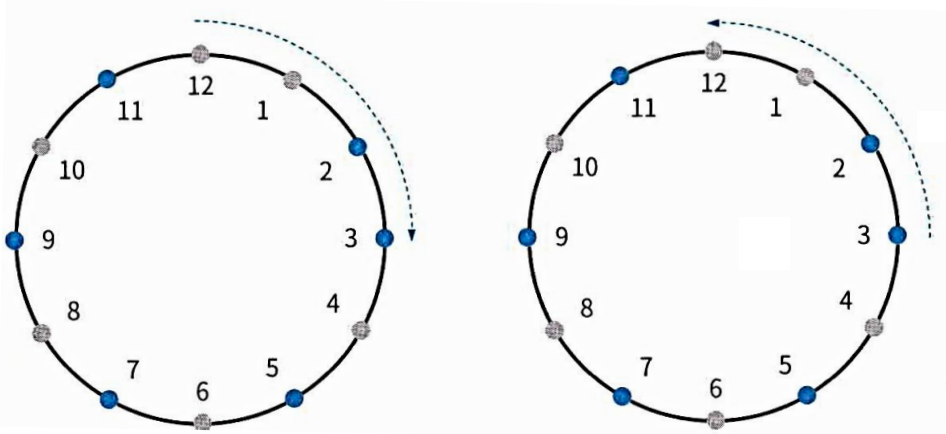


- **문제의 특성**
  - ✓친구들을 배치할 수 있는 위치가 다양함.
  - ✓친구들이 1시간 동안 이동할 수 있는 거리가 다름.
  - ✓따라서, 친구들을 배치하는 위치와 해당 친구의 이동 거리에 따라 모든 취약 지점을 방문하는 데 필요한 최소 친구 수가 달라짐.



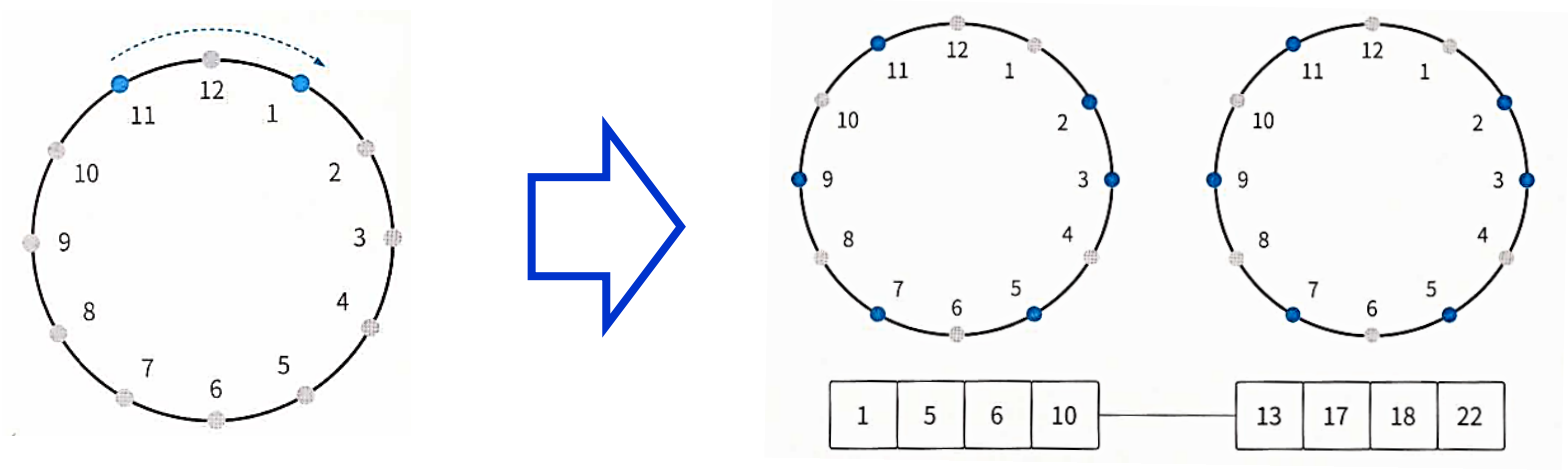
문제 3. 문제 분석

- **본 문제에서 고려해야 할 사항**
  - ✓ **친구의 1시간 이동 거리**
  - ✓ **친구 배치 순서**
    - ➢ 친구가 배치되는 지점과 이동 거리에 따라 커버 가능한 취약 지점의 개수가 달라짐.
  - ~~✓ 친구의 이동 방향~~
    - ➢ 친구의 이동 방향(시계/반시계 방향)에 따라 커버 가능한 취약 지점의 개수가 달라짐.
    - ➢ 단, 출발 지점을 다르게 설정한다면, 시계 이동 방향과 동일하게 반시계 이동 방향을 커버할 수 있음. → **따라서, 이동 방향은 고려할 필요가 없음.**

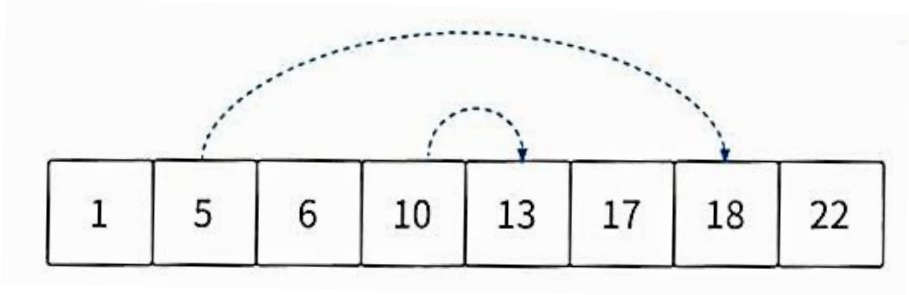


문제 3. 문제 해결 전략 수립

- **한 취약 지점에서 다른 취약 지점으로 이동하는 방법**
  - (예) 11에서 1로 이동하는 상황 → 11에서 13으로 이동하는 것과 동일하게 생각



  - 이를 위해 weak 배열의 길이가 n이라면, 이를 2n으로 늘리고, 새롭게 추가된 n개의 영역에는 weak 배열의 앞 n개의 값 각각에 n을 더한 값을 추가 (아래 그림 참고)



문제 3. 문제 해결 전략 수립

- **친구를 효과적으로 배치하는 방법**

① 순열로 친구를 배치할 순서 정하기

② 첫 번째 친구를 첫 번째 취약 지점에 세우고 점검할 수 있는 취약 지점을 제거하기

③ 그 다음 친구를 제거되지 않은 취약 지점에 세우고 점검할 수 있는 취약 지점 제거하기

④ 위 과정 ① ~ ③을 모든 취약 지점이 사라질 때 까지 반복적으로 수행

문제 3. 구현 및 검증

```
from itertools import permutations

def solution3(n, weak, dist):
    # 주어진 weak 지점들을 선형으로 만들기 위해 weak 리스트에 마지막 지점 + n을 추가
    length = len(weak)
    for i in range(length):
        weak. append(weak[i] + n)

    # 투입할 수 있는 친구들의 수에 1을 더한 값을 초기값으로 설정
    answer = len(dist) + 1

    # 모든 weak 지점을 시작점으로 설정하며 경우의 수를 탐색
    for i in range(length):
        for friends in permutations(dist, len(dist)):
            # frineds에 들어 있는 친구들을 순서대로 배치하여 투입된 친구 수(cnt)를 계산
            cnt = 1
            position = weak[i] + friends[cnt-1]
```

문제 3. 구현 및 검증

```
        # 현재 투입된 친구가 다음 weak 지점까지 갈 수 있는지 검사
        for j in range(i, i+length):
            if position < weak[j]:
                cnt += 1
                # 투입 가능한 친구의 수를 초과한 경우 탐색 중단
                if cnt > len(dist):
                    break
                position = weak[j] + friends[cnt-1]
        answer = min(answer, cnt)

return answer if answer <= len(dist) else - 1
```

문제 3. 구현 및 검증

테스트 케이스 #1

```
n = 12
weak = [1, 5, 6, 10]
dist = [1, 2, 3, 4]
print(solution3(n, weak, dist))
```

테스트 케이스 #2

```
n = 12
weak = [1, 3, 4, 9, 10]
dist = [3, 5, 7]
print(solution3(n, weak, dist))
```

문제 4. 조합 계산하기

문제 4. 조합 계산하기

제한시간: 50분

- 정수 N을 입력받아 1부터 N까지의 숫자 중에서 합이 10이 되는 조합을 리스트로 반환하는 **solution4** 함수를 작성하시오.
- **제약 조건**
  - **백트래킹을 활용**할 것.
  - 숫자 조합은 오름차순으로 정렬되어야 함.
  - 같은 숫자는 한 번만 선택할 수 있음.
  - N은 1이상 10 이하의 정수임.
- **입출력 예**

N	result
5	[[1, 2, 3, 4], [1, 4, 5], [2, 3, 5]]
2	[ ]
7	[[1, 2, 3, 4], [1, 2, 7], [1, 3, 6], [1, 4, 5], [2, 3, 5], [3, 7], [4, 6]]

문제 4. 문제 분석

- **주어진 문제의 목표**
  - ➢ 1부터 N까지의 숫자를 조합했을 때, 합이 K가 되는 조합을 찾는다.

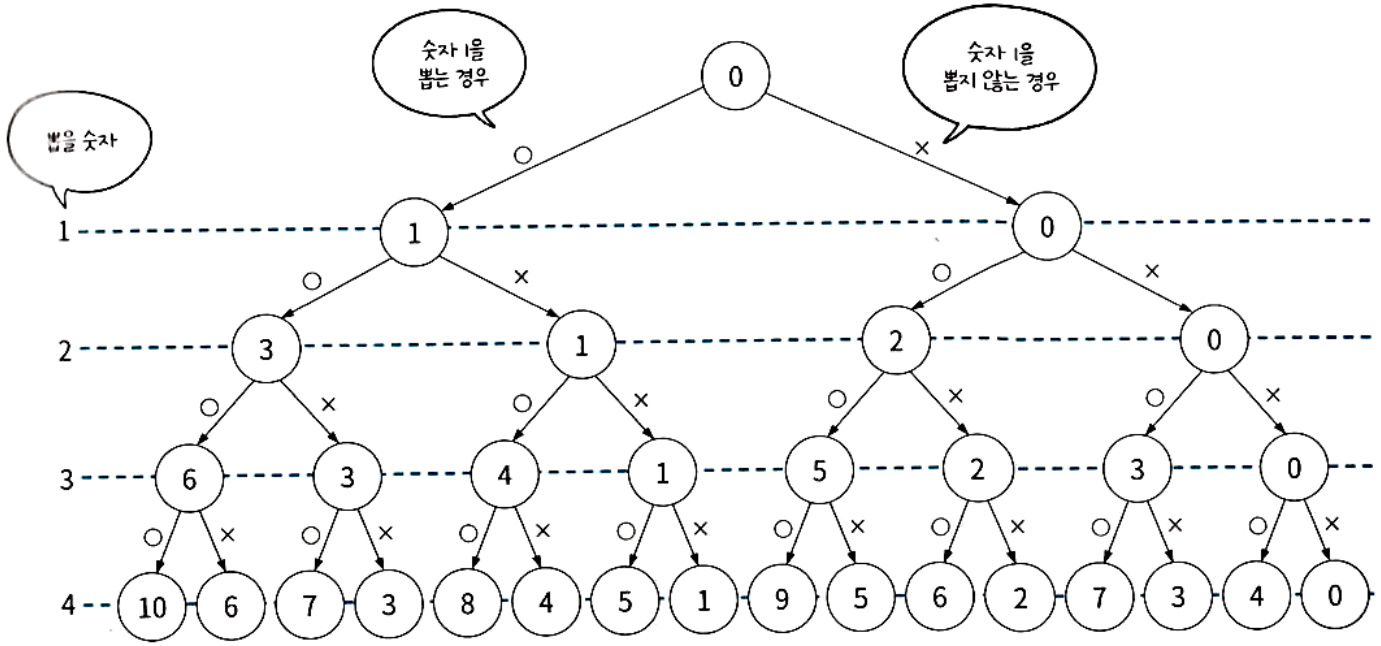
- **문제의 특성**
  - ✓ 조합 문제: 가능한 여러 상황을 조합하면서 최적의 해를 찾을 수 있다.
  - ✓ 대표적인 문제 풀이 방법
    - ✓ 완전 탐색 방법: 1부터 N까지의 모든 가능한 조합을 고려하는 경우
    - ✓ 백트래킹 방법: 위의 완전 탐색 방법에서, 탐색이 불필요한 부분에 대해서는 더 이상 탐색을 수행하지 않는 방법

문제 4. 문제 분석

- **완전 탐색으로 문제를 해결하는 방법**
  - 각 숫자는 뽑는 상태와 뽑지 않는 상태가 있음.
  - 각 숫자를 선택하는 과정은 다른 숫자에 대한 선택에 영향을 미치지 않음.

시간 복잡도는 $O(2^N)$

- **상태 공간 트리 설계**
  - 왼쪽의 숫자는 현재 뽑을 숫자, 원 안의 숫자는 현재까지 뽑은 숫자들의 합을 의미
  - O는 왼쪽의 "뽑을 숫자"를 실제로 뽑는 경우, X는 안 뽑는 경우를 의미함.



문제 4. 문제 해결 전략 수립

- **유망 함수의 필요성**
  - 상태 공간 트리를 살펴보면, 탐색이 불필요한 부분이 생각보다 많음을 확인할 수 있음.
  - 답이 될 가능성이 없는 조합이 생성된 경우, 더 이상의 탐색은 무의미함.
  - 따라서, 현재 주어진 조합이 답이 될 수 있는가 없는가를 검사하는 유망함수가 필요함.

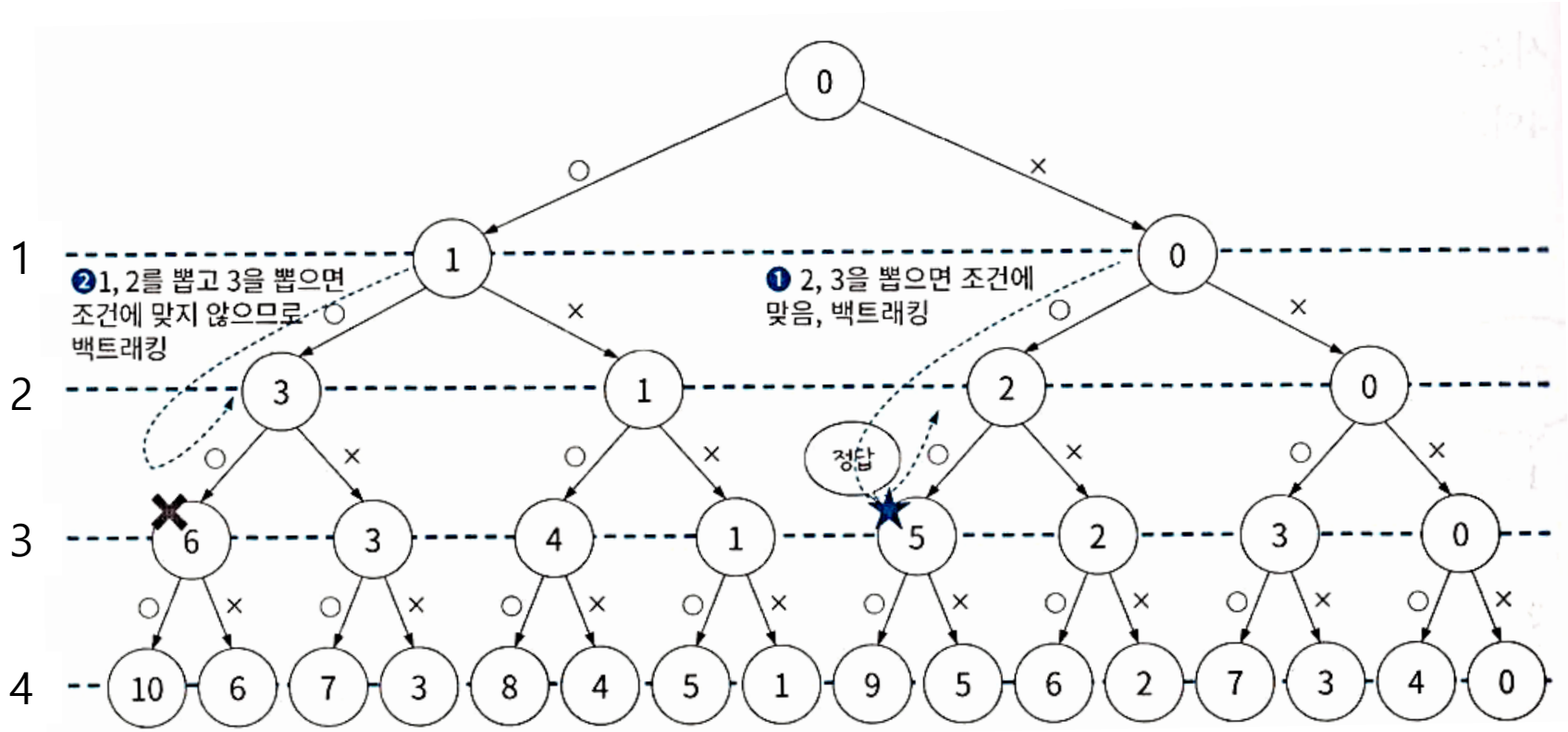
- **유망 함수**
  - 조건 1: 현재 조합으로 합이 K가 되는가를 검사
  - 조건 2: 해당 숫자를 조합하여 합이 K 이상이 되는가를 검사
  - 상기 두 조건 중 하나를 만족하면 더 이상 탐색을 수행하지 않음.

문제 4. 문제 해결 전략 수립

- **문제 해결 예**

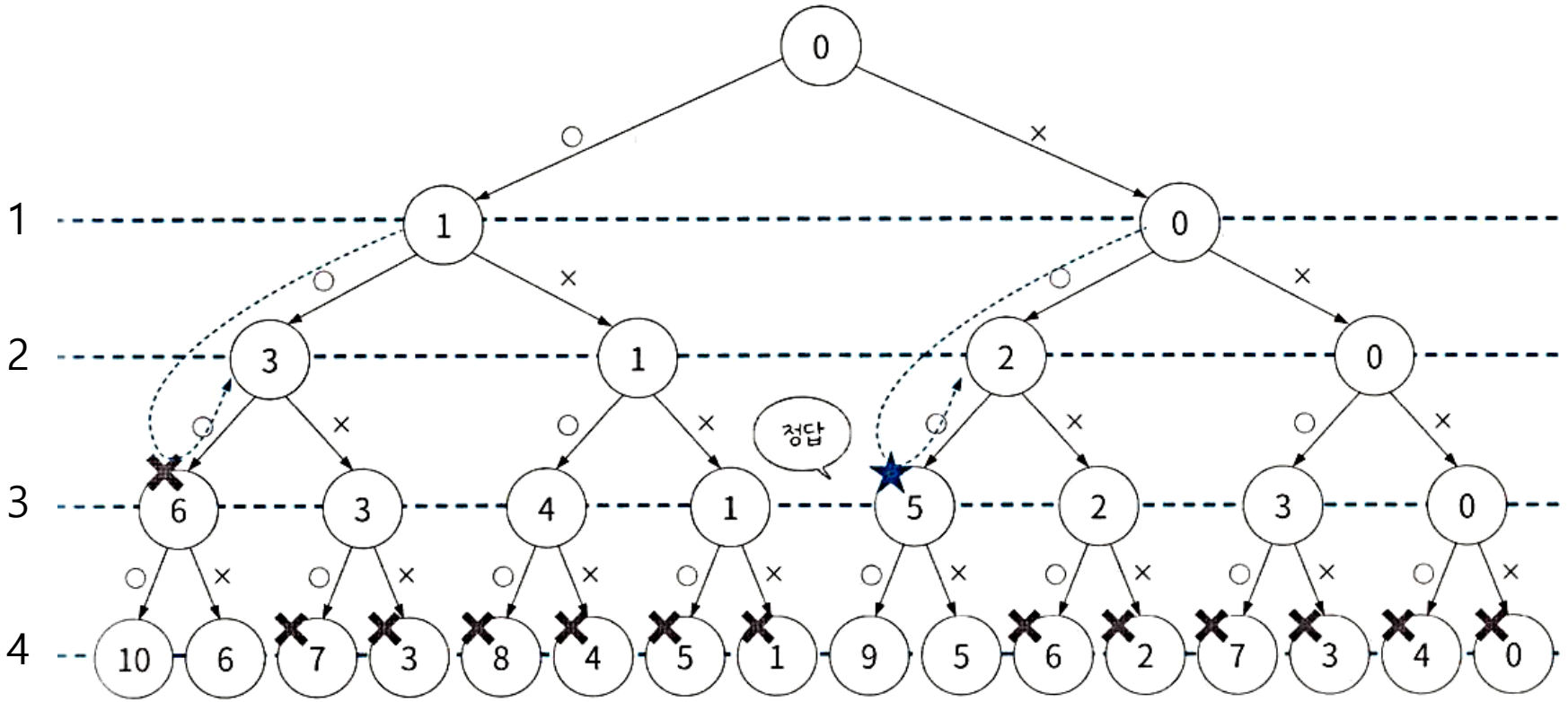
① 아래 그림에서, 2와 3을 뽑으면 이미 합이 5이므로 수를 더 뽑지 않아도 됨.

② 1과 2를 뽑은 상태에서 3을 뽑는다면 6이고, 이는 5보다 크므로 3 이후는 뽑지 않아도 됨.



문제 4. 문제 해결 전략 수립

- **문제 해결 예**
  - 백트래킹을 통해 아래와 같이 탐색이 배제되는 노드들이 많아졌음을 확인할 수 있음.



문제 4. 구현 및 검증

```
def solution4(N):
    results = []  # 조합 결과를 담을 리스트

    def backtrack(sum, selected_nums, start):
        if sum == 10:  # 합이 10이 되면 결과 리스트에 추가
            results.append(selected_nums)
            return

        # 다음에 선택할 수 있는 숫자들을 하나씩 선택
        for i in range(start, N+1):
            if sum + i <= 10:  # 선택한 숫자의 합이 10보다 작거나 같은 경우
                # 백트래킹 함수를 재귀적으로 호출
                backtrack(
                    sum + i, selected_nums + [i], i + 1
                )

    backtrack(0, [], 1)  # 백트래킹 함수를 호출
    return results
```

문제 4. 구현 및 검증

테스트 케이스 #1

```
N = 5
print(solution4(N))
```

테스트 케이스 #2

```
N = 2
print(solution4(N))
```

테스트 케이스 #3

```
N = 7
print(solution4(N))
```

Q & A