

Professional Coding Specialist

COS Pro 파이썬 1 급

21 강-24 강. 모의고사 5 차

1. 모의고사 5 차(1-10 번)

과정 소개

Cos Pro 1 급 파이썬 5 차 문제를 풀어보며 문제 유형을 익히고, 파이썬을 이용하여 알고리즘을 구현하기 위해 필요한 관련 지식을 익혀보도록 한다.

학습 목차

1. 문제 1
2. 문제 2
3. 문제 3
4. 문제 4
5. 문제 5
6. 문제 6
7. 문제 7
8. 문제 8
9. 문제 9
10. 문제 10

학습 목표

1. YBM IT(www.ybmit.com) 에서 제공하는 COS Pro 1 급 파이썬 샘플 문제를 풀어보며 파이썬을 이용하여 주어진 문제를 해결하기 위한 알고리즘을 구성하는 능력을 배양한다.
2. 많이 등장하는 문제 유형을 익혀서 COS Pro 1 급 시험에 대비한다.

1. 문제 1

1) 문제 코드

```

1  def solution(n):
2      answer = 0
3      steps = [0 for _ in range(n+1)]
4      steps[1] = 1
5      steps[2] = 2
6      steps[3] = 4
7      for i in range(4, n+1):
8          steps[i] = @@@
9      answer = steps[n]
10     return answer
11
12     #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
13     n1 = 3
14     ret1 = solution(n1)
15
16     #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
17     print("solution 함수의 반환 값은", ret1, "입니다.")
18
19     n2 = 4
20     ret2 = solution(n2)
21
22     #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
23     print("solution 함수의 반환 값은", ret2, "입니다.")

```

2) 문제 개요

- 제시된 과제를 해결하기 위해 코드를 완성하는 문제.
- 한 번에 1 계단, 2 계단, 3 계단씩 오를 수 있을 때, 매개변수로 전달된 n 개의 계단을 오르는 방법의 수를 구하는 코드를 동적 계획법의 타블레이션 방식을 이용하여 완성하는 문제.

3) 동적 계획법(Dynamic Programming)

- 복잡한 문제를 간단한 여러 개의 문제로 나누어 푸는 방법.
- 중복된 하위 문제들의 결과를 저장해 두었다가 상위 문제를 풀 때 그 결과를 재사용.
- 타블레이션 방식은 재귀 호출을 하지 않고, 하위 문제들의 첫 번째부터 모든 결과를 저장해 두었다가 사용.
- 메모이제이션 방식은 재귀 호출을 사용하지만, 하위 문제의 결과를 메모리에 저장한 후 재사용.

4) 정답

- 주요 아이디어 정리

오르는 계단	오르는 방법	경우의 수
1 계단	1 계단	1
2 계단	1 계단+1 계단	2

	2 계단	
3 계단	1 계단+1 계단+1 계단 1 계단+2 계단 2 계단+1 계단 3 계단	4
4 계단	1 계단 오른 뒤 + 3 계단 오르는 법 사용 2 계단 오른 뒤 + 2 계단 오르는 법 사용 3 계단 오른 뒤 + 1 계단 오르는 법 사용	7
...
n 계단	1 계단 오른 뒤 + (n-1)계단 오르는 법 2 계단 오른 뒤 + (n-2)계단 오르는 법 3 계단 오른 뒤 + (n-3)계단 오르는 법	(n-1)경우의 수 +(n-2)경우의 수 +(n-3)경우의 수

- ➔ 4 개 이상 n 개의 계단을 오르는 방법의 수를 구하려면 1 계단 오른 뒤에 n-1 계단을 오르는 방법, 2 계단 오른 뒤에 n-2 계단을 오르는 방법, 3 계단 오른 뒤에 n-3 계단을 오르는 방법을 모두 더해야 함.
- ➔ 1 개부터 n-1 개의 계단을 오르는 방법을 모두 구해야 n 개의 계단 오르는 방법을 구할 수 있음.

● 정답 코드

```

1 def solution(n):
2     answer = 0
3     steps = [0 for _ in range(n+1)]
4     steps[1] = 1
5     ① steps[2] = 2
6     steps[3] = 4
7     ② for i in range(4, n+1):
8         steps[i] = steps[i-1] + steps[i-2] + steps[i-3]
9     ③ answer = steps[n]
10    return answer
11
12    #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
13    n1 = 3
14    ret1 = solution(n1)
15
16    #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
17    print("solution 함수의 반환 값은", ret1, "입니다.")
18
19    n2 = 4
20    ret2 = solution(n2)
21
22    #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
23    print("solution 함수의 반환 값은", ret2, "입니다.")

```

- ①. 1 계단, 2 계단, 3 계단을 오르는 방법의 수를 step 리스트의 인덱스 번호 1, 2, 3 에 저장.
- ②. for 문을 이용하여 4 계단부터 n 계단까지 오르는 방법의 수를 이전 세 개의 항목들을 이용하여 구함.
- ③. step 리스트의 n 번째 항목을 answer 로 저장.

5) 다른 코드 제안

① 동적 계획법 - 메모이제이션 방식으로 구현

```
1 def func_step(steps, n):
2     if steps[n] > 0:
3         return steps[n]
4
5     if n==1:
6         steps[n] = 1
7     elif n==2:
8         steps[n] = 2
9     elif n==3:
10        steps[n] = 4
11    else:
12        steps[n] = func_step(steps, n-1) + func_step(steps, n-2) + func_step(steps, n-3)
13
14    return steps[n]
15
16 step=6
17 ret=func_step([0]*(step+1), step)
18 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
19 print("solution 함수의 반환 값은", ret, "입니다.")
```

② 재귀 함수 사용

```
1 def step(n):
2     if n == 1:
3         return 1
4     elif n == 2:
5         return 2
6     elif n == 3:
7         return 4
8     elif n >= 4:
9         return step(n-1)+step(n-2)+step(n-3)
10
11 n = 4
12 ret = step(n)
13
14 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
15 print("solution 함수의 반환 값은", ret, "입니다.")
```

③ 변수만 사용

```
1 def step(n):
2     step1=1
3     step2=2
4     step3=4
5     new_step=0
6
7     if n==3:
8         return step3
9     elif n>=4:
10        for i in range(4,n+1):
11            new_step=step1+step2+step3
12            step1=step2
13            step2=step3
14            step3=new_step
15        return new_step
16
17 n = 6
18 ret = step(n)
19
20 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
21 print("solution 함수의 반환 값은", ret, "입니다.")
```

2. 문제 2

1) 문제 코드

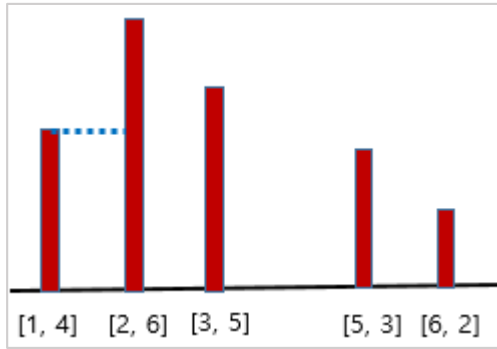
```
1 def solution(walls):
2     answer = 0
3     for i in range(len(walls)):
4         for j in range(i+1, len(walls)):
5             area = 0
6             if walls[i][1] > walls[j][1]:
7                 area = walls[i][1] * (walls[j][0] - walls[i][0])
8             else:
9                 area = walls[j][1] * (walls[j][0] - walls[i][0])
10            if answer < area:
11                answer = area
12        return answer
13
14 #아래는 테스트케이스 출력을 해보기 위한 코드입니다. 아래에는 잘못된 부분이 없으
15 walls = [[1, 4], [2, 6], [3, 5], [5, 3], [6, 2]]
16 ret = solution(walls)
17
18 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
19 print("solution 함수의 반환 값은", ret, "입니다.")
```

2) 문제 개요

- 제시된 과제를 해결하기 위해 작성된 프로그램에서 잘못된 부분을 찾아 수정하는 문제.
- 벽의 위치와 높이를 2 차원 리스트로 저장한 매개변수 walls 에서 각 항목값을 이용하여 최대로 담을 수 있는 물의 용량을 찾아내는 프로그램에서 잘못된 부분을 찾아 수정해야 함.

3) 정답

- 주요 아이디어 정리 : walls = [[1,4], [2,6], [3,5], [5,3], [6,2]] 인 경우



- ✓ walls 의 항목 구성 = [벽의 위치, 벽의 높이]
 - walls[0][0] 은 0 번째 벽의 위치=1
 - walls[0][1] 은 0 번째 벽의 높이=4
 - n 번째 벽의 위치 값 = walls[n][0]
 - n 번째 벽의 높이 값 = walls[n][1]
- ✓ 두 벽 사이에 물을 담을 수 있는 용량 = 두 벽 중 낮은 벽의 높이 * 두 벽 사이의 거리
 - 0 번째 벽과 1 번째 벽 사이에 물을 담을 수 있는 용량 = 0 번째 벽의 높이 * (1 번째 벽 위치 - 0 번째 벽 위치)
 - = walls[0][1] * (walls[1][0] - walls[0][0])
 - ∴ 0 번째 벽의 높이가 낮기 때문.

➔ 주어진 walls 를 이용해서 물을 담을 수 있는 최대 용량을 구하려면, 중첩 for 문을 이용하여 두 개의 벽을 가져와 물을 담을 수 있는 용량을 계산하고 그 중 최댓값을 찾음.

● 정답 코드

```

1  def solution(walls):
2      answer = 0
3      ① for i in range(len(walls)):
4          ② for j in range(i+1, len(walls)):
5              area = 0
6              ③ if walls[i][1] < walls[j][1]:
7                  area = walls[i][1] * (walls[j][0] - walls[i][0])
8              ④ else:
9                  area = walls[j][1] * (walls[j][0] - walls[i][0])
10             ⑤ if answer < area:
11                 answer = area
12         return answer
13
14     #아래는 테스트케이스 출력을 해보기 위한 코드입니다. 아래에는 잘못된 부분이 없으
15     walls = [[1, 4], [2, 6], [3, 5], [5, 3], [6, 2]]
16     ret = solution(walls)
17
18     #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
19     print("solution 함수의 반환 값은", ret, "입니다.")
    
```

- ①. 바깥쪽 for 문을 이용하여 하나의 벽에 대한 인덱스를 i 로 받아오고, 안쪽 for 문을 이용하여 그 다음에 위치하는 벽에 대한 인덱스를 j로 받음.
- ②. 물의 용량을 계산하는 변수 area 를 0 으로 초기화.
- ③. i 번째 벽 높이가 그 다음에 위치한 j 번째 벽보다 높이가 낮으면 물의 용량은 i 번째 벽의 높이 * (j 번째 벽의 위치 - i 번째 벽의 위치)로 계산. 문제 코드는 조건식으로 사용한 벽 높이의 비교 연산식이 반대로 작성되었음.

- ④. i 번째 벽 높이가 그 다음에 위치한 j 번째 벽보다 높이가 낮지 않으면 물의 용량은 j 번째 벽의 높이 * (j 번째 벽의 위치 - i 번째 벽의 위치)로 계산.
- ⑤. 최대 담수 용량을 가지고 있는 `answer` 가 현재 계산한 용량 `area` 보다 적으면 `answer` 를 `area` 로 재할당.

3. 문제 3

1) 문제 코드

```

1 def solution(numbers):
2     answer = []
3     numbers.sort()
4     mid = (len(numbers) - 1) // 2
5     numbers[mid], numbers[len(numbers)-1] = numbers[len(numbers)-1], numbers[mid]
6     left = mid + 1
7     right = len(numbers) - 1
8     while left <= right:
9         numbers[left], numbers[right] = numbers[right], numbers[left]
10        left = left + 1
11        right = right - 1
12    answer = numbers
13    return answer
14
15    #아래는 테스트케이스 출력을 해보기 위한 코드입니다. 아래에는 잘못된 부분이 없으니 위의 코드만 수정하세요.
16    numbers = [7, 3, 4, 1, 2, 5, 6]
17    ret = solution(numbers)
18
19    #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
20    print("solution 함수의 반환 값은", ret, "입니다.")

```

2) 문제 개요

- 제시된 과제를 해결하기 위해 작성된 프로그램에서 잘못된 부분을 찾아 수정하는 문제.
- 주어진 리스트에서 앞의 절반까지는 오름차순으로 정렬하고 절반의 바로 다음 항목부터 마지막 항목까지는 내림차순으로 정렬하도록 작성된 프로그램에서 잘못된 부분을 찾아 수정해야 함. (문제 조건으로 리스트의 항목 개수는 홀수로 지정됨.)

3) 정답

- 주요 아이디어 정리 : 리스트의 가운데 인덱스와 마지막 인덱스 구하기 - 리스트의 항목 개수가 = 7 인 경우

Index :0	1	2	3	4	5	6
7	3	4	1	2	5	6

- 가운데 항목의 인덱스 = (리스트의 길이(7)-1)//2=3
- 마지막 항목의 인덱스 = 리스트의 길이(7) -1

● 정답 코드

```

1 def solution(numbers):
2     answer = []
3     numbers.sort()
4     ① mid = (len(numbers) - 1) // 2
5     ② numbers[mid], numbers[len(numbers)-1] = numbers[len(numbers)-1], numbers[mid]
6     ③ left = mid + 1
7     right = len(numbers) - 2
8     ④ while left <= right:
9         ⑤ numbers[left], numbers[right] = numbers[right], numbers[left]
10        left = left + 1
11        ⑥ right = right - 1
12    answer = numbers
13    return answer
14
15 #아래는 테스트케이스 출력을 해보기 위한 코드입니다. 아래에는 잘못된 부분이 없으니 위의 코드만 수정하세요.
16 numbers = [7, 3, 4, 1, 2, 5, 6]
17 ret = solution(numbers)
18
19 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
20 print("solution 함수의 반환 값은", ret, "입니다.")

```

- ①. 리스트 numbers 의 가운데 인덱스의 값을 구하여 mid 에 저장.
 - 리스트 인덱스는 0 부터 시작하기 때문에 (리스트 길이-1) 을 2 로 나눈 몫으로 가운데 인덱스 값을 구함.
- ②. 가운데 인덱스가 나타내는 항목값과 마지막 인덱스가 나타내는 항목값을 맞교환.
- ③. 리스트의 가운데 이후의 항목부터 리스트의 마지막에서 두 번째 항목까지 내림차순으로 정렬하기 위해서 (가운데 인덱스 값 + 1)을 left 로, 마지막에서 두 번째 항목에 대한 인덱스를 right 로 지정. 리스트의 마지막 항목은 이미 앞줄의 코드를 통해서 교환이 이루어졌기 때문에 문제 코드에서 제시된 대로 right 를 len(numbers) - 1 로 지정하면 원하는 결과를 얻을 수 없음.
- ④. left 값이 right 값보다 작거나 같은 동안
- ⑤. left 값에 해당하는 인덱스의 항목과 right 값에 해당하는 인덱스의 항목을 맞교환.
- ⑥. left 는 1 만큼 증가 / right 는 1 만큼 감소하는 작업을 반복 실행.

4. 문제 4

1) 문제 코드

```

1  def solution(number):
2      answer = ''
3      number_count = [0 for _ in range(10)]
4      while number > 0:
5          number_count[number % 10] += 1
6          number //= 10
7      for i in range(10):
8          if number_count[i] != 0:
9              answer += (str(i) + str(number_count[i]))
10     return answer
11
12     #아래는 테스트케이스 출력을 해보기 위한 코드입니다. 아래에는 잘못된 부분이 없
13     number1 = 2433
14     ret1 = solution(number1)
15
16     #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
17     print("solution 함수의 반환 값은", ret1, "입니다.")
18
19     number2 = 662244
20     ret2 = solution(number2)
21
22     #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
23     print("solution 함수의 반환 값은", ret2, "입니다.")

```

2) 문제 개요

- 제시된 과제를 해결하기 위해 작성된 프로그램에서 잘못된 부분을 찾아 수정하는 문제.
- 매개변수 `number` 로 전달받은 수에서 사용된 숫자들과 각 숫자가 사용된 횟수를 `return` 하되 큰 숫자부터 먼저 문자열에 나타나도록 작성한 프로그램에서 잘못된 부분을 찾아 수정하는 문제.
- 0 부터 9 까지의 인덱스를 갖는 리스트에 `number` 에서 사용된 숫자 별 빈도수를 집계.

3) 정답

- 주요 아이디어 정리 : 어떤 수에서 일의 자리부터 각 자리 숫자를 가져오기
ex) 2307 에서 각 자리 숫자를 추출하기 위한 절차

실행 순서	구분	필요한 계산식	결과
1	일의 자리 숫자	$2307 \% 10$	7
2	십의 자리 숫자	$2307 // 10$ 의 결과 $230 \rightarrow 230 \% 10$	0
3	백의 자리 숫자	$230 // 10$ 의 결과 $23 \rightarrow 23 \% 10$	3
4	천의 자리 숫자	$23 // 10$ 의 결과 $2 \rightarrow 2 \% 10$	2

- 정답 코드

```

1  def solution(number):
2      answer = ''
3      ① number_count = [0 for _ in range(10)]
4      while number > 0:
5          ② number_count[number % 10] += 1
6              number //= 10
7          ③ for i in range(9, 0, -1):
8              ④ if number_count[i] != 0:
9                  answer += (str(i) + str(number_count[i]))
10     return answer
11
12     #아래는 테스트케이스 출력을 해보기 위한 코드입니다. 아래에는 잘못된 부분이 없으
13     number1 = 2433
14     ret1 = solution(number1)
15
16     #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
17     print("solution 함수의 반환 값은", ret1, "입니다.")
18
19     number2 = 662244
20     ret2 = solution(number2)
21
22     #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
23     print("solution 함수의 반환 값은", ret2, "입니다.")

```

- ①. 1 부터 9 까지의 숫자가 나타나는 빈도수를 집계하기 위해 10 개의 항목을 갖는 리스트 number_count 를 생성하고 0 으로 초기화.
- ②. number 가 0 보다 큰 동안 반복.
 - number 를 10 으로 나눈 나머지 값에 해당하는 인덱스의 항목 값을 1 만큼 증가.
 - number 를 10 으로 나눈 몫으로 number 값을 재설정.
- ③. 큰 숫자부터 결과 문자열에 붙이기 위해 range(9, 0,-1) 로 작성하면 9 부터 1 까지의 수가 차례로 i 로 할당됨. 문제에서 제시된 대로 for 문의 반복 범위를 지정할 때 range(10)을 사용하면 0 부터 9 까지의 수가 i 에 할당되고 결과 문자열을 저장하는 answer 에는 작은 수의 빈도 수부터 나타나게 됨.
- ④. 숫자 i 의 빈도수를 저장한 number_count[i]의 값이 0 이 아니면 answer 에 저장된 문자열 끝에 i 를 문자열로 변환한 값과 number_count[i]을 문자열로 변환한 값을 차례로 덧붙임.

5. 문제 5

1) 문제 코드

```
1  #다음과 같이 import를 사용할 수 있습니다.
2  #import math
3
4  def solution(enemies, armies):
5      #여기에 코드를 작성해주세요.
6      answer = 0
7      return answer
8
9  #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
10 enemies1 = [1, 4, 3]
11 armies1 = [1, 3]
12 ret1 = solution(enemies1, armies1)
13 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
14 print("solution 함수의 반환 값은", ret1, "입니다.")
15
16 enemies2 = [1, 1, 1]
17 armies2 = [1, 2, 3, 4]
18 ret2 = solution(enemies2, armies2)
19 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
20 print("solution 함수의 반환 값은", ret2, "입니다.")
```

2) 문제 개요

- 제시된 과제를 해결하기 위해 solution()에 프로그램 코드를 작성하는 문제.
- armies 리스트의 항목 값을 이용하여 enemies 리스트의 항목 값을 이길 수 있는 최대 개수를 구하도록 프로그램을 작성.
- armies 리스트와 enemies 리스트의 항목 값을 정렬한 후 각각의 리스트에서 하나씩 항목을 가져와 이길 수 있는 지 비교해서 이길 수 있는 항목 개수를 구해야 함.
- enemies 의 항목을 이기려면 armies 의 항목 값이 크거나 같아야 함.

3) 정답

```

1 def solution(enemies, armies):
2     answer = 0
3     enemies.sort()
4     armies.sort()
5     i, j = 0, 0
6     while i < len(enemies) and j < len(armies):
7         if enemies[i] <= armies[j]:
8             answer = answer + 1
9             i = i + 1
10            j = j + 1
11        else:
12            j = j + 1
13    return answer
14
15 #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
16 enemies1 = [1, 4, 3]
17 armies1 = [1, 3]
18 ret1 = solution(enemies1, armies1)
19
20 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
21 print("solution 함수의 반환 값은", ret1, "입니다.")
22
23 enemies2 = [1, 1, 1]
24 armies2 = [1, 2, 3, 4]
25 ret2 = solution(enemies2, armies2)
26
27 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
28 print("solution 함수의 반환 값은", ret2, "입니다.")

```

- ①. enemies 리스트와 armies 리스트를 오름차순으로 정렬.
- ②. enemies 의 인덱스로 사용될 변수 i 와 armies 의 인덱스로 사용될 변수 j 를 0 으로 초기화.
- ③. i 값이 enemies 의 항목 개수보다 작고 j 값이 armies 의 항목 개수보다 작은 동안 반복. 즉, 두 리스트에서 비교할 항목이 모두 존재할 동안 반복문 안쪽의 명령들을 실행.
- ④. i 번째 enemies 의 값이 j 번째 armies 값보다 작거나 같으면 armies 가 승리하는 횟수를 저장하는 변수 answer 를 1 만큼 증가시키고 i 와 j 를 모두 1 만큼 증가.
- ⑤. i 번째 enemies 의 값이 j 번째 armies 값보다 크면 armies 의 인덱스인 j 만 1 만큼 증가하여 armies 의 다음 항목을 가리키도록 조정.

6. 문제 6

1) 문제 코드

```

1  #다음과 같이 import를 사용할 수 있습니다.
2  #import math
3
4  def solution(s1, s2, p, q):
5      #여기에 코드를 작성해주세요.
6      answer = ''
7      return answer
8
9  #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
10 s1 = "112001"
11 s2 = "12010"
12 p = 3
13 q = 8
14 ret = solution(s1, s2, p, q)
15
16 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
17 print("solution 함수의 반환 값은", ret, "입니다.")

```

2) 문제 개요

- 제시된 과제를 해결하기 위해 solution()에 프로그램 코드를 작성하는 문제.
- p 진법 수 두 개가 문자열로 형태로 전달된 것을 더하고 그 결과를 q 진법 수의 문자열로 return 하도록 코드를 작성해야 함.


3) 정답

- 주요 아이디어 정리

- ✓ p 진법 수 → 10 진법 수로 변환하기

ex) 3 진법 수 112001 $= 1 \times 3^0 + 0 \times 3^1 + 0 \times 3^2 + 2 \times 3^3 + 1 \times 3^4 + 1 \times 3^5 = 10$ 진법 수 379
 : 오른쪽 끝의 마지막 자리 숫자부터 각 자리 숫자가 의미하는 수를 곱한 것을 더하여 십진수 값을 구함.

- ✓ 10 진법 수 → p 진법 수로 변환하기

ex) 10 진법 수 21 = $\begin{array}{r} 8 \overline{) 21} \\ 8 \overline{) 2} \cdots 5 \\ 0 \cdots 2 \end{array}$  = 8 진법 수 25

: 21 을 8 로 나눈 몫이 0 이 될 때까지 나누고, 나누는 과정에서 나온 나머지들을 역순으로 가져와 8 진법의 각 자리 숫자로 사용.

✓ 형 변환 함수

int(실수 or 문자열, [base]) :	<p>실수의 정수부만 가져오거나 정수로 된 문자열을 정수 형태로 가져옴.</p> <p>매개변수 base(진법) 에 대해서 추가로 값을 지정하면 해당 진법으로 표현된 문자열을 십진수 정수로 변환함.</p> <p>ex) int(1.2) → 1, int('12') → 12, int('110', 2) → 5</p>
str(실수 or 정수) :	<p>실수나 정수로 전달된 값을 문자열 형태로 변환.</p> <p>ex) str(1.2) → '1.2', str(12) → '12'</p>

● 정답 코드

```

1  numbers_int = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2  numbers_char = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
3
4  ① def char_to_int(ch):
5      for i in range(10):
6          if ch == numbers_char[i]:
7              return numbers_int[i]
8
9  ② def int_to_char(val):
10     for i in range(10):
11         if val == numbers_int[i]:
12             return numbers_char[i]
13
14  ③ def convert_scale(num, q):
15     if num == 0:
16         return ""
17     return convert_scale(num // q, q) + int_to_char(num % q)
18
19  ④ def parse_decimal(s, p):
20     num = 0
21     mul = 1
22     for s_i in reversed(s):
23         num += char_to_int(s_i) * mul
24         mul *= p
25     return num
26
27  def solution(s1, s2, p, q):
28     num1 = parse_decimal(s1, p)
29     num2 = parse_decimal(s2, p)
30     answer = convert_scale(num1 + num2, q)
31     return answer
32
33  #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
34  s1 = "112001"
35  s2 = "12010"
36  p = 3
37  q = 8
38  ret = solution(s1, s2, p, q)
39
40  #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
41  print("solution 함수의 반환 값은", ret, "입니다.")

```

- ①. char_to_int() 함수는 정수로 된 문자를 정수로 변환하는 것을 구현한 함수.
 - 매개변수 chr 와 같은 문자를 갖는 number_char 리스트의 항목을 찾아서 해당 항목의 인덱스와 같은 인덱스를 갖는 number_int 리스트의 항목을 return.
- ②. int_to_char() 함수는 정수를 문자로 변환하는 것을 구현한 함수.
 - 매개변수 val 와 같은 값을 갖는 number_int 리스트의 항목을 찾아서 해당 항목의 인덱스와 같은 인덱스를 갖는 number_char 리스트의 항목을 return.
- ③. convert_scale() 함수는 십진수를 q 진법으로 변환하는 것을 구현한 함수
 - 매개변수 num 을 q 로 나눈 몫이 0 이 아닌 동안, num 을 q 로 나눈 몫을 인수로 전달하여 재귀호출하고 동시에 num 을 q 로 나눈 나머지를 문자열로 변환하여 재귀 호출한 결과 뒤에 붙여서 return.
- ④. parse_decimal() 함수는 p 진법 수를 십진수로 변환하는 것을 구현한 함수
 - for 문을 이용하여 매개변수로 받은 정수 문자열의 마지막 문자부터 가져와 그 문자와 10 의 거듭제곱을 곱한 것을 누적 합산하여 십진수로 변환.

4) 다른 코드 제안

```

1  def solution(s1, s2, p, q):
2      answer = ''
3      #p 진법으로 표현된 숫자열을 10 진수 정수로 변환하여 덧셈
4      add_num = int(s1,p) + int(s2,p)
5
6
7      #덧셈 결과를 q 진법으로 변환하기
8      while add_num != 0:
9          b = add_num % q
10         add_num = add_num // q
11         answer = str(b) + answer
12     return answer
13
14     #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
15     s1 = "112001"
16     s2 = "12010"
17     p = 3
18     q = 8
19     ret = solution(s1, s2, p, q)
20
21     #[ 실행 ] 버튼을 누르면 출력 값을 볼 수 있습니다.
22     print("solution 함수의 반환 값은", ret, "입니다.")
    
```

- ①. 파이썬에서 제공하는 형 변환 함수를 사용하여 p 진수로 나타낸 문자열인 s1, s2 를 10 진수로 변환하여 덧셈 연산을 실행.
- ②. 덧셈 결과 add_num 을 q 로 나눈 몫이 0 이 아닌 동안 add_num 을 q 로 나눈 나머지를 문자열로 변환한 후, 결과 값을 갖는 문자열 변수 answer 의 앞에 붙이는 작업을 반복.

7. 문제 7

1) 문제 코드


```

1 def find(parent, u):
2     if u == parent[u]:
3         return u
4     parent[u] = @@@
5     return parent[u]
6
7 def merge(parent, u, v):
8     u = find(parent, u)
9     v = find(parent, v)
10    if u == v:
11        return True
12    @@@
13    return False
14
15 def solution(n, connections):
16     answer = 0
17     parent = @@@
18     for i, connection in enumerate(connections):
19         if merge(parent, connection[0], connection[1]):
20             answer = i + 1
21             break
22     return answer
23
24 #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
25 n = 3
26 connections = [[1, 2], [1, 3], [2, 3]]
27 ret = solution(n, connections)
28
29 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
30 print("solution 함수의 반환 값은", ret, "입니다.")

```

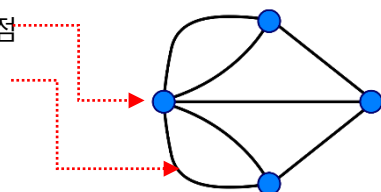
2) 문제 개요

- 제시된 과제를 해결하기 위해 작성된 프로그램에서 빈 곳을 채우는 문제.
- 주어진 노드와 노드 연결 순서에 의해서 사이클이 생성되는 연결 순서가 몇 번인지 찾아서 return 하는 프로그램에서 빈 곳에 알맞은 코드를 채워 넣는 문제.
- 사이클을 찾기 위해 Union-Find 알고리즘이 사용됨.

3) 그래프 이론 정리

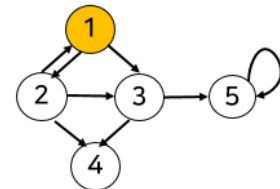
① 그래프(Graph) 소개

- 레온하르트 오일러가 코니히스베르크의 다리 문제를 풀기 위해 처음 도입한 이론.
- 7 개의 다리를 한 번씩만 건너서 출발지로 다시 돌아올 수 있는 지 확인하는 문제.
- 한 붓 그리기, 오일러의 경로는 모든 간선을 한 번씩 방문하는 유한 그래프의 대표적인 예.
- 그래프의 구성
 - 정점(Vertex) 혹은 노드(Node) : 그래프의 각 지점
 - 간선(Edge) : 노드 사이를 연결하는 선을 간선

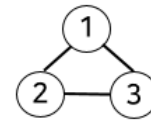


② 그래프 용어 정리

- 그래프는 (V, E) 의 쌍으로 표현.
- 각 정점(Vertex) 혹은 노드(Node)는 그래프의 특정 지점을 나타내며 숫자 혹은 알파벳으로 표시
- 간선(Edge) : 두 정점을 잇는 개체로 방향성이 있는 그래프에서는 화살표로 표시하고 방향이 없는 그래프에서는 일반 선으로 표시.
- 경로(Path) : 거쳐가는 정점들의 순서를 열거한 것.
ex) 2 에서 5 까지의 경로($2 \rightarrow 3 \rightarrow 5$ or $2 \rightarrow 1 \rightarrow 3 \rightarrow 5$)
- 경로의 길이(Path Length) : 간선의 수 or 가중치
- 차수(Degree) : 정점에 연결되어 있는 선의 개수
 - 입력차수(in-degree) : 정점으로 들어오는 선의 수
 - 출력차수(out-degree) : 정점에서 나가는 선의 수.
- 인접한다 : 정점과 정점 사이에 간선이 존재하는 것을 표현하는 용어.
 - 방향성 그래프 : 3 은 1 에 인접한다. / 1 은 3 에 인접하지 않는다.
 - 무 방향성 그래프 : 1 과 3 은 서로 인접한다.



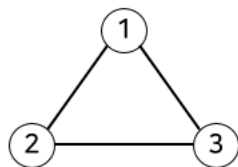
Directed Graph



Undirected Graph

③ 그래프의 종류

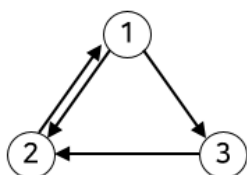
- 무방향 그래프(Undirected Graph) : 간선에 방향이 없는 그래프



노드의 집합 $V = \{ 1, 2, 3 \}$

간선의 집합 $E = \{ (1, 2), (1, 3), (2, 3) \}$
 $= \{ (2, 1), (3, 1), (2, 3) \}$

- 방향 그래프(Directed Graph) : 간선에 방향이 있는 그래프



노드의 집합 $V = \{ 1, 2, 3 \}$

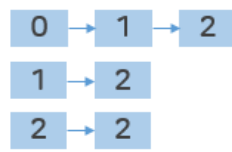
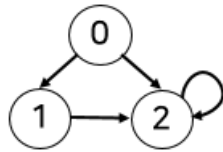
간선의 집합 $E = \{ (1, 2), (1, 3), (2, 1), (3, 2) \}$
 (출발지, 도착지) 순으로 표현.

④ 그래프의 표현

• 인접 리스트(Adjacency List)

- 출발 노드를 키로 하여 출발 노드에서 갈 수 있는 노드를 값으로 표현.

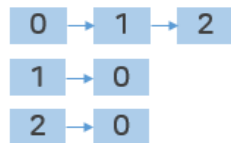
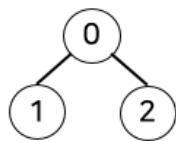
Directed Graph



0 노드에서 갈 수 있는 곳은 1, 2.
1 노드에서 갈 수 있는 곳은 2.
2 노드에서 갈 수 있는 곳은 2.

$E = \{ (0, 1), (0, 2), (1, 2), (2, 2) \}$

Undirected Graph



0 노드에서 갈 수 있는 곳은 1, 2.
1 노드에서 갈 수 있는 곳은 0.
2 노드에서 갈 수 있는 곳은 0.

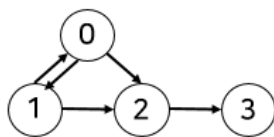
$E = \{ (0, 1), (0, 2), (1, 0), (2, 0) \}$

프로그래밍할 때는 양 방향
경로를 모두 지정해야 함.

• 인접 행렬(Adjacency Matrix)

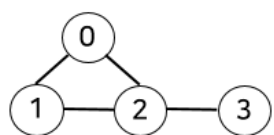
- 행 번호를 출발 노드, 열 번호를 도착 노드로 사용.
- 출발 노드에서 도착 노드로 갈 수 있는 곳에 대해서 각 번호가 교차하는 지점의 값을 1로 할당.

Directed Graph



	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	0	0	0	1
3	0	0	0	0

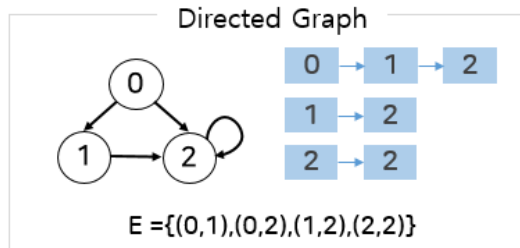
Undirected Graph



	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

⑤ 그래프를 파이썬 코드로 표현하는 방법

ex) 아래와 같은 Directed Graph 표현하기



- 인접 리스트(Adjacency List) : 2 차원 리스트를 활용.

<pre> gl=[[] for _ in range(3)] gl[0].append(1) gl[0].append(2) gl[1].append(2) gl[2].append(2) </pre>	<p>→ - 3 개의 빈 리스트를 항목으로 갖는 2 차원 리스트를 생성.</p> <p>→ - 0 번 노드에서 갈 수 있는 1, 2 를 0 번 리스트의 항목 값으로 추가.</p> <p>→ - 1 번 노드에서 갈 수 있는 2 를 1 번 리스트의 항목 값으로 추가.</p> <p>→ - 2 번 노드에서 갈 수 있는 2 를 2 번 리스트의 항목 값으로 추가.</p>
--	--

< 결과 >

$[[1, 2], [2], [2]]$

- 인접 행렬(Adjacency Matrix)

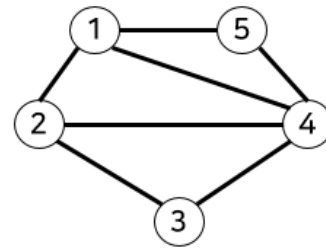
<pre> gm = [[0, 1, 1], [0, 0, 1], [0, 0, 1]] </pre>	<p>→ - 0 번 노드에서 갈 수 있는 1, 2 를 표현하기 위해 gm[0][1] 과 gm[0][2] 를 1 로 할당.</p> <p>→ - 1 번 노드에서 갈 수 있는 2 를 표현하기 위해 gm[1][2] 를 1 로 할당.</p> <p>→ - 2 번 노드에서 갈 수 있는 2 를 표현하기 위해 gm[2][2] 를 1 로 할당.</p>
---	---

< 결과 >

$[[0, 1, 0], [0, 0, 1], [0, 0, 1]]$

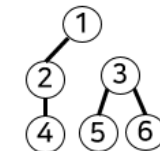
⑥ 그래프(Graph)와 순환(Cycle)

- 순환(Cycle) : 그래프에서 출발점과 도착점이 같은 경로를 말함($3 \rightarrow 4 \rightarrow 5 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 3$)
- 단일 순환(Simple Cycle)
 - 순환 경로에서 거치는 모든 정점이 다른 경우.
 - 반복되는 정점이 출발점과 도착점.
 - $3 \rightarrow 4 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 3$

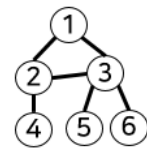


⑦ 그래프의 종류

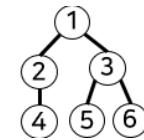
- 비순환 그래프(Acyclic Graph)
 - 순환이 없는 그래프.
- 연결 그래프(Connected Graph)
 - 모든 정점 사이에 두 정점을 양 끝으로 하는 경로가 존재하는 그래프.
- 트리(Tree)
 - 연결된 비순환 무방향성 그래프



비순환 그래프
무방향 그래프



연결 그래프
무방향 그래프

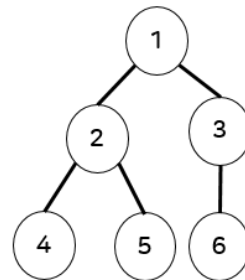


연결 그래프
비순환 그래프
무방향 그래프

⑧ 그래프 순회(Graph Traversals)

- 그래프 탐색(Graph Search) 라고도 함.
- 그래프에 있는 모든 노드들을 방문하는 방법
- 깊이 우선 탐색 : Depth First Search(DFS)
 - 스택, 재귀로 구현.
 - 백트래킹에 유용
 - 세로 방향에 있는 노드를 먼저 방문하는 방법.
 - 오른쪽 트리의 노드 방문 순서
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

ex)



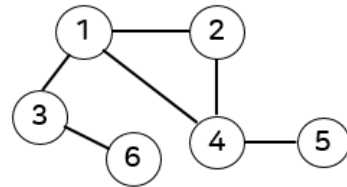
- 너비 우선 탐색 : Breadth First Search(BFS)
 - 큐로 구현.
 - 그래프의 최단 경로를 구하는 문제에 유용.
 - 가로 방향에 있는 노드를 먼저 방문하는 방법.
 - 오른쪽 트리의 노드 방문 순서
 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$

⑨ 그래프(Graph)에서 그래프 순회

♦ 깊이 우선 탐색 : Depth First Search(DFS)

- 스택 구조를 사용.
- 오른쪽 그래프의 노드 방문 순서
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

ex)

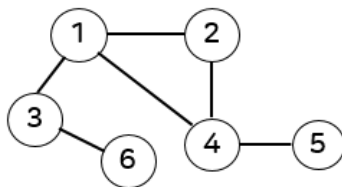


♦ 너비 우선 탐색 : Breadth First Search(BFS)

- 시작점에서 거리(간선의 수)를 하나씩 늘리면서 찾음.
- 오른쪽 그래프의 노드 방문 순서
 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5$

⑩ 깊이 우선 탐색(DFS)의 구현

: 탐색할 그래프



♦ 재귀호출 사용

- v 노드를 방문한 것으로 체크.
- 방문한 순서를 저장하는 result 에 v 를 추가.
- v 노드와 연결되어 있는 노드를 방문한 적이 없으면 재귀 호출.

```

1  result=[]
2
3  def dfs(graph, v, visited):
4      visited[v]=True
5      result.append(v)
6      for i in graph[v]:
7          if not visited[i]:
8              dfs(graph, i, visited)
9      return result
10
11  graph = [
12      [],
13      [2,3,4],
14      [1,4],
15      [1,6],
16      [1,2,5],
17      [4],
18      [3]
19  ]
20
21  visited = [False] * 7
22  print(dfs(graph, 1, visited))

```

< 결과 >

[1, 2, 4, 5, 3, 6]

◆ 딕셔너리 사용

```

1  def dfs2(graph, start):
2      visited2 = list()
3      stack = list()
4      stack.append(start)
5
6      while stack:
7          node = stack.pop()
8          if node not in visited2:
9              visited2.append(node)
10             stack.extend(reversed(graph[node]))
11      return visited2
12
13  graph = {
14      '1': ['2', '3', '4'],
15      '2': ['1', '4'],
16      '3': ['6'],
17      '4': ['1', '2', '5'],
18      '5': ['4'],
19      '6': ['3']
20  }
21
22  print(dfs2(graph, '1'))

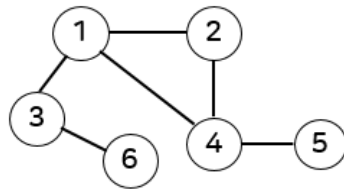
```

< 결과 >

['1', '2', '4', '5', '3', '6']

- 방문한 노드를 저장할 리스트 생성.
- 다음에 방문할 노드로 시작 노드를 추가.
- 현재 노드를 방문한 적이 없으면 visited2 에 추가하고 현재 노드와 연결되어 있는 노드의 순서를 뒤집어서 stack 에 추가.

- ⑪ 너비 우선 탐색(BFS)의 구현
: 탐색할 그래프



• deque 사용

```

1 from collections import deque
2 result = []
3 def bfs(graph, start, visited):
4     q = deque([start])
5     visited[start] = True
6     while q:
7         v = q.popleft()
8         result.append(v)
9         for i in graph[v]:
10            if not visited[i]:
11                q.append(i)
12                visited[i] = True
13    return result
14
15 visited = [False] * 7
16 graph = [
17     [],
18     [2, 3, 4],
19     [1, 4],
20     [1, 6],
21     [1, 2, 5],
22     [4],
23     [3]
24 ]
25 print(bfs(graph, 1, visited))
    
```

- start 노드를 deque 에 추가.
- start 노드를 방문한 것으로 저장.
- q 에 데이터가 있는 동안 q 의 가장 왼쪽(앞쪽) 데이터를 꺼내서 v 에 저장.
- 노드 방문 순서를 저장하는 result 에 추가.
- v 노드와 연결되어 있는 노드를 가져와 방문한 적이 없으면 q 에 넣고 visited 에 방문한 것으로 저장.

< 결과 >

[1, 2, 3, 4, 6, 5]

• 딕셔너리 사용

```

1 def bfs2(graph2, start_node):
2     visited2 = list()
3     q2 = list()
4     q2.append(start_node)
5     while q2:
6         node = q2.pop(0)
7         if node not in visited2:
8             visited2.append(node)
9             q2.extend(graph2[node])
10    return visited2
11
12 graph2 = {
13     '1': ['2', '3', '4'],
14     '2': ['1', '4'],
15     '3': ['6'],
16     '4': ['1', '2', '5'],
17     '5': ['4'],
18     '6': ['3']
19 }
20 print(bfs2(graph2, '1'))
    
```

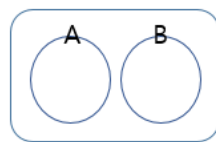
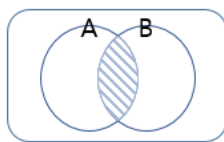
- 시작 노드 번호를 q2 에 저장.
- q2 의 맨 앞의 항목을 pop 하여 node 에 저장.
- node 를 방문한 적이 없으면 visited2 에 추가하고 현재 노드와 연결되어 있는 노드를 추가.

< 결과 >

['1', '2', '3', '4', '6', '5']

4) 합집합 – 찾기 (Union find)알고리즘

- 서로소 집합(disjoint set) 자료 구조 또는 병합 찾기 집합(merge find set) 자료구조라고도 함.
- 서로소 집합인지 판별하기 위해 사용하는 알고리즘.
- 어떤 두 노드를 선택했을 때 두 노드가 같은 그래프에 속하는 지 찾아내는 알고리즘.
- 무방향 그래프에서 사이클 판별 시에 사용됨.



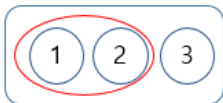
오른쪽 그림처럼 공통된 부분이 없는 두 집합의 관계를 서로소 라고 함.

- 어떤 노드 두 개를 선택했을 때 두 노드가 같은 그래프에 속하는 지 찾아내는 알고리즘으로 아래 두 가지 연산을 수행.
 - 찾기(Find) : 대표(부모, 루트)를 찾는 함수.
 - 합집합(Union) : 하나의 대표로 합해 주는 함수.



각 노드의 대표를 자기 자신으로 초기화.

Find(1) → 대표=1, Find(2) → 대표=2, Find(3) → 대표=3

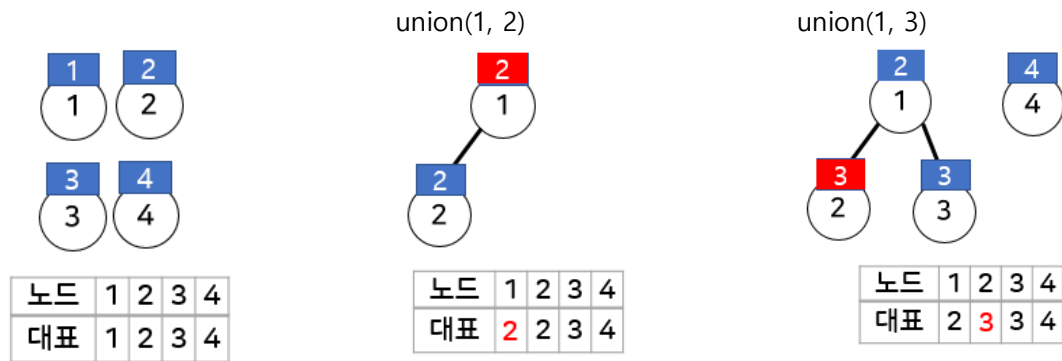


Union(1, 2) → 대표 = 2

Find(1) → 대표=2, Find(2) → 대표=2, Find(3) → 대표=3

※ '대표' 라는 용어 대신 부모(parent), 루트(root) 라고도 함.

- 합집합 – 찾기(Find-Union) 알고리즘의 구성
 - ① 대표 노드 번호를 자신의 노드 번호로 초기화
 - ② 파인드(Find) : 하나의 노드가 어떤 대표 노드에 속하는 지 확인.
(노드 번호와 대표 번호가 같지 않은 경우 계속해서 대표 번호를 찾음.)
 - ③ 유니온(Union) : 서로 다른 대표 노드에 속하는 두 노드를 하나의 대표 노드에 속하도록 병합.



- union(1, 2) 실행 후 노드 1 과 2 는 대표가 같으므로 같은 집합에 속함.
- union(1, 3) 실행 후 노드 2 의 대표가 3 으로 변경. 1 의 대표는 2, 2 와 3 의 대표는 3 이므로 결론적으로 노드 1, 2, 3 은 같은 집합에 속함.

♦ 합집합-찾기(union-find) 를 파이썬 코드로 구현

```

1 def find(parent, value):
2     ① if value == parent[value]:
3         return value
4     ② return find(parent, parent[value])
5
6 def union(parent, u, v):
7     u = find(parent, u)
8     v = find(parent, v)
9     if u < v:
10        ③ parent[u] = v
11    else:
12        parent[v] = u
13    return parent
14
15 def solution(n, connections):
16     ④ parent = [ i for i in range(n+1)]
17     ⑤ for connection in connections:
18         parent = union(parent, connection[0], connection[1])
19
20     # 각 원소의 부모에 들어간 값
21     print("각 원소의 대표(부모)값", parent)
22     # 각 원소의 진짜 부모 - 속한 집합 확인
23     print("각 원소가 속한 집합 확인하기", end="")
24     for i in range(1, n+1):
25         print(find(parent, i), end=' ')
26
27 n = 4
28 connections = [[1, 2], [1, 3]]
29 solution(n, connections)

```

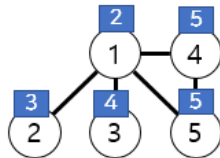
- ①. 노드와 노드의 대표 값이 같으면 노드를 리턴.
- ②. 노드와 노드의 대표 값이 같지 않으면 노드의 대표 값에 대한 대표 값을 찾기 위해 재귀 호출.
- ③. u 노드의 대표 값이 v 노드의 대표 값보다 작으면
 - u 노드의 대표 값을 v 노드의 대표 값으로 변경.
 - 그렇지 않으면 v 노드의 대표 값을 u 노드의 대표 값으로 변경.

- ④. 각 노드의 대표 값을 노드 자신으로 초기화.
- ⑤. connections 의 항목으로 들어온 노드 쌍에 대해서 합집합을 수행.

< 결과 >

각 원소의 대표(부모)값 [0, 2, 3, 3, 4]
 각 원소가 속한 집합 확인하기3334

- ♦ 합집합-찾기(Union-Find)의 문제점



노드 수 = 5

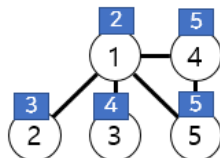
연결 순서 = [[1, 2], [1, 3], [1, 4], [4, 5], [1, 5]]

노드	0	1	2	3	4	5
대표	0	2	3	4	5	5

➔ 대표를 찾는 과정에서 재귀 호출로 인한 시간 낭비가 발생.

```
def find(parent, value):
    if value == parent[value]:
        return value
    return find(parent, parent[value])
```

- ♦ 합집합-찾기(Union-Find) 경로 압축



노드 수 = 5

연결 순서 = [[1, 2], [1, 3], [1, 4], [4, 5], [1, 5]]

노드	0	1	2	3	4	5
대표	0	5	3	5	5	5

➔ 대표 값을 찾기 위한 순회 중 방문한 각 원소들이 직접 대표 원소를 가리키도록 갱신.

```
def find(parent, value):
    if value == parent[value]:
        return value
    parent[value]=find(parent, parent[value])
    return parent[value]
```

< 결과 >

각 원소의 대표(부모)값: [0, 5, 3, 5, 5, 5]
 각 원소가 속한 집합 확인하기:5 5 5 5 5

5) 그래프 사이클 연결 확인

노드 수 = 3

연결 순서 = [[1, 2], [1, 3], [2, 3]]

	1 번째 연결 [1,2]	2 번째 연결 [1, 3]	3 번째 연결 [2, 3]																																
<table border="1"> <tr><td>노드</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>대표</td><td>1</td><td>2</td><td>3</td></tr> </table>	노드	1	2	3	대표	1	2	3	<table border="1"> <tr><td>노드</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>대표</td><td>2</td><td>2</td><td>3</td></tr> </table>	노드	1	2	3	대표	2	2	3	<table border="1"> <tr><td>노드</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>대표</td><td>2</td><td>3</td><td>3</td></tr> </table>	노드	1	2	3	대표	2	3	3	<table border="1"> <tr><td>노드</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>대표</td><td>2</td><td>3</td><td>3</td></tr> </table>	노드	1	2	3	대표	2	3	3
노드	1	2	3																																
대표	1	2	3																																
노드	1	2	3																																
대표	2	2	3																																
노드	1	2	3																																
대표	2	3	3																																
노드	1	2	3																																
대표	2	3	3																																

→ 연결하려는 노드의 대표(루트, 부모) 노드가 같을 때 사이클 생성.

6) 합집합-찾기를 활용하여 사이클 찾기

```

1 def find(parent, value):
2     if value == parent[value]:
3         return value
4     parent[value]=find(parent, parent[value])
5     return parent[value]
6
7 def union(parent, u, v):
8     u = find(parent, u)
9     v = find(parent, v)
10    ① if u == v:
11        return True
12    ② parent[u] = v
13
14    return False
15
16 def solution(n, connections):
17     answer=0
18     parent = [i for i in range(n+1)]
19    ③ for i, connection in enumerate(connections):
20    ④     if (union(parent, connection[0], connection[1])==True):
21    ⑤         answer = i + 1
22         break
23     print("사이클이 발생한 union 순번", answer)
24
25     n = 3
26     connections=[[1,2],[1,3],[2,3]]
27     solution(n, connections)
28     n = 5
29     connections=[[1,2],[1,3],[1,4],[4,5],[1,5]]
30     solution(n, connections)
31     n = 6
32     connections=[[1,2],[3,6],[4,5],[4,6]]
33     solution(n, connections)

```

①. 연결하려는 두 노드의 대표 노드가 같으면 사이클이 생성.

- ②. 대표 노드가 다른 유니온 연산을 수행.
- ③. 인덱스 번호와 연결 순서를 i, connection 에 받음.
- ④. union() 함수를 이용하여 연결하려는 두 노드의 대표 노드가 같은지 확인.
- ⑤. 연결 순서는 1 번부터 시작하지만 인덱스 번호는 0 부터 시작하므로 인덱스 번호 i + 1 을 연결 순서 값으로 산출하여 answer 에 할당.

7) 정답

- 주요 아이디어 정리
- ✓ 합집합-찾기(Union-Find) 알고리즘의 find 와 union 연산 과정대로 작성된 프로그램의 빈 칸을 완성.

● 정답 코드

```

1 def find(parent, u):
2     ① if u == parent[u]:
3         return u
4     ② parent[u] = find(parent, parent[u])
5     return parent[u]
6
7 def merge(parent, u, v):
8     u = find(parent, u)
9     v = find(parent, v)
10    ③ if u == v:
11        return True
12    ④ parent[u] = v
13    return False
14
15 def solution(n, connections):
16     answer = 0
17     ⑤ parent = [i for i in range(n+1)]
18     ⑥ for i, connection in enumerate(connections):
19         ⑦ if merge(parent, connection[0], connection[1]):
20             answer = i + 1
21             break
22     return answer
23
24 #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
25 n = 3
26 connections = [[1, 2], [1, 3], [2, 3]]
27 ret = solution(n, connections)
28
29 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
30 print("solution 함수의 반환 값은", ret, "입니다.")
    
```

❖ find() 함수 : 매개변수 u 로 전달된 노드의 대표 노드를 찾는 함수.

- ①. u 와 u 의 대표 노드가 같으면 노드를 리턴.
- ②. u 와 u 의 대표 노드가 다른 재귀 호출을 이용하여 u 의 대표 노드의 대표 노드를 찾음.

- ❖ `merge()` 함수 : `u` 와 `v` 가 나타내는 노드의 대표 노드를 찾고 두 노드의 대표 노드가 같으면 `True` 를 `return` 하고, 두 노드의 대표 노드가 다르면 `u` 의 대표 노드를 `v` 로 변경한 뒤에 `False` 를 `return`.
- ③. `u` 의 대표 노드와 `v` 의 대표 노드가 같으면 `u` 와 `v` 를 연결할 경우 사이클이 생성되므로 `True` 를 `return`.
- ④. `u` 의 대표 노드와 `v` 의 대표 노드가 다르면 `u` 와 `v` 를 연결해도 사이클이 생성되지 않으므로 `False` 를 `return` 하고, `u` 와 `v` 가 연결되기 때문에 `u` 의 대표 노드를 `v` 로 변경.
- ❖ `solution()` 함수 : 사이클이 생성되는 연결 횟수를 리턴.
- ⑤. 1 번 노드부터 `n` 번 노드까지의 대표 노드를 자기 자신으로 갖도록 초기화.
- ⑥. 인덱스와 연결할 노드 리스트를 `for` 문을 이용하여 `i`, `connection` 에 받음.
- ⑦. 두 노드를 연결한 후 사이클이 발생하면 해당 인덱스+1 을 `answer` 에 할당.

8. 문제 8

1) 문제 코드

```

1 def func_a(a, b):
2     mod = a % b
3     while mod > 0:
4         a = b
5         b = mod
6         mod = a % b
7     return b
8
9 def func_b(n):
10    answer = 0
11    for i in range(1, n+1):
12        if func_000(000):
13            answer += 1
14    return answer
15
16 def func_c(p, q):
17     if p % q == 0:
18         return True
19     else:
20         return False
21
22 def solution(a, b, c):
23     answer = 0
24     gcd = func_000(func_000(000)000)
25     answer = func_000(000)
26     return answer
27
28 #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
29 a = 24
30 b = 9
31 c = 15
32 ret = solution(a, b, c)
33 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
34 print("solution 함수의 반환 값은", ret, "입니다.")

```

2) 문제 개요

- 문제 코드 안에 작성된 함수를 파악한 후, 제시된 과제를 해결하기 위한 알고리즘대로 알맞은 함수를 호출하도록 코드를 완성하는 문제.
- 유클리드 호제법을 이용하여 세 수의 최대공약수를 구한 뒤, 산출된 최대공약수의 약수의 개수를 집계하여 return 하기 위해 알맞은 함수와 인수를 적는 문제.

3) 24, 9, 15 의 공약수의 개수 구하기

24 의 약수	1, 2, 3, 4, 6, 8, 12, 24	→ 세 수의 공약수 = 1, 3.
9 의 약수	1, 3, 9	→ 공약수의 개수 = 2 개
15 의 약수	1, 3, 5, 15	→ 최대공약수 = 3

- 최대공약수 구하기

```
def gcd(a,b,c):
    answer=0
    i=1
    while i<=a and i<=b and i<=c:
        if a%i==0 and b%i==0 and c%i ==0:
            answer =i
            i+=1
    return answer

print(gcd(24,9,15))
```

- 공약수의 개수 구하기

```
def cm_cnt(a,b,c):
    answer=0
    i=1
    while i<=a and i<=b and i<=c:
        if a%i==0 and b%i==0 and c%i ==0:
            answer +=1
            i+=1
    return answer

print(cm_cnt(24,9,15))
```

4) 유클리드 호제법

- 두 자연수의 최대공약수를 구하는 알고리즘.

- 두 자연수 a, b 에 대해서 a 가 b 보다 큰 경우 a 를 b 로 나눈 나머지를 r 이라고 할 때, a 와 b 의 최대공약수는 b 와 r 의 최대공약수와 같다는 특징을 이용.
- a 를 b 로 나눈 나머지 r 을 구하고, b 를 r 로 나눈 나머지 r_1 을 구하고, 다시 r 을 r_1 로 나눈 나머지를 구하는 연산을 반복하여 나머지를 0 으로 나오게 하는 나누는 수가 최대 공약수.
- 큰 수들의 최대공약수를 구할 때 많이 사용.
- 예) 72 와 27 의 최대공약수 구하기

a(큰 수)	b(작은 수)	r(a%b)
72	27	18
27	18	9
18	9	0

∴ 72 와 27 의 최대공약수 = 9

- 파이썬 코드로 구현
 - while 문을 사용

```
def gcd(a,b):
    r=a%b
    print(a, b, r)
    while r>0:
        a=b
        b=r
        r=a%b
        print(a, b, r)

    return b

print(gcd(72,27))
```

< 결과 >

```
a b r
72 27 18
27 18 9
18 9 0
9
```

- 재귀 함수 사용

```
def gcd2(a,b):
    if b==0:
        return a
    else:
        print(a,b,a%b)
        return gcd2(b,a%b)
    return b

print(gcd2(72,27))
```

< 결과 >

```
a b a%b
72 27 18
27 18 9
18 9 0
9
```

4) 정답


```

1  def func_a(a, b):
2      ① mod = a % b
3      while mod > 0:
4          ② a = b
5              b = mod
6              mod = a % b
7      ③ return b
8
9  def func_b(n):
10     answer = 0
11     for i in range(1, n+1):
12         ④ if func_c(n,i):
13             answer += 1
14     return answer
15
16 def func_c(p, q):
17     if p % q == 0:
18         ⑤ return True
19     else:
20         return False
21
22 def solution(a, b, c):
23     answer = 0
24     ⑥ gcd = func_a(func_a(a,b),c)
25     ⑦ answer = func_b(gcd)
26     return answer
27
28 #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
29 a = 24
30 b = 9
31 c = 15
32 ret = solution(a, b, c)
33
34 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
35 print("solution 함수의 반환 값은", ret, "입니다.")

```

- ❖ func_a() 함수 : 유클리드 호제법을 이용하여 a, b 의 최대공약수를 구하는 함수.
 - ①. a 를 b 로 나눈 나머지를 구하여 mod 로 저장.
 - ②. b → a로, mod → b로 바꾼 뒤 a 를 b 로 나눈 나머지를 다시 구하여 mod 에 저장하는 명령을 mod 가 0 보다 큰 동안 반복 실행.
 - ③. mod 값이 0 이 될 때의 b 값을 return.
- ❖ func_b() 함수 : 약수의 개수를 구하는 함수
 - ④. for 문을 이용하여 1 부터 n 까지의 수를 i 로 가져온 후, 함수 func_c()를 이용하여 i 가 n 의 약수인지 판별하여 약수의 개수를 집계.
- ❖ func_c() 함수 : q 가 p 의 약수인지 판별하여 그 결과를 return 하는 함수.
 - ⑤. p 를 q 로 나누었을 때 나눈 나머지가 0 이면 q 는 p 의 약수이므로 True 를 return, 그렇지 않으면 False 를 return .
- ❖ solution() 함수 : 세 수의 공약수의 개수를 리턴.
 - ⑥. func_a() 를 이용하여 세 수 a, b, c 의 최대공약수를 구해서 변수 gcd 에 저장.
 - ⑦. 함수 func_b() 를 이용하여 최대공약수 gcd 의 약수의 개수를 구하여 answer 에 저장.

9. 문제 9

1) 문제 코드

```

1  #다음과 같이 import를 사용할 수 있습니다.
2  #import math
3
4  def solution(number, target):
5      # 여기에 코드를 작성해주세요.
6      answer = 0
7      return answer
8
9  #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
10 number1 = 5
11 target1 = 9
12 ret1 = solution(number1, target1)
13 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
14 print("solution 함수의 반환 값은", ret1, "입니다.")
15
16 number2 = 3
17 target2 = 11
18 ret2 = solution(number2, target2)
19 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
20 print("solution 함수의 반환 값은", ret2, "입니다.")

```

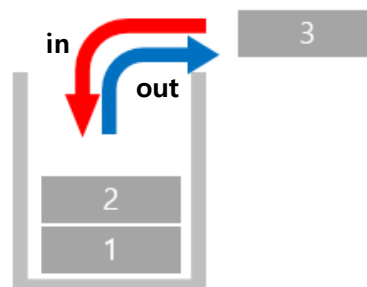
2) 문제 개요

- 제시된 과제를 해결하기 위해 프로그램을 작성하는 문제.
- number 가 target 이 될 때까지 곱하기 2, 더하기 1, 빼기 1 연산 중 최소 연산 수행 횟수를 구하는 프로그램을 작성해야 함.
- number 에 곱하기 2, 더하기 1, 빼기 1 연산을 한 결과를 차례로 큐에 저장.
- 큐에 저장된 결과 값을 먼저 저장된 순서대로 하나씩 가져와 다시 곱하기 2, 더하기 1, 빼기 1 연산을 수행 → 결과를 다시 큐에 저장 → 큐에 결과값 꺼내는 과정을 큐에서 가져오는 결과값이 target 에 도달할 때까지 반복 수행.

3) 스택(Stack) 자료구조

① 스택(Stack) 자료구조 소개

- LIFO(Last In First Out: **후입선출**
(마지막으로 들어온 데이터가 가장 먼저 나감)방식으로 자료를 관리하는 구조
- top : 스택에 들어 있는 데이터 중 가장 위(뒤).
- append() / put() : 마지막(top) 위(뒤)에 데이터 저장.
- pop() / get() : 마지막(top) 데이터를 꺼냄.



② 스택을 구현하는 방법

- 리스트를 이용한 스택

```
stack=[]  
  
stack.append(1)  
stack.append(2)  
stack.append(3)  
print(stack)  
  
stack.pop()  
print(stack)  
stack.pop()  
print(stack)
```

- stack 뒤에 1 을 추가
- stack 뒤에 2 을 추가
- stack 뒤에 3 을 추가
- stack 의 마지막 항목 삭제
- stack 의 마지막 항목 삭제

< 결과 >

```
[1, 2, 3]  
[1, 2]  
[1]
```

- deque 를 이용한 스택 : 파이썬이 스택과 큐를 쉽게 구현하기 위해 제공하는 모듈.

```
from collections import deque  
  
stack2=deque()  
stack2.append(1)  
stack2.append(2)  
stack2.append(3)  
print(stack2)  
  
stack2.pop()  
print(stack2)  
stack2.pop()  
print(stack2)
```

- deque 객체를 생성.
- stack 뒤에 1 을 추가
- stack 뒤에 2 을 추가
- stack 뒤에 3 을 추가
- stack 의 마지막 항목 삭제

< 결과 >

```
deque([1, 2, 3])  
deque([1, 2])  
deque([1])
```

- LifoQueue 를 이용한 스택 : 파이썬 큐 모듈에서 스택 구현을 위해 제공.

```

from queue import LifoQueue

stack3=LifoQueue(maxsize=3)
stack3.put(1)
stack3.put(2)
stack3.put(3)
print("스택 Size:", stack3.qsize())

print(stack3.get())
print(stack3.get())
print("스택 Size:", stack3.qsize())

```

- LifoQueue 객체를 생성.
 - stack 뒤에 1 을 추가
 - stack 뒤에 2 을 추가
 - stack 뒤에 3 을 추가
 - stack 의 마지막 항목 삭제

< 결과 >

```

스택 Size: 3
3
2
스택 Size: 1

```

4) 큐(Queue) 자료구조

① 큐(Queue) 자료구조 소개

- FIFO(First In First Out: 선입선출) (먼저 들어온 데이터가 가장 먼저 나감)방식으로 자료를 관리하는 구조.



② 큐를 구현하는 방법

- 리스트를 이용한 큐

```

queue=[]

queue.append(1)
queue.append(2)
queue.append(3)
print(queue)

queue.pop(0)
print(queue)
queue.pop(0)
print(queue)

```

- queue 뒤에 1 을 추가
 - queue 뒤에 2 을 추가
 - queue 뒤에 3 을 추가
 - queue 의 항목들을 앞에 있는 순서대로 삭제

< 결과 >

```

[1, 2, 3]
[1, 2]
[1]

```

- deque 를 이용한 큐

<code>from collections import deque</code>	- deque 객체 생성.
<code>queue2=deque()</code>	- queue 뒤에 1 을 추가
<code>queue2.append(1)</code>	- queue 뒤에 2 을 추가
<code>queue2.append(2)</code>	- queue 뒤에 3 을 추가
<code>queue2.append(3)</code>	
<code>print(queue2)</code>	
<code>queue2.popleft()</code>	- queue 의 항목들을 앞에
<code>print(queue2)</code>	있는 순서대로 삭제
<code>queue2.popleft()</code>	
<code>print(queue2)</code>	

< 결과 >

```
deque([1, 2, 3])
deque([2, 3])
deque([3])
```

- Queue 를 이용한 큐

<code>from queue import Queue</code>	- Queue 객체 생성.
<code>queue3=Queue(maxsize=3)</code>	- queue 뒤에 1 을 추가
<code>queue3.put(1)</code>	- queue 뒤에 2 을 추가
<code>queue3.put(2)</code>	- queue 뒤에 3 을 추가
<code>queue3.put(3)</code>	
<code>print("큐 Size:", queue3.qsize())</code>	
<code>print(queue3.get())</code>	- queue 의 항목들을 앞에
<code>print(queue3.get())</code>	있는 순서대로 삭제
<code>print("큐 Size:", queue3.qsize())</code>	- queue 에 있는 항목 개수 출력.

< 결과 >

```
큐 Size: 3
1
2
큐 Size: 1
```

5) queue 모듈

① queue 모듈 소개

: 큐(queue), 스택(stack), 우선순위 큐(priority queue) 형태의 자료구조와 그에 관련된 메소드를 제공하는 모듈.

② 제공하는 클래스

queue.Queue(maxsize)	- 선입선출(FIFO) 방식으로 항목을 넣고 빼는 큐를 생성.
queue.LifoQueue(maxsize)	- 후입선출(LIFO) 방식으로 항목을 넣고 빼는 후입선출 큐(스택)를 생성.
queue.PriorityQueue(maxsize)	- 우선순위 큐 객체를 생성. - 항목을 우선순위 큐에 넣을 때, 우선순위 값과 항목을 튜플로 묶어 입력하고 우선순위 값이 낮은 항목을 먼저 빼는 방식을 가짐.

➔ maxsize 에 값을 넘겨서 큐에 넣을 수 있는 항목의 개수를 지정할 수 있음. maxsize 의 기본값은 0 으로, maxsize 값이 0 인 경우 큐에 들어가는 항목 개수는 무한.

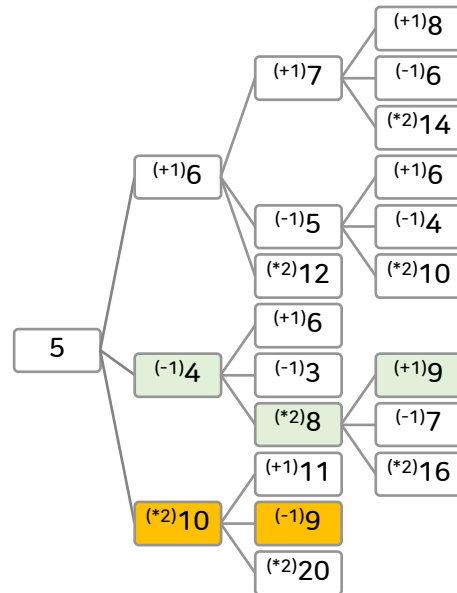
③ 제공하는 메소드 :

메소드 명	기능
empty()	현재 큐 객체가 비어 있으면 True, 아니면 False 를 return.
full()	큐가 가득 차면 True, 아니면 False 를 return.
qsize()	현재 큐의 대략적인 크기(큐에 들어 있는 항목 개수)를 return.
put(item)	item 을 큐 객체에 입력
get()	생성된 큐 객체의 특성(FIFO 혹은 LIFO 혹은 우선순위 큐)에 따라서 항목 하나를 큐 객체에서 빼고, 그 값을 return. 만일 큐에서 빼낼 항목이 없으면 queue.Empty 예외 발생.

6) 정답

- 주요 아이디어 정리
- ✓ 시작하는 수 number 에 문제에서 주어진 세 가지 연산에 대한 값을 산출.
- ✓ 위에서 구한 각각의 값들에 대해서 다시 세 가지 연산을 수행.
 - 이미 구했던 값이 나오면 skip.
 - 각 값이 몇 번의 연산을 통해 산출되었는 지 저장하는 공간이 요구됨.
- ✓ 목표값(target)이 나오면 종료하여 연산 횟수를 구함.

ex) 5 에서 시작하여 세 가지 연산을 통해 9 를 구하기



연산 횟수

$5 * 2 - 1 = 9$ → 2 (최소 연산 횟수)
 $(5 - 1) * 2 + 1 = 9$ → 3

- ✓ 0 ~ 10000 인덱스를 갖는 리스트를 생성하여 0 으로 값을 초기화하고, 산출된 적이 있는 수를 인덱스로 하는 항목에 대해 값을 연산횟수로 변경하여 다음 번 연산 시 재 산출되지 않도록 함.

visited 변수 : 각 숫자가 나오기 위해 연산한 횟수 (횟수 +1) 를 저장

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	3	2	1	2	3	3	3	2	3	3	0	0	0	0	0	0	0	3

- ✓ 큐를 사용하여 다음 번 계산에 사용할 숫자를 순서대로 저장해서 연산을 수행.

q : queue에 다음 계산할 숫자를 저장 <div style="border: 1px solid gray; padding: 5px; display: inline-block; background-color: #cccccc;">5</div>	- 시작하는 수가 5 인 경우, 5 를 큐에 넣고 연산 직전에 큐에서 5 를 가져온 후 세 가지 연산 수행.
q : queue에 다음 계산할 숫자를 저장 <div style="display: inline-block; background-color: yellow; padding: 5px;">6</div> <div style="display: inline-block; background-color: blue; padding: 5px;">4</div> <div style="display: inline-block; background-color: green; padding: 5px;">10</div>	- 5 를 이용한 연산 결과를 큐에 저장하고, 해당 결과들은 연산 실행 직전에 큐에서 순서대로 가져와서 연산 수행.

● 정답 코드

```

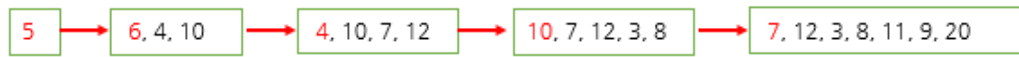
1  import queue
2
3  def solution(number, target):
4      answer = 0
5      ① visited = [0 for _ in range(10001)]
6      ② q = queue.Queue()
7      q.put(number)
8      visited[number] = 1
9      while not q.empty():
10         ③ x = q.get()
11         ④ if x == target:
12             break
13         if x+1 <= 10000 and visited[x+1] == 0:
14             visited[x+1] = visited[x]+1
15             q.put(x+1)
16         if x-1 >= 0 and visited[x-1] == 0:
17             visited[x-1] = visited[x]+1
18             q.put(x-1)
19         if 2*x <= 10000 and visited[2*x] == 0:
20             visited[2*x] = visited[x]+1
21             q.put(2*x)
22         ⑥ answer = visited[target]-1
23     return answer
24     #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
25     number1 = 5
26     target1 = 9
27     ret1 = solution(number1, target1)
28     #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
29     print("solution 함수의 반환 값은", ret1, "입니다.")
30
31     number2 = 3
32     target2 = 11
33     ret2 = solution(number2, target2)
34     #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
35     print("solution 함수의 반환 값은", ret2, "입니다.")

```

- ①. 10000 개의 0 을 항목으로 갖는 리스트 visited 를 생성.
- ②. 선입선출 방식을 갖는 Queue 객체를 생성하고 인수로 전달된 시작 값을 Queue 객체에 입력 후에 visited 리스트에서 시작 값과 동일한 인덱스의 항목 값을 1 로 설정. 예를 들어 시작 값이 5 인 경우 visited 리스트의 인덱스 5 항목이 갖는 값을 방문 횟수를 의미하는 1 을 저장.
- ③. 큐에 있는 값들 중 가장 먼저 입력된 값을 빼서 x 에 할당.
- ④. 큐에서 가져온 값이 목표 값과 같으면 반복문을 강제 종료.
- ⑤. (큐에서 가져온 값 + 1), (큐에서 가져온 값 - 1), (큐에서 가져온 값 * 2) 연산을 수행한 값을 인덱스로 갖는 visited 리스트의 항목 값이 0 이고, 연산을 수행한 결과값이 0 이상 10000 이하이면, (큐에서 가져온 값 + 1), (큐에서 가져온 값 - 1), (큐에서 가져온 값 * 2) 연산의 결과 값을 인덱스로 갖는 visited 리스트의 항목 값을 visited[큐에서 가져온 값] + 1 로 저장하고, 세 개의 연산 결과 값을 큐의 뒤쪽에 차례로 추가.

예) 초기값 5 에서 시작하여 목표 값 7 을 만들기 위해 필요한 최소 연산 횟수 구하기

- 큐에 저장된 값의 변화



- 연산 절차

x q.get()	x+1	v[x+1] =v[x]+1	put (x+1)	x-1	v[x-1] =v[x]+1	put (x-1)	x*2	v[x*2] =v[x]+1	put (x*2)
5	6	v[6]=v[5]+1	6	4	v[4]=v[5]+1	4	10	v[10]=v[5]+1	10
6	7	v[7]=v[6]+1	7	5			12	v[12]=v[6]+1	12
4	5			3	v[3]=v[4]+1	3	8	v[8]=v[4]+1	8
10	11	v[11]=v[10]+1	11	9	v[9]=v[10]+1	9	20	v[20]=v[10]+1	20
7									

- visited 리스트의 항목 값 변화

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	2	1	2	0	0	0	2	0	0	0	0	0	0	0	0	0	0
0	0	0	0	2	1	2	3	0	0	2	0	3	0	0	0	0	0	0	0	0
0	0	0	3	2	1	2	3	0	0	2	0	3	0	0	0	0	0	0	0	0
0	0	0	3	2	1	2	3	3	0	2	0	3	0	0	0	0	0	0	0	0
0	0	0	3	2	1	2	3	3	0	2	3	3	0	0	0	0	0	0	0	0
0	0	0	3	2	1	2	3	3	3	2	3	3	0	0	0	0	0	0	0	3

→ 마지막 get () 에 의해 나온 7 이 목표 값과 동일하므로 반복문은 강제 종료되고, 초기 값 5 에서 목표 값 7 을 만들기 위해 필요한 최소 연산 횟수는 visited[7] - 1 를 계산한 결과인 2 로 산출됨.

- ⑥. 시작값에서 시작하여 목표 값을 만들기 위해 필요한 연산 횟수 + 1 의 값이 visited 리스트에 있으므로 리턴하는 값은 (visited 리스트의 항목 값 - 1) 로 지정.

10. 문제 10

1) 문제 코드

```

1 class Job:
2     def __init__(self):
3         self.salary = 0
4
5     def get_salary(self):
6         return salary
7
8 class PartTimeJob@@@:
9     def __init__(self, work_hour, pay_per_hour):
10        super().__init__()
11        self.work_hour = work_hour
12        self.pay_per_hour = pay_per_hour
13
14    @@@:
15        self.salary = self.work_hour * self.pay_per_hour
16        if self.work_hour >= 40:
17            self.salary += (self.pay_per_hour * 8)
18        return self.salary
19
20 class SalesJob@@@:
21     def __init__(self, sales_result, pay_per_sale):
22         super().__init__()
23         self.sales_result = sales_result
24         self.pay_per_sale = pay_per_sale
25
26    @@@:
27        if self.sales_result > 20:
28            self.salary = self.sales_result * self.pay_per_sale * 3
29        elif self.sales_result > 10:
30            self.salary = self.sales_result * self.pay_per_sale * 2
31        else:
32            self.salary = self.sales_result * self.pay_per_sale
33        return self.salary
34
35 def solution(part_time_jobs, sales_jobs):
36     answer = 0
37     for p in part_time_jobs:
38         part_time_job = PartTimeJob(p[0], p[1])
39         answer += part_time_job.get_salary()
40     for s in sales_jobs:
41         sale_job = SalesJob(s[0], s[1])
42         answer += sale_job.get_salary()
43     return answer
44
45 #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
46 part_time_jobs = [[10, 5000], [43, 6800], [5, 12800]]
47 sales_jobs = [[3, 18000], [12, 8500]]
48 ret = solution(part_time_jobs, sales_jobs)
49
50 #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
51 print("solution 함수의 반환 값은", ret, "입니다.")

```

2) 문제 개요

- Job 클래스를 상속받도록 PartTimeJob 클래스와 SalesJob 클래스를 정의하는 문제.
- PartTimeJob 클래스와 SalesJob 클래스의 부모 클래스를 지정하고, 각각의 자식 클래스에서 부모 클래스인 Job 의 get_salary() 메소드를 재정의하여 문제에서 제시된 대로 급여를 계산하도록 메소드 정의 부분의 빈 칸을 채워야 함.

3) 정답

```

1 class Job:
2     ① def __init__(self):
3         self.salary = 0
4
5     ② def get_salary(self):
6         return salary
7
8     ③ class PartTimeJob(Job):
9         def __init__(self, work_hour, pay_per_hour):
10            ④ super().__init__()
11            ⑤ self.work_hour = work_hour
12            self.pay_per_hour = pay_per_hour
13
14        ⑥ def get_salary(self):
15            ⑦ self.salary = self.work_hour * self.pay_per_hour
16            ⑧ if self.work_hour >= 40:
17                self.salary += (self.pay_per_hour * 8)
18            return self.salary
19
20    ⑨ class SalesJob(Job):
21        def __init__(self, sales_result, pay_per_sale):
22            ⑩ super().__init__()
23            ⑪ self.sales_result = sales_result
24            self.pay_per_sale = pay_per_sale
25
26        ⑫ def get_salary(self):
27            if self.sales_result > 20:
28                self.salary = self.sales_result * self.pay_per_sale * 3
29            elif self.sales_result > 10:
30                self.salary = self.sales_result * self.pay_per_sale * 2
31            else:
32                self.salary = self.sales_result * self.pay_per_sale
33            return self.salary
34
35    def solution(part_time_jobs, sales_jobs):
36        answer = 0
37        for p in part_time_jobs:
38            part_time_job = PartTimeJob(p[0], p[1])
39            answer += part_time_job.get_salary()
40        for s in sales_jobs:
41            sale_job = SalesJob(s[0], s[1])
42            answer += sale_job.get_salary()
43        return answer
44
45    #아래는 테스트케이스 출력을 해보기 위한 코드입니다.
46    part_time_jobs = [[10, 5000], [43, 6800], [5, 12800]]
47    sales_jobs = [[3, 18000], [12, 8500]]
48    ret = solution(part_time_jobs, sales_jobs)
49
50    #[실행] 버튼을 누르면 출력 값을 볼 수 있습니다.
51    print("solution 함수의 반환 값은", ret, "입니다.")

```

❖ Job 클래스 정의

- ①. 생성자 메소드를 이용하여 멤버 변수 salary 를 생성하고 0 으로 초기화.
- ②. get_salary() 메소드는 멤버 변수 salary 에 저장되어 있는 값을 리턴.

❖ PartTimeJob 클래스 정의

- ③. PartTimeJob 클래스가 Job 클래스를 상속받도록 부모 클래스를 Job 으로 지정.
- ④. 부모 클래스의 생성자 메소드를 실행하여 멤버 변수 salary 를 생성하고 0 으로 초기화.
- ⑤. 매개 변수로 받은 값을 이용하여 멤버 변수 work_hour 와 멤버 변수 pay_per_hour 를 생성하고 초기화.
- ⑥. 부모 클래스 Job 에 있는 get_salary() 메소드를 오버라이드.
- ⑦. 멤버 변수 work_hour * pay_per_hour 를 계산한 값을 멤버 변수 salary 로 저장.
- ⑧. 멤버 변수 work_hour >= 40 이면 멤버 변수 pay_per_hour * 8 을 계산한 것을 멤버 변수 salary 에 추가로 더함.

❖ SalesJob 클래스 정의

- ⑨. SalesJob 클래스가 Job 클래스를 상속받도록 부모 클래스를 Job 으로 지정.
- ⑩. 부모 클래스의 생성자 메소드를 실행하여 멤버 변수 salary 를 생성하고 0 으로 초기화.
- ⑪. 매개 변수로 받은 값을 이용하여 멤버 변수 sales_result 와 멤버 변수 pay_per_sale 를 생성하고 초기화.
- ⑫. 부모 클래스 Job 에 있는 get_salary() 메소드를 오버라이드.