그리디 문제풀이

💡 Q1. 모험가 길드

문제

N 명의 모험가에 대한 공포도 정보가 주어진다. 공포도가 X인 모험가는 반드시 X명 이상으로 구성된 모험가 그룹에 참여해야 여행을 떠날 수 있다.

예를 들어 N = 5이고, [2, 3, 1, 2, 2]가 주어졌다면 [1, 2, 3], [2, 2]로 2개의 그룹으로 나눌 수 있다.

단, 모든 모험가를 특정 그룹에 넣을 필요는 없을 때, 여행을 떠날 수 있는 그룹 수의 최댓값을 구하여라

해설

1. 내가 생각한 방법
    - 배열을 내림차순 정렬 후, max 값을 기준으로 그룹을 짓는다. 배열의 길이가 0이거나 배열이 길이가 배열의 최댓값보다 작으면 반복을 멈추고 아니라면 배열을 최댓값 길이만큼 앞에서부터 잘라줍니다.

2. 정답
    - 위 방법과 반대… 오름차순 정렬 후 min 값을 기준으로 그룹을 지어야 그룹이 최대한 많이 지어진다…. ㅋㅋㅋ 또한 오름차순이 되었기 때문에 방식도 바뀌어야 하는데 만약 오름차순 기준으로 내 방식대로 풀면 1, 2, 2, 2, 3 배열에서 [1], [2, 2], [2, 3]이 되므로 마지막 배열이 틀려진다.

코드

1. 내가 생각한 방법 : 시간복잡도 O(NlogN) - 틀림

```
N = int(input())
adv = list(map(int, input().split()))

adv.sort(reverse=True) # O(NlogN)

cnt = 0

while True:
    if len(adv) == 0 or len(adv) < adv[0]:
        break
    else:
        adv = adv[adv[0]:]
        cnt += 1

print(cnt)
```

2. 정답 풀이 : 시간복잡도 O(NlogN)

```
N = int(input())
data = list(map(int, input().split()))
data.sort()

result = 0

count = 0

for i in data:
  count += 1

  if count >= i
    result += 1
    count = 0

print(result)
```

💡 Q2. 곱하기 혹은 더하기

문제

- 각 자리가 숫자(0~9)로만 이루어진 문자열이 주어졌을 때 중간중간에 ‘x’, ‘+’를 넣어 만들 수 있는 최대값을 출력하라

해설

- 각 자리가 0 혹은 1이면 덧셈을, 나머지는 곱셈을 이용한다. 단, 현재 계산 결과가 0인 경우 덧셈을 사용한다.

코드

1. 내가 생각한 방법

```
s = str(input())

result = int(s[0])

for i in range(1, len(s)):
    num = int(s[i])

    if result == 0 or num <= 1:
        result += num
    else:
        result *= num

print(result)
```

- result 값이 1일 경우에도 더해줘야 한다. 1+1이 1*1 보다 크다.

2. 정답

```
s = str(input())

result = int(s[0])

for i in range(1, len(s)):
    num = int(s[i])

    if result <= 1 or num <= 1:
        result += num
    else:
        result *= num
```

```
print(result)
```

💡 Q3. 문자열 뒤집기

문제

0과 1로만 이루어진 문자열 S가 있다. S에 있는 모든 숫자를 전부 같게 만들려고 한다. 할 수 있는 행동은 S에서 연속된 하나 이상의 숫자를 잡고 모두 뒤집는 것이다.

예를 들어 S = 0001100일 때,

1. 전체를 뒤집으면 1110011 이 된다.
2. 4번째 문자부터 5번째 문자까지 뒤집으면 1111111이 되어 두 번 만에 모두 같은 숫자가 된다.

그러나, 처음부터 4, 5문자를 뒤집으면 한 번에 0000000이 되어 1번 만에 모두 같은 숫자로 만들 수 있다.

이처럼 S가 주어질 때 모든 숫자를 같게 만드려면 최소 몇 번 행동해야 할까

해설

모르겠어서 답을 봤다.

기본적인 생각은 하나다. 어차피 어떤 문자열이 주어지든 하나로만 바뀌어야 하므로 전체가 0으로 바뀌던가 전체가 1로 바뀌던가 할 뿐이다.

그렇다면, 전체가 0으로 바뀌는 경우 count와, 전체가 1로 바뀌는 경우 count를 비교해 그 중 최소 값을 출력하면 된다.

코드

```
data = str(input())

cntFor0 = 0 # 1 -> 0
cntFor1 = 0 # 0 -> 1

if data[0] == '0':
    cntFor1 = 1
else:
    cntFor0 = 1

for i in range(len(data) - 1):
    if data[i] != data[i+1]:
        if data[i] == '0':
            cntFor1 += 1
        else:
            cntFor0 += 1

print(min(cntFor0, cntFor1))
```

만들 수 없는 금액

문제

N개의 동전이 있다. 이때, N개의 동전을 이용해 만들 수 없는 양의 정수 금액 중 최솟값을 구해야 한다.

예를 들어, N = 5이고, 각 동전이 3, 2, 1, 1, 9원 이라고 가정할 때, 만들 수 없는 양의 정수 금액 중 최솟값은 8원이다.

해설

1. 내가 생각한 방법
    - 모든 경우의 수에 대해 집합에 저장하고 1부터 순서대로 확인해 존재하지 않는 최초의 경우의 수를 출력한다.

    → 아무리 생각해봐도 이건 시간복잡도 초과라… 여기서 어떻게 발전시킬 수 있을까 생각해봤는데 답이 없다.

2. 정답

   배열을 오름차순 정렬 한다. 이후 맨 처음 동전부터 차례대로 동전을 추가해가며 만들 수 없는 금액을 찾는다. 예를 들어, 현재 동전 리스트가 [1, 1, 2]라면 만들 수 있는 금액은 1~4원이다. 그 다음 확인해야 할 금액(target)은 5이다.

   만약 전체 동전 리스트가 [1, 1, 2, 2]인 경우와 [1, 1, 2, 6]인 경우가 있다고 생각해보자.

   [1, 1, 2, 2]의 경우 1~4원까지 가능했던 [1, 1, 2]에 2가 추가로 들어갔기 때문에 1~6원까지 만들 수 있다. 이 경우 target은 7이 된다.

   [1, 1, 2, 6]의 경우 1~4원에서 1~10원까지 만들 수 있어진다. 하지만 이 경우 문제가 있는데 [1, 1, 2]일 경우의 target인 5보다 새로 추가된 동전 6이 더 크기 때문에 에러가 생긴다. 이 경우 기존의 [1, 1, 2]의 target인 5가 답이 된다.

   즉, 새로운 동전이 추가될 때 마다 이전 target + 새 동전의 금액을 하면서 target을 올려나가고, target 보다 새 동전이 클 경우 종료한다.

내 궁금증

그럼 [1, 1, 2, 5]의 경우는 된다는 건데 한 번 세보자.

[1, 1, 2]까지는 1~4가 된다. 그럼 5가 추가된 경우, 1~9까지가 되어야 한다는 건데

5 : [5]

6 : [1, 5]

7 : [2, 5]

8 : [1, 2, 5]

9 : [1, 1, 2, 5]

이거 뭔가 원래 있던 수학 식인 것 같다.

코드

```
N = int(input())
data = list(map(int, input().split()))

data.sort()
```

```
target = 1

for i in data:
    if i > target:
        break
    target += i

print(target)
```

💡 볼링공 고르기

문제

A, B 두 사람이 볼링을 치고 있다. 두 사람이 서로 무게가 다른 볼링공을 고르려 한다. 볼링공은 N개가 있으며 각 볼링공마다 무게가 있고, 공의 번호는 1번부터 순서대로 부여된다. 또한 같은 무게의 공이 여러 개 있을 수 있지만, 서로 다른 공으로 간주한다. 볼링공의 무게는 1~M까지의 자연수 형태로 존재한다.

예를 들어, N = 5, M = 3이고 [1, 3, 2, 3, 2]일 때 각 공의 번호가 차례대로 1, 2, 3, 4, 5번으로 부여된다. 이때 두 사람이 고를 수 있는 볼링공 번호의 조합을 구하면 다음과 같다.

(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 5), (3, 4), (4, 5) → 8가지 경우의 수

이와 같이 입력이 주어졌을 때 두 사람이 고를 수 있는 볼링공 조합의 경우의 수를 구하여라.

해설

1. 내가 생각한 해설
    - 각 모든 경우의 수에 대해 검증한다.
    - 이중 반복을 돌려 첫 반복 조건은 0 ~ N-1, 두 번째 반복조건은 i+1 ~ N으로 두고 서로 값이 다르면 카운트를 증가시킨다.
    - 다만, 이중 반복을 돌려서 문제를 푸는 것이 최적인가는 모르겠다. 시간 제한이 1초로 되어 있어 N의 개수가 최대라면 좀… 안될 것 같다

2. 정답 해설
    - 볼링공의 무게는 1~10 이라고 명시되어 있다. 그렇기 때문에 각 무게의 볼링공이 몇 개가 있는지 리스트에 담을 수 있으며 이를 이용해 아래의 예시와 같이 문제를 풀 수 있다.
    - 예를 들어, [1, 2, 2, 3, 3]이라면 아래와 같이 계산이 가능한데, 이때 이미 계산했던 경우의 수는 제외한다.
        1. A가 무게가 1인 공을 선택했을 경우 = 1 (무게가 1인 공) * 4 (B가 선택하는 경우의 수) = 4
        2. A가 무게가 2인 공을 선택했을 경우 = 2 (무게가 2인 공) * 2 (B가 선택하는 경우의 수) = 4
        3. A가 무게가 3인 공을 선택했을 경우 = 2 (무게가 3인 공) * 0 (B가 선택하는 경우의 수) = 0

코드

1. 내가 생각한 코드

```
N, M = map(int, input().split())
data = list(map(int, input().split()))

cnt = 0

for i in range(N - 1):
    for j in range(i + 1, N):
        if data[i] != data[j]:
            cnt += 1

print(cnt)
```

두 번째 반복문의 반복 횟수가 1~N, 2~N, 3~N, … N-1 ~ N으로 계속 줄어든다. 이에 대한 평균을 구하면 1~N-1 까지의 평균이므로 N(N+1)/2N → O(N)이 나오고 첫 번째 반복문 또한 N의 시간을 가지니 최종적으로 가지는 시간복잡도는 O(N^2) 이다.

2. 정답 코드 - O(N)

```
N, M = map(int, input().split())
data = list(map(int, input().split()))

arr = [0] * 11

for x in data:
  arr[x] += 1

result = 0

for i in range(1, m + 1):
  n -= arr[i] # 무게가 i인 볼링공의 개수(A가 선택할 수 있는 개수) 제외
  result += arr[i] * n
```

무지의 먹방 라이브

문제

특별한 규칙을 가지고 먹방을 하는 미친 무지의 이야기이다. 규칙은 다음과 같다.

- 회전판에 N개의 음식이 있다.
- 각 음식에는 1~N까지 번호가 붙어있으며, 각 음식을 섭취하는데 일정 시간이 소요된다.
- 무지는 1번부터 음식을 먹기 시작하며, 회전판은 번호가 증가하는 순서대로 음식을 무지 앞에 가져다 놓는다.
- 마지막 번호의 음식을 섭취한 후에는 회전판에 의해 다시 1번 음식이 무지 앞으로 온다.
- 무지는 음식 하나를 1초 동안 섭취한 후 남은 음식은 그대로 두고, 다음 음식을 섭취한다.
    - 여기서 다음 음식이란, 아직 남은 음식 중 다음으로 섭취해야 할 가장 가까운 번호의 음식이다.
- 회전판이 다음 음식을 무지 앞으로 가져오는데 걸리는 시간은 없다고 가정한다.

무지가 먹방을 시작한 지 K초 후에 네트워크 장애로 인해 방송이 잠시 중단되었다. 무지는 네트워크 정상화 후 다시 방송을 이어갈 때, 몇 번 음식부터 먹어야 할지를 알고자 한다.

각 음식을 먹는데 필요한 시간이 담겨있는 배열 food_times, 네트워크 장애가 발생한 시간 K초가 주어질 때, 몇 번 음식부터 다시 섭취하면 되는지 return 하도록 solution 함수를 완성하라.

단, 더 섭취해야 할 음식이 없다면 -1을 반환한다.

해설

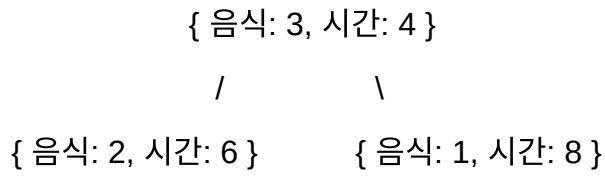
1. 내가 생각한 해설

   …은 아래 코드의 주석으로 대체한다.

2. 정답 해설

   최소 힙을 이용한다. 시간이 적게 걸리는 음식부터 제거해 나가는 방식을 이용해 문제를 해결한다. (최소 힙은 항상 루트 노드가 최소이기 때문에 이런 문제 해결에 용이하다.)

   예를 들어 K = 15, [8, 6, 4] 라는 데이터가 있다고 가정했을 때, 최소 힙을 구성하면 아래와 같아진다.

   1. 음식 3을 다 먹는데는 음식 1, 2, 3이 전체 4바퀴를 돌아야 하므로 12초가 소요된다. 이때 남은 시간은 3(15 - 12)초이다. → { 음식: 2, 시간: 2 } - { 음식: 1, 시간: 4 }
   2. 다음으로 시간이 적게 걸리는 음식은 2번이다. 2번을 다 먹는데 걸리는 시간은 4초이지만 남은 시간이 3초밖에 없다. 이렇게 되면 빼지 않고 3번 과정을 수행한다.
   3. 남은 음식은 남은 시간 % 남은 음식 개수 로 구할 수 있다.
      - 현재 남은 시간 = 3초이고, 남은 음식은 2개이므로 음식 2번이 먹어야 될 음식이다.(인덱스는 0부터 시작이다.)
      - 만약 남은 시간이 8초이고, 남은 음식이 3개라면 3번 음식이 먹어야 될 음식이다.
      - 예전에 고등학교 수학 시간에 배운 내용인데, 기억을 되살려 보면

> 1~6까지 써진 숫자 카드가 반복되어 나올 때 1003번째 나올 숫자 카드가 무엇일까요?

라는 문제와 같은 내용이다.

코드

1. 내가 생각한 코드

```
def solution(food_times, k):
    answer = 0

    foodIdx = 0
    foodLen = len(food_times)

    for i in range(k):
        if food_times[foodIdx] != 0: # 현재 음식이 남아있을 경우
            food_times[foodIdx] -= 1 # 해당 음식에서 1만큼 뺀다

            # 다음 음식으로 인덱스 이동
            if foodIdx != foodLen - 1:
                foodIdx += 1
            else:
                foodIdx = 0
        else: # 현재 음식이 남아있지 않을 경우
            zeroCnt = 0 # 전체 음식중에 0이 몇 개인지 확인

            while zeroCnt != foodLen: # 전체 음식이 0이면 종료
                # 현재 음식이 0인 경우
                if food_times[foodIdx] == 0:
                    zeroCnt += 1 # 현재 음식이 0이므로 +1

                    # 현재 음식이 0이므로 인덱스 up
                    if foodIdx != foodLen - 1:
                        foodIdx += 1
                    else:
                        foodIdx = 0

                # 현재 음식이 0이 아닌 경우
                else:
                    food_times[foodIdx] -= 1 # 해당 음식에서 1만큼 뺀다

                    # 다음 음식으로 인덱스 이동
                    if foodIdx != foodLen - 1:
                        foodIdx += 1
                    else:
                        foodIdx = 0

                    break
```

```
            if zeroCnt == foodLen:
                answer = -1
                break

    answer = foodIdx + 1

    return answer
```

누가 봐도 더럽고 시간 복잡도 또한 말이 안된다. 실행 결과 또한 60%만 정답이 되었다. 타임아웃으로 안된건지 그냥 안된 케이스가 있는 건지도 모르겠다.

2. 정답 코드

```
import heapq

def solution(food_times, k):
    # 전체 음식을 먹는 시간보다 k가 크거나 같으면 -1을 출력한다.
    # k가 더 작다면 어떤 음식이든 출력될 것이기에 -1을 출력할 경우의 수는 없어진다.
    if sum(food_times) <= k:
        return -1

    # 최소힙을 사용할 변수 지정
    q = []
    for i in range(len(food_times)):
        # heapq는 기본적으로 최소 힙 방식으로 구성된다.
        # q 변수에 (음식 시간, 음식 번호)을 저장한다.
        heapq.heappush(q, (food_times[i], i + 1))

    # 먹기 위해 사용한 시간
    sum_value = 0
    # 직전에 다 먹은 음식 시간
    previous = 0
    # 남은 음식 개수
    length = len(food_times)

    # 먹기 위해 사용한 시간 + (현재 음식 시간 - 직전에 다 먹은 음식 시간) * 남은 음식 개수 <= k
    while sum_value + ((q[0][0] - previous) * length) <= k:
        # 최소 힙에서 pop 한 음식의 시간 데이터
        now = heapq.heappop(q)[0]
        # 먹기 위해 사용한 시간 += (현재 값 - 직전에 다 먹은 음식 시간) * 남은 음식 개수
        sum_value += (now - previous) * length
        # 1개 빠졌으니 -1
        length -= 1
        # 이제 빠진게 직전에 다 먹은 음식 시간이 된다.
        previous = now

    # 음식 번호 기준으로 정렬
    result = sorted(q, key = lambda x: x[1])
    # 남은 음식 중 몇 번째 음식인지 확인해 출력
    # (k - sum_value) % length = 위 예시에서...
    # (15초 - 12초) % 2 = 1
    return result[(k - sum_value) % length][1]
```