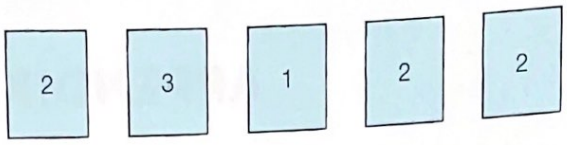
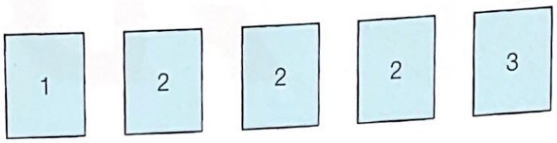
A 01 모험가 길드

일단 공포도를 기준으로 오름차순으로 정렬을 수행해보자. 이후에 공포도가 가장 낮은 모험가부터 하나씩 확인하며, 그룹에 포함될 모험가의 수를 계산할 수 있다. 만약에 현재 그룹에 포함된 모험가의 수가 현재 확인하고 있는 공포도보다 크거나 같다면, 그룹을 결성할 수 있을 것이다.

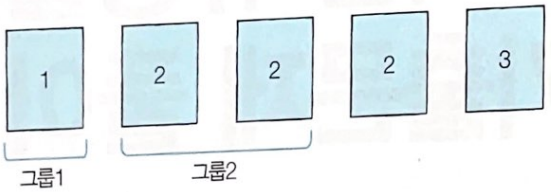
예를 들어 문제에서의 예시 입력을 그림으로 표현하면 다음과 같다.



가장 먼저 5명의 공포도를 오름차순으로 정렬하면 다음과 같이 구성된다.



이제 앞에서부터 공포도를 하나씩 확인하며, '현재 그룹에 포함된 모험가의 수'가 '현재 확인하고 있는 공포도'보다 크거나 같다면, 이를 그룹으로 설정하면 된다. 현재 예시에서는 다음과 같이 2개의 그룹이 형성된다. 남은 2명의 모험가는 그룹에 속하지 못하고 그대로 남아 있게 된다.



이러한 방법을 이용하면 공포도가 오름차순으로 정렬되어 있다는 점에서, 항상 최소한의 모험가의 수만 포함하여 그룹을 결성하게 된다. 따라서 최대한 많은 그룹이 구성되는 방법이므로, 항상 최적의 해를 찾을 수 있다. 이를 소스코드로 나타내면 다음과 같다.

A01.py 답안 예시

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

result = 0 # 총 그룹의 수
```

```
count = 0 # 현재 그룹에 포함된 모험가의 수

for i in data: # 공포도를 낮은 것부터 하나씩 확인하며
    count += 1 # 현재 그룹에 해당 모험가를 포함시키기
    if count >= i: # 현재 그룹에 포함된 모험가의 수가 현재의 공포도 이상이라면, 그룹 결성
        result += 1 # 총 그룹의 수 증가시키기
        count = 0 # 현재 그룹에 포함된 모험가의 수 초기화

print(result) # 총 그룹의 수 출력
```

A 02 곱하기 혹은 더하기

일반적으로 특정한 두 수에 대하여 연산을 수행할 때, 대부분은 '+'보다는 '×'가 더 값을 크게 만든다. 예를 들어 5와 6이 있다고 해보자. 이때 더하기를 수행하면 5 + 6 = 11이 되고, 곱하기를 수행하면 5 × 6 = 30이 된다. 즉, 대부분의 경우에서는 곱하기를 수행한 결괏값이 더 크다.

하지만 두 수 중에서 하나라도 '0' 혹은 '1'인 경우, 곱하기보다는 더하기를 수행하는 것이 효율적이다. 예를 들어 1과 2가 있다고 해보자. 이때 더하기를 수행하면 1 + 2 = 3이 되고, 곱하기를 수행하면 1 × 2 = 2가 된다.

다시 말해 두 수에 대하여 연산을 수행할 때, 두 수 중에서 하나라도 1 이하인 경우에는 더하며, 두 수가 모두 2 이상인 경우에는 곱하면 된다.

이러한 원리를 이용하면 쉽게 문제를 해결할 수 있다. 문자열이 입력되었을 때 모든 숫자를 기준으로 나눈 뒤에, 앞에서부터 연산을 수행하면 된다. 다시 말해 현재까지의 계산 결과를 result에 담은 상태로, 매 숫자에 대하여 연산을 수행하면 된다. 그래서 result가 1 이하이거나, 현재 처리하고 있는 숫자가 1 이하라면 더하기 연산을 수행하고, 그렇지 않은 경우 곱하기 연산을 수행하면 항상 최적의 결과를 얻을 수 있다.

A02.py 답안 예시

```
data = input()

# 첫 번째 문자를 숫자로 변경하여 대입
result = int(data[0])
```

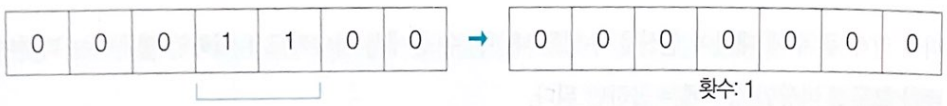
```
for i in range(1, len(data)):
    # 두 수 중에서 하나라도 '0' 혹은 '1'인 경우, 곱하기보다는 더하기 수행
    num = int(data[i])
    if num <= 1 or result <= 1:
        result += num
    else:
        result *= num

print(result)
```

A 03 문자열 뒤집기

다솜이는 모든 숫자를 전부 같게 만드는 것이 목적이다. 따라서 전부 0으로 바꾸는 경우와 전부 1로 바꾸는 경우 중에서 더 적은 횟수를 가지는 경우를 계산하면 된다.

예를 들어 문자열이 "0001100"이라고 가정해보자. 이때 '모두 0으로 만드는 경우'와 '모두 1로 만드는 경우'를 고려했을 때 각각 뒤집기 횟수를 계산하면 다음과 같다.

1. 모두 0으로 만드는 경우



2. 모두 1로 만드는 경우

이를 실제로 구현할 때는 전체 리스트의 원소를 앞에서부터 하나씩 확인하며, 0에서 1로 변경하거나 1에서 0으로 변경하는 경우를 확인하는 방식으로 해결할 수 있다. 소스코드는 다음과 같다.

A03.py 답안 예시

```
data = input()
count0 = 0 # 전부 0으로 바꾸는 경우
count1 = 0 # 전부 1로 바꾸는 경우

# 첫 번째 원소에 대해서 처리
```

```
if data[0] == '1':
    count0 += 1
else:
    count1 += 1

# 두 번째 원소부터 모든 원소를 확인하며
for i in range(len(data) - 1):
    if data[i] != data[i + 1]:
        # 다음 수에서 1로 바뀌는 경우
        if data[i + 1] == '1':
            count0 += 1
        # 다음 수에서 0으로 바뀌는 경우
        else:
            count1 += 1

print(min(count0, count1))
```

A 04 만들 수 없는 금액

이 문제는 정렬을 이용한 그리디 알고리즘으로 해결할 수 있는 문제이다. 문제 해결을 위한 정확한 아이디어를 떠올리기 위해서는 충분히 고민을 해야 하는 문제이므로, 그리디 알고리즘에 익숙하지 않은 독자라면 문제 해결이 쉽지 않을 수 있다.

문제 해결 아이디어는 다음과 같다. 일단 동전에 대한 정보가 주어졌을 때, 화폐 단위를 기준으로 오름차순 정렬한다. 이후에 1부터 차례대로 특정한 금액을 만들 수 있는지 확인하면 된다. 1부터 target - 1까지의 모든 금액을 만들 수 있다고 가정해보자. 우리는 화폐 단위가 작은 순서대로 동전을 확인하며, 현재 확인하는 동전을 이용해 target 금액 또한 만들 수 있는지 확인하면 된다. 만약 target 금액을 만들 수 있다면, target 값을 업데이트하는(증가시키는) 방식을 이용한다.

기본적으로 그리디 알고리즘은, 현재 상태에서 매번 가장 좋아 보이는 것만을 선택하는 알고리즘이라고 하였다. 구체적으로 현재 상태를 '1부터 target - 1까지의 모든 금액을 만들 수 있는 상태'라고 보자. 이때 매번 target인 금액도 만들 수 있는지(현재 확인하는 동전의 단위가 target 이하인지) 체크하는 것이다. 만약 해당 금액을 만들 수 있다면, target의 값을 업데이트(현재 상태를 업데이트)하면 된다.

예를 들어 3개의 동전이 있고, 각 화폐의 단위가 1, 2, 3이라고 하자. '원'은 생략하겠다.

그러면 1부터 6까지의 모든 금액을 만들 수 있다.

- 1원: 1
- 2원: 2
- 3원: 3
- 4원: 1 + 3
- 5원: 2 + 3
- 6원: 1 + 2 + 3

그다음 우리는 금액 7도 만들 수 있는지 확인을 하면 된다. 이때 화폐 단위가 5인 동전 하나가 새롭게 주어졌다고 가정하자. 이제 화폐 단위가 5인 동전이 주어졌기 때문에, 1부터 11까지의 모든 금액을 만들 수 있다. 예를 들면 다음과 같이 1부터 11까지의 모든 금액을 만들 수 있다. (당연히 금액 7도 만들 수 있다는 것이 자동으로 성립한다.)

- 1원: 1
- 2원: 2
- 3원: 3
- 4원: 1 + 3
- 5원: 5
- 6원: 1 + 5
- 7원: 2 + 5
- 8원: 3 + 5
- 9원: 1 + 3 + 5
- 10원: 2 + 3 + 5
- 11원: 1 + 2 + 3 + 5

이후에 우리는 금액 12도 만들 수 있는지 확인을 하면 되는 방식이다. 이때 화폐 단위가 13인 동전 하나가 새롭게 주어졌다고 가정하자. 이때 금액 12를 만드는 방법은 존재하지 않는다. 그래서 이 경우에는 정답이 12가 된다.

이제 또 다른 예시를 확인해보자. 이번에는 문제를 해결하는 과정을 단계별로 보이겠다. 만약에 동전을 4개 가지고 있고, 화폐 단위가 각각 1, 2, 3, 8이라고 해보자.

step 0 처음에는 금액 1을 만들 수 있는지 확인하기 위해, target = 1로 설정한다.

step 1 target = 1을 만족할 수 있는지 확인한다. 우리에게는 화폐 단위가 1인 동전이 있다. 우리는 이 동전을 이용해서 금액 1을 만들 수 있다. 이어서 target = 1 + 1 = 2로 업데이트를 한다. (1까지의 모든 금액을 만들 수 있다는 말과 같다.)

step 2 target = 2를 만족할 수 있는지 확인한다. 우리에게는 화폐 단위가 2인 동전이 있다. 따라서 target = 2 + 2 = 4가 된다. (3까지의 모든 금액을 만들 수 있다는 말과 같다.)

step 3 target = 4를 만족할 수 있는지 확인한다. 우리에게는 화폐 단위가 3인 동전이 있다. 따라서 target = 4 + 3 = 7이 된다. (6까지의 모든 금액을 만들 수 있다는 말과 같다.)

step 4 target = 7을 만족할 수 있는지 확인한다. 우리에게는 이보다 큰, 화폐 단위가 8인 동전이 있다. 따라서 금액 7을 만드는 방법은 없다. 따라서 정답은 7이 된다.

이러한 원리를 이용하면, 단순히 동전을 화폐 단위 기준으로 정렬한 뒤에, 화폐 단위가 작은 동전부터 하나씩 확인하면서 target 변수를 업데이트하는 방법으로 최적의 해를 계산할 수 있다.

이 문제는 그리디 알고리즘 유형의 문제를 여러 번 풀어보았다면 풀이 방법을 떠올릴 수 있지만, 그리디 알고리즘이 익숙하지 않다면 쉽게 이해되지 않을 수 있는 문제이다. 따라서 이 문제가 어렵다면 그리디 알고리즘 유형의 문제를 더욱 많이 접해보자.

참고로 이 문제는 앞서 이론 파트에서 다루었던 '거스름돈' 문제와는 다른 문제이다. 거스름돈 문제는 각 화폐 단위마다 무한 개의 동전이 존재한다고 가정했는데, 여기서는 동전의 수가 한정적이라는 점이 다르다.

A04.py 답안 예시

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

target = 1
for x in data:
    # 만들 수 없는 금액을 찾았을 때 반복 종료
    if target < x:
        break
    target += x

# 만들 수 없는 금액 출력
print(target)
```

A 05 볼링공 고르기

이 문제를 효과적으로 해결하기 위해서는, 먼저 무게마다 볼링공이 몇 개 있는지를 계산해야 한다. 문제에서 등장했던 예시를 기준으로 보면 다음과 같다.

- 무게가 1인 볼링공: 1개
- 무게가 2인 볼링공: 2개
- 무게가 3인 볼링공: 2개

이때 A가 특정한 무게의 볼링공을 선택했을 때, 이어서 B가 볼링공을 선택하는 경우를 차례대로 계산하여 문제를 해결할 수 있다. A를 기준으로 무게가 낮은 볼링공부터 무게가 높은 볼링공까지 순서대로 하나씩 확인을 한다고 했을 때 다음과 같다.

step 1 A가 무게가 1인 공을 선택할 때의 경우의 수는
1(무게가 1인 공의 개수) × 4(B가 선택하는 경우의 수) = 4가지 경우가 있다.

step2 A가 무게가 2인 공을 선택할 때의 경우의 수는
2(무게가 2인 공의 개수) × 2(B가 선택하는 경우의 수) = 4가지 경우가 있다.

step 3 A가 무게가 3인 공을 선택할 때의 경우의 수는
2(무게가 3인 공의 개수) × 0(B가 선택하는 경우의 수) = 0가지 경우가 있다.

따라서 가능한 경우의 수는 총 8가지이다.

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

단계(step)가 진행됨에 따라, 'B가 선택하는 경우의 수'는 줄어드는데, 이미 계산했던 경우(조합)는 제외하기 때문이다. 또한 볼링공의 무게가 1부터 10까지만 존재할 수 있다고 명시되어 있다. 따라서 하나의 리스트 변수를 선언해서, 각 무게별로 볼링공이 몇 개가 존재하는지 기록할 수 있다.

이를 소스코드로 옮기면 다음과 같다.

A05.py 답안 예시

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

# 1부터 10까지의 무게를 담을 수 있는 리스트
array = [0] * 11
```

```
for x in data:
    # 각 무게에 해당하는 볼링공의 개수 카운트
    array[x] += 1

result = 0
# 1부터 m까지의 각 무게에 대하여 처리
for i in range(1, m + 1):
    n -= array[i] # 무게가 i인 볼링공의 개수(A가 선택할 수 있는 개수) 제외
    result += array[i] * n # B가 선택하는 경우의 수와 곱하기

print(result)
```

A 06 무지의 먹방 라이브

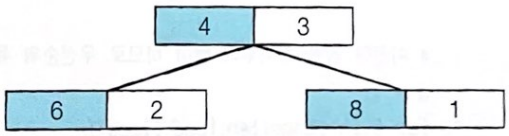
이 문제는 시간이 적게 걸리는 음식부터 확인하는 탐욕적Greedy 접근 방식으로 해결할 수 있다. 모든 음식을 시간을 기준으로 정렬한 뒤에, 시간이 적게 걸리는 음식부터 제거해 나가는 방식을 이용하면 된다. 이를 위해 우선순위 큐를 이용하여 구현할 수 있는데, 문제를 풀기 위해 고려해야 하는 부분이 많아서 까다로울 수 있다.

간단한 예시로 다음과 같이 3개의 음식이 있으며, K를 15초라고 해보자.

- 1번 음식: 8초 소요
- 2번 음식: 6초 소요
- 3번 음식: 4초 소요

step 0 초기 단계에서는 모든 음식을 우선순위 큐(최소 힙)에 삽입한다. 또한 마지막에는 K초 후에 먹어야 할 음식의 번호를 출력해야 하므로 우선순위 큐에 삽입할 때 (음식 시간, 음식 번호)의 튜플 형태로 삽입한다. 그 형태는 오른쪽과 같다.

- 전체 남은 시간(K): 15초
- 남은 음식: 3개



step 1 첫 단계에서는 가장 적게 걸리는 음식인 3번 음식을 뺀다. 다만, 음식이 3개 남아 있으므로 3(남은 음식의 개수) × 4(3번 음식을 먹는 시간) = 12를 빼야 한다. 다시 말해 3번 음식을 다 먹기 위해서는 12초가 걸린다는 의미이다. 결과적으로 전체 남은 시간이 15초에서 3초로 줄어들게 된다.