Space Crews

codility
WE TEST CODERS

The last challenge task was codenamed Kappa. If you didn't get a chance to play with it, you can still do so here, though you can no longer get a certificate for this task.

In this task we asked how many ways are there to form a space crew. There were many countries participating in the mission and each country has trained several astronauts, but could only delegate some of them to join the crew.

First of all, how many ways are there to choose astronauts from one country? That's easy, the answer is binomial coefficient. The choice that one country makes is independent from other countries' choices, hence, in order to obtain the total number of possible choices, we just have to multiply the number of choices per country. In other words, we have to calculate the following number:

$$B(T[0], D[0]) \cdot B(T[1], D[1]) \cdot \cdots \cdot B(T[N-1], D[N-1])$$

Values of binomial coefficient can be calculated using the following formula involving factorials:

$$B(N, K) = N!/(K! \cdot (N-K)!)$$

This approach leads us to the following solution (written in Python):

1: Slow solution

```python
MOD = 1410000017

def factorial(N):
    if N == 0:
        return 1
    else:
        return N * factorial(N-1)

def binomial_coefficient(N, K):
    return factorial(N) // (factorial(K) * factorial(N-K))

def space_crews(T, D):
    R = 1
    for t, d in zip(T, D):
        R *= binomial_coefficient(t, d)
    return R % MOD
```

This solution is too slow. Imagine a scenario where all values in array T are equal to 1 000 000. The code above will recalculate the value of `factorial(1000000)` from scratch each time. This is not acceptable, but there is a simple trick to fix this. All we have to do is to cache the values returned by the `factorial` function. This is also known as memoization. After modifications the code looks as follows:

2: Solution with memoization

```
MOD = 1410000017
F = [1]

def factorial(N):
    global F
    K = len(F)
    while K <= N:
        F.append(K * F[-1])
        K += 1
    return F[N]

def binomial_coefficient(N, K):
    return factorial(N) // (factorial(K) * factorial(N-K))

def space_crews(T, D):
    R = 1
    for t, d in zip(T, D):
        R *= binomial_coefficient(t, d)
    return R % MOD
```

We solved our performance problem, but we're still not there yet. The values of the factorial function can be very large. Imagine calculating the value `binomia_coefficient(1000000, 1)`. Although the result is 1 000 000, the numbers involved in the calculation are huge! The value of `factorial(1000000)` has millions of digits, so it won't fit in a 32-bit integer, 64-bit integer or even a floating-point variable.

Is this really a problem? Python handles arbitrarily long numbers after all. But this comes at a price: handling millions of digits takes a considerable amount of time. We can do better than that, enter modular arithmetic!

It is fairly easy to perform multiplication in modular arithmetic, but there is a problem with division. A positive integer can be mapped to zero in modular arithmetic, so there is a risk that we will be unable to divide by `factorial(K)` or by `factorial(N-K)` when we calculate the value of `binomial_coefficient` function.

We have now identified the main difficulty in the task and there is only one question left unanswered: why the result is supposed to be returned modulo this strange looking number 1 410 000 017? Because it's a prime number (also, year 1410 was full of events). Considering that we only need to calculate factorial up to 1 000 000, its value will never become zero modulo such a large prime number. Why? Because otherwise we would obtain a factorization of a prime, which is not possible!

Multiplication is pretty easy to implement, division is considerably harder. In order to divide by a value, we will multiply by its modular multiplication inverse and in order to find this inverse we will use the extended Euclidean algorithm. Our solution becomes the following:

3: Model solution

```python
MOD = 1410000017
F = [1]

def extended_euclidean_algorithm(A, B):
    if B == 0:
        return A, 1, 0
    Q, R = divmod(A, B)
    D, K, L = extended_euclidean_algorithm(B, R)
    return D, L, K - Q*L

def modular_multiplicative_inverse(A):
    D, K, L = extended_euclidean_algorithm(A, MOD)
    return K

def factorial(N):
    global F
    K = len(F)
    while K <= N:
        F.append((K * F[-1]) % MOD)
        K += 1
    return F[N]

def binomial_coefficient(N, K):
    R = 1
    R = (R * factorial(N)) % MOD
    R = (R * modular_multiplicative_inverse(factorial(K))) % MOD
    R = (R * modular_multiplicative_inverse(factorial(N-K))) %
       MOD
    return R

def space_crews (T, D):
    N = len(T)
    R = 1
    for t, d in zip(T, D):
        R = (R * binomial_coefficient(t, d)) % MOD
    return R
```

This is the final solution. It runs fast, because it caches the values of the factorial function and avoids performing operations on arbitrarily long numbers by employing modular arithmetic. It is worth noting that both of these techniques are widely used: caching is ubiquitous nowadays and modular arithmetic is the building block of virtually all cryptographic systems.