1 Problem Description

According to the fundamental theorem of arithmetic in number theory, every number can be uniquely represented as a product of one or more prime numbers. But we want to find out the number of arrangements for prime factors. Define $f(k)$ is the number of arrangements of the prime numbers of k. Given a number n, we want to figure out the smallest number k, so that $f(k) =$ n.

2 Mathematical Formulation

Given an integer n, which is the number of arrangement of prime factors, we want to find the smallest interger k .

Input
Input consists of at most 1,000 cases, and each case, on each separate line, is a positive integer $n < 2^63$.

Output
For each one, print out n and the smallest $k < 2^63$such that $f(k) = n$.

Formulation
k can be represented uniquely as a product of prime factors. Let's say k is represented as
k = $2^{p_1} \cdot 3^{p_2} \cdot ...m^{p_h}$
where the ordering of p_i is decreasing that $p_1 \geq p_2 \geq ... \geq p_h$

Let's have all distinct prime integers, such that $p_i = 1, (i = 1..h)$, then let's define X as a sequence of distinct prime factors. Then, the number of arrangement of distinct prime factors would be $X!$.
And according to the problem's constraint that $X! < 2^63!$, we could only have the maximum value of X is 20 because $20! < 2^63 < 21!$.
Then we could only have at most a sequence of 20 prime factors, which means X = 20. Thus, the maximum prime number in the sequence X would be only 71, as each of prime numbers is unique and increasing as followed:
k = $2^{p_1} \cdot 3^{p_2} \cdot ... \cdot 71^{p_20}$
(the first 20 prime numbers are 2, 3 ,5 , 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71)
Then the number of arrangement for prime factors would be the number of permutation of these 20 prime numbers p_i where $p_i \geq 0$. In the general case, then n would be the permutation of arranging each of these 20 p_i in their total sum

$$\text{n} = \binom{p_1+p_2+p_3+...+p_20}{p_1,p_2,p_3,...,p_20} = \frac{p_1+p_2+p_3+...+p_20}{p_1!p_2!p_3!...p_20!}$$

The major key idea here in this formulation is that finding the number of ways to arrange each of these 20 p_i would ensure the smallest value for k such that $f(k) = n$. Because the ordering of p_i, as we classified above, is descending such that
$p_1 \geq p_2 \geq ... \geq p_20$
Then, if we find out the sequence of these 20 prime numbers given the number of permutation of them- n, we know that k is able to be represented as the product of increasing prime numbers, with each of them rising up to the power of $p_1, p_2, ..., p_20$. So, here, it's true that k is minimum since the smallest prime number in the product representation gets the biggest exponent p_1, and similarly the largest prime number gets the smallest exponent p_20.

3 Solution

Although we're given the number of permutation of the first 20 prime numbers, n, we still have to compute all the numbers of arranging these prime factors by calculating the binomial function given a sequence of $p_1, p_2, ..., p_20$.
So, our idea here is to build up the look-up hash table, called *table*, which maps the integer n, as the number of permutation to choose 20 prime numbers out of their total sum, to another integer k, as the product of prime numbers rising up to the exponents. Then, we first have to enumerate different sequences of choosing the first 20 prime numbers based on the (binomial) chose function, which is stored in the array called *xSequence*. Constructing our array *xSequence* in which each element is produced by using the binomial function, and then iterating through these elements with the use of complete search and pruning cases when the possible values in *xSequence* would produce k or $n \geq 2^{63}$
Then, essentially, for each given number n, we add to the *table*, the value of k.

Global vars:
int[] primes //explicitly initialize the array of the first 20 prime numbers
BigInteger MAX //the maximum upper bound 2^{63}
int *anchor*, *inc*

Algorithm 1 building the lookup hash table

```
procedure MAIN
    initialize table as a hash map BigInteger to BigInteger and xSequence[0] = 1
    while true do //BigInteger vars: top, bot, key, val
        if xSequence[0] == 63 then
            break
        if val.compareTo(MAX) > 0 then
            val = calcVal()
            if val.compareTo(MAX) >0 then
                reset()
            top = calcTop(), bot = calcBot(), key = top.divide(bot)
            if table.containsKey(key) then
                stored = table.get(key)
                if val.compareTo(stored) < 0 then
                    table.put(key,val)
            else
                table.put(key,val)
            if inc +1 < xSequence.length then
                xSequence[++inc]++
            else
                reset()
```

Algorithm 2 Helpers

```
procedure RESET(anchor, xSequence, inc)
    prev, next ← anchor-1, anchor+1
    if anchor != inc && next != xSequence.length && xSequence[anchor] xSequence[next] then
        anchor++
    else
        while prev ≥ 0 && xSequence[prev] == xSequence[anchor] do
            anchor–, prev–
    zeroAfter() // make all zeros after the new anchor
    xSequence[anchor]++;
    inc = anchor;
procedure CalcVal
    val = 1
    for x ∈ xSequence do
        val = val * primes[indexOf(x)]^x
    return val
procedure CalcTop
    top = 0
    for x ∈ xSequence do
        top = top + x
    top = top!
    return top
procedure CalcBot
    bot = 1
    for x ∈ xSequence do
        if x > 0 then
            bot = bot * x!
    return bot
```

4 Correctness

Proposition 1. *Our algorithmName algorithm will, for any integer $n < 2^{63}$, produce output k s.t. $k = xSequence_1^{p_1} * xSequence_2^{p_2} * ... * xSequence_{20}^{p_{20}}$, and k is as small as possible.*

Proof. To begin, we construct our array *xSequence* s.t. each $xSequence \in X$ corresponds to a prime number $2, 3, 5, 7, ...67$ where *xSequence* contains 20 distinct primes, and each prime in *xSequence* is as small as possible. We know that the combination of $xSequence \in X$ can produce any number n because our method of incrementing the elements in *xSequence* will, through simple implementation, produce every possible value that can be produced by operating the *calcTop* can *calcBottom* operations

on $xSequence$, stopping only if $k > 2^{63}$. Secondly, we construct each element of $xSequence$ s.t. $xSequence[1] == p_1, xSequence[2] = p_2, ..., xSequence[20] = p_{20}$. We also state that $p_i \geq p_{i+1}$, meaning that $xSequence[i] \geq xSequence[i+1]$. These two facts taken together that the smallest primes in the sequence will have the highest exponents. This means that for the sequence of n combinations, the sequence will contain as many small numbers as possible ≥ 0, and that therefore k will be the product of this set of the smallest possible prime integers. Since the product of smaller integers ≥ 0 is necessarily smaller than the product any two integers where both are either larger or equal two the two smaller integers, k is the smallest possible value for a combination of $xSequence$ choose n □

5 Analysis

Proposition 2. *Our algorithmName algorithm is constructed in a way such that the running time is independent of the input, and the actual time taken by the input is constant. In relation to Big Oh notation, we will define n as 2^{63}, the largest possible value that we will compute up to, and use that as a simulation of our input. Let us define p as the possible values for each possible prime in the array, which is what we are actually modifying. With that in mind, the algorithm runs in $O(n * log(p)!)$ time.*

Proof. Our algorithm performs multiple operations on the array $xSequence$, in order to both ensure that all possible values of n are reached, and that pruning is performed so that combinations which produce out-of-index values are not counted. To that end, we iterate over the array several times, maintaining that $p_i \geq p_{i+1}$, meaning that we do not strictly brute-force the computations, and do not calculate all values since we do not need them. We perform n possible operations, but since we only modify the elements of p s.t. modifying them obeys $p_i \geq p_{i+1}$, we have an operating time of $O(n * log(p)!)$ □