Factoring: Existence

In this section, we discuss the following (almost obvious) statement:

Theorem

Every integer $m > 1$ can be represented as the product of primes: there exists a positive integer k and primes $p_1, \ldots, p_k$ such that $m = p_1 \ldots p_k$.

Stop and think! What happens if m is prime?

In this case, $k = 1$ and $p_1 = m$. So the phrase "product of primes" should not be taken too literally: in this case there is only one (prime) number in the "product" and the product is equal to this number. (One could even allow $m = 1$ by considering an "empty product" with no factors that is equal to 1, but we will not go that far.)

Note also that we do not require the $p_1, \ldots, p_k$ to all be distinct: for example, $4 = 2 \cdot 2$ and $12 = 3 \cdot 2 \cdot 2$.

Stop and think! Do you see how to prove the theorem?

Let us look at this statement from the algorithmic point of view: given some $m > 1$, how can we find a prime decomposition (existence of which we want to prove)?

If m is prime, we already know that this decomposition consists of m only. What if m is composite (not prime)? By definition then, $m = uv$ for some u, v where $1 < u, v < m$. How can we then find the decomposition of m? It is easy: just combine the decompositions of u and v, forming a long product from the two shorter ones. Both u and v are smaller that m, so we may assume (by induction in our proof, or a recursive call in our algorithm) that for u and v the decompositions exist and can be found.

```
# Finds the minimal divisor>1 of the given integer m>1
def min_divisor(m):
    for d in range(2, m + 1):
        if m % d == 0:
            return d
        # optimization:
        if d * d > m:
            return m


def is_prime(m):
    return m == min_divisor(m)


def factoring(m):
    if is_prime(m):
        return [m]
    else:
        divisor = min_divisor(m)
        factors = factoring(m // divisor)
        factors.append(divisor)
        return factors


for i in (7, 60, 1001, 2 ** 32 + 1, 2 ** 64 + 1):
    print(f'Factoring of {i}: {factoring(i)}')
```

Run

Reset

```
Factoring of 7: [7]
Factoring of 60: [5, 3, 2, 2]
Factoring of 1001: [13, 11, 7]
Factoring of 4294967297: [6700417, 641]
Factoring of 18446744073709551617: [67280421310721, 274177]
```

```
Factoring of 7: [7]
Factoring of 60: [5, 3, 2, 2]
Factoring of 1001: [13, 11, 7]
Factoring of 4294967297: [6700417, 641]
Factoring of 18446744073709551617: [67280421310721, 274177]
```

Stop and think! Examine the algorithm and compare it with the description above. What is the discrepancy and why is the algorithm correct?

The algorithm finds the smallest divisor `d` and then decomposes `m` into `d` $\cdot$ (`m // d`). If~we followed our description literally, we should be applying the algorithm recursively to both `d` and `m // d` and concatenate the resulting lists. Instead, we just append `d` to the decomposition of `m // d`.

Stop and think! Why is it OK to do this?

In other words, why may we assume that the smallest divisor `d` of any integer `m>1` is a prime? The answer is in the word "smallest": if `d` were not prime, then its divisors would be smaller divisors of `m`. (Recall that a divisor of a divisor is a divisor: $u|d$ and $d|m$ implies $u|m$, as we have seen.)

Problem

Note that in our examples the factors go in the non-increasing order in the decomposition provided by our algorithm. Will it be always the case?

Solution

In fact it will, and it is easy to see why, also using inductive (recursive) argument. In our algorithm, `d` is the *smallest* divisor of `m`, so all divisors of `m//d` (being also divisors of `m`) are at least `d`. So appending `d` to the ordered list for `m//d` (induction assumption), we do not destroy the ordering.

Problem

There is a funny trick for children that learn division. Take any three digit number, say 358. Write it twice: 358358. Then divide the resulting number by 7, then by 11 and then by 13. Miraculously all divisions do not give a remainder and the result is the original number 358 (indeed, $358358/7 = 51194, 51194/11 = 4654$ and $4654/13 = 358$). Can you explain why this trick always works?

Hint: look at the second line in our factorization examples.

The numbers of the form $2^{2^n} + 1$ were considered long ago by Pierre Fermat (1607--1665, Wikipedia), famous for the Last Fermat theorem; we will see the other result of him later in this chapter. He noticed that $2^0 + 1, 2^1 + 1, 2^2 + 1, 2^4 + 1, 2^8 + 1, 2^{16} + 1$ are all primes and conjectured that all subsequent numbers are also prime. Only later Leonhard Euler (1707--1783, Wikipedia) discovered the factorization of $2^{32} + 1$ in 1732, and more than a century after that was needed to find the factorization of $2^{64} + 1$. Now our simple program gives these factorization in seconds. (But do not forget two optimization lines, otherwise it would take much longer.)

For the next Fermat numbers one should use more advanced factorization algorithms. There is a module `sympy` in python that provides function `factorint` that factors $2^{128} + 1$ and $2^{256} + 1$ in minutes, but Fermat numbers grow really fast (and only few more factorizations are known, even if we use the most advanced factorization tools).

Completed **Go to next item**

Like **Dislike** **Report an issue**