Codility_

Chapter 10

Prime and composite numbers

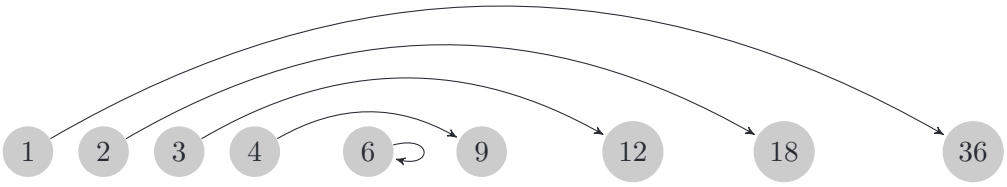
People have been analyzing prime numbers since time immemorial, but still we continue to search for fast new algorithms that can check the primality of numbers. A prime number is a natural number greater than 1 that has exactly two divisors (1 and itself). A composite number has more than two divisors.

In the above picture the primes are highlighted in white and composite numbers are shown in gray.

10.1. Counting divisors

Let's count the number of divisors of n. The easiest approach is to iterate through all the numbers from 1 to n and check whether or not each one is a divisor. The time complexity of this solution is $O(n)$.

There is a simple way to improve the above solution. Based on one divisor, we can find the symmetric divisor. More precisely, if number a is a divisor of n, then $\frac{n}{a}$ is also a divisor. One of these two divisors is less than or equal to $\sqrt{n}$. (If that were not the case, n would be a product of two numbers greater than $\sqrt{n}$, which is impossible.)



Thus, iterating through all the numbers from 1 to $\sqrt{n}$ allows us to find all the divisors. If number n is of the form k^2, then the symmetric divisor of k is also k. This divisor should be counted just once.

10.1: Counting the number of divisors — $O(\sqrt{n})$.

```
1 def divisors(n):
2     i = 1
3     result = 0
```

```
4       while (i * i < n):
5           if (n % i == 0):
6               result += 2
7           i += 1
8       if (i * i == n):
9           result += 1
10      return result
```

10.2. Primality test

The primality test of n can be performed in an analogous way to counting the divisors. If we find a number between 2 and $n-1$ that divides n then n is a composite number. Otherwise, n is a prime number.

10.2: Primality test — $O(\sqrt{n})$.

```
1  def primality(n):
2      i = 2
3      while (i * i <= n):
4          if (n % i == 0):
5              return False
6          i += 1
7      return True
```

We assume that 1 is neither a prime nor a composite number, so the above algorithm works only for $n \geqslant 2$.

10.3. Exercises

Problem: Consider n coins aligned in a row. Each coin is showing heads at the beginning.

Then, n people turn over corresponding coins as follows. Person i reverses coins with numbers that are multiples of i. That is, person i flips coins $i, 2 \cdot i, 3 \cdot i, \ldots$ until no more appropriate coins remain. The goal is to count the number of coins showing tails. In the above example, the final configuration is:

Solution $O(n \log n)$: We can simulate the results of each person reversing coins.

10.3: Reversing coins — $O(n \log n)$.

```
1  def coins(n):
2      result = 0
3      coin = [0] * (n + 1)
4      for i in xrange(1, n + 1):
5          k = i
6          while (k <= n):
7              coin[k] = (coin[k] + 1) % 2
8              k += i
9          result += coin[i]
```

```
10      return result
```

The number of operation can be estimated by $\frac{n}{1}+\frac{n}{2}+\ldots+\frac{n}{n}$, what equals $n\cdot(\frac{1}{1}+\frac{1}{2}+\ldots+\frac{1}{n})$. The sums of multiplicative inverses (reciprocals) of the first n numbers are called harmonic numbers, which asymptotically equal $O(\log n)$. In summary, the total time complexity is $O(n \log n)$.

Solution $O(\log n)$: Notice that each coin will be turned over exactly as many times as the number of its divisors. The coins that are reversed an odd number of times show tails, meaning that it is sufficient to find the coins with an odd number of divisors.

We know that almost every number has a symmetric divisor (apart from divisors of the form $\sqrt{n}$). Thus, every number of the form k^2 has an odd number of divisors. There are exactly $\lfloor\sqrt{n}\rfloor$ such numbers between 1 and n. Finding the value of $\lfloor\sqrt{n}\rfloor$ takes logarithmic time (or constant time if we use operations on floating point numbers).

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.