Arithmetic – While loop – II

Our study of numbers is further developed with the loop "while". For this chapter you need your function `is_prime()` *built in the part "Arithmetic – While loop – I".*

Activity 1 (Goldbach's conjecture(s)).

Goal: study two Goldbach conjectures. A conjecture is a statement that you think is true but you can't prove it.

1. **Goldbach's good guess:** *Every even integer greater than* 4 *is the sum of two prime numbers.*
   For example $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, $10 = 3 + 7$ (but also $10 = 5 + 5$), $12 = 5 + 7$,... For $n = 100$ there are 6 solutions: $100 = 3 + 97 = 11 + 89 = 17 + 83 = 29 + 71 = 41 + 59 = 47 + 53$.
   No one can prove this conjecture, but you will see that there are good reasons to believe it is true.
   (a) Program a function `number_solutions_goldbach(n)` which for a given even integer n, finds how many decompositions $n = p + q$ there are with p and q two prime numbers and $p \leqslant q$.
   For example, for $n = 8$, there is only one solution $8 = 3 + 5$, but for $n = 10$ there are two solutions $10 = 3 + 7$ and $10 = 5 + 5$.
   Hints.
   - It is therefore necessary to test all p including 2 and $n/2$;
   - set $q = n - p$;
   - we have a solution when $p \leqslant q$ and p and q are both prime numbers.

   (b) Proves with the machine that the Goldbach conjecture is verified for all even integers n between 4 and 10 000.
2. **Goldbach's bad guess:** *Every odd integer n can be written as*
   $$n = p + 2k^2$$
   where p is a prime number and k is an integer (possibly zero).
   (a) Program a function `is_decomposition_goldbach(n)` that returns "True" when there is a decomposition of the form $n = p + 2k^2$.
   (b) Show that Goldbach's second guess is wrong! There are two integers smaller than 10 000 that do not have such decomposition. Find them!

Activity 2 (Numbers with 4 or 8 divisors).

Goal: disprove a conjecture by doing a lot of calculations.

Conjecture: *Between* 1 *and* N*, there are more integers that have exactly* 4 *divisors than integers that have exactly* 8 *divisors.*

You will see that this conjecture looks true for N rather small, but you will show that this conjecture is false by finding a large N that contradicts this statement.

1. **Number of divisors.**
   Program a function `number_of_divisors(n)` that returns the number of integers dividing n.
   For example: `number_of_divisors(100)` returns 9 because there are 9 divisors of $n = 100$:
   $$1, 2, 4, 5, 10, 20, 25, 50, 100$$
   Hints.
   - Don't forget 1 and n as divisors.
   - Try to optimize your function because you will use it intensively: for example, there are no divisors strictly larger than $\frac{n}{2}$ (except n).
2. **4 or 8 divisors.**
   Program a function `four_and_eight_divisors(Nmin,Nmax)` that returns two numbers: (1) the number of integers n with $N_{\text{min}} \leqslant n < N_{\text{max}}$ that admit exactly 4 divisors and (2) the number of integers n with $N_{\text{min}} \leqslant n < N_{\text{max}}$ that admit exactly 8 divisors.
   For example `four_and_eight_divisors(1,100)` returns `(32,10)` because there are 32 integers between 1 and 99 that admit 4 divisors, but only 10 integers that admit 8.
3. **Proof that the conjecture is false.**
   Experiment that for "small" values of N (up to $N = 10\,000$ for example) there are more integers with 4 divisors than 8. But calculate that for $N = 300\,000$ this is no longer the case.
   Hints. As there are many calculations, you can split them into slices (the slice of integers $1 \leqslant n < 50\,000$, then $50\,000 \leqslant n < 100\,000$,...) and then add them up. This allows you to share your calculations between several computers.

Activity 3 (121111... is never prime?).

Goal: study a new false conjecture!

We call U_k the following integer:
$$U_k = 12\underbrace{111\ldots111}_{k \text{ occurrences of } 1}$$
formed by the digit 1, then the digit 2, then k times the digit 1.
For example $U_0 = 12$, $U_1 = 121$, $U_2 = 1211$, ...

1. Write a function `one_two_one(k)` that returns the integer U_k.
   Hint. You can notice that starting with $U_0 = 12$, we have the relationship $U_{k+1} = 10 \cdot U_k + 1$. So you can start with `u = 12` and repeat a number of times `u = 10*u + 1`.
2. Check with the machine that $U_0, \ldots, U_{20}$ are not prime numbers.

   You might think it's still the case, but it's not true. The integer U_{136} is a prime number! Unfortunately it is too big to be verified with our algorithms. In the following we will define what is a almost prime number *to be able to push the calculations further.*
3. Program a function `is_almost_prime(n,r)` that returns "True" if the integer n does not admit any divisor d such as $1 < d \leqslant r$ (we assume $r < n$).
   For example: $n = 143 = 11 \times 13$ and $r = 10$, then `is_almost_prime(n,r)` is "True" because n does not allow any divisor less than or equal to 10. (But of course, n is not a prime number.)
   Hint. Adapt your function `is_prime(n)`!
4. Find all the integers U_k with $0 \leqslant k \leqslant 150$ which are almost prime for $r = 1\,000\,000$ (i.e. they are not divisible by any integer d with $1 < d \leqslant 1\,000\,000$).

Hint. In the list you must find U_{136} (which is a prime number) but also U_{34} which is not prime but whose smallest divisor is 10 149 217 781.

Activity 4 (Integer square root)**.**

Goal: calculate the integer square root of an integer.

Let $n \geqslant 0$ be an integer. The ***integer square root of n*** is the largest integer $r \geqslant 0$ such as $r^2 \leqslant n$. Another definition is to say that the integer square root of n is the integer part of $\sqrt{n}$.
Examples:

- $n = 21$, then the integer square root of n is 4 (because $4^2 \leqslant 21$, but $5^2 > 21$). In other words, $\sqrt{21} = 4.58\ldots$, and we only keep the integer part (the integer to the left of the dot), so it is 4.
- $n = 36$, then the integer square root of n is 6 (because $6^2 \leqslant 36$, but $7^2 > 36$). In other words, $\sqrt{36} = 6$ and the integer square root is of course also 6.

1. Write a first function that calculates the integer square root of an integer n, first calculating $\sqrt{n}$, then taking the integer part.
   Hints.
   - For this question only, you can use the module `math` of `Python`.
   - In this module `sqrt()` returns the real square root.
   - The function `floor()` of the same module returns the integer part of a number.
2. Write a second function that calculates the integer square root of an integer n, but this time according to the following method:
   - Start with $p = 0$.
   - As long as $p^2 \leqslant n$, increment the value of p by 1.

   Test carefully what the returned value should be (beware of the offset!).
3. Write a third function that still calculates the integer square root of an integer n with the algorithm described below. This algorithm is called the Babylonian method (or Heron's method or Newton's method).

   Algorithm.
   Input: a positive integer n
   Output: its integer square root
   - Start with $a = 1$ and $b = n$.
   - as long as $|a - b| > 1$:
     - $a \leftarrow (a + b)//2$;
     - $b \leftarrow n//a$
   - Return the minimum between a and b: this is the integer square root of n.

   We do not explain how this algorithm works, but it is one of the most effective methods to calculate square roots. The numbers a and b provide, during execution, an increasingly precise interval containing of $\sqrt{n}$.
   Here is a table that details an example calculation for the integer square root of $n = 1664$.

Step	a	b
$i=0$	$a=1$	$b=1664$
$i=1$	$a=832$	$b=2$
$i=2$	$a=417$	$b=3$
$i=3$	$a=210$	$b=7$
$i=4$	$a=108$	$b=15$
$i=5$	$a=61$	$b=27$
$i=6$	$a=44$	$b=37$
$i=7$	$a=40$	$b=41$

In the last step, the difference between a and b is less than or equal to 1, so the integer square root is 40. We can verify that this is correct because: $40^2 = 1600 \leqslant 1664 < 41^2 = 1681$.

Bonus. Compare the execution speeds of the three methods using `timeit()`. See the chapter “Functions”.

Lesson 1 (Exit a loop).
It is not always easy to find the right condition for a loop “while”. `Python` has a command to immediately exit a loop “while” or a loop “for”: this is the instruction `break`.
Here are some examples that use this command `break`. As it is rarely an elegant way to write your program, alternatives are also presented.

Example.
Here are different codes for a countdown from 10 to 0.

```
# Countdown
n = 10
while True: # Infinite loop
    print(n)
    n = n - 1
    if n < 0:
        break # Immediate stop
```

```
# Better (with a flag)
n = 10
finished = False
while not finished:
    print(n)
    n = n - 1
    if n < 0:
        finished = True
```

```
# Even better
# (reformulation)
n = 10
while n >= 0:
    print(n)
    n = n - 1
```

Example.
Here are programs that search for the integer square root of 777, i.e. the largest integer i that satisfies $i^2 \leqslant 777$. In the script on the left, the search is limited to integers i between 0 and 99.

```
# Integer square root
n = 777
for i in range(100):
    if i**2 > n:
        break
print(i-1)
```

```
# Better
n = 777
i = 0
while i**2 <= n:
    i = i + 1
print(i-1)
```

Example.
Here are programs that calculate the real square roots of the elements in a list, unless of course the number is negative. The code on the left stops before the end of the list, while the code on the right handles the problem properly.

```
# Square root of the elements
# of a list
mylist = [3,7,0,10,-1,12]
for element in mylist:
    if element < 0:
        break
    print(sqrt(element))
```

```
# Better with try/except
mylist = [3,7,0,10,-1,12]
for element in mylist:
    try:
        print(sqrt(element))
    except:
        print("Warning, I don't know how to
        compute the square root of",element)
```