Project 2

Rodda John

06/24/2020

1 Overview

There are 3 different projects, sorted by Python comfort level.

If you finish one as a group, feel free to move up (or down) in level.

Feel free to ask for help!

2 Project A

More! Project! Euler!

This is a curated list of Project Eulers to try.

2.1 Euler Hints

- Generate functions to write separately

    - Find inputs and outputs for each function

    - Derive tests prior to writing the functions

    - Test each function separately

- If your algorithm works on small numbers but not larger ones, start thinking about the efficiency of the algorithm you've written

2.2 Euler 7

By listing the first six prime numbers: 2, 3, 5, 7, 11, and 13, we can see that the 6th prime is 13.

What is the 10 001st prime number?

2.3 Euler 2

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

2.4 Euler 23 (a bit harder)

A perfect number is a number for which the sum of its proper divisors is exactly equal to the number. For example, the sum of the proper divisors of 28 would be 1 + 2 + 4 + 7 + 14 = 28, which means that 28 is a perfect number.

A number n is called deficient if the sum of its proper divisors is less than n and it is called abundant if this sum exceeds n.

As 12 is the smallest abundant number, 1 + 2 + 3 + 4 + 6 = 16, the smallest number that can be written as the sum of two abundant numbers is 24. By mathematical analysis, it can be shown that all integers greater than 28123 can be written as the sum of two abundant numbers. However, this upper limit cannot be reduced any further by analysis even though it is known that the greatest number that cannot be expressed as the sum of two abundant numbers is less than this limit.

Find the sum of all the positive integers which cannot be written as the sum of two abundant numbers.

3 Project B

Project B introduces you to what's called a 'recursive' function – or a function that calls itself.

The easiest example of this is the factorial function.

3.1 Factorial function

`5! = 5 * 4 * 3 * 2 * 1`
OR
`5! = 5 * 4!` such that `1! = 1`
THUS
`n! = n * (n - 1)!`

The case of `n = 1` is considered your "base case" – where the function is defined as a constant, and not in relation to itself.

The general pseudocode of a recursive function looks like:

```
def recursive_function(inputs...):
   if <base_case>:
      return <answer_for_base_case>
   return <something and a call to itself>
```

THUS

```
def factorial(n):
    if n == 1:
return 1
    return n * factorial(n - 1)
```

3.2 Fibonacci

Try writing a function that given an input `n` gives you that element Fibonnacci number.

A Fibonacci number is defined as:

$F_n = F_{n-1} + F_{n-2}$
such that
$F_1 = 1$ and $F_0 = 0$
thus
$F_2 = F_{2-1} + F_{2-2}$
$F_2 = F_1 + F_0$
$F_2 = 1 + 0$
$F_2 = 1$

Here is a table to test your function when you have a working function:

n	F_n
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89
12	144
13	233
14	377

3.3 Sum of `n` integers

Write a recursive function that finds the sum of the first `n` integers.

ie:

For `n = 1` the answer is 1
For `n = 2` the answer is 3
For `n = 5` the answer is `5 + 4 + 3 + 2 + 1` or `15`

3.4 Sieve of Erastothenes

This is the Sieve of Erastothenes

```
1-Create a list of integers from two to n: 2, 3, 4, ..., n
2-Start with a counter i set to 2, i.e. the first prime number
3-Starting from i+i, count up by i and remove those numbers from
the list, i.e. 2*i, 3*i, 4*i, etc..
4-Find the first number of the list following i. This is the next
prime number.
5-Set i to the number found in the previous step
6-Repeat steps 3 and 4 until i is greater than n. (As an improvement:
It's enough to go to the square root of n)
7-All the numbers, which are still in the list, are prime numbers

You can easily see that we would be inefficient, if we strictly used
this algorithm, e.g. we will try to remove the multiples of 4, although
they have been already removed by the multiples of 2. So it's enough to
produce the multiples of all the prime numbers up to the square root of n.
We can recursively create these sets.
```

This is the most efficient way to find prime numbers (sans random heuristic optimizations)

4 Project C

This should only be attempted if you have familiarity with recursive functions and/or have completed Project B.

4.1 Binary Search

A search algorithm is an algorithm which accepts a (sorted) list of values, and a value to find. These are often termed the `needle` and the `haystack`. The goal is to return the index of the found element.

The way a binary search algorithm works is it checks the `needle` against the middle value of a list, there are three cases:

1. The `needle` is the middle index –> return the middle index.

2. The `needle` is less than the value at the middle index –> run the algorithm on the list to the left of the middle index.

3. The `needle` is greater than the value at the middle index –> run the algorithm on the list to the right of the middle index.

This is the fastest simple search algorithm.

I suggest you write the function like so:
`def binary_search(needle, haystack, start_index, end_index)`
The call to `binary_search` will pass in a `needle` (value to find), a `haystack` (list), as well as the starting and ending index, these will be `0` and `len(haystack) - 1` to begin.

4.2 More

If you have finished everything – talk to Rodda, he will give you two options