F28PL - Programming Languages

Question 1 (More Python)

Tommy Lamb

H00217505

Where answers comprise of more than one function, each is listed and commented on separately, in an order approximating lowest to highest hierarchically. That is independent functions first and then in increasing number of dependencies.

Note:

Certain functions utilise other functions defined in earlier questions. Any such prerequisites are noted in blue for each applicable question.

1. Write a program `factors` that inputs a number `n`, which you may assume is greater than 2, and outputs the list of its prime factors (prime numbers are 2, 3, 5, 7, 11, 13, 17, and so on). You may assume basic arithmetic operations, including modulus `n % k` and divisibility. However, if you find a library call "prime_factors_of", you can't use that, sorry.

   ```
   from math import sqrt, ceil
   ```

   - Imports from the Math Library functions to ceiling a number, and to calculate the square root of a number.

```
def isprime(number):
    if (number == 2) or (number == 3):
        return True

    if (number % 2 == 0) or (number % 3 == 0) or (number == 1):
        return False

    for counter in range(5, ceil(sqrt(number)), 6):
        if ((number % counter) == 0) or ((number % (counter + 2)) == 0):
            return False
    return True
```

- A function to carry out primality testing on a given number. The algorithm was adapted from that featured in the Wikipedia article on primality tests, and was used and tested extensively as part of coursework for F28DA. First it checks if the number is 2 or 3, which are prime. Then it checks if the number can be divided by 2, 3, or is 1 and thus is not prime. Getting past these checks the program then iterates through all of the numbers not in the 2 or 3 times tables and checks if these are factors of the number, starting at 5. It does this by setting `counter` to 5, and checking against counter (5) and counter+2 (7) before incrementing by 6 - everything in between is a multiple of 2 or 3 (except 7, which was checked). Another trick used to make the algorithm more efficient is by limiting the iteration by the square root of the number rather than the number itself. This works because any value which is larger than $\sqrt{number}$must multiply another value smaller than $\sqrt{number}$ to achieve *number*. Any such smaller value must have been checked by the algorithm before reaching this point, and to reach this point no such value can exist (else the `return` statement is fired). Thus no value larger than $\sqrt{number}$ can exist which is a factor of *number*, such that the algorithm would otherwise return true than for its presence.

```
def factors(number):
    accumulator = []
    for testcase in range(2, number+1, 1):
        if (number % testcase == 0) and isprime(testcase):
            accumulator.append(testcase)
       return accumulator
```

- A simple iteration over all values from 2 to the number inclusive along with a conditional is used to accumulate a list containing all prime factors in this function. Each value is concatenated to the list iff it is both a factor of the number, and a prime number. In the end the function returns this accumulated list. Note that the function (rightly) includes the number itself in the list of prime factors, given that number is itself prime of course.

2. Write a program `largest` that inputs a nonempty list of numbers and outputs its greatest element. You do not need to worry about error-handling, e.g. if the input is the empty list.

```
def largest(intlist):
    maxvalue = intlist[0]
    for value in intlist[1:]:
        if maxvalue < value:
            maxvalue = value
    return value
```

   - The program assumes the head of the list to be the largest value, before iterating through and comparing each successive list element to the stored largest value, updating it as necessary, and in the end returning this stored value.

3. Write a program `largest_factor` that inputs a number n and outputs its largest prime factor.

```
def largest_factor(number):
    return largest(factors(number))
```

Questions 1 & 2 are prerequisites for this method

   - This function simply returns the result of calling the previous method `largest` on the result of calling `factors` (Q1) on the argument list.

4. Write a program `firstbigfib` that inputs a number n and outputs the index of the first Fibonacci number to contain n digits.

5. Write a program `firstbigf` that inputs a number n and a function `f` mapping numbers to numbers, and outputs the least strictly positive number `i` (so `i` is in `[1,2,3,...]`) such that `f(i)` contains n digits. So for example if `f=lambda x:10**x` then `firstbigf f 10` should calculate 9 (because `10**9`, which is equal to `1000000000`, has ten digits).

Please note that Question 4 here utilises the method defined as an answer to Question 5. This is because Q5 was completed before Q4, and reusing Q5 code seemed the best way to achieve the required result. As such, note that method `firstbigf` is the answer to Q5, and `firstbigfib` the answer to Q4. Method `fibonacci` is the function passed to `firstbigf` within the `firstbigfib` method. In the code listing the ordering of Q4 and Q5 reflects the order that they must be declared to the interpreter, not the order of questions as here.

```
# Question 4A - Independent method
# This program treats the Fibonacci sequence as "0,1,1,2...", with "0" as the zeroeth value
def fibonacci(index):
    fiblist = [0, 1]
    if index == 0:  # edge case handling
        return fiblist[0]

    while index > (len(fiblist) - 1):
        nextfib = fiblist[-1] + fiblist[-2]
        fiblist.append(nextfib)
       return fiblist[-1]
```

- This function returns the Fibonacci number of specified index by iteratively constructing a list of all Fibonacci numbers up to and including that index. Starting with a list containing zero and one, the functions intuitively computes the next Fibonacci number before concatenating this to the list and continuing.

Function `firstbigf` (on next page) must be declared first before the following function will run.

```
#Question 4B - Dependent function - Answer compliant
def firstbigfib(number):
       return firstbigf(number, fibonacci)
```

- This function returns the result of calling `firstbigf` (Q5) on the function `fibonacci`. This is done as `firstbigf` is the higher-order version of the question-defined `firstbigfib` (Q4), and it is logical to use the higher order function alongside `fibonacci` to achieve the same results. It meant that only a function to return a Fibonacci number of specified index need be created, namely `fibonacci`, rather than a whole method to deal with both the length testing and iteration done by `firstbigf` and the generation performed by `fibonacci`. The only downside to this methodology is that it significantly complicates the question-answer numbering system.

```
# Question 5 - Answer
def firstbigf(number, function):
    count = 1
    while True:
        if len(str(function(count))):
            return count
        count += 1
```

- This function continuously tests the length of the string representation of the return value of calling the argument function on a number. Starting at 1, the function tests in steps of 1 until a case where a return value of the specified length is found. The function will run until the process is interrupted or killed, the system crashes, or an answer is found. This is the only way to ensure that any answer that exists will be found, without resorting to mathematical analysis of the argument function.
  The casting to string is required, as length function `len` does not accept integer arguments.

6. Suggest, in very general terms, how we might optimise firstbigf to run on an m-core architecture where m>1, with a roughly m-fold speedup.

   Given that the result of calling the function (`f`) on any number has no impact on the result of `f` applied to any other number, the process can be highly parallelised - ideal for multi-core architectures. Each application of `f` to a number can be independently performed by any number of cores/threads/logical processors to compute a result. The only limitation on parallelisation is in evaluating the final return value, which of all the successful applications of `f` was the result of calling `f` on the smallest value. This requires a core/thread to be dedicated to managing the parallelisation and parsing the results of each `f` application into the correct answer. If there is an inbuilt `map` method or similar in the language in question, it is likely that the compiler/interpreter can optimise it better than a self-defined method, and quite possibly in the aforementioned manner.

7. A **Pythagorean triple** is a 3-tuple of strictly positive numbers `(x,y,z)` such that `z*z=x*x+y*y`. Using Python generators (page 97 of these slides or the generator that yields items example here), write a Python generator function `triples()` to generate all Pythagorean triples. So:
   `x=triples()` and then `next(x)` should return a first triple, and then
   `next(x)` should return a second triple, and then
   `next(x)` should return a third one, and so on.
   Test your program further by running it on `list(itertools.islice(triples(), 1, 300))` (using the Internet or otherwise, make sure you understand what this does; you may need to import itertools).

```
def triples():
    counter = 2
    previousresult = (4, 3, 5)
    while True:
        yield previousresult
        counter += 1
        nextresulta = previousresult[0] + previousresult[1] + previousresult[2]
        nextresultb = fibonacci((2 * counter) - 1) - previousresult[1]
        nextresultc = fibonacci(2 * counter)
           previousresult = (nextresulta, nextresultb, nextresultc)
```

`Question 4A (fibonacci)` is a prerequisite for this function.

- The difference between this generator function and any other method is the use of the `yield` keyword. With `yield`, the state of the method is saved upon the yielding (returning) of a value unlike with `return`. When an appropriate call is made, rather than starting from the first line of the method as normal the function instead continues executing from the `yield` statement, using the stored state information. This is a very simple method for creating an iterateable object, which can easily be converted to an iterator object. The benefit of this is that it helps to avoid storing large lists of data, as with the `fibonacci` function in Q4. That could be much improved by use of a generator function.

  With this difference in mind, the function can otherwise be interpreted much like any other. With the starting point of Pythagorean triple (4,3,5) this function loops forever, calculating successive Pythagorean triples and yielding them. The `counter` variable is used to maintain an index of Fibonacci numbers used in the algorithm for actually calculating the triple values.

  The algorithm - which is based on Fibonacci sequence numbers - was taken from the Wikipedia article on Algorithms for generating Pythagorean triples. Neither knowledge nor understanding of the underlying maths is claimed, only a knowledge of the process. Note that the listed algorithm uses 1-based indexing of Fibonacci numbers whereas `fibonacci` (Q4A) uses 0-based counting, thus the counter value is 1 less than that in the Wikipedia article.