## **CS Problem Set #1**

These problems cover the essential understandings of the Problem Solving with Python unit. Many of the questions are reflective in nature and have a variety of correct answers. **Questions #1 - 6 are required.** Challenge problems are optional, though encouraged. Submit your answers to the CS Problem Set #1 Gradescope assignment:

**This assignment is due on Sunday, July 7th by 11:59 pm**. You should upload a single python file with all the code you used for these problems encapsulated as stand-alone functions. You can type your answers questions 1- 6 into Gradescope. Answers to challenge problems should be uploaded as a single .pdf document in the Gradescope assignment.

Required Problems

1. Consider the following problem definition given in Project 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.***

1. List key vocabulary terms from this problem definition. How will these terms affect the code that you’ll write?
2. If you were to generalize PE#2 as a function:
   1. What would you name the function?
   2. What parameter(s) would you use and why?
3. Write five test cases for a generalized form of PE#2 in an input → output format. Explain why you chose each test case.

a)Fibonacci, adding two previous, sum, even, four million. This terms tell us how to generate Fibonacci numbers. Then it tells us that we have to sum even Fibonacci numbers under four million.

b) i)SumOfEvenFibonacci(n).

ii) n – upper limit for Fibonacci numbers, in our case it’s 4000000

1. I would test -5, 0, 1, 2, 8. For -5 and 0 the function should return 0, because there is no Fibonacci numbers less then -5 and 0. For 1 the function should return 0, because 1 is not even-valued. For 2 the function should return 2, because 2 is only number that is even-valued and Fibonacci. And for 8 it should return 10.
2. Given what you now know about **is\_prime**, list at least 5 good test cases and explain your reasoning.

-5, because the program might return True, if you didn't check whether n is bigger than 1. For the same reason 1 would be a great case to check. Then I would check the case 7, to check whether the program identify primes correctly. Then I would check 2 more cases both products of large enough primes. I would check 11\*13 = 143 and 17\*19 = 323.

1. If it takes 7 nanoseconds to divide 2 64-bit integers on your laptop, how many seconds (rounded to nearest whole number) would it take to determine that 2971215073 (a fibonacci prime) is a prime number using **is\_prime\_exhaustive** vs **is\_prime\_factor\_fold**? Justify your answer.

If we dont count the time to determine the square root of 2971215073, then we only have to do 54509 checks in is\_prime\_factor\_fold. On other hand in is\_prime\_exhaustive we will have to do 2971215073 checks. So for is\_prime\_factor\_fold the time going to be 2971215073\*7 nanoseconds and for is\_prime\_exhaustive the time going to be 54509\*7 nanoseconds.

1. Encryption works because of the “asymmetry in timing between multiplying, dividing, and factoring”. What does this mean? Provide an example. Feel free to use the internet for research, though you should use your own words in forming your response.   
     
   Basically, due to asymmetry in timing between multiplying, dividing, and factoring, to encrypt something we can use multiplication operations which is going to be very fast, but if want to decrypt the same thing now we have use division and factorization which is going to be much slower. If we take extreme cases for example multiplying n large prime numbers, using computer we can do it relatively fast, but factoring of the same number going to take much more time.
2. Solve 2 additional prime-related Project Euler problems:
   1. [PE #7](https://projecteuler.net/problem=7): Find the 10001st Prime

I wrote 2 functions, one function checks if the number is prime, another function returns the n-th prime number. In the first function I checked every if our number n is divisible by another number from 2 to square root of n. In the second functions I just looked through every integer and check it if it is prime or not, and when I meet prime I reduce n by 1. And i do it until n becomes zero and then stop. The resulting prime will be n-th prime number.

I didn’t have any problems.

* 1. [PE #10](https://projecteuler.net/problem=10): Summation of Primes

I wrote 2 functions, one function checks if the number is prime, another function returns the n-th prime number. In the first function I checked every if our number n is divisible by another number from 2 to square root of n. In the second functions I just looked through every integer that is less then n and check it if it is prime. If yes then I add it to the total sum. At the end I return the sum.

I didn’t have any problems.

* 1. [PE #27](https://projecteuler.net/problem=27): Quadratic Primes
  2. [PE# 35](https://projecteuler.net/problem=35): Circular Primes
  3. [PE# 37](https://projecteuler.net/problem=37): Truncatable Primes
  4. [PE# 41](https://projecteuler.net/problem=41): Pandigital Primes

Answer the following questions for each of the two Project Euler problems:

1. Which Project Euler problem did you solve?
2. How did you solve the problem? Describe your algorithm in English.
3. What major roadblocks did you hit in your attempts to understand how to solve the problem? How did you get over the roadblocks?

(Challenge Problems on next page)

CS Problem Set #1: Challenge Problems

**Challenge problems are optional for CS2.**

**Challenge #1**: The Trouble with Roots

[Square root algorithms use loops to successively refine approximations](https://drive.google.com/file/d/19ckd1MUR4qz9imq8XxmWKPpaKsm5F2oj/view?usp=drive_link) of the square root of a given number. Consequently, the **sqrt** operation is a relatively slow procedure.

* 1. Describe an experiment to test the hypotheses “the square root function is a relatively slow procedure.”
     1. Compare at least three different square root algorithms with multiplication and division
        1. Verify each algorithm is correct by establishing at least 10 test cases
     2. Produce at least one graph
  2. What did you measure in your experiment?
  3. What inputs did you use to conduct the experiment? Why?

**Challenge #2:** Prime Densities

Read about the [prime counting problem](https://www.quantamagazine.org/mathematicians-will-never-stop-proving-the-prime-number-theorem-20200722/). This value is called π(n), where π is the “prime counting function.” For example:

π(10) = 4 since there are four primes less than or equal to 10: 2, 3, 5 and 7

π(100) = 25

π(1,000) = 168

As you calculate π(10) , π(100) , π(1000) the percentage of primes decreases from 40% (4/10) to 25% (25/100) to 16.8% (168/1000).

1. Implement two versions of prime counting functions:
   1. basic counting using **is\_prime\_factor\_fold**
   2. [sieve of eratosthenes](https://www.youtube.com/watch?v=klcIklsWzrY)
2. Generate graphs demonstrating that:
   1. The percentage of prime numbers (ie prime\_count(n)/n) at or below n decreases as n gets larger.
   2. The error rate between the actual prime count and the approximation provided by n/ln(n) decreases as n gets larger.
3. Each version of **is\_prime** starts at 2, then works its way up to n. Would it make a difference to move in the opposite direction?
   1. Explain your position in English
   2. Generate a graph that makes an empirical justification for your answer.

**Challenge #3:**  Turn Fermat’s little theorem into a primality test.

1. Read about [the use of randomness in algorithms](https://www.quantamagazine.org/how-randomness-improves-algorithms-20230403/?mc_cid=6b703ccaff&mc_eid=ef94aa122d), then turn Fermat’s little theorem into a primality test called **is\_prime\_fermat,** and test with at least 10 random picks.
2. Generate graphs comparing the accuracy and efficiency of **is\_prime\_fermat** with **is\_prime\_factor\_fold**.