EECE 230X – Introduction to Computation and Programming
Programming Assignment 2

- This programming assignment consists of 4 problems
- Prerequisites: Topic 3
  - Problems 1 and 2: Topic 3 up to Lesson 7
  - Problems 3: Topic 3.Lesson 8
  - Problems 4: Topic 3.Lesson 9
- Related material: Selection and Repetition, testing primality, and bisection method

Problem 1. Factorial

If n is a positive integer, the factorial of n, which is denoted by $n!$, is defined as the product of all the positive integers less than or equal to n, i.e.,

$$n! = 1 \times 2 \times \ldots \times (n-1) \times n.$$

If the integer n is zero, $n!$ is defined to be 1.

Examples:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 1 \times 2 = 2 \\ 3! &= 1 \times 2 \times 3 = 6 \\ 4! &= 1 \times 2 \times 3 \times 4 = 24 \end{aligned}$$

Write a Python script that asks the user to enter an nonegative integer n and then finds and displays the factorial of n. If n is negative, your program should display the error message ”Negative number!”.

In this problem you are *NOT allowed* to use the `math` module, which has a function for computing the factorial.

Sample Input/Output:

```
Enter an integer n:0
Factorial of n: 1
---------------------
Enter an integer n:5
Factorial of n: 120
---------------------
Enter an integer n:-3
Negative number!
---------------------
Enter an integer n:51
Factorial of n: 1551118753287382280224243016469303211063259720016986112000000000000
```

Problem 2. Max in a sequence

Write a program which asks the user to enter:

- a nonnegative integer n, and
- a sequence of n integer.

Your program is supposed to find the maximum integer in the input sequence. If $n = 0$, your program should display the message "Empty sequence".

For example, if $n = 6$ and the sequence of integers is

10 1 100 2 7 5,

then the max is 100.

(*Hint:* As in the "find sum and average problem"[Topic 3], we can solve this problem without storing all the user input in memory (anyways, at this stage we don't know how to do that; we will study Python Lists next week). It is enough to use one variable `x` to keep track of the current user input and another variable `maximum` to keep track of the maximum. We also need a counter `i` to keep track of the number of integers entered by the user. First check if $n == 0$ and handle this case separately. If $n \geq 1$, do the following. Initialize `maximum` to the first user input (10 in the above example). Then read the remaining $n - 1$ integers entered by the user input in a loop. Each time you read a number, update `maximum` if needed. Complete the following code.

```
n = int(input("Enter number n of  integers:"))
if n==0:
    -------------
else:
    maximum = int(input("Enter an integer:"))
    i = ---------
    while -------
         x = ----
         --------
         --------
         --------
    print("Maximum:", maximum)
```

)

Problem 3. Primes

A positive integer n is called *a prime number* if it is not equal to 1 and if the only positive integers which divide n are 1 and n. For instance, 2,3,5,7,11,13,... are primes.

In Topic 3, we designed following primality test:

```
n = int(input("Enter n:"))
if n<=1:
     print(n,"is not prime")
else:
     isPrime = True
     d=2
     while d<=n-1:
         if n%d ==0:
             isPrime = False
             break
         d=d+1
     if isPrime:
         print(n,"is prime")
     else:
         print(n,"is not prime")
```

a) **Speedup.** Do you have to check all the integers $2, 3, \ldots, n-1$?

Argue that it is enough to check if n is divisible by an integer d such that $2 \leq d \leq \sqrt{n}$.

Use this observation to write a more efficient version of the above primality test.

Note that you can check if $d \leq \sqrt{n}$ without finding the square-root of n: check if $d * d \leq n$.

Note: The problem of testing primality has puzzled people since ancient times. The idea underlying the test in this problem was discovered before 200 BC. The most recent breakthrough related to efficient primality tests was made 2002 (google "AKS primality test"). To see why fast primality tests are needed, try running your code on the following prime numbers: 67280421310721 and 170141183460469231731687303715884105727. To stop your program, use Control-C. One application of large primes is cryptography.

b) **Density of primes.** Write a python script which given a positive integer x, computes the fraction $f(x)$ of prime numbers less than or equal to x, i.e., $f(x) = y/x$, where y is the the number of prime numbers less than or equal to x.

(*Hint:* Use nested loops: put your solution of Part (a) (after suitable modifications) in an outer loop.)

Sample Input/Output:

```
Enter x:2
Fraction of primes less than or equal to 2 : 0.5
---------------------------------------------------
Enter x:10
Fraction of primes less than or equal to 10 : 0.4
---------------------------------------------------
Enter x:500
Fraction of primes less than or equal to 500 : 0.19
```

Note: Asymptotically, $f(x)$ decays like $\frac{1}{\log x}$ as x tends to infinity. Display few values of $f(x) \times \log x$ for large x. You should get values close to 1. This is the so called Prime Numbers Theorem (1890's).

Problem 4. Squares

In this problem, you are *NOT allowed* to use the power operator $**$ or the `math` module.

a) **Square Test.** Write a python script which prompts the user to enter an integer n, and checks whether or not n is a square of another integer. That is, your program should check whether or not there exists an integer x such that $n = x^2$.

For instance, the following are squares: 0 (since $0 = 0^2$), 1 (since $1 = 1^2$), 4 (since $4 = 2^2$), 25 (since $25 = 5^2$), and 81 (since $81 = 9^2$). On the other hand, -25, 2, 3, 10 are not squares.

If the answer is YES, your program is supposed to print x as shown below.

(*Hint:* Check if such an x exists using a loop by first trying $x = 0$, then $x = 1$, then $x = 2$, and so on. Figure out when to stop. At a high level, the idea is similar the primality test. A key difference is that in the primality test we are searching for an evidence of the NO answer, i.e., nonprimality. Here we are searching for an evidence of the YES answer, i.e., squareness.)

Sample Input/Output:

```
Enter an intger n:3
3 is not a square
----------------------------
Enter an intger n:81
YES square: 81 = 9 ^2
```

b) **Faster test using the bisection method.** To see why a faster test is needed, try your solution of Part (a) on the integer

$$7053469893815737412161804472740352737004092269220283090412003703627788135835749348 01$$
$$= 83984938493849820834982094848938493894384 9^2.$$

It won't stop! In this part, you are asked to implement a much faster square test using ideas from the bisection method. For instance, it should run in a split second on the above integer.

Note that in Topic 3, we used the bisection method to find the square-root of a given *real number* x such that $0 \leq x \leq 1$, hence $x \leq \sqrt{x} \leq 1$. Here you are asked to use the idea of the bisection method to check if a given *integer* n is square. Thus, ignoring the trivial cases when $n < 0$ and $n = 0$, we have $1 \leq \sqrt{n} \leq n$.

(*Hints:*

- ⋆ Handle the cases $n < 0$ and $n = 0$ separately
- ⋆ Since $n \geq 1$, we have $1 \leq \sqrt{n} \leq n$. Hence, the initial search interval is $[1, n]$
- ⋆ Keep the boundaries of the search interval integers: instead of setting `mid` to the float `(low+high)/2`, set it to `(low+high)//2`, i.e., the integer part of `(low+high)/2`. One advantage of keeping `mid` an integer is that it enables checking if `mid*mid==n` without having to worry about approximation errors which can be significant for large values of n (another concern for large values of n is reaching the maximum number allowed by the float type).
- ⋆ If `mid*mid<n`,the square-root bisection method sets `low` to `mid`. Here we can do slightly better as our search space consists of integers.
- ⋆ A similar remark holds for the case when `mid*mid>n`
- ⋆ Figure out when to stop the loop. Note that here we don't have a variable called `epsilon` as in the square-root bisection method.)