COL 765	August 25, 2021
Assignment 1	
Instructor: Subodh Sharma	Due: September 5, 2021

NOTE: Unless explicitly specified, solution to each question should be accompanied with a brief correctness proof (for recursive solutions)/ invariants (for the tail-recursive solutions), and timing analysis. These arguments should be written as comments in the program file itself. A single solution file should be submitted containing all the attempts.

Problem 1: Lists and Sorting (150 Marks).

- Implement tail-recursive reversal of a list.
- Implement tail-recursive merge function.
- Implement tail-recursive Fibonacci function.
- Implement tail-recursive insertion sort.
- Implement recursive quick sort.
- Implement recursive binary search given a list, the element and the low and high indices of the list.

Problem 2: Primality Testing (50 marks)

Develop a recursive functional implementation for primality testing based on the following computational theory.

Fermat's little theorem: If n is a prime and $a < n$ is any positive integer, then $a^n \equiv a \texttt{ modulo } n$ (congruent modulo n: Two numbers are said to be congruent modulo n if they both have the same remainder when divided by n).

If n is not a prime, then, in general, most of the numbers $a < n$ will not satisfy the above relation. Thus, given a number n, we can pick a random number $a < n$ and then compute the remainder $a^n \texttt{modulo } n$. If this is not equal to a, n is certainly not a prime. Otherwise, chances are good that n is a prime. We can assume that the probability that n is a prime is 0.5. We can keep repeating the above experiment and stop if either (i) at any stage we find that n is not a prime, or (ii) we find that the probability that n is not a prime has decreased to an acceptable level.

With the above information, implement a recursive `prime n q` where n is the number whose primality is to be tested and q is the number of times the Fermat's test is to be applied. Note, no proof of correctness or timing analysis needs to be provided for this question.

Problem 3: Higher order functions (50 marks)

- Implement Newton's method for computing root of an arbitrary function f as a higher-order function. It should accept a function f, a parameter `guess` and an accuracy factor ϵ as input and evaluate the root as the output.

- Implement a recursive higher-order double summation function to compute:

$$\sum_{i=a}^{b}\sum_{j=c}^{d} f(i,j)$$