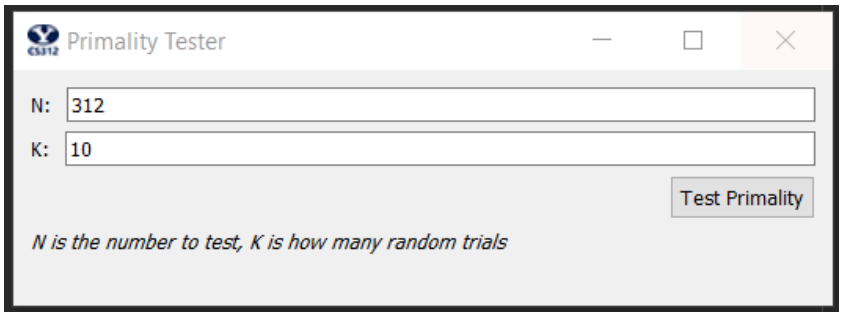
Project #1: Fermat's Primality Test



Instructions: Download the provided code for this project. Before you can run the provided GUI, you will need to set up Python 3 and install PyQT5 (see the python section in LS Content).

You will implement the code that is executed when the "Test Primality" button is clicked (see image above which is part of the provided GUI). The provided file fermat.py includes 6 functions, 3 of which are called from the GUI. You will implement five of these functions (the 6th simply passes your results back to the GUI).

1. Code up the Fermat primality test pseudocode from Figure 1.8 of the text. You may set *k* to any value you like (see p. 27), this value indicates how many random trials (values of *a*) are used.
2. Implement modular exponentiation (pseudo-code in Figure 1.4 of the text). Your primality test will use your modular exponentiation function and must work for numbers up to $2^{30} = 1073741824$.
3. Code the probability that *k* Fermat trials gave you the correct answer -- see the discussion between Figure 1.7 and Figure 1.8.
4. Implement the Miller-Rabin primality test. There is no pseudo-code in the book for this, but you can find what you need in the sidebar on p. 28 and in the discussion below.
5. Code the probability that *k* Miller-Rabin trials give you the correct answer.

Report: Your report should consist of:

1. [10 points] At least one screenshot of your application with a working example (distinct from the one above).
2. All of the code that you wrote. For this and all future code submissions note the following. In each project, be sure to always document your algorithm with legible comments in important places in your code that make it obvious to the reader what your solution is doing. Note the time/space complexity of each key portion of your code. This documentation also provides evidence of your comprehension. With the aid of adequate documentation, the correctness of your approach should be easy for the reader (e.g., TA) to verify. If your code spans multiple files, make sure that you include all of it in your PDF appendix, and make sure that it is organized in a way that makes it easy for the reader (TA) to follow. Your code must include the following:
    a. [20 points] A correct implementation of modular exponentiation.
    b. [20 points] A correct implementation of the Fermat primality tester.
    c. [10 points] A correct implementation of the Miller-Rabin algorithm.
    d. [10 points] A brief discussion of some experimentation you did to identify inputs for which the two algorithms disagree and give your best effort to explain why. Include a screenshot showing a case of disagreement.
3. [10 points] Explain the time and space complexity of your algorithm by showing and summing up the complexity of each significant subsection of your code. This should appear as a subsection of your report. Even though your full code in the appendix has this in its comments, this part is a much briefer pseudo-code version of your program that lets you point out the time and space complexity of each critical code portion. In fact, it should probably look very similar to the sketch pseudo-code you put together for your design experience.

4. [10 points] Discuss the two equations you used to compute the probabilities *p* of correctness for the two algorithms (Fermat and Miller-Rabin).

Notes:

- There is no performance requirement for this project. Correctness is the only criterion.
- This report is worth 90/100 points for the project, with the remaining 10 points accounted for in the design experience.
- Submit your PDF report following the submission directions in the Project Content section.

The Miller-Rabin test

Let's say the number we are testing is N=97; just as in the Fermat case, we choose a random test in the range $1 \leq a < N$; suppose for our test we chose a=3. Then,

a^(N-1) = 3^96 = 6362685441135942358474828762538534230890216321 ≣ 1 (mod 97),

as Fermat's theorem says it should. Since this is 1 (mod 97), if we take the square root, we would expect the result to be either 1 or -1. We can check this by computing

(3^96)^(1/2) = 3^(48) = 79766443076872509863361 ≣ 1 (mod 97),

just as expected. But, since this is also 1, we can take the square root again by computing

(3^48)^(1/2) = 3^24 = 282429536481 ≣ 96 ≣ -1 (mod 97). <-- Passed. Still looks prime.

Continuing the sequence, we can take yet another square root, but since we are now taking the square root of -1, we don't expect such nice behavior, and indeed, we get

(3^24)^(1/2) = 3^12 = 531441 ≣ 75 (mod 97). We can complete the sequence with
(3^12)^(1/2) = 3^6 = 729 ≣ 50 (mod 97) and
(3^6)^(1/2) = 3^3 = 27 ≣ 27 (mod 97),

and we can't divide the exponent by 2 anymore, so we are done. To summarize, what we are doing here is repeatedly taking the square root of a number that is ≣ 1 (mod N). For awhile, the result is, not unexpectedly, 1 (mod N), but at some point it is -1 (mod N). As we know, taking the square root of -1 is weird (though in this modular case that doesn't mean complex numbers; rather, we just get away from 1). What Miller and Rabin showed is that for prime numbers, for all choices of a, $1 \leq a < N$ this sequence of square roots, starting with a 1 (mod N) will either consist of all 1s (mod N), or, if at some point it changes to something else, that something else will always be N-1 ≣ -1 (mod N), which is exactly what we saw in our example. What they also showed was that for composites (*including* Carmichael numbers), for at least 3/4 (compared to the ½ we used for basic Fermat) of the possible choices for a, this will not be the case---either the initial test will not equal 1 (mod N), just as in the Fermat case, or, if it does, in taking the series of square roots, the first number to show up after the sequence of 1s will be something other than N-1 ≣ -1 (mod N).

Here is another example: suppose N=561 and we choose a=4. Then, computing our sequence of square roots, we get

4^560 = a really big number ≣ 1 (mod 561)
4^280 = a smaller but still really big number ≣ 1 (mod 561)
4^140 = a still pretty big number ≣ 1 (mod 561)
4^70 = 139379657490816394634598239204052259412377 6 ≣ 67 (mod 561) <-- Failed. Composite.
4^35 = 11805916207174113034 24 ≣ 166 (mod 561)

Notice that this time, that when we encountered something other than 1 (mod N) in the sequence, it was also not N-1 ≡ -1 (mod N). This is common for composites, but never happens for primes. Also notice in both examples that we stopped the sequence when taking the square root (halving the exponent) resulted in an odd exponent. Hopefully now it is becoming clear how we can implement an alternative to the Fermat-test-based primality testing algorithm with this new information---if our number N passes the initial test a (which is equivalent to a Fermat test, right?), we then compute this sequence of square roots (until we get to an odd exponent), looking for the first result that is not 1 (mod N). If there isn't one (because the entire sequence is 1s) or if the first such number is N-1 ≡ -1 (mod N), N has passed Miller-Rabin test a, and we still don't know anything for sure (so, we should continue to choose another random a and repeat our test); however, if we instead find that there is a first result in the sequence that is neither 1 nor N-1 ≡ -1 (mod N), then we know the number is composite.