

Programming-Language-Name-Of-Lab

Author: Jim Burnham - TopClown@STEAMClown.org

First Authored: Jan 13, 2018

Last Revised: 2019/01/21

Note to TopClown next time you edit: add hyperlinks to the python structures.

License: Distributed as Open Source. See the Appendix A, B, C for source and references.

Overview, Introduction and Objective:

This is a Python Lab (but could be C++ too) where you will write a program that prints *all* prime numbers. The objective is to use your prior knowledge of Python coding to implement this program.

Prior Knowledge:

- Doing Math, and checking the results with conditional `if/else`
- Understand conditional loops, including `for` and `while` loops
- Understand how to nest loops

What You Will Know & Be Able To Do:

- Have a greater level of understanding on how and when to use conditional statements including `if/else`, `for` and `while` loops
- Understand how to nest loops
- Be able to describe what a Prime number is and how to calculate them

Resources & Materials Needed:

- PC, Laptop or Raspberry Pi
- Link to GutHub for Source
 - [Python Template](#) to start from
- Link to online C++ or Python Compiler
 - [Python 3 On-Line Python Interpreter](#) - Tutorials Point
 - [Python 3 Interpreter](#) - Online GDB

How You Will Be Measured:

- Programming Lab Rubric link (coming Soon)
- You will turn your code as a `Lastname-Firstname-Prime.py` in to the [Google classroom...](#) Check the Stream or the Programing category for C++ or Python

Scenario & Lab Instructions:

Write a program that prints *all* prime numbers. (Note: if your programming language does not support arbitrary size numbers, printing all primes up to the largest number you can easily represent is fine too.)

Tip: Python integers - The maximum value a variable of type integer can take is usually $2^{31} - 1$ on a 32-bit platform and $2^{63} - 1$ on a 64-bit platform.

Extra Credit #1: use a type(long) to calculate and output larger numbers than an integer can hold.

Extra Credit #2: Write a program that asks the user to input a number and you then tell them if it is a Prime number. Rather than calculating all the Prime numbers up to and including that number, try to solve it using this algorithm described in this [Large Prime Numbers](#) explanation. If that link has died, see the Hint section below.

Copy, Edit & Execute Code

Instructions for accessing any example code on Github

- You can start by copying [Python Template](#) to start from. Update the template with your header comments and your code.
- Python Compiler
 - Use your Raspberry Pi
 - [Python 3 On-Line Python Interpreter](#) - Tutorials Point
 - [Python 3 Interpreter](#) - Online GDB

Expected Output:

2, 3, 5, 7, 11, 13,... → up to the largest number your platform can hold in an Integer.

Turn In Your Code:

Turn your code as a **Lastname-Firstname-Prime.py** as specified in the **How You Will Be Measured** section above.

Hint:

The Sieve of Eratosthenes

Eratosthenes (275-194 B.C., Greece) devised a 'sieve' to discover prime numbers. A sieve is like a strainer that you use to drain spaghetti when it is done cooking. The water drains out, leaving your spaghetti behind. Eratosthenes's sieve drains out composite numbers and leaves prime numbers behind.

To use the sieve of Eratosthenes to find the prime numbers up to 100, make a chart of the first one hundred positive integers (1-100):

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

1. Cross out 1, because it is not prime.
2. Circle 2, because it is the smallest positive even prime. Now cross out every multiple of 2; in other words, cross out every second number.
3. Circle 3, the next prime. Then cross out all of the multiples of 3; in other words, every third number. Some, like 6, may have already been crossed out because they are multiples of 2.
4. Circle the next open number, 5. Now cross out all of the multiples of 5, or every 5th number.

Continue doing this until all the numbers through 100 have either been circled or crossed out. You have just circled all the prime numbers from 1 to 100!

For more on Prime number check out [Ask Dr Math - Primality Testing](#)

Hint for Extra Credit: Large Prime Numbers

Date: 12/17/97 at 13:55:22

From: Lynne & Dave Ware

Subject: Prime Numbers

To the Math Swat Team:

I believe that I read somewhere (Ivars Pederson [spelling?], I think) that it is possible to determine if a very large number is prime using a simple procedure. (But not to determine its factors). It didn't say how this is done. If this is true, what is the algorithm?

Thanks for your consideration.

Dave Ware

Date: 12/17/97 at 15:50:01

From: Doctor Rob

Subject: Re: Prime Numbers

Good question! The person is probably Ivars Peterson, who is a popular science writer.

The situation is a bit more complicated than you remember. The fact is that the simple procedure does not provide a proof of primality, but may provide a proof of compositeness. It is based on Fermat's Little Theorem, which says:

Theorem: If p is a prime number, a is an integer, and p is not a divisor of a , then p is a divisor of $a^{(p-1)} - 1$.

Given a number N we want to test, pick any old a , and find the greatest common divisor of a and N , $\text{GCD}(a,N)$. If this is not 1, then it is a factor of N , and N is composite. If it is 1, then compute the remainder r of $a^{(N-1)}$ when divided by N . If r is not 1, then by Fermat's Little Theorem, N cannot be prime, so is composite. This gives the proof of compositeness.

By the way, you might as well choose $1 < a < N-1$, since a and $a-N$ will give the same value of r , and since $a = 1$ or $N-1$ will always

give $r = 1$.

How to compute the remainder r ? If N is even moderately large, computing $a^{(N-1)}$ and then dividing by N will be a bad idea, since the number $a^{(N-1)}$ will have many, many digits. The trick is to divide by N and keep only the remainder at all intermediate steps. It may not be obvious that this works, but it does. If $N = 67$, $N-1 = 66$, you might compute a^{66} by doing 65 multiplications. After each one, divide by 67 and keep only the remainder.

Better than doing a^{66} by 65 multiplications (and 65 divisions by N), you can shortcut the computation by the following trick:
 $a^{66} = (a^{33})^2$, $a^{33} = a \cdot a^{32}$, $a^{32} = (a^{16})^2$, $a^{16} = (a^8)^2$,
 $a^8 = (a^4)^2$, $a^4 = (a^2)^2$, $a^2 = a \cdot a$. Working from the last equation backwards, you will need only 7 multiplications (and 7 divisions by N).

If $r = 1$, then what can we say? All prime numbers will give $r = 1$, but there are a few composite numbers which will do so, too. For example, if $a = 2$, $N = 341 = 11 \cdot 31$ is composite, but $r = 1$. Such N 's are called Fermat pseudoprimes with respect to base a . 341 is a Fermat pseudoprime with respect to base 2.

It turns out that Fermat pseudoprimes with respect to any fixed base are uncommon. The chance of picking one at random from some large set of N 's is very small. Thus, in some sense, which is again a complicated matter, if you get $r = 1$, N is "probably" prime.

A tactic you could use if you get $r = 1$ is to pick another base a and repeat the calculation. If you get $r = 1$ again, try still another a . Continue this until you either find a proof that N is composite, or you decide that you couldn't possibly be unlucky enough to have an N which is a Fermat pseudoprime to all the bases you used. Then declare the number to be prime, and you will be wrong only rarely.

A flaw with the above tactic is that there exist numbers called Carmichael numbers, for which, for all bases a such that N passes the GCD test, r will equal 1. The smallest one is $561 = 3 \cdot 11 \cdot 17$. Every base a not divisible by 3, 11, or 17 will give $r = 1$. These are even rarer than Fermat pseudoprimes with respect to a given base a , but there are known to be infinitely many. If you happened to choose one of these, you might try very many bases, getting $r = 1$ over and over, then declare the number prime, and be wrong.

There are variations of the above method which get rid of the last flaw, give you proofs of compositeness, but only probabilistic statements about primality. If you need to know more about them, write again.

If you want a true proof of primality, there are more complicated methods which will do this, but I am quite sure the above is what you remember reading about. The complicated methods can be proven to run on a computer in a relatively short time, and produce a certificate of primality which can be checked even faster. They could not, however, be termed "simple" in most senses of the word!

-Doctor Rob, The Math Forum

Check out our web site! <http://mathforum.org/dr.math/>

Date: 12/18/97 at 10:36:38

From: Lynne & Dave Ware

Subject: Follow up question on testing for primes

Doctor Rob:

Thank you for your quick answer to my question.

I think I was looking for the true test of primality you mentioned in your last paragraph. I think Ivars Peterson did cover some of the theory on Fermat's little theorem. I also remember something about a method he only mentioned that could be run on a computer. How can I program my computer to do this complicated method?

I am working on a prime number program (in Pascal) that uses the sieve method. I saw an answer on your web site about testing for primes. I go further in simplification by only testing with prime numbers less than the square of the numbers and only odd numbers. I want to be able to cross check on its operation by using a check for primes. The "simple method" looks too messy for large numbers as I am trying to push my program to work up to 10 billion.

Dave Ware

Date: 12/18/97 at 13:22:02

From: Doctor Rob

Subject: Re: Follow up question on testing for primes

If you think the "simple method" is too messy for large numbers, then you will really hate the more sophisticated ones. They are too complicated to go into via e-mail, but I will point you to a reference. See

Henri Cohen, *_A Course in Computational Algebraic Number Theory_*, Springer-Verlag, Graduate Texts in Mathematics 138, Chapter 9.

You should be able to find a copy in any medium-to-large university library, and probably other places. You will need a background in either algebraic number theory, or the theory of elliptic curves, to understand why the algorithms work, but only a knowledge of programming to implement them.

Here is another idea. Since your range is only up to 10^{10} (as opposed to 10^{1000} !!!), here is an alternative. Use the Strong Compositeness Test (below) with bases 2, 3, 5, and 7. If it fails all of these, and is not equal to 3215031751, then it is prime.

The Strong Compositeness Test with Base a: To test a number N for compositeness:

1. If $\text{GCD}(a,N) > 1$, stop and declare that N is composite.
2. Write $N - 1 = 2^s \cdot d$, where d is odd.
3. Compute R, the remainder of a^d when divided by N. (Do this as described in the last message.)
4. If $R = 1$, stop and declare failure.
5. For $i = 0, 1, \dots, s-1$, do the following:
 - a. If $R = N-1$, stop and declare failure.
 - b. Replace R with the remainder of R^2 when divided by N.
 - c. If $R = 1$, stop and declare that N is composite.
6. Stop and declare N composite.

A number which fails the Strong Compositeness Test with Base a, yet is composite, is called a "strong pseudoprime with respect to base a." Every strong pseudoprime is a Fermat pseudoprime to the same base, but the reverse is false. The smallest strong pseudoprime with respect to base 2 is 2047.

Theorem: The only number less than $2.5 \cdot 10^{10}$ which is a strong pseudoprime with respect to bases 2, 3, 5, and 7, is 3215031751.

-Doctor Rob, The Math Forum

Check out our web site! <http://mathforum.org/dr.math/>



Appendix

Please maintain this License and Attribution information with any changes you make. Where to get more information about this lab and the presentation that may go with it? Please visit STEAMClown.org or [jim.The.STEAM.Clown's Google Site](http://jim.The.STEAM.Clown's.Google.Site)

Appendix A: License & Attribution

- This interpretation is primarily the Intellectual Property of Jim Burnham, Top STEAM Clown, at STEAMClown.org
- This presentation and content is distributed under the [Creative Commons License CC-BY-NC-SA 4.0](http://creativecommons.org/licenses/by-nc-sa/4.0/)
- My best attempt to properly attribute, or reference any other sources or work I have used are listed in Appendix C

Under the following terms:

-  **Attribution** — You must give [appropriate credit](#), provide a link to the license, and [indicate if changes were made](#). You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
-  **NonCommercial** — You may not use the material for [commercial purposes](#).
-  **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the [same license](#) as the original.

No additional restrictions — You may not apply legal terms or [technological measures](#) that legally restrict others from doing anything the license permits.

Appendix B: Coding License & Attribution

- This interpretation is primarily the Intellectual Property of Jim Burnham, Top STEAM Clown, at STEAMClown.org
- The programming code found in this presentation or linked to on my Github site is distributed under the:
 - [GNU General Public License v3.0](http://www.gnu.org/licenses/gpl-3.0.html)
 - European Union Public Licence [EUPL 1.2 or later](http://www.eu-licence.com/)
- My best attempt to properly attribute, or reference any other sources or work I have used are listed in Appendix C



EUPL



Appendix C: Primary Sources & Attribution for Material Used

-