Numerical methods with JavaScript

Fred Richman
Florida Atlantic University

May 13, 2011

0 Writing programs in JavaScript

You can't do computational mathematics without writing programs. In this course, all of the programs are to be written in JavaScript. If you have a browser on your computer, then you have a JavaScript interpreter, so you don't need anything special to run these programs. JavaScript is a powerful programming language. A lot of its constructs are like those of the programming language C. It is amply documented on the web. For example, you can go to http://www.w3schools.com/js/default.asp. Normally I just Google. If you Google "javascript tutorial" you will find the w3schools site. You can also Google particular constructs, like "javascript loop" or "javascript if then". You can create and modify JavaScript programs with any ASCII editor. I usually use Notepad. I've created a shell which will allow you to start writing and running programs immediately.

You can work with Internet Explorer as your JavaScript interpreter for local files on your computer, but I have had some trouble with it. I've been using Firefox, which hasn't given me any headaches yet. Firefox 4 can be downloaded (free) at

http://www.mozilla.com/en-US/firefox/fx/

Internet Explorer seems to work okay. The difficulty I've had with it is that each time I fire up a JavaScript program, it asks me twice if I really want to do it.

Another browser which runs quite well is Opera. It seems faster than Explorer or Firefox. It can be downloaded (free) at

http://www.opera.com/

Actually, I tend to use Chrome now:

www.google.come/chrome

The home page for this course, MAD 3400, is at

http://math.fau.edu/Richman/Numerical/

If you click on "Program 1" there you will be taken to a page that tells you how to get started on the your first project. Begin by clicking on "RichF0.htm". This should bring up a JavaScript program. Read what it says and click on "Go". A blue screen should come up which, among other things, lists the first 40 powers of 2. You can get it to list more by going back and changing the number 40 in the interface to the JavaScript program before clicking on "Go".

Afterwards, follow the instructions there to download the program to your hard drive, change its name appropriately, and run it locally on your computer.

1 Project 1: the largest integer

The first project is to figure out the largest integer that JavaScript can handle. JavaScript doesn't really distinguish between a number that is an integer and a number that is a real number (a floating point number) like many programming languages do. Very large numbers are held only approximately, as are most fractions. Although we are not particularly interested in working with large integers, we are interested in how many decimal digits in a number we can trust, and this is directly related to how large an integer can be held exactly. We can find this out by generating a large integer n and comparing it with $n + 1$. If JavaScript holds $n + 1$ exactly, then n and $n + 1$ will be different, otherwise they will be the same.

Download the program RichF0.htm at

http://math.fau.edu/Richman/Numerical/programs.htm

Here are some html tags that are used in that program:

 	line break	<sup>	begin superscript
<hr>	horizontal line	</sup>	end superscript

All the output from a JavaScript program is written using html: **h**yper**t**ext **m**arkup **l**anguage. A (constant) string is a sequence of alphanumeric characters enclosed by single quotation marks. For example, `'MAS 2502'` is a string. The command

```
document.write('Hello')
```

prints `Hello`. If you add two strings, the result is their concatenation:

```
'MAD 3400' + 'Hello' = 'MAD 3400Hello'
```

If you add a string to an integer, the result is a string:

```
'Hello' + 5 = 'Hello5'
```

If you add two integers, the result is an integer:

```
7 + 5 = 12
```

So the associative law fails:

```
('Hello' + 5) + 7 = 'Hello57'
'Hello' + (5 + 7) = 'Hello12'
```

What happens if you leave out the parentheses altogether? Try it on

```
'1' + 2 + 3
```

2 Project 2: finding roots

2.1 The smallest prime factor

To start, we want a program that takes a number n and computes the smallest prime factor p of n. This factor is the smallest positive integer p greater than 1 which divides n. So we take the string that represents the number n (that's what gets entered), and apply the built-in JavaScript function parseInt to it, as in the first project. Then you might as well take its absolute value in case, by accident, a negative number was entered. If the number is 0, then I guess its smallest prime factor is 2; what do you think? If the number is 1, then

it has no prime factors. (Nobody considers 1 to be a prime, although one can argue that it is.) Those are the only exceptional cases: The number 0 is exceptional because its smallest prime factor is bigger than it; the number 1 is exceptional because it has no prime factor.

So dispose of the cases $n = 0$ and $n = 1$ before the main computation. Print out something informative, and quit. You are faced with an integer n that is greater than 1; you want to try to divide n successively by $2, 3, 4, 5, \ldots$ until you are successful. You can do that with the function $n \ \% \ m$ which returns the remainder that you get when you divide n by m. I have no idea why they use the percent sign to indicate that function—the same thing is done in the language C. So we want to see when $n \ \% \ m$ is first equal to zero. That condition is written

$$n \ \% \ m == 0$$

with two equal signs (also as in C). One equal sign is an assignment statement: $x = a + b$ says to set the value of x equal to $a + b$. The expression $x == a + b$ doesn't say to do anything—it has the value **true** if x is equal to $a + b$, and the value **false** if x is not equal to $a + b$. An expression that takes on the values **true** or **false** is called a **boolean expression**. Another boolean expression is $x < y$.

2.2 Using an if statement

The typical use of boolean expressions is in an **if** statement. The statement

```
if( n % m == 0 ) out.put('Hello');
```

prints out 'Hello' if m divides n. The first program had the lines

```
if( j < j + 1 ) {
   out.put('looks ok <br>');
} else { out.put('overflow <br>'); }
```

These lines print out 'looks ok', followed by a line break, if $j < j + 1$, and print out 'overflow', followed by a line break, otherwise. How could j not be less than $j + 1$? If j were so large that adding 1 to it had no effect.

2.3 Using a for statement

So we want to run through the integers m, starting at 2, to see if any of them divide n. When we come to the first one that does, we print it out. There is

at least one that does, namely n itself. We can do this as follows:

```
for( m = 2; m < n; m++ ) if( n % m == 0 ) break;
out.put(m + '<br>');
```

The "break" instruction says "get out of this for-loop". So there are two ways to exit this particular for-loop. One is when we run into m with $2 \leq m < n$ such that m divides n. The other is when the for-loop runs its course, and since the condition $m < n$ first fails when $m = n$, that will be the value of m when we exit. In either case, the value of m when we exit the for-loop will be the smallest integer greater than 1 that divides n.

2.4 Speeding the algorithm up

This is a very unsophisticated algorithm. For one thing, it's pointless to see if m divides n when m^2 is greater than n. Why is that? Well, if $m^2 > n$ and m divides n, then $n/m < m$ also divides n so we would have already broken out of the loop when we tested n/m. Once m^2 is greater than n, we already know that n is prime, so we should just print out n and quit. After trying the program out, you will want to rewrite it to eliminate all that superfluous testing. Notice that if n is around 1000000, this means that you will be doing around 1000 tests rather than 1000000 tests, so this improvement is well worth doing. The program, as it stands, is a little too clever about what it does in the second case, so there might be more work in rewriting it than appears.

2.5 Testing the algorithm

Before you rewrite the program, you should test it as it stands to see if it works. Of course you just could try it out by hand on a few numbers, and you should do that. Another thing you could do is print out what it does on the first 100 numbers (for example) and eyeball the results to see if they look okay. You could interpret the input n as telling you to find the smallest prime factors of the numbers from 1 to n, or maybe from 2 to n. This, of course, would be done with another for-loop, which would look like:

```
for( m = 2; m <= n; m++ ) {

}
```

where we have to fill in the lines between the curly brackets. Note the awful symbol m <= n which stands for $m \leq n$. Inside that for-loop we have to put our original for-loop, but there are two things we have to modify. What used to be called n in the original for-loop should now be called m, and we will need another variable, say i, to use instead of the m in the original loop. Moreover, we should not just print out i, we should print out both m and i, so we will get a list of pairs: the number, and its smallest prime factor. Something like

```
out.put(m + '. ' + i + '<br>');
```

which prints out m followed by a period and a space, followed by i and a line break.

2.6 Using a JavaScript function

Better yet, you should define a function that takes a natural number $n > 1$ and returns its least prime factor. Then you can call that function instead of having to worry about nested for-loops. You already have the required line of code, you just have to write it down as a function:

```
function spf(n) {
  for( var m = 2; m < n; m++ ) if( n % m == 0 ) break;
  return m;
}
```

The command "return m" makes m the result of applying the function spf to n. The word "var" before the first occurrence of m assures that this m is isolated to this function program—that is, changing m within the function program will not affect the value of any m's that occur outside the function program. It makes m a *local variable*.

Put in this definition of the function spf just before tess. Now use it to rewrite tess so that tess prints out each number from 2 to n together with the smallest prime that divides it. Your print statement will look something like

```
out.put(m + '. ' + spf(m) + '<br>');
```

2.7 Complete factorization

How do we get a program to tell us what *all* the prime factors of a number are? We want to enter a number n and have the output be a list of the primes

dividing n. Probably we want to allow repetitions on this list, so when we enter the number 72, the list that gets returned is $2, 2, 2, 3, 3$ rather than just $2, 3$. The natural way to handle this problem is via a **recursive program**, one that calls on itself.

Let's denote the program we want to write by $\text{allpf}\,(n)$. Then, roughly, what we want to do is first compute

$$m = \text{spf}\,(n)$$

and then set

$$\text{allpf}\,(n) = m, \text{allpf}\,(n/m)$$

where the comma indicates that we are forming a list whose first element is m and whose remaining elements form the list $\text{allpf}\,(n/m)$. This last equation is the **recursion**: we define allpf in terms of itself. How does it work in practice? If $n = 12$, then $m = \text{spf}\,(12) = 2$, and $n/m = 6$, so we have

$$\text{allpf}\,(12) = 2, \text{allpf}\,(6)$$

Proceeding, we get

$$\text{allpf}\,(6) = 2, \text{allpf}\,(3)$$

so

$$\text{allpf}\,(12) = 2, 2, \text{allpf}\,(3)$$

Finally,

$$\text{allpf}\,(3) = 3, \text{allpf}\,(1)$$

so

$$\text{allpf}\,(12) = 2, 2, 3, \text{allpf}\,(1)$$

Now the "exit strategy" here is that $\text{allpf}\,(1)$ is the empty list, so $\text{allpf}\,(12) = 2, 2, 3$.

This works because of two things. One is that the number n/m that we apply the function allpf to on the right, is always smaller than the number n that we apply allpf to on the left. The other is that if n is small enough (equal to 1), then we know what $\text{allpf}\,(n)$ is independent of the recursion.

So here is the general idea. If $n = 1$, then return the empty list. If $n > 1$, then set $m = \text{spf}\,(n)$, and return the list $m, \text{allpf}\,(n/m)$.

How do we deal with lists in JavaScript? In this case, all we want to do is print out the list at the end, so it will be convenient to use strings. When

the input is the number 12, we want the output to be the string '2 2 3 '. You could of course use commas, but I like spaces better and they have the advantage that you don't notice whether there is a space at the end or not, while you would notice if there were a comma at the end. The empty list is simply ' ' (no space between the single quotes). The recursion

$$\text{allpf}\,(n) = m, \text{allpf}\,(n/m)$$

becomes

$$\mathsf{allpf(n) = m + ' ' + allpf(n/m)}$$

remembering that addition of strings is simply concatenation, and that when you add an integer to a string, you get a string. If you wrote ',' instead of ' ', you would get commas between the primes, and also a comma at the end of the list. How could you get rid of that comma at the end?

2.8 More on recursive programs

The prototype recursive program is one that computes the factorial function, the product of the integers from 1 to n:

$$n! = n\,(n-1)\,(n-2)\cdots 3\cdot 2\cdot 1$$

The key fact is that $n!$ is equal to n times $(n-1)!$, at least if n is positive, as you can see by looking at the product. Thus if we could compute $(n-1)!$ we would know how to compute $n!$. The computation for 4! would go as follows:

$$\begin{aligned} 4! &= 4\cdot 3! \\ 3! &= 3\cdot 2! \\ 2! &= 2\cdot 1! \\ 1! &= 1\cdot 0! \end{aligned}$$

but now we can't go further, we have to know what 0! is. Well, 0! is equal to 1. That's our exit condition. Then we work our way backwards through the equations to find that

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

The program would look like:

```
function fac(n) {
  if(n == 0) return 1;
  return n*fac(n-1);
}
```

What happens when it runs on $n = 4$? When we run fac(4), it tries to return 4*fac(3), so it has to run fac(3). We now have two programs running at once: fac(4) and fac(3), with fac(4) waiting for the result from fac(3). Similarly fac(3) tries to return 3*fac(2), so it has to run fac(2). We now have three programs running at once. And so on. We can indicate this by a table:

program	returns
fac(4)	4*fac(3)
fac(3)	3*fac(2)
fac(2)	2*fac(1)
fac(1)	1*fac(0)
fac(0)	1

At the end we have five fac programs running at once. But fac(0) knows what to do without calling fac any more: it returns 1. Now fac(0) = 1 is passed to the fac(1) program giving the result fac(1) = 1*1 = 1. Then fac(1) = 1 is passed to the fac(2) program giving the result fac(2) = 2*1 = 2. This result is passed to the fac(3) program, yielding fac(3) = 3*2 = 6, and finally this result is passed to the fac(4) program yielding fac(4) = 4*6 = 24.

2.9 Working with arrays

An array is a sequence indexed by the integers $0, 1, 2, \ldots, n-1$. The n elements of the array a are denoted a[0], a[1], a[2], ..., a[n-1]. The number n is the length of the array and is denoted a.length.

A string s can be thought of as a special type of array whose elements are alphanumeric characters. Its length is also denoted s.length, but a string is handled a little bit differently from an array.

The simplest way to create an array is by the statement

```
var a = [];
```

which creates an array with no elements—an empty array. An empty array has length 0. So, if after defining the array a above, you printed out a.length, you should see 0.

To put elements into your array, simply use an assignment statement like

```
a[3] = 17;
```

If you do this, the array will have length 4, because arrays always start at 0. The first three elements of the array, `a[0]`, `a[1]`, `a[2]`, will be `undefined`. This is not a particularly good situation, so you usually want to assign elements consecutively, starting at 0. For example,

```
for( i := 0; i < 10; i++ ) a[i] = i*i;
```

gives an array of length 10 containing the squares of the numbers from 0 to 9.

One handy array operation on an array `a` is `a.push()`. This sticks an element on the end of the array and so increases its length by one. If we took the array `a` of length 10 above and wrote `a.push(35)`, the array `a` would now have length 11 and consist of the number $0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 35$. We can stick an element on the beginning of the array by using the operation with the implausible name `a.unshift()`. Thus if we wrote `a.unshift(22)` instead of `a.push(35)`, the array `a` would become $22, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81$.

Your fourth assignment is to modify your program that factors positive integers into primes so that it uses arrays instead of strings. In addition, you can check your program by multiplying the elements in the array of primes together to see whether you get the original number.

3 Sieving for primes

One way to construct a list of the primes from 1 to 1000 is to test each number from 1 to 1000 to see if it is a prime. A more efficient way is to use the "Sieve of Eratosthenes". This is the same Eratosthenes who, around 240 BC, calculated the circumference of the Earth.

The idea is to list the numbers from 2 to 1000

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 ...

then eliminate all multiples of 2 (except itself) which is the first number

2 3 5 7 9 11 13 15 17 19 21 23 25 ...

then eliminate all multiples of 3 which is the next number remaining

2 3 5 7 11 13 17 19 23 25 ...

then eliminate all multiples of 5 which is the next number remaining

2 3 5 7 11 13 17 19 23 ...

If we continue in this way until we eliminate all multiples of 31, then we are left with exactly the primes from 1 to 1000.

The typical way to implement this in a program is to create a one-dimensional array with 999 entries, indexed by the numbers from 2 to 1000. Fill this array with the boolean value **true**, indicating that, as far as we now know, it is true that each of these numbers is prime. Now we change the entries whose indexes are multiples of 2 to the boolean value **false**, because we know that $4, 6, 8, 10, 12, \ldots$ are not primes. Next we change the entries whose indexes are multiples of 3 to **false**, because we know that $6, 9, 12, 15, \ldots$ are not primes. Notice that 6 and 12 get eliminated twice, but that is harmless. And so on.

In JavaScript, and other languages, the entry in the i-th position of an array A is denoted by $A[i]$. The easiest way to declare an array A is simply to have a line

```
var A = [];
```

in your program. Then you can simply set $A[i]$ equal to **true** with the command $A[i] =$ true. So, to set all the entries in A equal to **true**, you would have a line like

```
for( i = 2; i <= 1000; i++ ) A[i] = true;
```

That's the initial set up. The core of the program eliminates a bunch of numbers (they fall through the sieve), which you do by setting some entries equal to **false**. You will have some prime p, which initially is 2, and you will eliminate $2p, 3p, 4p, \ldots$, that is, you will set $A[ip]$ equal to **false** for $i = 2, 3, 4, \ldots$. That line might look like

```
for( i = 2; i*p <= 1000; i++ ) A[i*p] = false;
```

or

```
for( i = 2*p; i <= 1000; i+=p) A[i] = false;
```

The command "i+=p" increments i by p, just like the command "i++" increments i by 1. Actually, it is not necessary to start at i = 2 in the first loop because we have already eliminated $A[ip]$ for $i < p$. So we can start that loop with i = p (and the second loop with i = p*p).

We start with $p = 2$. If we've just finished with a prime p, how do we get the next prime p? We proceed to the next index i so that $A[i]$ is **true**. The following two lines do that, almost.

```
p++;
while( A[p] == false ) p++;
```

The problem here is that $A[p]$ may be the last place in array that is true. So it is safer to write

```
p++;
while( (A[p] == false) && (p <= 1000) ) p++;
```

although maybe we would be okay if we allowed p to go beyond the bound of the array A. The symbol "&&" means "and". I tend to be overly cautious with parentheses in this situation. The "while" construction here will increment p by 1 as long as $A[p]$ is false and p is less than 1000. So either p hits 1000, or we have reached the next place where $A[p]$ is true.

3.1 Counts

Once we have the array A set up, we can do various counts. The simplest is just to count the number of primes between 1 and n. This is accomplished by counting how many of the entries of A are equal to **true**. The following function does that:

```
function nprimes(n) {
  var count = 0;
  for( var m = 2; m <= n; m++ ) if( A[m] ) count++;
  return count;
}
```

Twin primes are two primes that differ by 2, like 17 and 19, or 71 and 73. While we have known since Euclid that there are an infinite number of primes, no one has succeeded in proving that there are an infinite number of twin primes. We can count the number of (pairs of) twin primes in the same way we counted the number of primes:

```
function ntwins(n) {
  var count = 0;
  for( var m = 2; m+2 <= n; m++ ) if( A[m] && A[m+2] ) count++;
  return count;
}
```

Notice the use of "&&" here. Recall that it is the JavaScript symbol for "and", so the conditional statement count++ is executed exactly when both A[m] and A[m+2] are **true**, that is, when $m, m+2$ is a pair of twin primes. So the gap between consecutive primes p and q is $q-p-1$. The gap between 2 and 3 is zero (there are no nonprimes between 2 and 3) while the gap between 7 and 11 is three (there are three nonprimes, $8, 9, 10$, between 7 and 11).

Our last task is to count the number of gaps of different sizes between consecutive primes from 1 to n. Our result should be an array G such that $G[i]$ is the number of gaps of size i. It will be convenient to focus on the larger of the two consecutive primes. So our main loop will run i from 3 to n. But we have have to compare i with the value of the previous prime, so we will need a variable to take care of that: say lastprime. Since we are starting at $i = 3$, we initially set the value of lastprime equal to 2. So our function will contain the lines:

```
function gaps(n) {
  var lastprime = 2;
  for( var i = 3; i <= n; m++ ) {

  }
  return G;
}
```

Of course we have to set up G as an array at the beginning, and we will only do something in the loop when i is prime. Also, when we find the next prime i, and we do something in the loop, we have to set lastprime equal to i before we go looking for the next prime after i. So our function will contain the lines:

```
function gaps(n) {
 var G = [];
  var lastprime = 2;
  for( var i = 3; i <= n; m++ ) {
```

```
    if(A[i]){

      lastprime = i;
    }
  }
  return G;
}
```

Now what do we want to do with the numbers lastprime and the next prime after it, which will be i? The gap g between those two consecutive primes is i-lastprime-1. We want to increase G[g] by 1. A quirky problem here is that G[g] may not be defined because this may be the first gap of size g that we have encountered. So what we do is test whether G[g] is defined. If it is, we increase its value by 1, if it isn't, we set its value equal to 1. A simple test is to use the conditional if(G[g]). It turns out that G[g] acts like **true** in this context if it is defined, and like **false** if it is not. So our function becomes:

```
function gaps(n) {
 var G = [];
  var lastprime = 2;
  for( var i = 3; i <= n; m++ ) {
    if(A[i]){
      var g = i - lastprime - 1;
      if(G[g]) G[g]++; else G[g] = 1;
      lastprime = i;
    }
  }
  return G;
}
```

One last refinement. The way the program is written now, the returned array G is not defined where it should be 0. To remedy that, we can put the following line just before the return:

```
for( var i = 0; i <= G.length; i++ ) if(!G[i]) G[i] = 0;
```

The JavaScript symbol "!" means "not", so the statement G[i] = 0, which is conditioned on if(!G[i]), is executed exactly when G[i] is **false** or undefined. Thus the effect of this line is to replace all undefined entries of G by zeros, which is what we want.

4 Euclidean algorithm and Euler phi function

The Euclidean algorithm is one of the oldest and best algorithms we have. It computes the greatest common divisor of two positive integers a and b. You can find it in Book VII Proposition 2 of Euclid's Elements. Euclid also showed that any number that divides both a and b divides their greatest common divisor. I think of that fact as being the "algebraic gcd property".

The Euclidean algorithm is most naturally written recursively. Euclid pointed out is that the common divisors of a and b (those numbers that divide both a and b) are the same as the common divisors of a and $b - a$. So if $0 < a \leq b$, we can reduce the problem of finding $\gcd(a, b)$ to that of finding $\gcd(a, b-a)$. The latter is a simpler problem because the numbers are smaller (certainly their sum is smaller). We can refine that observation a bit for our purposes by noticing that we can repeatedly subtract a from b until we get a number that is smaller than b. Actually, Euclid noted that also: he talked about repeatedly subtracting the smaller number from the larger number. If you repeatedly subtract a from b until b becomes smaller than a, you end up with the remainder you get when you divide b by a. (Recall that division of positive integers can be thought of as repeated subtraction.) So using the JavaScript % notation for the remainder, the key equation is

$$\gcd(a, b) = \gcd(b \,\%\, a, a)$$

Here the $b \,\%\, a$ is written first because it is (strictly) smaller than a. Of course $b \,\%\, a$ might be equal to zero, in which case we are done because $\gcd(0, a) = a$. This is our exit condition from the recursion. The usual convention is that $\gcd(0, 0) = 0$ even though, strictly speaking, there is no greatest common divisor of 0 and 0. However, zero is obviously the algebraic gcd of 0 and 0 because any number that divides both 0 and 0 divides zero (and zero is the only number with that property).

Rather than making your algorithm foolproof, at least at first, just make it work when it is handed integers a and b with $0 \leq a \leq b$. Use the exit condition

$$\gcd(0, b) = b$$

and the recursion displayed above. Test your algorithm on a few small pairs a and b where you know what the answer is.

Two positive integers a and b are **relatively prime** if they have no common divisors except for 1. Notice that 1 is relatively prime to any positive

integer. It's easy to see that two numbers a and b are relatively prime exactly when $\gcd(a, b) = 1$, so the Euclidean algorithm provides a test for when a and b are relatively prime. The Euler φ function is defined, for $n > 1$, by setting $\varphi(n)$ equal to the number of positive integers less than n that are relatively prime to n. Thus $\varphi(10) = 4$ because the numbers less than 10 that are relatively prime to 10 are 1, 3, 7, and 9. Usually people set $\varphi(0) = \varphi(1) = 1$, but we really don't care about those values, especially $\varphi(0)$. We can justify $\varphi(1) = 1$ by modifying the definition of $\varphi(n)$ to read "the number of positive integers *less than or equal to* n that are relatively prime to n." This, in fact, is the usual definition.

Once you get your gcd function working, you can use it to compute the φ function. For each m just run through the numbers $1, 2, \ldots, m$, checking each one to see if it is relatively prime to m, and counting how many times that happens. After writing such a function, test it by printing out, in two columns, the values m and $\varphi(m)$ for $m = 2, \ldots, n$.

As a final check on your φ function, you can compute it in a different way. The formula is

$$\varphi(n) = n \prod_{p \in P} \left(1 - \frac{1}{p}\right)$$

where P is the set of primes that divide n. So

$$\varphi(10) = 10 \left(1 - \frac{1}{2}\right) \left(1 - \frac{1}{5}\right)$$

and

$$\varphi(12) = 12 \left(1 - \frac{1}{2}\right) \left(1 - \frac{1}{3}\right)$$

If you write it that way, you will be dealing with real numbers, at least as intermediate values. In order to deal only with integers, you can rewrite that formula as

$$n \prod_{p \in P} \left(1 - \frac{1}{p}\right) = n \prod_{p \in P} \left(\frac{p-1}{p}\right) = \frac{n}{\prod_{p \in P} p} \prod_{p \in P} (p - 1)$$

So for $n = 10$, the set P is $\{2, 5\}$ and the product $\prod_{p \in P} p$ is 10. The product $\prod_{p \in P} (p - 1)$ is $1 \cdot 4 = 4$ so

$$\varphi(10) = \frac{10}{10} 4 = 4$$

For $n = 12$, the set P is $\{2, 3\}$ and the product $\prod_{p \in P} p$ is 6. The product $\prod_{p \in P} (p - 1)$ is $1 \cdot 2 = 2$ so

$$\varphi(12) = \frac{12}{6} 2 = 4$$

You can use allpf(n) to find the set P. What you need to do is to eliminate the duplicates from the array that allpf(n) returns. Or you could modify the function allpf(n) so that it doesn't return any duplicates. That's probably the more elegant solution. When you find the smallest prime dividing n, keep dividing n by that prime until you can't anymore, then go on to find the smallest prime dividing what's left. You should probably give the modified algorithm a different name.

Once you get an array A whose entries are the set P with no duplicates, you can easily compute the two required products $\prod_{p \in P} p$ and $\prod_{p \in P} (p - 1)$. Now you can get a third column to your output consisting of the values of $\varphi(n)$ computed by this formula, and you can see if it agrees with the values of $\varphi(n)$ that you got by counting.

5 Extended Euclidean algorithm

The extended Euclidean algorithm is a refinement of the Euclidean algorithm that finds integers s and t so that $sa + tb$ divides both a and b. If $sa + tb$ is positive, which we can always arrange, it follows that $sa + tb$ is the greatest common divisor of a and b. The equation

$$sa + tb = \gcd(a, b)$$

is known as **Bezout's equation**.

An interesting aspect of Bezout's equation is that if the left-hand side, $sa + tb$, divides both a and b, then $sa + tb$ must be the greatest common divisor of a and b. Indeed, if d is any common divisor of a and b, then d must divide $sa + tb$ (by the distributive law) so, if everything is positive, d must be smaller (or equal to) $sa + tb$.

We will assume that $0 \leq a \leq b$. If that's not the case, we can always replace a and b by their absolute values, and interchange them if necessary. The simplest, if not the most understandable, algorithm is a recursive one. The "exit strategy" is that if $a = 0$, then we will choose $s = 0$ and $t = 1$ because $\gcd(0, b) = b = 0 \cdot 0 + 1 \cdot b$. (Is $\gcd(0, 0) = 0$?)

The idea behind the Euclidean algorithm is that if $b = qa + r$, where $0 \leq r < a$, then $\gcd(a, b) = \gcd(r, a)$. Thus we can reduce the computation of $\gcd(a, b)$ to the computation of $\gcd(r, a)$, and repeat. Note that in JavaScript, or in C, the number r is equal to `b%a`. Because $r + a < a + b$, these reductions eventually end with $\gcd(0, b) = b$.

The same thing happens with the extended Euclidean algorithm. Suppose again that $b = qa + r$ with $0 \leq r < a$. If

$$s'r + t'a = \gcd(r, a)$$

then, substituting $r = b - qa$ into this equation gives

$$s'(b - qa) + t'a = \gcd(r, a)$$

so

$$(t' - s'q)\, a + s'b = \gcd(r, a) = \gcd(a, b)$$

Thus taking $s = t' - s'q$ and $t = s'$ solves the Bezout equation. That gives us the following (informal, not JavaScript) algorithm that accepts two integers $0 \leq a \leq b$ and returns a pair of integers s and t such that $sa + tb = \gcd(a, b)$:

bez (a, b)
If $a = 0$ return $(0, 1)$
Set $r = b \bmod a$
Set $q = (b - r)/a$
Set $(s, t) = \text{bez}(r, a)$
return $(t - sq, s)$

Of course you have to write that in JavaScript. A pair is implemented as an array of length 2 so that you can return the pair at one go. So the function `bez` takes two integers, a and b, and returns an integer array of length 2. How do you return the pair $(0, 1)$? You simply return the array `[0,1]`. For the final return, you return `[t - s*q, s]`. What are `s` and `t`? If you set a variable `y` equal to bez (r, a), then `s` is `y[0]` and `t` is `y[1]`.

Watch out for the "If $a = 0$" because "$a = 0$" in JavaScript, as in C, sets the variable a equal to zero, it does not test for whether a is equal to zero. That's done with "$a == 0$". Everybody makes that mistake, more than once.

How do you test your function `bez` once you write it? Run through a bunch of numbers a and b, and print them out together with the corresponding s and t, and possibly also $sa + tb$. Maybe run a from 5 to n and b from a to n for modest values of n. See if the answers look correct.

You can automate the checking also. Compute $sa + tb$ and compare it with the greatest common divisor of a and b that is computed by looking for the common divisors of a and b and recording the biggest one. Then you can print out both $sa + tb$ and the greatest common divisor, in addition to a, b, s, t, and compare them visually.

6 Orders of units modulo n

If $\gcd(a, b) = 1$, we say that a and b are **relatively prime**. Now fix a number n, say $n = 15$. The numbers that are less than n and relatively prime to n are 1, 2, 4, 7, 8, 11, 13, and 14. Those are called the **units** modulo 15. If we multiply two of them, and reduce modulo n, we get another. The multiplication table for $n = 15$ is

$\times$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

If the number a is relatively prime to n, then the **order** of a modulo n is the least exponent $m \geq 1$ such that $a^m \equiv 1 \pmod{n}$. So the order of 2 modulo 15 is 4 as you can see by the sequence of powers of 2

$$2, 4, 8, 1$$

The order of 2 modulo 19 is 18 as you can see by the sequence of powers of 2

$$2, 4, 8, 16, 13, 7, 14, 9, 18, 17, 15, 11, 3, 6, 12, 5, 10, 1$$

What is the order of 2 modulo 11?

The number of units modulo n is denoted by $\varphi(n)$. This φ is the **Euler phi function**. Its first few values are

n	2	3	4	5	6	7	8	9	10
$\varphi(n)$	1	2	2	4	2	6	4	6	4

Verify this table.

7 The sum of the squares of the digits of a number

We are interested in the function f that takes a number to the sum of the squares of its digits. So

$$f(340779) = 3^2 + 4^2 + 0^2 + 7^2 + 7^2 + 9^2 = 9 + 16 + 0 + 49 + 49 + 81 = 204$$

After we figure out how to compute this function, we will want to iterate it and see what happens. That is, after computing $f(340779) = 204$, we will compute $f(204) = 20$, then $f(20) = 4$, then $f(4) = 16$ and so on. The sequence we get starting with 340779 is

$$340779,\ 204,\ 20,\ 4,\ 16,\ 37,\ 58,\ 89,\ 145,\ 42,\ 20$$

and now, of course, the sequence starts repeating. We have found a **periodic point**, 20, of the function f. That is,

$$f^8(20) = 20$$

where f^8 indicates the function f applied 8 times, that is

$$f^8(m) = f(f(f(f(f(f(f(f(m))))))))$$

We call f^8 the 8-th **iterate** of f. We want to investigate the sequence $m,\ f(m),\ f^2(m),\ f^3(m),\ \ldots$ for various values of m. We just did it for $m = 340779$. Notice that not only is 20 a periodic point of f, but so are 4, 16, 37, 58, 89, 145, and 42.

If we want to use a program to study these sequences, we will have to write a function that computes $f(m)$. How do we get hold of the digits of m so that we can square them and add up the results? Consider $m = 340779$. The easiest digit of m to get hold of is the last one, 9, because $9 = m\%10$. To get at the next-to-the-last digit, we arrange for it to be the last digit of another number. To do this, we subtract 9, the last digit, from m to get 340770. Now, dividing by 10, we get 34077, and the next digit we want is the last digit of this number. We continue this computation as in the following

table

m	$m\%10$	$m - m\%10$	$\dfrac{m - m\%10}{10}$
340779	9	340770	34077
34077	7	34070	3407
3407	7	3400	340
340	0	340	34
34	4	30	3
3	3	0	0

Notice that we generate the digits in reverse order, the last one first. This makes no difference for the computation of f because we are just going to square the digits and add up the results, and it doesn't matter in which order we add.

There is a recursion for f, which you can pretty much see from our computations, namely

$$f(m) = (m\%10)^2 + f\left(\frac{m - m\%10}{10}\right)$$

This tells us what to do with any nonzero m, reducing the computation of $f(m)$ to computing f of a number smaller than m. What about $f(0)$? You should now be able to write a recursive JavaScript function that computes f. Write it and test it out on the numbers from 0 to n where, say, $n = 100$.

But, of course, what we want to see are not the sequences $f(m)$, where m goes from 0 to n, but the sequences $f^k(m)$ where m is fixed and k goes from 1 to n. This will require two input boxes, one for m and one for n.

There are two numbers we can change in the definition of $f(m)$, the sum of the squares of the digits of m. One is the number 2 that we use because we are summing the **squares** of the digits. We could just as well sum the cubes of the digits, or the fourth powers of the digits. We can call that number e, for exponent. For the original f, we have $e = 2$. It's easy to change the recursion for $f(m)$ so that it now computes the sum of the cubes of the digits of m. How do you do that?

The other number we can change is the number 10. That is the base b for our system of representing numbers by strings of digits. If we change the number $b = 10$ to, say, $b = 8$, then the digits we get are the digits in the base-8 representation of m, that is, we are thinking of writing m in **octal**. The octal representation of the number 340779 is 1231453, or 1231453_8 if we

want to indicate that the number is written in base 8. So the sum of the squares of its digits is

$$1 + 4 + 9 + 1 + 16 + 25 + 9 = 65 = 101_8$$

and if we continue summing the squares of the digits we get 2, 4, $16 = 20_8$, 4 and we have run into a periodic point, 4, with period 2. You can easily change your program for computing $f(m)$ to one that computes $f(m, e, b)$, the sum of the e-th powers of the digits of m, where m is written in base b. Your first function $f(m)$ was $f(m, 2, 10)$.

8 Fermat's theorem

Fermat's theorem says that if n is a prime, and a is an integer not divisible by n, then

$$a^{n-1} \equiv 1 \pmod{n}$$

This is sometimes referred to as "Fermat's little theorem" as opposed to "Fermat's great theorem" better known as "Fermat's last theorem" because it was the last of Fermat's theorems to be proved (not by Fermat, as far as we know). Fermat's last theorem says that if $n > 2$, then there are no nonzero integer solutions x, y, z to the equation $x^n + y^n = z^n$. Fermat's last theorem is very difficult to prove, and remained unproved for over three hundred years after Fermat stated it. Fermat's little theorem is fairly easy to prove and you can use Google to find lots of proofs of it on the web.

Our first project with Fermat's theorem will be to check it on lots of examples. So we want to take some primes n, and some numbers a with $1 \leq a < n$, and see if $a^{n-1} \equiv 1 \pmod{n}$. Of course this equation holds for $a = 1$, so we need only look at numbers a such that $1 < a < n$.

Just checking the equation on primes n is a little boring. It is also a little dangerous because the output of the program will just be, "yes, it checks", so we won't have any evidence as to whether the program is working right or not. More interesting would to check the equation not only on primes n but also on composites. In fact, we will be more interested in what happens with composites than with primes because we are going to use Fermat's theorem to test whether or not n is a prime. The test will be: choose a number a from $\{2, 3, \ldots, n-1\}$ and compute a^{n-1} modulo n. If the result is 1, then n might be a prime, or it might not. But if the result is not 1, then n is definitely not a prime.

So our first project will be to run through the integers n from 2 to 100, and for each n run through the integers a from 2 to $n-1$ to see whether or not $a^{n-1} \equiv 1 \pmod{n}$. What do we want the output to be? For each n, we will count the number of integers a from 2 to $n-1$ such that a^{n-1} is *not* equal to 1 modulo n. So if n is prime, this count should be zero according to Fermat's theorem. We want to print out the number n together with the count. A check on the program will be to see if the numbers n where we get a count of zero are exactly the primes. The first few lines of output should be:

2. 0
3. 0
4. 2
5. 0
6. 4
7. 0
8. 6
9. 6

8.1 Raising to powers

For small numbers it won't matter, but to handle larger numbers we really need an efficient method for raising a number to a power. The straightforward way to compute a^{n-1} is to compute a^2, then a^3, then a^4 and so on until we get to a^{n-1}. This will take $n-2$ multiplications. We can view this process as a recursion:

$$\begin{aligned} a^1 &= a \\ a^i &= a\left(a^{i-1}\right) \text{ if } i > 1 \end{aligned}$$

A quicker way to compute a^8 is to compute a^2, then square that to get a^4, then square that to get a^8. That takes 3 multiplications instead of the 7 multiplications used by the recursion listed above. Squaring allows us to get to our goal quicker. If we wanted to compute a^9, we would compute a^8 using 3 multiplications, and then multiply the result by a giving us a^9 using only 4 multiplications rather than 8. This idea can also be put into the form of a

recursion:

$$\begin{aligned} a^1 &= a \\ a^i &= \begin{cases} a\left(a^{i-1}\right) & \text{if } i > 1 \text{ is odd} \\ \left(a^{i/2}\right)^2 & \text{if } i > 1 \text{ is even} \end{cases} \end{aligned}$$

So to compute a^{14} by this method we would write

$$a^{14} = \left(a^7\right)^2 = \left(a\left(a^6\right)\right)^2 = \left(a\left(\left(a^3\right)^2\right)\right)^2 = \left(a\left(\left(a\left(a^2\right)\right)^2\right)\right)^2$$

which takes 5 multiplications—square, multiply by a, square, multiply by a, square—rather than 13 multiplications.

Of course since JavaScript allows recursive programs, we don't have to write things out like that, we just use the recursion itself. Note for i to be odd means that the boolean expression $1 == i\%2$ is true. Be sure to reduce modulo n each time. We don't want to try to compute the number a^{n-1} itself, because that will generally be an integer that is way bigger than anything JavaScript can handle. Because we will have to multipy or square before reducing modulo n, we will not be able to handle values of n bigger than about 9 digits because the products before reducing modulo n would have more than 18 digits.

A **Carmichael number** is a number n so that $a^{n-1} \equiv 1 \pmod{n}$ whenever $\gcd(a, n) = 1$. So a Carmichael number will pass the Fermat test unless you happen to choose a number a that has a common factor with n. If that were easy to do, then you could just choose such a number a, compute $\gcd(a, n)$, which is a quick computation, and you would have a factor of n. Carmichael numbers are sometimes called *pseudoprimes*. The first few Carmichael numbers are 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341. Two bigger ones are 294409 and 56052361.

9 The Miller-Rabin test

The most commonly used probabilistic primality test is the Miller-Rabin test. This test, which is a variant of Fermat's test, cannot be fooled by Carmichael numbers. If the numbers a that we use to test are chosen at random, then the probability that a composite n will pass the test is less than $1/2$. So if we test 20 times in succession, choosing the numbers a independently, then

the probability of a composite passing these 20 tests is less than $1/2^{20}$, that is, less than 1 in a million.

Here is the test. You are given a number n that wish to test for primality. Write $n-1 = 2^s d$ where d is odd. This is always possible: you just keep dividing $n-1$ by 2 until you get an odd number. For the test, you choose a number a at random so that $0 < a < n$. Then you compute $x = a^d \pmod{n}$. The test is to consider, modulo n, the sequence

$$x, x^2, x^4, x^8, \ldots, x^{2^s}$$

There are $s+1$ terms in this sequence (the exponents of x are $2^0, 2^1, 2^2, 2^3, \ldots, 2^s$), and the last is $x^{2^s} = a^{d2^s} = a^{n-1}$. Of course the last term should be 1—that's Fermat's test.

The key observation for the test is that if $z^2 = 1$ modulo n, and n is a prime, then $z = \pm 1$ modulo n. That's because if $z^2 - 1 = (z-1)(z+1)$ is divisible by a prime n, then n has to divide either $z-1$ or $z+1$.

Now each term in our sequence is the square of the previous term. So if a term is 1, and the last term must be 1 or we would reject the hypothesis that n was a prime, then the previous term must be ± 1. If that doesn't happen, we reject the hypothesis that n is a prime. So the good sequences look like

$$1, 1, 1, 1, 1, 1, 1, 1$$
$$-1, 1, 1, 1, 1, 1, 1$$
$$*, *, *, -1, 1, 1, 1$$

where $*$ is a number different from ± 1. The bad sequences look like

$$*, *, *, 1, 1, 1, 1$$
$$*, *, *, *, *, *, -1$$
$$*, *, *, *, *, *, *$$

If we denote the sequence by $t_0, t_1, t_2, \ldots, t_s$, then for the sequence to be good we must have $t_s = 1$ and there must not be an index i such that $t_i \neq \pm 1$ and $t_{i+1} = 1$. I don't think you need an array to perform this test. Just compute x, keep squaring it, looking for the forbidden $z \neq \pm 1$ and $z^2 = 1$, and make sure that $x^{2^s} = 1$. While developing this program you might want to print out the sequence $t_0, t_1, t_2, \ldots, t_s$.

10 Ciphers

A standard method for enciphering a piece of text is to replace the letters by other letters according to a fixed scheme. One famous example is attributed to Julius Caesar. The idea was to replace the letter a by the letter d, the letter b by the letter e, the letter c by the letter f, and so on until the letter z is replaced by the letter c. The scheme can be summarized in the following table

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c

where the letters in the second row are substituted for the letters in the first row. So the word "caesar" would become "fdhvu". If we think of the letters of the alphabet as being the numbers $0, 1, 2, \ldots, 25$, this amounts to replacing the number n by the number $n + 3 \pmod{26}$.

More briefly, we can think of the plain alphabet as being the string abcdefghijklmnopqrstuvwxyz and what we need to do is to specify a cipher alphabet which is some permutation of these 26 letters. In the Caesar cipher case, we choose the cipher alphabet to be the string defghijklmnopqrstuvwxyzabc.

Suppose p is the plain alphabet and c is the cipher alphabet. How do we encrypt a message t? We want to form a new string r which is the encrypted message. The i-th character in the message t is $t\,[i]$, although Internet Explorer doesn't seem to recognize that. If that's your browser, you will need to use `t.charAt(i)` instead of $t\,[i]$.

We need to know the position j of $t\,[i]$ in the plain alphabet p. (Remember that positions start at 0.) Then we can set $r\,[i]$ equal to $c\,[j]$. There is a JavaScript function associated with the string p that does just that. It is called `p.indexOf`. If we write `p.indexOf('u')`, we will get the position of the first occurrence of the letter `u` in the string p. If the letter `u` does not appear in the string p, then we get the number -1. So we want to set `r[i] = c[j]` where `j = p.indexOf(t[i])`. To decipher a message, we interchange the roles of p and c.

There is a quick way to generate a cipher alphabet from a word, called a **keyword**. It's much easier to remember a keyword than to remember a whole cipher alphabet. Here is how you generate a cipher alphabet from the phrase "boca raton". First you eliminate the space and all the duplicate letters, so you are left with "bocartn". Then you form the following table, with the letters bocartn at the top, and the rest of the alphabet written in

rows underneath:

b	o	c	a	r	t	n
d	e	f	g	h	i	j
k	l	m	p	q	s	u
v	w	x	y	z		

Finally, you read off the cipher alphabet by going down each column:

bdkvoelwcfmxagpyrhqztisnju

What happens when you take the keyword to be the one-letter word "a"? The one-letter word "b"? The one-letter word "z"?

Can you develop a formula, like $n + 3 \pmod{26}$ for the Caesar cipher, for how to encipher using the two-letter keyword "ab"?

11 Finding orbits for sums of e-th powers

We are interested in the function f_e that takes a number to the sum of the e-th powers its digits. We examined f_2 in a previous section. The function f_e maps the positive integers into the positive integers. That gives us a *discrete dynamical system*. One thing we are interested in is *fixed points*, number m such that $f_e(m) = m$. The fixed points of f_3 are 1, 153, 370, 371, and 407. We are also interested in *periodic points*. These come in groups called *orbits*. An orbit is a finite set S of positive integers $\{s_1, s_2, \ldots, s_k\}$ so that $f_e(s_i) = s_{i+1}$, for $i < k$, and $f_e(s_k) = s_1$.

It turns out that if we choose any positive integer whatsoever and keep applying f_e to it, then eventually we end up in an orbit.

12 Possible topics

1. Graph theory
   (a) Maximal cliques
   (b) Maximum degree
   (c) Number of independent cycles modulo 2
   (d) Number of components
   (e) Planar

(f) Chromatic number

2. Number theory

   (a) Order of 2 in U_n for n odd. Large n.

   (b) Euclidean algorithm—extended

   (c) Orders of elements in U_p for p a prime. Maximal order of element of U_n. The function $\varphi(n)$.

   (d) Primality testing. Naive test. Sieve. Count the number of primes. Probabilistic tests. Efficient powers.

   (e) Quadratic residues—square roots

   (f) Carmichael numbers

   (g) Perfect numbers

   (h) Sums of two squares

   (i) Emirps—reverse a prime in various bases to see if it's a (different) prime

3. Permutations: S_n

   (a) Representations—back and forth

   (b) Number of subgroups

   (c) Number of elements of each order

   (d) Number of normal subgroups

   (e) Products

4. Text

   (a) Frequency count

   (b) Encode and decode simple substitutions

   (c) Encode and decode Playfair