The Sieve of Eratosthenes

Generating Lists of Prime Numbers
Basic Algorithm
Automating the Algorithm

Reading: SoC Chapter 3

Prime numbers have a long and interesting history
- a number is prime if it cannot be written as the product of two smaller numbers
- non-prime numbers are called composite numbers
- these are primes: 5, 11, 73, 9967, ...
- these are composite: 10 (2 × 5), 99 (3 × 3 × 11), ...

Ancient Greek, Persian, and Chinese philosophers all studied properties of prime numbers
- the Sieve of Eratosthenes, the algorithm we’re looking at today, is one of the oldest known algorithms
- dates to at least 200 BC

See [http://primes.utm.edu](http://primes.utm.edu)

History

Motivation

- Until the middle of the 20th century prime numbers were of interest only to number theorists
- In 1970 a new method for encrypting messages was invented
  - the security of the RSA algorithm relies on the difficulty of finding the factors of large numbers
  - here “large” means numbers with 500 digits or more
  - RSA has renewed interest in properties of prime numbers and methods for generating them

Big Number

- In Aug 2008 a project named GIMPS (Great Internet Mersenne Prime Search) set a new record
  - the largest known prime number is now \(2^{43,112,609} - 1\)
  - around 13 million digits
  - the group that found it claimed a $100,000 prize....

- At 10 digits/inch:
  - 12” thick book, or
  - a string 20 miles long

See "The Dinette Set" by Julie Larson

UCLA
The Sieve

- The basic idea is simple:
  - make a list of numbers, starting with 2
  - repeat:
    - the first number in the list is prime
    - cross off multiples of the most recent prime

See “Sieve of Eratosthenes” at Wikipedia

Technology

- The method described on the previous slide works well for short lists
- But what if you want to find prime numbers between 2 and 100? 1000?
  - it’s a tedious process to write out a list of 100 numbers
  - takes a lot of paper to make a list of 1000 numbers
  - chances are you will make a few arithmetic mistakes (this is a boring job)
- You could improve accuracy by using an abacus or calculator
  - but it’s still very boring....

Technology (cont’d)

- Can we turn this method into an algorithm?
  - detailed specification of starting and ending conditions are there
- What about the steps?
  - “cross off” and “next number” need to be defined if we’re going to use Ruby
  - when do we stop the iteration?

Isn’t it “obvious” there are no multiples of 11 in this list?

The Sieve with Ruby

- Eventually we’re going to see a complete Ruby program that implements the Sieve of Eratosthenes
- As a first step, though, let’s just use IRB to replace the paper-and-pencil technology
  - use Ruby array objects to represent lists of integers
  - use an iterator to remove multiples of a recently discovered prime
  - \[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\]
- We will still be “in control” and making the main decisions:
  - when to “circle” a number as prime
  - when to stop iterating
Lists of Numbers

- From the previous set of slides:
  - an array in Ruby is a container that holds references to other objects
  - we can make an array of integers by enclosing a list in brackets
    >>> a = [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
- Or, if we want to use an iterator (we'll need this for longer lists):
  >>> a = []
  => []
  >>> 19.times { |n| a << n+2 }
  => 19
  >>> a
  => [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]


Divisibility

- An important question: how do we determine whether or not to remove a number from the list?
- Useful fact: if a number $x$ is the product of two numbers $n$ and $m$ then
  - the remainder of $x \div m$ is 0
  - the remainder of $x \div n$ is also 0
- Example:
  - 77 is not prime because $77 = 7 \times 11$
  - $77 \div 7 = 11 \, R\, 0$
  - $77 \div 11 = 7 \, R\, 0$
  - So the way to implement the step that deletes multiples:
    - remove any number $x$ if the remainder of $x \div p$ is 0 (where $p$ is the most recent prime)

Modulo Arithmetic

- Ruby has a very useful arithmetic operator for this step:
  $x \% y$ is the remainder after dividing $x$ by $y$
- Examples:
  >>> 10 % 3
  => 1
  >>> 77 % 10
  => 7
  >>> 77 % 11
  => 0
  >>> 77 % 7
  => 0
- This new operator is called the “mod” operator
  - based on the “modulo” function of number theory
  - Mathematicians usually write the name of the operator instead of a symbol
  - 77 mod 10 = 7
  - 77 mod 11 = 0
  - Modulo arithmetic is often introduced as “clock arithmetic”
  - if it’s 8 o’clock now, what time will it be 30 hours from now?
  - $8 + 30 \, \text{mod} \, 12 = 2$
  - in Ruby:
    >>> (8 + 30) % 12
    => 2
Even Numbers

- Can you think of an expression in Ruby that will tell us if a number \( n \) is even?
  - hint: a number is even if the remainder of dividing by 2 is 0

\[ n \% 2 == 0 \]

Note!
To test for equality use ==
A single = is used for assignment

- Some experiments:
  - \( 8 \% 2 \)
    - \( \Rightarrow 0 \)
  - \( 8 \% 2 == 0 \)
    - \( \Rightarrow true \)
  - \( 9 \% 2 == 0 \)
    - \( \Rightarrow false \)

Even Numbers in a List

- The slides on arrays also introduced the idea of an iterator
  - a method that “visits” every item in an array

- Let’s use the each iterator and the expression on the previous slide to tell us whether a number in the array is even

\[ \text{true} \]
\[ \text{false} \]

A New Iterator: **delete_if**

- A very useful iterator for this project is named **delete_if**
  - like each, it visits each item in the array
  - the expression in the block is expected to evaluate to true or false
  - if it’s true, the current item is deleted

- This expression deletes every even number from our list:

\[ \text{evaluates to true or false} \]

- Note the similarities with the call to each on the previous slide:

\[ \text{prints true or false} \]

**both iterate over the entire array**

**both operate by placing an array element in \( n \), one at a time**

Outline

- We now have all the pieces we need to make a list of prime numbers
  - we’re going to replace the paper list with Ruby arrays
  - we’ll have Ruby “cross off” multiples by using the **delete_if** iterator to remove them from a list

- The plan:
  - use two arrays
  - **worklist** will hold the list of numbers to sift
  - **primes** will hold the list of known primes
  - repeat until we’re done:
    - copy the first item from worklist to primes
    - remove all multiples of the most recent prime from worklist

(see the picture on the next slide)
Outline (cont’d)

start  worklist:  [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
       primes:  [ ]

copy  worklist:  [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
       primes:  [2]
delete worklist:  [3,5,7,9,11,13,15,17,19]
       primes:  [2]
copy  worklist:  [3,5,7,9,11,13,15,17,19]
       primes:  [2,3]
delete worklist:  [5,7,11,13,17,19]
       primes:  [2,3]

Sieve in IRB

Making the arrays is simple:

>> worklist = []               Use the expressions shown earlier to
>> => []                      make a list of numbers from 2 to 20
>> 19.times { |n| worklist << n+2 }  Always a good idea to make sure it’s
>> => 19                      what we expected
>> worklist
>> => [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
>> primes = []               A second array for the prime numbers
>> => []

Sieve in IRB (cont’d)

To copy the first item in worklist to the end of primes:

>> primes << worklist.first
== [2]

To delete multiples of the most recent prime:

>> worklist.delete_if { |n| n % primes.last == 0 }
=> [3,5,7,9,11,13,15,17,19]

Almost the same expression we used
for remove even numbers

Returns true for multiples of the item
at the end of the primes list

Sieve in IRB (cont’d)

To repeat the previous two steps either

- cut and paste the expressions in your terminal emulator window, or
- if your emulator has “command history” use it to re-execute the commands

On OS/X: type the `!` key twice, then return

>> primes << worklist.first               Note these two lines are identical to the
>> => [2,3]                                expressions typed before....
>> worklist.delete_if { |n| n % primes.last == 0 }
=> [5,7,11,13,17,19]

Live Demo
Sieve in IRB (cont’d)

- When you decide you don’t need to do any more filtering you will have lists that look something like this:
  ```
  >> primes 
  => [2, 3, 5, 7, 11] 
  >> worklist 
  => [13, 17, 19] 
  ```
- The prime numbers are spread across both lists
  - primes has the values you know are prime -- they have passed the filter
  - worklist has the remaining primes
- You can combine them using this expression, which uses a string operator:
  ```
  >> primes += worklist 
  => [2, 3, 5, 7, 11, 13, 17, 19] 
  ```

A sieve Method

- This little “computational experiment” has given us some insight into how the Sieve of Eratosthenes works
- To make an algorithm based on this process we need to solve two more problems
  1. How do we automate the execution of the steps?
     - we want the computer to take control and repeat the copy and delete steps
  2. How do we know when to stop?
     - can we formalize the intuition that “obviously” no primes are left in worklist?
- Once we solve these problems we can make a method named `sieve` that will generate the list for us
  ```
  >> sieve(50) 
  => [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47] 
  ```

sieve 1.0

- A simple way to get Ruby to automate the steps is to simply iterate until the working list is empty
- Ruby will do too much work, but it’s worth implementing this simple version
  - it’s a good demonstration for the type of Ruby statement that implements repetition
  - it will help us “debug” the method
  - when we are satisfied this simpler version works the way we want it to we can change it to use a more sophisticated stopping condition
- This strategy is widely used in the field of software engineering
  - several variations, with names such as “incremental development”, “continual testing”, and “extreme programming”

Another Form of Iteration

- The last set of slides introduced two forms of iteration
  - iterating over the items in a container
  - repeating the evaluation of an expression a fixed number of times
- A generalization of the second form of iteration is to repeat the evaluation until a specified condition is reached
- Example: make an array with 10 random numbers
  ```ruby
  while a.length < 10 
  a << rand(10) 
  end 
  ```
Another Form of Iteration (cont’d)

- This shows what happened when I used this statement in an IRB session:
  ```ruby
  >> include Math
  >> a = []
  => []
  >> while(a.length < 10
  a << rand(10)
  end
  >> a
  => [8, 7, 8, 9, 1, 2, 5, 8, 5]
  ```

- I split the statement into three lines (the way it's usually entered in a file)

- A call to `rand(10)` returns a random integer between 0 and 9

- After the 10th number is appended to `a` the call to `a.length` returns 10 and the expression is false

while Loops

- Another name for this type of statement in Ruby is “while loop”
  - an old name, dating to the early days of programming

- You’ve probably heard the phrase “infinite loop”
  - it’s (usually) a bug in a program
  - it occurs when the controlling expression never becomes false

- When we write a while expression we have to make sure something inside the loop changes a value in the expression
  - e.g. make sure we add something to the array so the length eventually reaches 10

Extra Credit

- The code at right is from the Zune media player
- On Jan 1 2009 people started reporting their machines had frozen
- The reason: a bug in this code led to an infinite loop
- Can you explain why?
- What happens when the variable named `days` has the value 366 and the call to `IsLeapYear` returns true?

```ruby
while (days > 365)
  { if (IsLeapYear(year))
      { if (days > 366)
          { days -= 366; year += 1; }
      }
      else
      { days -= 365; year += 1; }
  }
end
```

This means “subtract 366 from `days` and add 1 to `year`”

sieve 1.0 (cont’d)

- To get back to the sieve algorithm, here is the code for the first version of the method:

```ruby
def sieve(n)
  worklist = []
  (n-1).times { |i| worklist << i+2 }
  primes = []
  while(worklist.length > 0)
    primes << worklist.first
    worklist.delete_if { |x| x % primes.last == 0 }
  end
  return primes
end
```

The while loop tells Ruby to keep iterating until there are no more numbers in `worklist`

The body of the loop and the expressions before the loop are identical to the expressions we typed in IRB

(except for this, which makes a list of the specified size)
Aside: Monitoring Progress

- A common technique used in software development is to print a trace as a program runs.
  - Add print statements to the code as the program runs these statements print strings on the terminal.
  - Example: keep track of primes and worklist in the sieve method.

```ruby
while worklist.length > 0
  primes << worklist.first
  worklist.delete_if { |x| x % primes.last == 0 }
  puts primes
  p worklist
end
```

Aside: Monitoring Progress (cont’d)

- This is what I saw when I called `sieve(50)`:

  ```ruby
  >> sieve(50)
  [2]
  [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49]
  [2, 3]
  [5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49]
  [2, 3, 5]
  [7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49]
  ....
  ```

A bit hard to read...

Aside: Monitoring Progress (cont’d)

- Add a third statement that “dresses up” the output a bit.
  - Prints a string that stands out from the others, to mark the start of each iteration.
  - Provides information about the state of the algorithm (i.e., the last prime found).

```ruby
while worklist.length > 0
  primes << worklist.first
  worklist.delete_if { |x| x % primes.last == 0 }
  puts "** after filtering by #{primes.last}:"
  p primes
  p worklist
end
```

Aside: Monitoring Progress (cont’d)

- The trace from the revised version:

  ```ruby
  >> sieve(50)
  ** after filtering by 2:
  [2]
  [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49]
  ** after filtering by 3:
  [2, 3]
  [5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49]
  ....
  ```

Better -- long lists of numbers have been broken up into sections.
Aside: Comments

- Instead of deleting the statements that produce the trace turn them into comments
  - a comment is a line in a program that is not executed by Ruby
  - it can contain any text
  - comments are used to describe the program to other people
- When Ruby sees a # character it ignores the rest of the line

```ruby
# Simple version -- iterate until the working list is empty
def sieve(n)
  worklist = []
  ...
  # p primes
  ...
  end
```

Tests

- When we are satisfied the program is working we can run a few more tests
  ```ruby
  >> sieve(50)
  => [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
  >> sieve(1000)
  >> sieve(1)
  => []
  ```

Good programmers also test for unexpected or unusual parameters values

Upper Bound

- When we call `sieve` we pass it `n`, the upper limit of the interval to search
  - e.g. `sieve(1000)` means we want find primes between 1 and 1000
- Any number removed from `worklist` must be
  (i) less than `n`
  (ii) the product of two numbers `a` and `b`
- A useful observation: `a` or `b` or both must be less than `sqrt(n)`
- If `a > sqrt(n)` and `b > sqrt(n)` then `a * b > sqrt(n) * sqrt(n) > n`
- That means we can stop scanning `worklist` when the newest prime is larger than `sqrt(n)`
  ```ruby
  while primes.last < sqrt(n)
    ...
  end
  ```

sieve 2.0

- Let’s return to the sieve method and the question of how many times we need to execute the loop
  - current version: until worklist is empty
  - Is there a way to formalize the intuition that well before the list is empty there is no need to keep looking for multiples?
- No multiples of 11 in worklist

```ruby
primes:   [2, 3, 5, 7, 11]
worklist: [13, 17, 19]
```
Details

- Unfortunately if we just change the expression in the while statement we introduce two “bugs”
- This is an error message from Ruby after we make the change, reload the method, and call it:
  ```ruby
  >> sieve(100)
  NoMethodError: undefined method `<' for nil:NilClass
  ```
- The cause of this error: when primes is empty, primes.last is nil
  ```ruby
  primes = []
  while primes.last < sqrt(n) 
    ...
  end
  ```

One way to fix this problem is to initialize primes to [1] so there is always a value for primes.last

Details (cont’d)

- The second error:
  ```ruby
  >> load "sieve.rb"
  >> sieve(100)
  => [2, 3, 5, 7, 11]
  ```
- We seem to have lost most of the primes!
- The problem: when the loop ends, the first few prime numbers are in the array called primes, the rest are still in worklist

If x and y are both arrays, x + y is a new array made by attaching y to the end of x

Change the return value to primes + worklist

sieve 2.0

Here is the final version of the method:

```ruby
def sieve(n)
  worklist = []
  (n-1).times { |i| worklist << i+2 }
  primes = [1]
  while primes.last < sqrt(n) 
    primes << worklist.first
    worklist.delete_if { |x| x % primes.last == 0 }
  end
  primes.shift
  return primes + worklist
end
```

Does It Make a Difference?

- For small values of $n$ you won’t notice much difference when you call the two methods
- But when $n$ is larger it makes a big difference

<table>
<thead>
<tr>
<th>Iterations</th>
<th>simple</th>
<th>sqrt</th>
<th>Time (min:sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Measurements made on Apple iMac with 3GHz CPU and OS/X 10.5
A New Tool for the Workbench

- Now that we have a method for making lists of prime numbers we can save it for later use.
- We can use it to answer questions about primes:
  - how many less than \( n \)?
  - largest gap between successive primes
  - prime pairs
  - Mersenne primes

Abstraction

- This is a good example of abstraction:
  - the ⭐ topic from a previous lecture.
- We have made a nice, neat package that we can save and reuse:
  - in the future we don’t have to worry about the implementation details.
  - we (and people who use it) just need to know that \( \text{sieve}(n) \) makes a list of prime numbers between 2 and \( n \).
  - `load "sieve.rb"`
  - `true`
  - `p = sieve(1000)`
  - `p.last` \( \rightarrow 997 \)
  - `p.length` \( \rightarrow 168 \)

Aside: Every Method Call Returns an Object

- One of the nice things about Ruby is that it is so consistent:
  - major complaint about Perl: too many special cases
  - constantly need to look things up in a manual to decide if they’re possible
- Every method call returns an object:
  - the return value might be \( \text{nil} \) (“nothing”) but \( \text{nil} \) is an object.
- A common idiom: \textit{chain} of method calls
- Example: how many prime numbers less than 1000?
  - `p = sieve(1000)`
  - `p.length` \( \rightarrow 168 \)

The Sieve in Practice

- This project on the Sieve of Eratosthenes was motivated by the importance of prime numbers in modern cryptography.
- Methods like RSA use huge prime numbers — 500 or more digits.
- These numbers are not generated by the sieve:
  - there are too many prime numbers, and the lists would be way too long.
  - an estimate: the number of primes less than \( n \) is \( n / \ln n \)
  - for \( n = 10^{500} \) there are around \( 10^{167} \) prime numbers.
- But the sieve is an important “building block” in the RSA algorithm.
The Sieve in Practice (cont’d)

To make a huge prime number:
1. let $p = \text{sieve}(1000)$
2. let $n$ be a 500-digit number chosen at random
3. if $n$ is a multiple of any number in $p$ go back to step 2
4. apply the “Rabin-Miller test”; if the test fails go back to step 2

It seems like this method would have a hard time finding a prime number but there are so many that the odds are pretty good
- for 500-digit numbers nearly 1 out of every 1000 is prime

Summary

- The Sieve of Eratosthenes is a process for making lists of prime numbers
- For over 2000 years people used this process -- aided by paper and pencil, abacus, or whatever technology was available
- In this unit we
  - used IRB to manage the lists, explore how the sieve works
  - created an algorithm that controls each step in the process
  - improved the algorithm after analyzing properties of composite numbers

<table>
<thead>
<tr>
<th>$n$</th>
<th>simple</th>
<th>sqrt</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>1,000</td>
<td>168</td>
<td>12</td>
</tr>
<tr>
<td>10,000</td>
<td>1229</td>
<td>26</td>
</tr>
<tr>
<td>100,000</td>
<td>9592</td>
<td>66</td>
</tr>
<tr>
<td>1,000,000</td>
<td>78498</td>
<td>169</td>
</tr>
</tbody>
</table>

Summary (cont’d)

- A final comment:
  - computer science is more than just “programming”
  - computer science is the study of computation
  - computations may be performed by people or by machines
- A programmer is concerned with writing instructions in a programming language in order to implement an algorithm
- A computer scientist is also interested in the ideas behind the algorithm and finding ways to improve the algorithm
  - how many iterations? can we make fewer iterations?
  - use mathematical properties of prime numbers to derive a new condition for stopping the iteration
- Good programmers are also computer scientists....