Lecture 11
Number Theoretic Algorithms 2
Getting Prime Numbers

COMP26120

Giles Reger

May 2019

Today

Many applications (such as cryptography, as we saw last time) require prime numbers... very large numbers.

How large?

Today

Many applications (such as cryptography, as we saw last time) require prime numbers... very large numbers.

How large? In the order of 512 - 1024 bits...
(2^{512} consists of roughly 154 digits in decimal)

Today

Many applications (such as cryptography, as we saw last time) require prime numbers... very large numbers.

How large? In the order of 512 - 1024 bits...
(2^{512} consists of roughly 154 digits in decimal)

So, where they do they come from?

Today we look at why finding prime numbers is hard

We then look at how *almost always* finding them can be (relatively) easy

Prime number theorem

Let $\pi(n)$ be the *prime distribution function* specifying the number of primes that are less than or equal to n.

Some examples, $\pi(10) = 4$ and $\pi(10^9) = 50,847,534$.

We have the **prime number theorem**:

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln\ n} = 1$$

In fact, $\frac{n}{\ln\ n}$ is a reasonable approximation of $\pi(n)$ even for small n, which means...

Finding Primes versus Checking Primality

How hard are prime numbers to find?

From the prime number theorem, the probability that a randomly chosen number n is prime is $\frac{1}{\ln\ n}$

So if we want a prime near n we can pick $\ln\ n$ numbers and check if they're prime (primality testing)

For example, finding a 1024-bit prime would require checking $\ln\ 2^{1024} \approx 710$ numbers - half that if we ignore even numbers.

We reduce finding primes to checking primality... so how hard is that?

Checking Primality Naively

(this process is called **trial division**)

```
isPrime(x):
  y := x-1
  while y > 0:
    if y divides x then break
    else y := y-1
  if y is 1 return ‘is prime’
  else return ‘is composite’
```

Can you suggest any optimisations?

Checking Primality Naively

(this process is called **trial division**)

```
isPrime(x):
  y := x-1
  while y > 0:
    if y divides x then break
    else y := y-1
  if y is 1 return ‘is prime’
  else return ‘is composite’
```

Can you suggest any optimisations? Answer: start at $\sqrt{x}$

Checking Primality Naively

(this process is called **trial division**)

```
isPrime(x):
  y := x-1
  while y > 0:
    if y divides x then break
    else y := y-1
  if y is 1 return ‘is prime’
  else return ‘is composite’
```

Can you suggest any optimisations? Answer: start at $\sqrt{x}$

What is the complexity?

Checking Primality Naively

(this process is called **trial division**)

```
isPrime(x):
  y := x-1
  while y > 0:
    if y divides x then break
    else y := y-1
  if y is 1 return ‘is prime’
  else return ‘is composite’
```

Can you suggest any optimisations? Answer: start at $\sqrt{x}$

What is the complexity? $O(\sqrt{x})$ but if x is n bits this is...

Checking Primality Naively

(this process is called **trial division**)

```
isPrime(x):
  y := x-1
  while y > 0:
    if y divides x then break
    else y := y-1
  if y is 1 return ‘is prime’
  else return ‘is composite’
```

Can you suggest any optimisations? Answer: start at $\sqrt{x}$

What is the complexity? $O(\sqrt{x})$ but if x is n bits this is... $O(2^{\frac{n}{2}})$

Another try. Remember Fermat's Little Theorem?

Fermat's Little Theorem

Let p be a prime, and let x be an integer such that $x \bmod p \neq 0$. Then

$$x^{p-1} \equiv 1 \pmod{p}$$

What if we pick a random x and raise it to the power $p-1$. If the result is *not* 1 then p is *not* prime.

But if it is... we don't know. But, what if we pick enough x?

Bad news: *Carmichael numbers* have $x^{n-1} \equiv 1 \bmod n$ for all $1 \leq x \leq n-1$ but n is composite e.g. 561 and 1105.

So that's never going to give us a 100% guarantee

Randomized Primality Testing

We cannot use Fermat's Little Theorem directly but we can create a probabilistic approach that uses it.

Randomized Primality Testing

Assume that we have a function witness(x, n) that takes a random variable x and a number n that works as follows:

If n is

1. prime then witness(x, n) always returns *false*
2. composite then witness(x, n) returns *false* with probability $q < 1$

e.g. it sometimes incorrectly identifies a composite as prime.

We can use this function to create a probabilistic primality testing algorithm that takes two inputs, a number n and a confidence parameter k, and decides whether n is prime with error probability 2^{-k}.

Randomized Primality Testing

Here's the algorithm

$$
\begin{array}{l}
\mathit{RandomizedPrimalityTesting}(n, k): \\
\quad t = \lceil \frac{k}{log_2(\frac{1}{q})} \rceil \\
\quad \textbf{for } i \leftarrow 1 \textbf{ to } t \textbf{ do} \\
\quad\quad x \leftarrow \text{random}() \\
\quad\quad \textbf{if } \text{witness}(x, n) \textbf{ then} \\
\quad\quad\quad \textbf{return } \mathit{true} \\
\quad \textbf{return } \mathit{false}
\end{array}
$$

But where does $\frac{k}{log_2(\frac{1}{q})}$ come from?

We want $q^t \leq 2^{-k}$ so we rearrange $t = log_q(2^{-k})$ as follows

$$\frac{log_2(2^{-k})}{log_2(q)} = \frac{-k}{log_2(q)} = \frac{k}{-log_2(q)} = \frac{k}{log_2(1) - log_2(q)} = \frac{k}{log_2(\frac{1}{q})}$$

Rabin-Miller Primality Testing

Where do we get this witness function from?

Let us build one using Fermat's little theorem and the following result.

Let p be a prime number > 2 and x be $0 < x < p$ such that

$$x^2 \equiv 1 \pmod{p}$$

then either

$$x \equiv 1 \pmod{p}$$

or

$$x \equiv -1 \pmod{p}$$

A *nontrivial square root of the unity* for n is an integer $1 < x < n-1$ such that $x^2 \equiv 1 \pmod{n}$. The above states that if n is prime then there are no nontrivial square roots of unity for n.

Rabin-Miller Primality Testing

This leads to this witness function:

$witness(x, n)$:
  Write $n-1$ as $2^k m$, where m is odd
  Compute $y \leftarrow x^m \pmod{n}$
  if $y \equiv 1 \pmod{n}$ **then**
    return false (n is probably prime)
  for $i \leftarrow 1$ **to** $k-1$ **do**
    if $y \equiv 1 \pmod{n}$ **then**
      return false
    $y \rightarrow y^2 \pmod{n}$
  return true (n is definitely composite)

If n is composite then there are at most $(n-1)/4$ positive values of $1 < x < n-1$ such that $witness(x, n)$ returns true.
So $q = \frac{1}{4}$ and in the above we get $t = \frac{k}{\log_2(4)} = \frac{k}{2}$.

Rabin-Miller Primality Testing

Note that the main arithmetic operation is modular exponentiation - we know how to do this linearly in the size of the input.

The whole algorithm requires asymptotically no more work than k modular exponentiations.

So, given an odd positive integer n and a confidence parameter $k > 0$, the Rabin-Miller algorithm determines whether n is prime, with error probably 2^{-k}, by performing $O(k \log n)$ arithmetic operations.

Sieve of Eratosthenes (non-examinable)

Finally, we can generate all primes not greater than a given n in $O(n \log \log n)$ with $O(n)$ memory as follows:

1. Create an array of consecutive numbers from 2 to n
2. Let $p = 2$
3. Enumerate multiples of p in increments of p from $2p$ to n and mark them in the array (e.g. $2p, 3p, 4p, \ldots$)
4. Find the first number $> p$ in the array that is not marked. If it exists let it equal p and GOTO 3. Otherwise, stop.
5 . When we stop all non-marked numbers are primes below n

This is the general idea - try implementing it as a fun task:

Summary

Because of the density of primes we can find them with reasonable probability by sampling a certain number.

Given a witness for checking primality with a certain error rate, we can achieve a given much lower error rate.

We can use a result related to Fermat's Little Theorem to create such a witness that works efficiently

We can also use the Sieve of Eratosthenes to generate prime numbers up to a given value