BILL, RECORD LECTURE!!!!

BILL RECORD LECTURE!!!

Primality Testing

RECAP

We seek a protocl where Alice and Bob do easy computations and Eve has to do a hard one in order to crack the code.

RECAP

We seek a protocl where Alice and Bob do easy computations and Eve has to do a hard one in order to crack the code.

1. Exponentiation mod p is easy.

RECAP

We seek a protocl where Alice and Bob do easy computations and Eve has to do a hard one in order to crack the code.

1. Exponentiation mod p is easy.
2. Discrete Log is hard.

RECAP

We seek a protocl where Alice and Bob do easy computations and Eve has to do a hard one in order to crack the code.

1. Exponentiation mod p is easy.
2. Discrete Log is hard.
3. In order for Discrete Log to be used we need a prime p and a generator g.

RECAP

We seek a protocl where Alice and Bob do easy computations and Eve has to do a hard one in order to crack the code.

1. Exponentiation mod p is easy.
2. Discrete Log is hard.
3. In order for Discrete Log to be used we need a prime p and a generator g.
4. If p is a safe prime then it is easy to find a generator.

RECAP

We seek a protocl where Alice and Bob do easy computations and Eve has to do a hard one in order to crack the code.

1. Exponentiation mod p is easy.
2. Discrete Log is hard.
3. In order for Discrete Log to be used we need a prime p and a generator g.
4. If p is a safe prime then it is easy to find a generator.
5. **Goal One** Primality Testing. Can easily use this to test if a number is a prime and also if a number is a safe prime.

RECAP

We seek a protocl where Alice and Bob do easy computations and Eve has to do a hard one in order to crack the code.

1. Exponentiation mod p is easy.
2. Discrete Log is hard.
3. In order for Discrete Log to be used we need a prime p and a generator g.
4. If p is a safe prime then it is easy to find a generator.
5. **Goal One** Primality Testing. Can easily use this to test if a number is a prime and also if a number is a safe prime.
6. **Goal Two** Finding a Safe Prime: Given L we will want to quickly generate a safe prime of bit-length L.

Primality Testing

Warning The next few slides will culminate in a test for primality that may FAIL.

Primality Testing

Warning The next few slides will culminate in a test for primality that may FAIL.

It is **not** used.

Primality Testing

Warning The next few slides will culminate in a test for primality that may FAIL.

It is **not** used.

But the **ideas** are used in real algorithms.

Is This a Natural Number?

Is the following a natural number?

$$\frac{1002!}{417!585!}$$

Is This a Natural Number?

Is the following a natural number?

$$\frac{1002!}{417!585!}$$

Yes

Is This a Natural Number?

Is the following a natural number?

$$\frac{1002!}{417!585!}$$

Yes

Hard Proof Look at factors and stuff.

Is This a Natural Number?

Is the following a natural number?

$$\frac{1002!}{417!585!}$$

Yes

Hard Proof Look at factors and stuff.

Easy Proof
The number of ways to pick 417 people out of 1002 is

$$\frac{1002!}{417!585!}.$$

Is This a Natural Number?

Is the following a natural number?

$$\frac{1002!}{417!585!}$$

Yes

Hard Proof Look at factors and stuff.

Easy Proof
The number of ways to pick 417 people out of 1002 is

$$\frac{1002!}{417!585!}.$$

So $\frac{1002!}{417!585!}$ is the answer to a question that has a nat numb answer.

Is This a Natural Number?

Is the following a natural number?

$$\frac{1002!}{417!585!}$$

Yes

Hard Proof Look at factors and stuff.

Easy Proof
The number of ways to pick 417 people out of 1002 is

$$\frac{1002!}{417!585!}.$$

So $\frac{1002!}{417!585!}$ is the answer to a question that has a nat numb answer.
Yes that really is the proof.

More Generally: Yes, This is a Natural Number

Theorem NAT For all $k, n \in \mathbb{N}$, $k \leq n$, $\frac{n!}{k!(n-k)!} \in \mathbb{N}$.

Proof

$\frac{n!}{k!(n-k)!}$ is the number of ways to choose k objects out of n.

So it answers a question that has a nat numb answer.

So its a natural number.

End of Proof

More Generally: Yes, This is a Natural Number

Theorem NAT For all $k, n \in \mathbb{N}$, $k \le n$, $\frac{n!}{k!(n-k)!} \in \mathbb{N}$.

Proof

$\frac{n!}{k!(n-k)!}$ is the number of ways to choose k objects out of n.

So it answers a question that has a nat numb answer.

So its a natural number.

End of Proof

Notation $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

The Binomial Theorem

Recall

The Binomial Theorem

For any $n \in \mathbb{N}$,

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i}.$$

Lemma on $\frac{p!}{i!(p-i)!}$

Lemma If $\frac{a}{b} \in \mathbb{N}$, p is a prime, p divides a, but p does not divide b then p divides $\frac{a}{b}$.
Proof Factor a and b into primes.

Lemma on $\frac{p!}{i!(p-i)!}$

Lemma If $\frac{a}{b} \in \mathbb{N}$, p is a prime, p divides a, but p does not divide b then p divides $\frac{a}{b}$.
Proof Factor a and b into primes.
Let $p_1, \ldots, p_k$ be primes that divide either a or b or both. Let $p = p_1$.

Lemma on $\frac{p!}{i!(p-i)!}$

Lemma If $\frac{a}{b} \in \mathbb{N}$, p is a prime, p divides a, but p does not divide b then p divides $\frac{a}{b}$.

Proof Factor a and b into primes.

Let $p_1, \ldots, p_k$ be primes that divide either a or b or both. Let $p = p_1$.

$a = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$

Lemma on $\frac{p!}{i!(p-i)!}$

Lemma If $\frac{a}{b} \in \mathbb{N}$, p is a prime, p divides a, but p does not divide b then p divides $\frac{a}{b}$.
Proof Factor a and b into primes.
Let $p_1, \ldots, p_k$ be primes that divide either a or b or both. Let $p = p_1$.
$a = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$
$b = p_1^{0} p_2^{b_2} \cdots p_k^{b_k}$
Since $\frac{a}{b} \in \mathbb{N}$, we have $a_2 \geq b_2$, ..., $a_k \geq b_k$.

$$\frac{a}{b} = p_1^{a_1} p_2^{a_2-b_2} \cdots a_k^{a_k-b_k}$$

Since $a_1 \geq 1$ and all of the exponents are ≥ 0. p divides $\frac{a}{b}$.
End of Proof
Corollary If p prime, $1 \leq i \leq p-1$, then $\frac{p!}{i!(p-i)!} \in \mathbb{N}$ and is divisible by p.

Primality Testing

Fermat's Little Thm

Lemma If p prime, $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.

Primality Testing

Fermat's Little Thm

Lemma If p prime, $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.

Proof Fix prime p. By induction on a. **Base Case** $1^p \equiv 1$.

Primality Testing

Fermat's Little Thm
Lemma If p prime, $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.
Proof Fix prime p. By induction on a. **Base Case** $1^p \equiv 1$.
Ind Hyp $a^p \equiv a \pmod{p}$.

Primality Testing

Fermat's Little Thm

Lemma If p prime, $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.

Proof Fix prime p. By induction on a. **Base Case** $1^p \equiv 1$.

Ind Hyp $a^p \equiv a \pmod{p}$.

Ind Step $(a+1)^p = \binom{p}{p} a^p + \binom{p}{p-1} a^{p-1} + \cdots + \binom{p}{1} a^1 + \binom{p}{0} a^0$.

Primality Testing

Fermat's Little Thm

Lemma If p prime, $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.

Proof Fix prime p. By induction on a. **Base Case** $1^p \equiv 1$.

Ind Hyp $a^p \equiv a \pmod{p}$.

Ind Step $(a+1)^p = \binom{p}{p}a^p + \binom{p}{p-1}a^{p-1} + \cdots + \binom{p}{1}a^1 + \binom{p}{0}a^0$.

By previous lemma $\binom{p}{1} \equiv \binom{p}{2} \equiv \cdots \equiv \binom{p}{p-1} \equiv 0$. Hence

Primality Testing

Fermat's Little Thm

Lemma If p prime, $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.

Proof Fix prime p. By induction on a. **Base Case** $1^p \equiv 1$.

Ind Hyp $a^p \equiv a \pmod{p}$.

Ind Step $(a+1)^p = \binom{p}{p}a^p + \binom{p}{p-1}a^{p-1} + \cdots + \binom{p}{1}a^1 + \binom{p}{0}a^0$.

By previous lemma $\binom{p}{1} \equiv \binom{p}{2} \equiv \cdots \equiv \binom{p}{p-1} \equiv 0$. Hence

$$(a+1)^p \equiv \binom{p}{p}a^p + \binom{p}{0}a^0 \equiv a^p + 1 \equiv a + 1.$$

(Used $a^p \equiv a \pmod{p}$ which is from Ind Hyp.)

End of Proof

A Primality Testing Algorithm

Prior Slides If p is prime and $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.

A Primality Testing Algorithm

Prior Slides If p is prime and $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.
What has been observed If p is **not** prime then **usually** for **most** a, $a^p \not\equiv a \pmod{p}$.

A Primality Testing Algorithm

Prior Slides If p is prime and $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.
What has been observed If p is **not** prime then **usually** for **most** a, $a^p \not\equiv a \pmod{p}$.
Primality Algorithm

A Primality Testing Algorithm

Prior Slides If p is prime and $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.
What has been observed If p is **not** prime then **usually** for **most** a, $a^p \not\equiv a \pmod{p}$.
Primality Algorithm

1. Input p. (In algorithm all arithmetic is mod p.)

A Primality Testing Algorithm

Prior Slides If p is prime and $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.
What has been observed If p is **not** prime then **usually** for **most** a, $a^p \not\equiv a \pmod{p}$.
Primality Algorithm

1. Input p. (In algorithm all arithmetic is mod p.)
2. Form rand $R \subseteq \{2, \ldots, p-1\}$ of size $\sim \lg p$.

A Primality Testing Algorithm

Prior Slides If p is prime and $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.
What has been observed If p is **not** prime then **usually** for **most** a, $a^p \not\equiv a \pmod{p}$.
Primality Algorithm

1. Input p. (In algorithm all arithmetic is mod p.)
2. Form rand $R \subseteq \{2, \ldots, p-1\}$ of size $\sim \lg p$.
3. For each $a \in R$ compute a^p.

A Primality Testing Algorithm

Prior Slides If p is prime and $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.
What has been observed If p is **not** prime then **usually** for **most** a, $a^p \not\equiv a \pmod{p}$.
Primality Algorithm

1. Input p. (In algorithm all arithmetic is mod p.)
2. Form rand $R \subseteq \{2, \ldots, p-1\}$ of size $\sim \lg p$.
3. For each $a \in R$ compute a^p.
    3.1 If ever get $a^p \not\equiv a$ then p **not prime**(we are sure).

A Primality Testing Algorithm

Prior Slides If p is prime and $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.
What has been observed If p is **not** prime then **usually** for **most** a, $a^p \not\equiv a \pmod{p}$.
Primality Algorithm

1. Input p. (In algorithm all arithmetic is mod p.)
2. Form rand $R \subseteq \{2, \ldots, p-1\}$ of size $\sim \lg p$.
3. For each $a \in R$ compute a^p.
   3.1 If ever get $a^p \not\equiv a$ then p **not prime**(we are sure).
   3.2 If for all a, $a^p \equiv a$ then PRIME (we are not sure).

Two reasons for our uncertainty:

A Primality Testing Algorithm

Prior Slides If p is prime and $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.
What has been observed If p is **not** prime then **usually** for **most** a, $a^p \not\equiv a \pmod{p}$.
Primality Algorithm

1. Input p. (In algorithm all arithmetic is mod p.)
2. Form rand $R \subseteq \{2, \ldots, p-1\}$ of size $\sim \lg p$.
3. For each $a \in R$ compute a^p.
   3.1 If ever get $a^p \not\equiv a$ then p **not prime**(we are sure).
   3.2 If for all a, $a^p \equiv a$ then PRIME (we are not sure).

Two reasons for our uncertainty:

- p is composite but we were unlucky with R.

A Primality Testing Algorithm

Prior Slides If p is prime and $a \in \mathbb{N}$ then $a^p \equiv a \pmod{p}$.
What has been observed If p is **not** prime then **usually** for **most** a, $a^p \not\equiv a \pmod{p}$.
Primality Algorithm

1. Input p. (In algorithm all arithmetic is mod p.)
2. Form rand $R \subseteq \{2, \ldots, p-1\}$ of size $\sim \lg p$.
3. For each $a \in R$ compute a^p.
   3.1 If ever get $a^p \not\equiv a$ then p **not prime**(we are sure).
   3.2 If for all a, $a^p \equiv a$ then PRIME (we are not sure).

Two reasons for our uncertainty:

- p is composite but we were unlucky with R.
- There are some composite p such that **for all** a, $a^p \equiv a$.

Primality Testing – What is Really True

Primality Testing – What is Really True

1. Exists algorithm that only has first problem, possible bad luck.

Primality Testing – What is Really True

1. Exists algorithm that only has first problem, possible bad luck.
2. That algorithm has prob of failure $\leq \frac{1}{2^p}$. Good enough!

Primality Testing – What is Really True

1. Exists algorithm that only has first problem, possible bad luck.
2. That algorithm has prob of failure $\leq \frac{1}{2^p}$. Good enough!
3. Exists deterministic poly time algorithm but is much slower.

Primality Testing – What is Really True

1. Exists algorithm that only has first problem, possible bad luck.
2. That algorithm has prob of failure $\leq \frac{1}{2^p}$. Good enough!
3. Exists deterministic poly time algorithm but is much slower.
4. n is a **Carmichael Number** if it is composite and, for all a, $a^n \equiv a$. These are the numbers my algorithm FAILS on.

Primality Testing – What is Really True

1. Exists algorithm that only has first problem, possible bad luck.
2. That algorithm has prob of failure $\le \frac{1}{2^p}$. Good enough!
3. Exists deterministic poly time algorithm but is much slower.
4. n is a **Carmichael Number** if it is composite and, for all a, $a^n \equiv a$. These are the numbers my algorithm FAILS on. The first few are

Primality Testing – What is Really True

1. Exists algorithm that only has first problem, possible bad luck.
2. That algorithm has prob of failure $\leq \frac{1}{2^p}$. Good enough!
3. Exists deterministic poly time algorithm but is much slower.
4. n is a **Carmichael Number** if it is composite and, for all a, $a^n \equiv a$. These are the numbers my algorithm FAILS on. The first few are

$$561, 1105, 1729, 2465, 2821, 6601, 8911.$$

Primality Testing – What is Really True

1. Exists algorithm that only has first problem, possible bad luck.
2. That algorithm has prob of failure $\leq \frac{1}{2^p}$. Good enough!
3. Exists deterministic poly time algorithm but is much slower.
4. n is a **Carmichael Number** if it is composite and, for all a, $a^n \equiv a$. These are the numbers my algorithm FAILS on. The first few are

   $$561, 1105, 1729, 2465, 2821, 6601, 8911.$$

   There are an infinite number of Carmichael numbers, but they are rare.

Generating Primes

- We just gave a fast algorithm for **testing** if p is prime.

Generating Primes

- We just gave a fast algorithm for **testing** if p is prime.
- We want to **generate** primes.

Generating Primes

- We just gave a fast algorithm for **testing** if p is prime.
- We want to **generate** primes.

New Problem Given L, return an L-bit prime.

Generating Primes

- We just gave a fast algorithm for **testing** if p is prime.
- We want to **generate** primes.

New Problem Given L, return an L-bit prime.
Clarification An L-bit prime has a 1 as left most bit.

Alg for Generating Primes

First Attempt at, given L, generating a prime of length L.

Alg for Generating Primes

First Attempt at, given L, generating a prime of length L.

1. Input(L).

Alg for Generating Primes

First Attempt at, given L, generating a prime of length L.

1. Input(L).
2. Pick $y \in \{0, 1\}^{L-1}$ at rand.

Alg for Generating Primes

First Attempt at, given L, generating a prime of length L.

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (so x is a L-bit number).

Alg for Generating Primes

First Attempt at, given L, generating a prime of length L.

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (so x is a L-bit number).
4. Test if x is prime.

Alg for Generating Primes

First Attempt at, given L, generating a prime of length L.

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (so x is a L-bit number).
4. Test if x is prime.
5. If x is prime then output x and stop, else goto step 2.

Alg for Generating Primes

First Attempt at, given L, generating a prime of length L.

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (so x is a L-bit number).
4. Test if x is prime.
5. If x is prime then output x and stop, else goto step 2.

Is this a good algorithm?

Alg for Generating Primes

First Attempt at, given L, generating a prime of length L.

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (so x is a L-bit number).
4. Test if x is prime.
5. If x is prime then output x and stop, else goto step 2.

Is this a good algorithm?
PRO Math: returns a prime $\sim 3L^2$ tries with high prob.

Alg for Generating Primes

First Attempt at, given L, generating a prime of length L.

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (so x is a L-bit number).
4. Test if x is prime.
5. If x is prime then output x and stop, else goto step 2.

Is this a good algorithm?
PRO Math: returns a prime $\sim 3L^2$ tries with high prob.
CON Tests lots of numbers that are obv not prime—e.g, evens.

Generating Safe Primes

Definition p is a *safe prime* if p is prime and $\frac{p-1}{2}$ is prime.

First Attempt at, given L, generating a safe prime of length L

Generating Safe Primes

Definition p is a *safe prime* if p is prime and $\frac{p-1}{2}$ is prime.

First Attempt at, given L, generating a safe prime of length L

1. Input(L).

Generating Safe Primes

Definition p is a *safe prime* if p is prime and $\frac{p-1}{2}$ is prime.

First Attempt at, given L, generating a safe prime of length L

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.

Generating Safe Primes

Definition p is a *safe prime* if p is prime and $\frac{p-1}{2}$ is prime.

First Attempt at, given L, generating a safe prime of length L

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (note that x is a L-bit number).

Generating Safe Primes

Definition p is a *safe prime* if p is prime and $\frac{p-1}{2}$ is prime.

First Attempt at, given L, generating a safe prime of length L

1. Input(L).
2. Pick $y \in \{0, 1\}^{L-1}$ at rand.
3. $x = 1y$ (note that x is a L-bit number).
4. Test if x and $\frac{x-1}{2}$ are prime.

Generating Safe Primes

Definition p is a *safe prime* if p is prime and $\frac{p-1}{2}$ is prime.

First Attempt at, given L, generating a safe prime of length L

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (note that x is a L-bit number).
4. Test if x and $\frac{x-1}{2}$ are prime.
5. If they both are then output x and stop, else goto step 2.

Generating Safe Primes

Definition p is a *safe prime* if p is prime and $\frac{p-1}{2}$ is prime.

First Attempt at, given L, generating a safe prime of length L

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (note that x is a L-bit number).
4. Test if x and $\frac{x-1}{2}$ are prime.
5. If they both are then output x and stop, else goto step 2.

Is this a good algorithm?

Generating Safe Primes

Definition p is a *safe prime* if p is prime and $\frac{p-1}{2}$ is prime.

First Attempt at, given L, generating a safe prime of length L

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (note that x is a L-bit number).
4. Test if x and $\frac{x-1}{2}$ are prime.
5. If they both are then output x and stop, else goto step 2.

Is this a good algorithm?

PRO Math: returns prime quickly with high prob.

Generating Safe Primes

Definition p is a *safe prime* if p is prime and $\frac{p-1}{2}$ is prime.

First Attempt at, given L, generating a safe prime of length L

1. Input(L).
2. Pick $y \in \{0,1\}^{L-1}$ at rand.
3. $x = 1y$ (note that x is a L-bit number).
4. Test if x and $\frac{x-1}{2}$ are prime.
5. If they both are then output x and stop, else goto step 2.

Is this a good algorithm?

PRO Math: returns prime quickly with high prob.

CON Tests lots of numbers that are obv not prime—e.g, evens.

Speed Prime-Finding: $n \not\equiv 0 \pmod 2$

We use $L - 1$-bit strings, including ones that end in 0, which are even.

Speed Prime-Finding: $n \not\equiv 0 \pmod 2$

We use $L-1$-bit strings, including ones that end in 0, which are even.
IDEA Pick $L-2$ bit string, put 1 on its right and on its left.
Is this a good idea? Vote.

Speed Prime-Finding: $n \not\equiv 0 \pmod{2}$

We use $L-1$-bit strings, including ones that end in 0, which are even.
IDEA Pick $L-2$ bit string, put 1 on its right and on its left.
Is this a good idea? Vote.

PRO Do not waste time testing even numbers.

Speed Prime-Finding: $n \not\equiv 0 \pmod 2$

We use $L-1$-bit strings, including ones that end in 0, which are even.
IDEA Pick $L-2$ bit string, put 1 on its right and on its left.
Is this a good idea? Vote.

PRO Do not waste time testing even numbers.
CON Does it really save that much time?

Speed Prime-Finding: $n \not\equiv 0 \pmod{2}$

We use $L-1$-bit strings, including ones that end in 0, which are even.
IDEA Pick $L-2$ bit string, put 1 on its right and on its left.
Is this a good idea? Vote.

PRO Do not waste time testing even numbers.
CON Does it really save that much time?
CAVEAT Extend so we don't test numbers div by 3? Discuss.

Speed Prime-Finding: $n \not\equiv 0 \pmod{2}$

We use $L-1$-bit strings, including ones that end in 0, which are even.
IDEA Pick $L-2$ bit string, put 1 on its right and on its left.
Is this a good idea? Vote.

PRO Do not waste time testing even numbers.
CON Does it really save that much time?
CAVEAT Extend so we don't test numbers div by 3? Discuss.
Yes.

Speed Up Prime-Finding: $\not\equiv 0 \pmod{2, 3}$

2 divides n iff $(\exists k)[n = 2k]$
2 does not divide n iff $(\exists k)[n = 2k + 1]$

Speed Up Prime-Finding: $\not\equiv 0 \pmod{2, 3}$

2 divides n iff $(\exists k)[n = 2k]$
2 does not divide n iff $(\exists k)[n = 2k + 1]$

3 divides n iff $(\exists k)[n = 3k]$
3 does not divide n iff $(\exists k)(\exists i \in \{1, 2\})[n = 3k + i]$

Speed Up Prime-Finding: $\not\equiv 0 \pmod{2, 3}$

2 divides n iff $(\exists k)[n = 2k]$
2 does not divide n iff $(\exists k)[n = 2k + 1]$

3 divides n iff $(\exists k)[n = 3k]$
3 does not divide n iff $(\exists k)(\exists i \in \{1, 2\})[n = 3k + i]$

How to get both?

Speed Up Prime-Finding: $\not\equiv 0 \pmod{2, 3}$

2 divides n iff $(\exists k)[n = 2k]$
2 does not divide n iff $(\exists k)[n = 2k + 1]$

3 divides n iff $(\exists k)[n = 3k]$
3 does not divide n iff $(\exists k)(\exists i \in \{1, 2\})[n = 3k + i]$

How to get both?
Neither 2 nor 3 divides n iff $(\exists k)(\exists i \in \{1, 5\})[n = 6k + i]$

Speed Up Prime-Finding: $\not\equiv 0 \pmod{2,3}$

2 divides n iff $(\exists k)[n = 2k]$
2 does not divide n iff $(\exists k)[n = 2k + 1]$

3 divides n iff $(\exists k)[n = 3k]$
3 does not divide n iff $(\exists k)(\exists i \in \{1, 2\})[n = 3k + i]$

How to get both?
Neither 2 nor 3 divides n iff $(\exists k)(\exists i \in \{1, 5\})[n = 6k + i]$

So need to generate numbers of the form $6k + 1$ and $6k + 5$.

Speed Up Prime-Finding: $\not\equiv 0 \pmod{2, 3}$

2 divides n iff $(\exists k)[n = 2k]$
2 does not divide n iff $(\exists k)[n = 2k + 1]$

3 divides n iff $(\exists k)[n = 3k]$
3 does not divide n iff $(\exists k)(\exists i \in \{1, 2\})[n = 3k + i]$

How to get both?
Neither 2 nor 3 divides n iff $(\exists k)(\exists i \in \{1, 5\})[n = 6k + i]$

So need to generate numbers of the form $6k + 1$ and $6k + 5$.
Caveat Might get a prime **of length $L - 1$**. We ignore this.

Alg for Gen Primes that Ignores $n \equiv 0 \pmod{2, 3}$

Alg for Gen Primes that Ignores $n \equiv 0 \pmod{2,3}$

1. Input L, want L bit prime.

Alg for Gen Primes that Ignores $n \equiv 0 \pmod{2, 3}$

1. Input L, want L bit prime.
2. Pick $y \in \{0, 1\}^{L-3}$ (an $(L-3)$-bit number).

Alg for Gen Primes that Ignores $n \equiv 0 \pmod{2, 3}$

1. Input L, want L bit prime.
2. Pick $y \in \{0, 1\}^{L-3}$ (an $(L - 3)$-bit number).
3. Let $x = 1y$ (an $L - 2$ bit number).

Alg for Gen Primes that Ignores $n \equiv 0 \pmod{2, 3}$

1. Input L, want L bit prime.
2. Pick $y \in \{0,1\}^{L-3}$ (an $(L-3)$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Test if $6x+1$ is prime. ($(L-1)$-bit or L-bit number). If yes then output $6x+1$. If not then goto Step 2.

Alg for Gen Primes that Ignores $n \equiv 0 \pmod{2,3}$

1. Input L, want L bit prime.
2. Pick $y \in \{0,1\}^{L-3}$ (an $(L-3)$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Test if $6x+1$ is prime. ($(L-1)$-bit or L-bit number). If yes then output $6x+1$. If not then goto Step 2.

Is this a good idea? Vote

Alg for Gen Primes that Ignores $n \equiv 0 \pmod{2, 3}$

1. Input L, want L bit prime.
2. Pick $y \in \{0,1\}^{L-3}$ (an $(L-3)$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Test if $6x+1$ is prime. ($(L-1)$-bit or L-bit number). If yes then output $6x+1$. If not then goto Step 2.

Is this a good idea? Vote

PRO Do not waste time testing numbers $\equiv 0$ mod 2 or 3.

Alg for Gen Primes that Ignores $n \equiv 0 \pmod{2, 3}$

1. Input L, want L bit prime.
2. Pick $y \in \{0, 1\}^{L-3}$ (an $(L-3)$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Test if $6x + 1$ is prime. ($(L-1)$-bit or L-bit number). If yes then output $6x + 1$. If not then goto Step 2.

Is this a good idea? Vote

PRO Do not waste time testing numbers $\equiv 0$ mod 2 or 3.
CON Uses primes of form $6k + 1$. Not random enough?

Alg for Gen Primes that Ignores $n \equiv 0 \pmod{2,3}$

1. Input L, want L bit prime.
2. Pick $y \in \{0,1\}^{L-3}$ (an $(L-3)$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Test if $6x+1$ is prime. ($(L-1)$-bit or L-bit number). If yes then output $6x+1$. If not then goto Step 2.

Is this a good idea? Vote

PRO Do not waste time testing numbers $\equiv 0$ mod 2 or 3.
CON Uses primes of form $6k+1$. Not random enough?
CAVEAT Can we modify to avoid this problem?

Alg for Gen Primes that Ignores $n \equiv 0 \pmod{2, 3}$

1. Input L, want L bit prime.
2. Pick $y \in \{0, 1\}^{L-3}$ (an $(L-3)$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Test if $6x + 1$ is prime. ($(L-1)$-bit or L-bit number). If yes then output $6x + 1$. If not then goto Step 2.

Is this a good idea? Vote

PRO Do not waste time testing numbers $\equiv 0$ mod 2 or 3.
CON Uses primes of form $6k + 1$. Not random enough?
CAVEAT Can we modify to avoid this problem?
Yes.

Speed Up Alg Prime-Finding: $\not\equiv 0 \mod 2, 3$

Speed Up Alg Prime-Finding: $\not\equiv 0 \mod 2, 3$

1. Input L.

Speed Up Alg Prime-Finding: $\not\equiv 0 \mod 2, 3$

1. Input L.
2. Pick $y \in \{0,1\}^{L-3}$ (an $L-3$-bit number).

Speed Up Alg Prime-Finding: $\not\equiv 0 \mod 2, 3$

1. Input L.
2. Pick $y \in \{0,1\}^{L-3}$ (an $L-3$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).

Speed Up Alg Prime-Finding: $\not\equiv 0 \pmod{2, 3}$

1. Input L.
2. Pick $y \in \{0,1\}^{L-3}$ (an $L-3$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Pick $i \in \{1,5\}$ at random.

Speed Up Alg Prime-Finding: $\not\equiv 0 \pmod{2,3}$

1. Input L.
2. Pick $y \in \{0,1\}^{L-3}$ (an $L-3$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Pick $i \in \{1,5\}$ at random.
5. Test if $6x+i$ is prime (($L-1$)-bit or L-bit number). If yes then done, if not then try goto step 2.

Speed Up Alg Prime-Finding: $\not\equiv 0 \pmod{2, 3}$

1. Input L.
2. Pick $y \in \{0, 1\}^{L-3}$ (an $L - 3$-bit number).
3. Let $x = 1y$ (an $L - 2$ bit number).
4. Pick $i \in \{1, 5\}$ at random.
5. Test if $6x + i$ is prime (($L - 1$)-bit or L-bit number). If yes then done, if not then try goto step 2.

Is this a good idea? Vote.

Speed Up Alg Prime-Finding: $\not\equiv 0 \mod 2, 3$

1. Input L.
2. Pick $y \in \{0,1\}^{L-3}$ (an $L-3$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Pick $i \in \{1,5\}$ at random.
5. Test if $6x + i$ is prime (($L-1$)-bit or L-bit number). If yes then done, if not then try goto step 2.

Is this a good idea? Vote.
PRO Do not waste time testing numbers $\equiv 0 \pmod{2, 3}$.

Speed Up Alg Prime-Finding: $\not\equiv 0 \pmod{2, 3}$

1. Input L.
2. Pick $y \in \{0,1\}^{L-3}$ (an $L-3$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Pick $i \in \{1,5\}$ at random.
5. Test if $6x+i$ is prime ($(L-1)$-bit or L-bit number). If yes then done, if not then try goto step 2.

Is this a good idea? Vote.

PRO Do not waste time testing numbers $\equiv 0 \pmod{2,3}$.

CON Getting more complicated. Is it worth it? Do not know.

Speed Up Alg Prime-Finding: $\not\equiv 0 \mod 2, 3$

1. Input L.
2. Pick $y \in \{0,1\}^{L-3}$ (an $L-3$-bit number).
3. Let $x = 1y$ (an $L-2$ bit number).
4. Pick $i \in \{1,5\}$ at random.
5. Test if $6x+i$ is prime ($(L-1)$-bit or L-bit number). If yes then done, if not then try goto step 2.

Is this a good idea? Vote.

PRO Do not waste time testing numbers $\equiv 0 \pmod{2,3}$.

CON Getting more complicated. Is it worth it? Do not know.

CAVEAT Extend to 2,3,5? 2,3,5,7? etc.

BILL, STOP RECORDING LECTURE!!!!

BILL STOP RECORDING LECTURE!!!