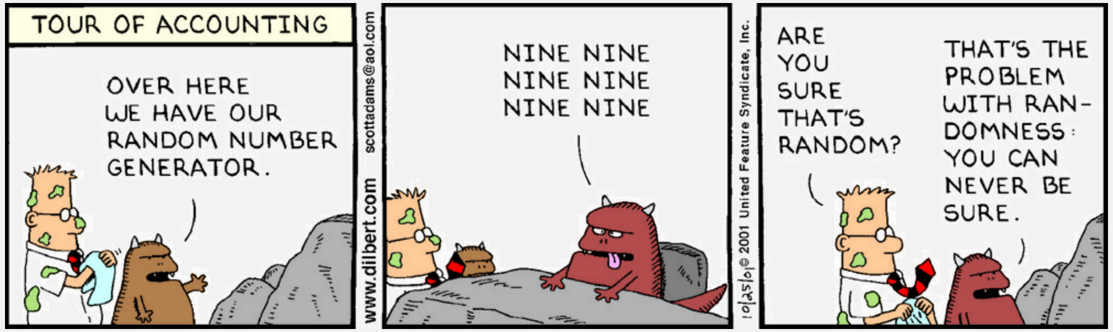
UMB CS622

Randomized Algorithms

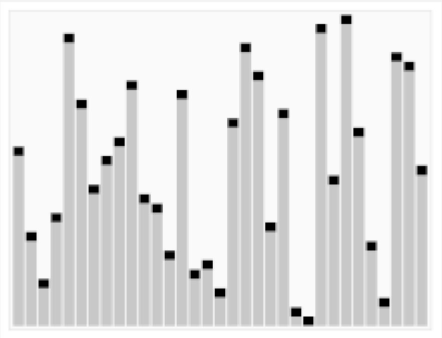
Monday, December 13, 2021



Announcements

- HW 11
  - Due Tues 12/14 11:59pm EST
- Last class! 🙁

Quicksort



SORT = On input A, where A is an array length n:

- Let:
  - `pivot` = $A[0]$
  - `partition1` = all $x \in A, x \leq \text{pivot}$
  - `partition2` = all $x \in A, x > \text{pivot}$
- Return *SORT*(`partition1`) ∘ [`pivot`] ∘ *SORT*(`partition2`)

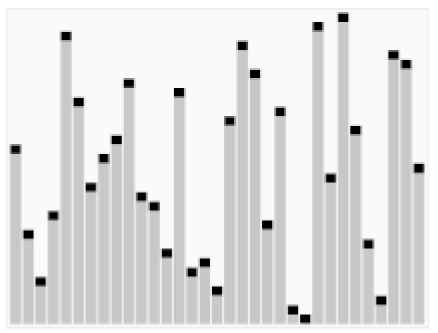
"Divide and conquer"

Worst case run time (should be $O(n \log n)$?):

- Time for each recursive call (to partition elements) = $O(n)$
- # recursive calls = $O(n)$ (if list is already sorted!)

Total: $O(n^2)$

Quicksort (with randomness)



$SORT$ = On input A, where A is an array length n:

- Let:
  - `pivot` = A[`random( )`] ← "coin flips"
  - `partition1` = all $x \in A, x \leq \text{pivot}$
  - `partition2` = all $x \in A, x > \text{pivot}$
- Return $SORT$(`partition1`) ∘ [`pivot`] ∘ $SORT$(`partition2`)

Randomness can help make worst case less likely to happen

or **wrong answer** (this is what we will look at)

Worst case run time (should be $O(n \log n)$?):

- Time for each recursive call (to partition) = $O(n)$
- # recursive calls = $O(n)$ (if the worst pivot is picked every time!)

Total: still $O(n^2)$!! (but much less likely)

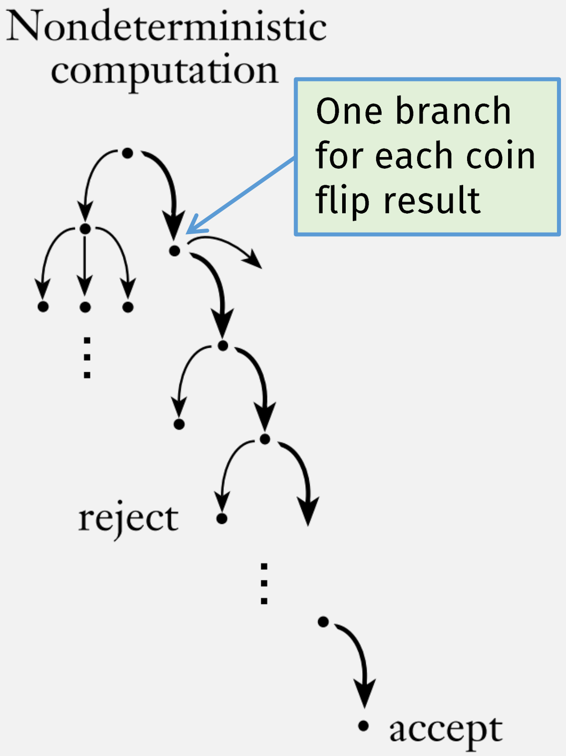
A Coin-Flipping (Probabilistic) TM

DEFINITION

A ***probabilistic Turing machine*** M is a type of nondeterministic Turing machine in which each nondeterministic step is called a ***coin-flip step*** and has two legal next moves. We assign a probability to each branch b of M's computation on input w as follows. Define the probability of branch b to be

$$\Pr[b] = 2^{-k},$$

where k is the number of coin-flip steps that occur on branch b.



A Coin-Flipping (Probabilistic) TM

This is the low-level model …

… but most probabilistic TM definitions just say “**randomly select …**”

DEFINITION

A ***probabilistic Turing machine*** M is a type of nondeterministic Turing machine in which each nondeterministic step is called a ***coin-flip step*** and has two legal next moves. We assign a probability to each branch b of M's computation on input w as follows. Define the probability of branch b to be

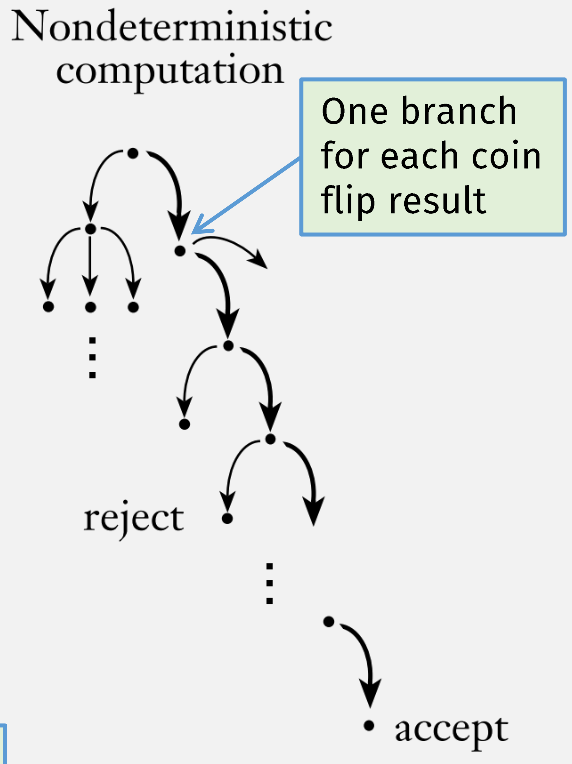
$$\Pr[b] = 2^{-k},$$

where k is the number of coin-flip steps that occur on branch b. Define the probability that M accepts w to be

$$\Pr[M \text{ accepts } w] = \sum_{\substack{b \text{ is an} \\ \text{accepting branch}}} \Pr[b].$$

Sum probability of all accepting branches

$$\Pr[M \text{ rejects } w] = 1 - \Pr[M \text{ accepts } w]$$



A Probabilistic TM Example

$PRIMES = \{n \mid n \text{ is a prime number in binary}\}$

$PRIME$ = "On input p:

1. If p is even, *accept* if $p = 2$; otherwise, *reject*.
2. Select $a_1, \ldots, a_k$ randomly in $\mathcal{Z}_p^+$.
3. For each i from 1 to k:
4. Compute a_i^{p-1} mod p and *reject* if different from 1.
5. Let $p - 1 = s \cdot 2^l$ where s is odd.
6. Compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, a_i^{s \cdot 2^2}, \ldots, a_i^{s \cdot 2^l}$ modulo p.
7. If some element of this sequence is not 1, find the last element that is not 1 and *reject* if that element is not -1.
8. All tests have passed at this point, so *accept*."

Probabilistic TM: Chance of Wrong Answer

Error Rate
(can depend on length of input $\boldsymbol{n}$)

M decides language A with error probability $\boldsymbol{\epsilon}$ if

1. $w \in A$ implies $\Pr[M \text{ accepts } w] \geq 1 - \epsilon$, and

Probabilistic TM: Chance of Wrong Answer

Error Rate
(can depend on length of input $\boldsymbol{n}$)

M decides language A with error probability $\boldsymbol{\epsilon}$ if

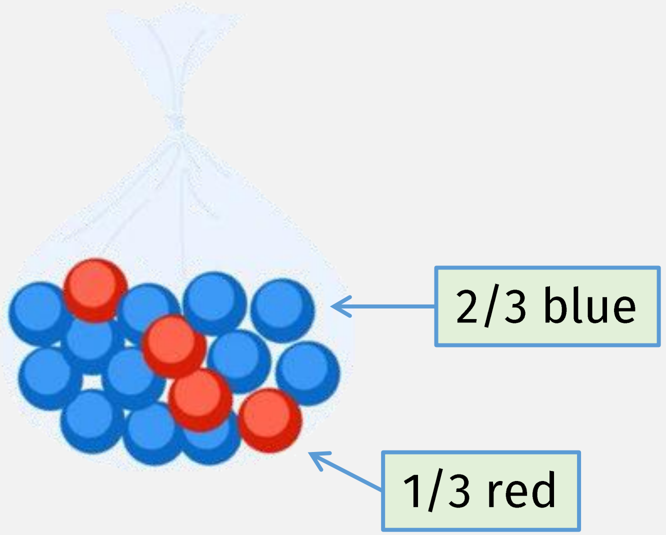
1. $w \in A$ implies $\Pr[M \text{ accepts } w] \geq 1 - \epsilon$, and

2. $w \notin A$ implies $\Pr[M \text{ rejects } w] \geq 1 - \epsilon$.

Balls in a Jar Analogy

Goal: determine the majority color of balls in a jar

Example Input:
J has 2/3 blue and 1/3 red balls
Error rate ϵ = 1/3



Probabilistic Algorithm = On input J, where J is a jar of balls:

- Randomly choose a ball from J
- Return the color of the chosen ball

BPP Complexity Class

DEFINITION

BPP is the class of languages that are decided by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$.

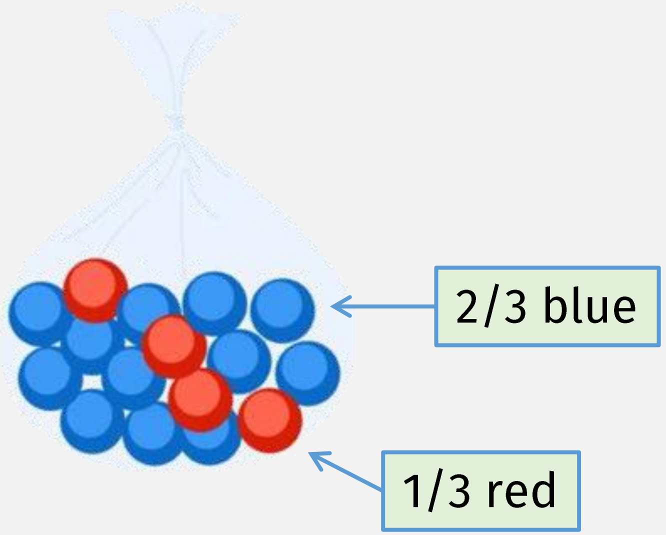
Count worst case # steps in any one branch (like NTM)

Arbitrary constant (anything between 0 and 0.5 works)

Balls in a Jar Analogy: Reducing Error

Goal: determine the majority color of balls in a jar

Example:
J has 2/3 blue and 1/3 red balls
Error rate ϵ =
P[choosing ≥ 5 red balls in 9 tries]



Probabilistic Algorithm = On input J, where J is a jar of balls:

- Randomly choose **9 balls** from J
- Return the majority color

Law of Large Numbers

Law of large numbers

From Wikipedia, the free encyclopedia

In probability theory, the **law of large numbers** (**LLN**) is a theorem that describes the result of performing the same experiment a large number of times. According to the law, the average of the results obtained from a large number of trials should be close to the expected value and will tend to become closer to the expected value as more trials are performed.[1]

Amplification Lemma

Let ϵ be a fixed constant strictly between 0 and $\frac{1}{2}$. Then for any polynomial $p(n)$, a probabilistic polynomial time Turing machine M_1 that operates with error probability ϵ has an equivalent probabilistic polynomial time Turing machine M_2 that operates with an error probability of $2^{-p(n)}$.

Convert an M_1 to M_2 with less error

PROOF IDEA M_2 simulates M_1 by running it a polynomial number of times and taking the majority vote of the outcomes. The probability of error decreases exponentially with the number of runs of M_1 made.

Amplification Lemma

Let ϵ be a fixed constant strictly between 0 and $\frac{1}{2}$. Then for any polynomial $p(n)$, a probabilistic polynomial time Turing machine M_1 that operates with error probability ϵ has an equivalent probabilistic polynomial time Turing machine M_2 that operates with an error probability of $2^{-p(n)}$.

PROOF Given TM M_1 deciding a language with an error probability of $\epsilon < \frac{1}{2}$ and a polynomial $p(n)$, we construct a TM M_2 that decides the same language with an error probability of $2^{-p(n)}$.

M_2 = "On input x:

1. Calculate k (see analysis below).
2. Run $2k$ independent simulations of M_1 on input x.
3. If most runs of M_1 accept, then *accept*; otherwise, *reject*."

Amplification Lemma: k

PROOF Given TM M_1 deciding a language with an error probability of $\epsilon < \frac{1}{2}$ and a polynomial $p(n)$, we construct a TM M_2 that decides the same language with an error probability of $2^{-p(n)}$.

M_2 = "On input x:
1. Calculate k (see analysis below).
2. Run $2k$ independent simulations of M_1 on input x.
3. If most runs of M_1 accept, then *accept*; otherwise, *reject*."

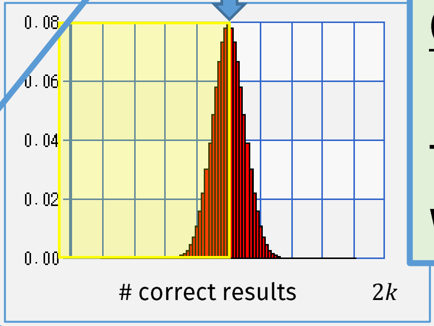
If M_1 is run $2k$ times (err ϵ), let $w + c = 2k$ where:

- c = # correct results
- w = # wrong results

Probability of this run: $\epsilon^w(1-\epsilon)^c$

$w = c = k$

$\epsilon < ½$, so $\epsilon < 1-\epsilon$



Conclusion:
If M_1 runs in poly time, then M_2 runs in poly time, with much smaller error

Wrong results:

Want: $\Pr[\text{wrong result}] \le 2^{-p(n)}$

- A run's result is wrong when: $w \ge c$
- Overall, Pr[wrong result]
  $= \sum_{w,c} \Pr[\text{run where } w \ge c] = \sum_{w,c} \epsilon^w(1-\epsilon)^c$
- Most likely wrong result: $w = c = k$
- Pr[wrong result]
  $\le \sum \epsilon^k(1-\epsilon)^k = 2^{2k}\epsilon^k(1-\epsilon)^k = (4\epsilon(1-\epsilon))^k$

2^{2k} = # combinations of w and c

Chernoff bound

Solve for k:

- $(4\epsilon(1-\epsilon))^k = 2^{-p(n)}$
- $k = \log_{(4\epsilon(1-\epsilon))} 2^{-p(n)}$ (log both sides)
  $= \log_2 2^{-p(n)} / \log_2(4\epsilon(1-\epsilon))$
  $= -p(n)/\log_2(4\epsilon(1-\epsilon))$

$\log_a b = \log_c a / \log_c b$

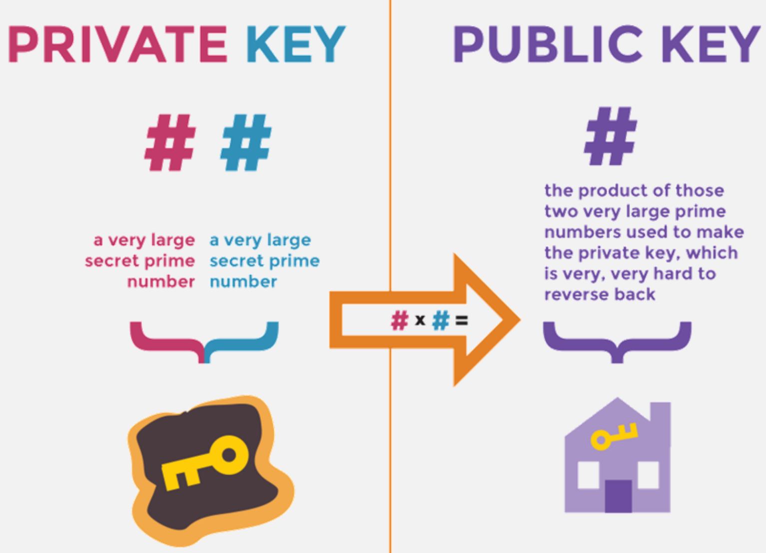
Prime Numbers

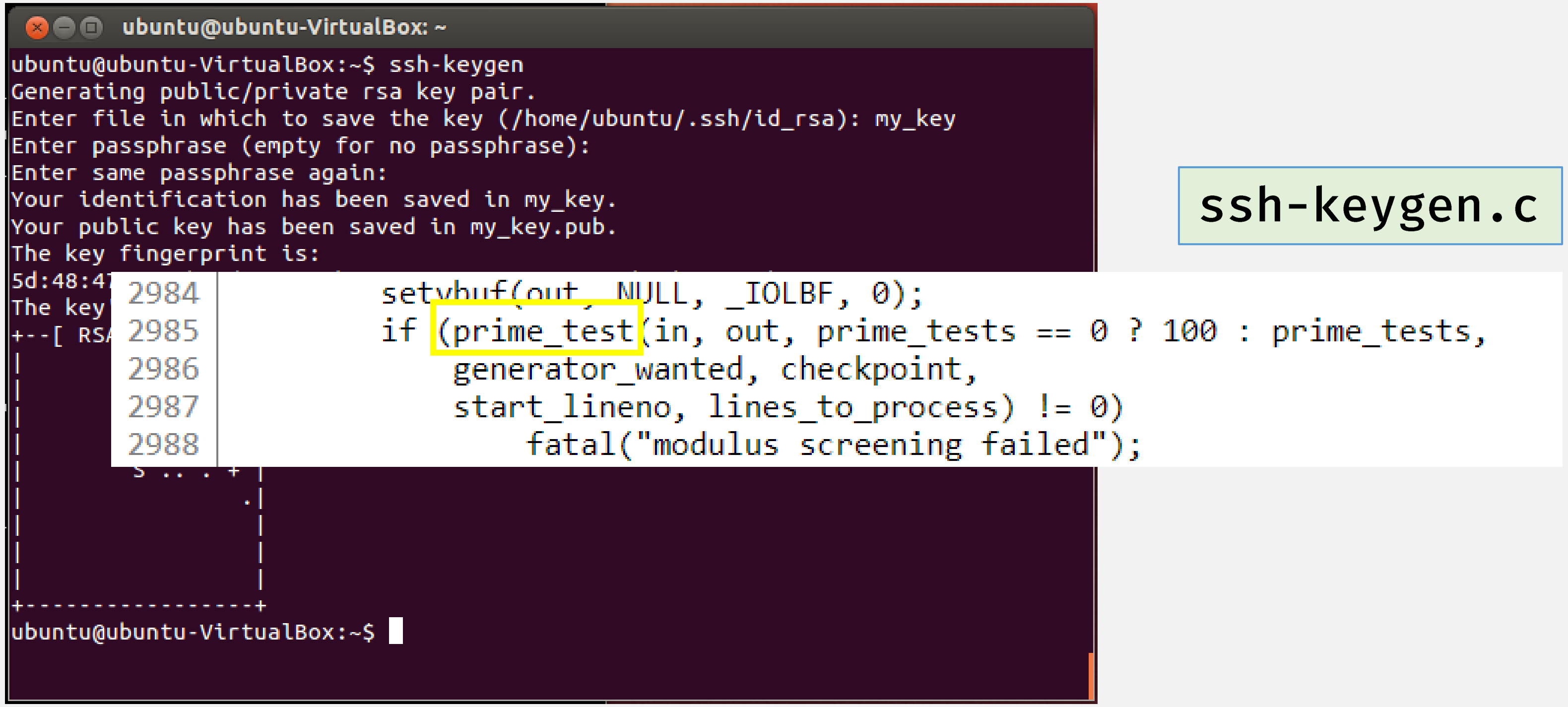
- A **prime number** is an integer > 1 with factors 1 and itself
- A **composite** number is a nonprime > 1

- Extremely important in cryptography, e.g., generating keys

Primality: Applications

- Cryptography impossible without an efficient primality test





ssh-keygen.c

Primality Test Algorithms

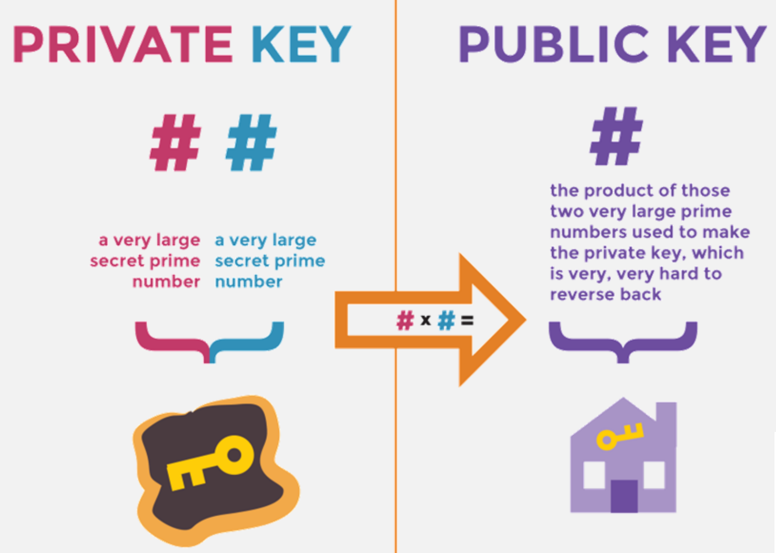
- EXPTIME: Try all possible factors
- POLYTIME: AKS algorithm (discovered in 2004)
  - Long and difficult to understand
  - $O(\log^{12}(n))$
- Probabilistic POLYTIME: Miller-Rabin, Solovay-Strassen
  - Simple(r) to understand
  - And more efficient!

Note:

- poly time primality tests don't seach for factors
- (so factoring still not poly time)

Primality: Applications

- Cryptography impossible without an efficient primality test




```
ubuntu@ubuntu-VirtualBox: ~
ubuntu@ubuntu-VirtualBox:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa): my_key
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in my_key.
Your public key has been saved in my_key.pub.
The key fingerprint is:
```

ssh-keygen.c

```
        setvbuf(out, NULL, _IOLBF, 0);
        if (prime_test(in, out, prime_tests == 0 ? 100 : prime_tests,
            generator_wanted, checkpoint,
            start_lineno, lines_to_process) != 0)
                fatal("modulus screening failed");
```

```
/*
 * perform a Miller-Rabin primality test
 * on the list of candidates
 * (checking both q and p)
 * The result is a list of so-call "safe" primes
 */
int
prime_test(FILE *in, FILE *out, u_int32_t trials, u_int32_t generator_wanted,
    char *checkpoint_file, unsigned long start_lineno, unsigned long num_lines)
{
```

Miller-Rabin Probabilistic Primality Test

$PRIMES = \{n \mid n \text{ is a prime number in binary}\}$

$PRIME$ = "On input p:

1. If p is even, *accept* if $p = 2$; otherwise, *reject*.
2. Select $a_1, \ldots, a_k$ randomly in $\mathcal{Z}_p^+$.
3. For each i from 1 to k:
4. Compute $a_i^{p-1} \bmod p$ and *reject* if different from 1.
5. Let $p - 1 = s \cdot 2^l$ where s is odd.
6. Compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, a_i^{s \cdot 2^2}, \ldots, a_i^{s \cdot 2^l}$ modulo p.
7. If some element of this sequence is not 1, find the last element that is not 1 and *reject* if that element is not -1.
8. All tests have passed at this point, so *accept*."

Fermat's Little Theorem

???

Primality "tests" (comes from number theory)

Fermat's Little Theorem

THEOREM

If p is prime and $a \in \mathcal{Z}_p^+$, then $a^{p-1} \equiv 1 \pmod{p}$.

Primality "test"

Modular Equivalence

<u>Definition:</u>

- Written: $x \equiv y \pmod{p}$
- Two numbers x and y are "equivalent (or congruent) modulo p" if ...
- ... $x - y = kp$, for some k
  - i.e., the difference is a multiple of p
- ... $x \bmod p = y \bmod p$
  - i.e., they have the same remainder when divided by p

<u>Example</u>

- $38 \equiv 14 \pmod{12}$
- Because: 38 - 14 = 24 = 2·12
- Or because: 38/12 has remainder 2, and 14/12 has remainder 2

For <u>every number</u> x, $x \equiv$ some $y \pmod{p}$ where $y \in \mathcal{Z}_p = \{0, \ldots, p-1\}$

Fermat's Little Theorem

$\mathcal{Z}_p = \{0, \ldots, p-1\}$
$\mathcal{Z}_p^+ = \{1, \ldots, p-1\}$

Alternatively, $a^{p-1}-1$ is divisible by p

THEOREM

If p is prime and $a \in \mathcal{Z}_p^+$, then $a^{p-1} \equiv 1 \pmod{p}$.

Must be true for all a

Primality "test", given number x:
- Contrapositive (true): if $a^{x-1}-1$ is not divisible by x, then x is ...
  ... not prime!
- Converse (not always true): if $a^{x-1}-1$ is divisible by x, then x is ...
  ... maybe prime? (called a pseudoprime!)

Fermat's Little Theorem

THEOREM

If p is prime and $a \in \mathcal{Z}_p^+$, then $a^{p-1} \equiv 1 \pmod{p}$.

$\mathcal{Z}_p^+ = \{1, \ldots, p-1\}$

Example # 1

- $p = 7$ (prime)
- $\forall a \in \{1, ..., 6\}$, $a^{p-1}-1$ is divisible by 7
- E.g., if $a = 2$,
  - $2^{7-1}-1 = 2^6-1 = 64-1 = 63 = 7 \cdot 9$

Example # 3 (converse)

- $p = 15$ (composite)
- If $a = 4$
  - $4^{15-1}-1 = 4^{14}-1 = 268{,}435{,}455$
  - $268{,}435{,}455 / 15 = 17{,}895{,}697$
- So 15 passes the primality "test" but is not prime!

Example # 2 (contrapositive)

- $p = 6$ (composite)
- if $a = 2$
  - $2^{6-1}-1 = 2^5-1 = 32-1 = 31$
- 31 is not divisible by 6 so 6 is not prime

Pseudoprime Algorithm

THEOREM

If p is prime and $a \in \mathcal{Z}_p^+$, then $a^{p-1} \equiv 1 \pmod{p}$.

$$\mathcal{Z}_p^+ = \{1, \ldots, p-1\}$$

Checking all a_i takes exponential time, so randomly sample instead

PSEUDOPRIME = "On input p:

1. Select $a_1, \ldots, a_k$ randomly in $\mathcal{Z}_p^+$.
2. Compute $a_i^{p-1} \bmod p$ for each i.
3. If all computed values are 1, *accept*; otherwise, *reject*."

If machine rejects, then $a_i^{p-1} \not\equiv 1 \pmod{p}$ for some a_i

- So p is composite (a_i is a "compositeness witness")
- Error rate: 0%

If machine accepts, then $a_i^{p-1} \equiv 1 \pmod{p}$ for all a_i

- p could be composite or prime
- Error Rate:
  - depends on $\Pr[p$ is a non-prime pseudoprime$]$

Need another primality "test"

Too high!

Miller-Rabin Probabilistic Primality Test

$PRIMES = \{n \mid n \text{ is a prime number in binary}\}$

Primality "test" #2

$PRIME$ = "On input p:

1. If p is even, *accept* if $p = 2$; otherwise, *reject*.
2. Select $a_1, \ldots, a_k$ randomly in $\mathcal{Z}_p^+$.
3. For each i from 1 to k:
4. Compute $a_i^{p-1} \bmod p$ and *reject* if different from 1.
5. Let $p - 1 = s \cdot 2^l$ where s is odd.
6. Compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, a_i^{s \cdot 2^2}, \ldots, a_i^{s \cdot 2^l}$ modulo p.
7. If some element of this sequence is not 1, find the last element that is not 1 and *reject* if that element is not -1.
8. All tests have passed at this point, so *accept*."

Primality Test #2: Modular Square Root

If $r^2 \equiv a \pmod{p}$ …

… then r is a "modular square root" of $a \pmod{p}$

- If p is prime …
  - … then the modular square root of $1 \pmod{p}$ = 1 or -1
- If p is a composite pseudoprime…
  - … then $1 \pmod{p}$ has ≥ 4 possible modular square roots

<u>Example</u>

- Modular square root of $1 \pmod{15}$ = 1 or -1 or 4 or -4

Fermat Test + Modular Square Root

- If p is prime, modular sqrt of $1 \pmod{p} = 1$ or -1
- If p is a composite pseudoprime, $1 \pmod{p}$ has ≥ 4 sqrts

If $a^{p\text{-}1} \equiv 1 \pmod{p}$ (from Fermat test), then modular sqrt = $a^{(p\text{-}1)/2}$

- If sqrt = 1, keep taking square root, because $a^{(p\text{-}1)/2} = 1 \pmod{p}$
  - i.e., keep dividing exponent by 2
- If sqrt = -1, consider test "passed"
  - i.e., number is prime
- If sqrt ≠ ±1, reject

Computing modular square root:

- Let $p\text{-}1 = s2^d$
- Then modular square root of $a^{(p\text{-}1)} = a^{s2\text{^}d} = a^{s2\text{^}(d\text{-}1)}$ (keep decreasing power of 2)

Miller-Rabin Probabilistic Primality Test

$PRIMES = \{n | \, n \text{ is a prime number in binary}\}$

$PRIME$ = "On input p:

1. If p is even, *accept* if $p = 2$; otherwise, *reject*.
2. Select $a_1, \ldots, a_k$ randomly in $\mathcal{Z}_p^+$.
3. For each i from 1 to k:
4. Compute a_i^{p-1} mod p and *reject* if different from 1.
5. Let $p - 1 = s \cdot 2^l$ where s is odd.
6. Compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, a_i^{s \cdot 2^2}, \ldots, a_i^{s \cdot 2^l}$ modulo p.
7. If some element of this sequence is not 1, find the last element that is not 1 and *reject* if that element is not -1.
8. All tests have passed at this point, so *accept*."

modular exponentiation is poly time

Repeated squaring is poly time

So this machine runs in (probabilistic) poly time

First compute Fermat's test, so $a_i^{p\text{-}1}$ mod p = 1

Then compute (repeated) sqrt, reject if ≠ ±1

If both tests pass for all a_i, then accept as prime

$PRIMES \in \mathbf{BPP}$

DEFINITION

BPP is the class of languages that are decided by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$.

***M* decides language *A* with error probability ϵ** if

1. $w \in A$ implies $\Pr[M \text{ accepts } w] \geq 1 - \epsilon$, and
2. $w \notin A$ implies $\Pr[M \text{ rejects } w] \geq 1 - \epsilon$.

PRIME = "On input p:

1. If p is even, *accept* if $p = 2$; otherwise, *reject*.
2. Select $a_1, \ldots, a_k$ randomly in $\mathcal{Z}_p^+$.
3. For each i from 1 to k:
4. Compute $a_i^{p-1} \bmod p$ and *reject* if different from 1.
5. Let $p - 1 = s \cdot 2^l$ where s is odd.
6. Compute the sequence $a_i^{s\cdot 2^0}, a_i^{s\cdot 2^1}, a_i^{s\cdot 2^2}, \ldots, a_i^{s\cdot 2^l}$ modulo p.
7. If some element of this sequence is not 1, find the last element that is not 1 and *reject* if that element is not -1.
8. All tests have passed at this point, so *accept*."

All $a_i^{p-1} \bmod p = 1$ (Fermat)

And sqrt $a_i^{p-1} = \pm 1$

If p is an odd prime number, $\Pr\left[\, PRIME \text{ accepts } p \,\right] = 1$.

$PRIMES \in$ **BPP**

DEFINITION

BPP is the class of languages that are decided by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$.

M decides language A with error probability ϵ if

1. $w \in A$ implies $\Pr[M \text{ accepts } w] \geq 1 - \epsilon$, and

2. $w \notin A$ implies $\Pr[M \text{ rejects } w] \geq 1 - \epsilon$.

PRIME = "On input p:

1. If p is even, *accept* if $p = 2$; otherwise, *reject*.
2. Select $a_1, \ldots, a_k$ randomly in $\mathcal{Z}_p^+$.
3. For each i from 1 to k:
4. Compute $a_i^{p-1} \bmod p$ and *reject* if different from 1.
5. Let $p - 1 = s \cdot 2^l$ where s is odd.
6. Compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, a_i^{s \cdot 2^2}, \ldots, a_i^{s \cdot 2^l}$ modulo p.
7. If some element of this sequence is not 1, find the last element that is not 1 and *reject* if that element is not -1.
8. All tests have passed at this point, so *accept*."

$\Pr[\, a \text{ is a witness} \,] \geq \frac{1}{2}$

If p is an odd composite number, $\Pr[\, \textit{PRIME} \text{ accepts } p \,] \leq 2^{-k}$

$$\Pr[\, a \text{ is a witness} \,] \geq \frac{1}{2}$$

- More Number Theory!
  - Chinese Remainder Theorem!

- Sipser shows how to find a real witness for every false witness
  - So $\epsilon \leq 1/2$

- Actual error rate of Miller-Rabin: $\epsilon \leq 1/4$

$PRIMES \in \mathbf{BPP}$

DEFINITION

BPP is the class of languages that are decided by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$.

M decides language A with error probability ϵ if

1. $w \in A$ implies $\Pr[M \text{ accepts } w] \geq 1 - \epsilon$, and

2. $w \notin A$ implies $\Pr[M \text{ rejects } w] \geq 1 - \epsilon$.

PRIME = "On input p:

1. If p is even, *accept* if $p = 2$; otherwise, *reject*.
2. Select $a_1, \ldots, a_k$ randomly in $\mathcal{Z}_p^+$.
3. For each i from 1 to k:
4. Compute $a_i^{p-1} \bmod p$ and *reject* if different from 1.
5. Let $p - 1 = s \cdot 2^l$ where s is odd.
6. Compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, a_i^{s \cdot 2^2}, \ldots, a_i^{s \cdot 2^l}$ modulo p.
7. If some element of this sequence is not 1, find the last element that is not 1 and *reject* if that element is not -1.
8. All tests have passed at this point, so *accept*."

If p is composite, then a randomly selected a_i will be a witness 75% of the time

If p is an odd prime number, $\Pr\big[\textit{PRIME} \text{ accepts } p\big] = 1$.

If p is an odd composite number, $\Pr\big[\textit{PRIME} \text{ accepts } p\big] \leq 2^{-k}$

1-sided error

RP

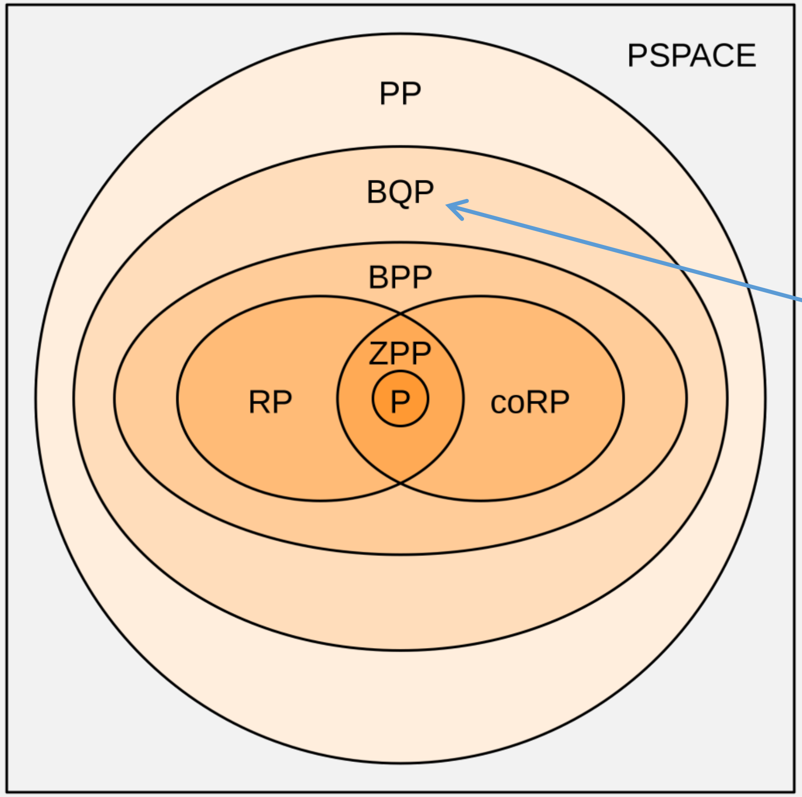
DEFINITION

RP is the class of languages that are decided by probabilistic polynomial time Turing machines where inputs in the language are accepted with a probability of at least $\frac{1}{2}$, and inputs not in the language are rejected with a probability of 1.

One-sided error, like *PRIMES*

So *PRIMES* ∈ **RP**

Probabilistic Complexity Classes



Quantum computing (quantum TM) version of **BPP**

It's unknown if any of these containments are strict!

No Quiz 12/13

Thank You For a Great Semester!