Lecture 18 Primality Testing

Let us recall Miller-Rabin algorithm. The input is an odd integer $n \geq 3$.

Step 0. Check if $n = a^b$ for integers $a, b \geq 2$.
If so then return "composite".

Step 1. Select $a \in \{1, \ldots, n-1\}$ uniformly at random.
Compute $a^{n-1} \bmod n$. If this is not 1
then return "composite".

Step 2 Let $n-1 = 2^k \cdot t$ where t is odd.
Compute $a^t, a^{2t}, a^{4t}, a^{8t}, \ldots$ till a 1 is seen.
(mod n) — recall that $a^{2^k t} = 1 \pmod n$.

If the number before 1 is not -1
then return "composite"; else return "prime".

→ In case $a^t = 1 \pmod n$ then there is no number before 1 ⇝ so we will return "prime".

Observe that whenever the algo. returns "composite", the number n is composite. The algo. may make a mistake by calling a composite number "prime" but it never calls a prime number "composite".

We will show that the prob. of calling a composite number "prime" is $\leq 1/2$. So by repeating the algo. twice & returning "prime" only if both times the algo. says "prime", the error probability becomes $\leq 1/4$.

The most non-trivial case is when n is Carmichael. That is, $\forall\, a \in \mathbb{Z}_n^* : a^{n-1} = 1 \pmod n$.

Consider the following table whose rows are indexed by the elements of $\mathbb{Z}_n^*$.

- Its columns are indexed by $t, 2t, 4t, 8t, \ldots$
- Recall that the algorithm computes $\underbrace{a^t, a^{2t}, a^{4t}, a^{8t}, \ldots}_{\text{modulo } n}$ where a is the element in $\{1, \ldots, n-1\}$ picked in Step 1.
- The entries in this table are $\underbrace{a^t, a^{2t}, a^{4t}, \ldots, a^{2^k t}}_{\text{modulo } n}$ for all $a \in \mathbb{Z}_n^*$. Recall that $n-1 = 2^k \cdot t$ and $a^{n-1} = 1 \pmod n$ $\forall a \in \mathbb{Z}_n^*$.

	t	$2t$	$4t$	$\ldots$	$2^h \cdot t$	$2^{h+1} \cdot t$		$2^k \cdot t$
a_1						1		1
a_2						1		1
						$\vdots$		$\vdots$
a_r						1		1

Let $a^{2^{h+1} \cdot t} = 1 \pmod n$ $\forall a \in \mathbb{Z}_n^*$, however it is not the case that $a^{2^h \cdot t} = 1 \pmod n$ for all $a \in \mathbb{Z}_n^*$. Why does such an h exist?

Claim. For at least half the elements x in $\mathbb{Z}_n^*$, it is the case that $x^{2^h \cdot t} \neq \pm 1 \pmod n$.

We will first show that there exists at least 1 element $b \in \mathbb{Z}_n^*$ such that $b^{2^h \cdot t} \neq \pm 1 \pmod n$.

Since the column in the above table that corresponds to h_{Date} is not all 1's, if there is no non ± 1 entry, then there has to be a -1 in this column.

That is, $\exists\ c \in \mathbb{Z}_n^*$ such that $c^{2^h \cdot t} = -1 \pmod n$.

Let $f(c) = (c_1, c_2)$ ⇝ recall the isomorphism f between $\mathbb{Z}_n^*$ and $\mathbb{Z}_\alpha^* \times \mathbb{Z}_\beta^*$ where $n = \alpha \cdot \beta$

So $f(c^{2^h \cdot t}) = (c_1^{2^h \cdot t}, c_2^{2^h \cdot t})$

$\| $

$f(-1) = (-1, -1)$. So $c_1^{2^h \cdot t} = -1 \pmod \alpha$

and $c_2^{2^h \cdot t} = -1 \pmod \beta$.

Consider $f^{-1}(c_1, 1)$. Call this element d.

$f(d) = (c_1, 1)$. What is $d^{2^h \cdot t} \pmod n$?

$f(d^{2^h \cdot t}) = (c_1^{2^h \cdot t}, 1) = (-1, 1)$.

Thus $\exists\ d \in \mathbb{Z}_n^*$ such that $d^{2^h \cdot t} \neq \pm 1 \pmod n$.

So $d^{2^h \cdot t} \neq \pm 1 \pmod n$ since $f(1) = (1, 1)$ and $f(-1) = (-1, -1)$.

Now we need to amplify this to show that for at least half the elements $x \in \mathbb{Z}_n^*$, we have $x^{2^h \cdot t} \neq \pm 1 \pmod n$.

– We will use Lagrange's theorem.

$\mathbb{Z}_n^*$ is a group wrt multiplication mod n.

Define $H = \{a \in \mathbb{Z}_n^* : a^{2^h \cdot t} = \pm 1 \pmod n\}$.

Check that H is a subgroup of $\mathbb{Z}_n^*$.

So $|H|$ divides $|\mathbb{Z}_n^*|$.

However $|H| \neq |\mathbb{Z}_n^*|$ since $d \notin H$.

Thus $|H| \leq \frac{|\mathbb{Z}_n^*|}{2}$. Hence there are $\geq \frac{|\mathbb{Z}_n^*|}{2}$ elements outside H.

So whenever we pick an element $a \in \mathbb{Z}_n^* \setminus H$ in Step 2, the algo. returns "composite".

Thus for Carmichael n, the prob. of saying "prime" is $\leq 1/2$.

What is the running time of Miller-Rabin algorithm? Easy to see that Steps 1 & 2 take poly(log n) time.

Exercise. Show that Step 0 can be implemented in poly(log n) time.

We have seen 2 Monte Carlo algorithms so far - the global min-cut algo. & the primality testing algo. Let us see a Las Vegas algorithm now.

Randomized Quick Sort

We know how to sort n numbers in $O(n \log n)$ time. For instance, the following algorithm (this is Quicksort) does so:

- find the median in $O(n)$ time.
- partition the input set S into 2 sets:
  $S_1 = \{\text{numbers} < \text{median}\}$
  $S_2 = \{\text{numbers} > \text{median}\}$
  * Recursively sort S_1 and S_2. Output the elements of S_1 in sorted order, followed by all occurrences of the median, followed by the sorted order of S_2.

The running time of QS is given by:

$$T(n) \leq 2T(n/2) + cn \quad \text{for } n \geq 2$$
$$T(1) = O(1)$$

Note that we used the fact that $|S_1| \leq n/2$ and $|S_2| \leq n/2$. $T(n) = O(n \log n)$.

How about a randomized version of the above algo?

RandQS: Rather than explicitly find the median, choose an element of S uniformly at random as the pivot.

Rand QS : This is exactly the same as the earlier algo. except that instead of computing the median, we ~~xxxx~~ choose an element uniformly at random from S as the pivot.

$S_1 = \{$ numbers $<$ pivot $\}$

$S_2 = \{$ numbers $>$ pivot $\}$.

and so on.

Analysis : The output is always correct. We always get S in sorted order. What changes from one invocation of this algo. to another invocation is the running time of the algo. Or more explicitly, the number of comparisons made by the algorithm.

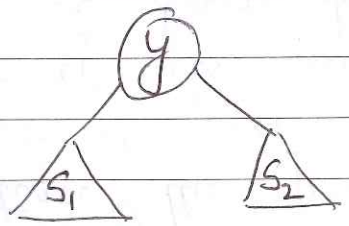
Define $X_{ij} = \begin{cases} 1 & \text{if } S[i] \text{ and } S[j] \text{ get compared in the algorithm} \\ 0 & \text{otherwise} \end{cases}$ → the i-th elmt in sorted order

The total number of comparisons made is $\sum_{i<j} X_{ij}$.

What we look to bound here is the expected running time of the algo. or the expected number of comparisons made by the algo. which is $E[\sum_{i<j} X_{ij}] = \sum_{i<j} E[X_{ij}]$

$E[X_{ij}] = p_{ij}$ where p_{ij} is the probability that $S[i]$ & $S[j]$ are compared in our algo.

Visualize the execution of our algo. as a binary tree.



Here y is the pivot.

Observe that no comparison is made between an element of the left subtree and an element of the right subtree.

Thus there is a comparison between $S[i]$ and $S[j]$ if and only if one of these elements is an ancestor of the other in the above tree.

The probability that $S[i]$ and $S[j]$ are compared = the probability that the first pivot chosen among $\{S[i], S[i+1], \ldots, S[j]\}$ is either $S[i]$ or $S[j]$.

The probability of this event $= \frac{2}{j-i+1}$

since every element in this $(j-i+1)$-sized set is equally likely to be chosen as the first pivot from this set.

Hence the expected number of comparisons

$$= \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j-i+1} \leq \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k}$$

$$= 2n H_n = 2n \ln n$$

Hashing

- The static dictionary problem: We are given a set S of keys and we must organize S into a data structure that supports FIND queries efficiently.

- The dynamic dictionary problem: The set S is not provided in advance. Instead it is constructed by a series of INSERT and DELETE operations that are intermingled with FIND queries.

These problems can be solved using balanced search trees and such data structures. For a set S, these data structures require $\Omega(\log |S|)$ time to process any search or update operation. These time bounds are optimal in the sense that for data structures based on pointers and search trees, there is a lower bound of $\Omega(\log |S|)$.

Date

In the next lecture we will see a new approach to avoid this lower bound and achieve $O(1)$ search time.