CS271 Randomness & Computation **Spring 2020**

Lecture 3: January 28

Instructor: Alistair Sinclair

3.1 Fingerprinting

Suppose we have a large, complex piece of data that we seek to send over a long communication channel which is both error-prone and expensive. In such situations, we would like to shrink our data down to a much smaller *fingerprint* which is easier to transmit. Of course, this is useful only if the fingerprints of different pieces of data are unlikely to be the same.

We model this situation with two spatially separated parties, Alice and Bob, each of whom holds an n-bit number (where n is very large). Alice's number is $a = a_0a_1 \dots a_{n-1}$ and Bob's is $b = b_0b_1 \dots b_{n-1}$. Our goal is to decide if $a = b$ without transmitting all n bits of the numbers between the parties.

Algorithm

Alice picks a prime number p u.a.r. from the set $\{2, \dots, T\}$, where T is a value to be determined. She computes her fingerprint as $F_p(a) = a \bmod p$. She then sends p and $F_p(a)$ to Bob. Using p, Bob computes $F_p(b) = b \bmod p$ and checks whether $F_p(b) = F_p(a)$. If not he concludes that $a \neq b$, else he presumes that $a = b$.

Observe that if $a = b$ then Bob will always be correct. However, if $b \neq a$ then there may be an error: this happens iff the fingerprints of a and b happen to coincide. We now show that, even for a modest value of T (exponentially smaller than a and b), if $a \neq b$ then $\Pr[F_p(a) = F_p(b)]$ is small.

First observe that, if $F_p(a) = F_p(b)$, then $a = b \bmod p$, so p must divide $|a - b|$. But $|a - b|$ is an n-bit number, so the number of primes p that divide it is (crudely) at most n (each prime is at least 2). Thus the probability of error is at most $\frac{n}{\pi(T)}$, where $\pi(x)$ is defined as the number of primes less than or equal to x.

We now appeal to a standard result in Number Theory:

Theorem 3.1 [Prime Number Theorem]

$$\pi(x) \sim \frac{x}{\ln x} \quad \textit{as } x \to \infty.$$

Moreover,

$$\frac{x}{\ln x} \leq \pi(x) \leq 1.26 \frac{x}{\ln x} \quad \forall x \geq 17.$$

Thus we may conclude that

$$\Pr[\text{error}] \leq \frac{n}{\pi(T)} \leq \frac{n \ln T}{T}.$$

Picking $T = cn \ln n$ for a constant c gives $\Pr[\text{error}] \leq \frac{1}{c} + o(1)$.

Actually, we can improve this analysis slightly, using another fact from Number Theory: the number of primes that divide any given n-bit number is at most $\pi(n)$. Thus we get the improved bound

$$\Pr[\text{error}] \le \frac{\pi(n)}{\pi(T)} \le 1.26\frac{n\ln T}{T\ln n}.$$

Setting $T = cn$ gives us an error probability of only

$$\frac{1.26}{c}\left(1+\frac{\ln c}{\ln n}\right),$$

which is small even for modest c. (As usual, this can be improved further by running the algorithm repeatedly, i.e., sending multiple independent fingerprints.)

The numbers transmitted in the above protocol are integers mod p, where $p = O(n)$. Hence the number of bits transmitted is only $O(\log n)$, an exponential improvement over the deterministic scenario.

Example: If $n = 2^{33}$ ($\approx$ 1 gigabyte) and $T = 2^{64}$ (so that fingerprints are the size of a 64-bit word), then

$$\Pr[\text{error}] \le 1.26\frac{n\ln T}{T\ln n} = 1.26\cdot\frac{2^{33}}{2^{64}}\cdot\frac{64}{33} \approx 1.25\times 10^{-9}.$$

The above protocol requires us to pick a random prime in $[0, T]$. A simple algorithm for this is to pick a random number in the interval and check if it is a prime. The prime number theorem tells us that we will only need to test approximately $\ln T$ numbers before we find a prime. Later, we shall see how to check efficiently whether a given number is a prime; this will actually involve another important use of randomization.

We now look at some applications of the fingerprinting technique.

3.2 Applications of Fingerprinting

3.2.1 Arithmetic modulo a prime

In Lecture 2 we saw an algorithm due to Schwartz and Zippel for testing polynomial identities. That algorithm involved evaluating a polynomial at a random point. If the polynomials are over the reals (rather than over a finite field), this may lead to very large intermediate values (even when the final result of the computation may be small). We can use the above fingerprinting method to do all the arithmetic modulo a small prime, at the cost of only a small probability of error; this additional error probability can be absorbed into the error of the Schwartz-Zippel algorithm itself. The same idea can be used in other applications to avoid computation on large numbers.

3.2.2 Pattern matching

Suppose that there is a long source text $X = x_1x_2...x_n$ and a shorter text $Y = y_1y_2...y_m$ with $m < n$, over some common alphabet, which for simplicity we assume is $\{0, 1\}$. Our goal is to determine whether or not Y occurs as a contiguous substring of X, i.e., whether $Y = X(j) \equiv x_jx_{j+1}...x_{j+m-1}$ for some j.

The standard deterministic algorithm that compares the pattern to the source text bit-by-bit until either a match is found or the end of the source is reached clearly runs in $O(mn)$ time.

There are clever deterministic algorithms due to Boyer/Moore [BM77] and Knuth/Morris/Pratt [KMP77] that run in $O(m + n)$ time, but they are difficult to implement and have a rather large overhead.

We now present a very simple and practical randomized algorithm, due to Karp and Rabin [KR81], that also runs in $O(m+n)$ time. This algorithm computes a fingerprint of Y, and compares it to the fingerprints of successive substrings of X.

```
pick a random prime $p \in [2, ..., T]$
compute $F_p(Y) = Y \bmod p$
**for** $j = 1$ **to** $n - m + 1$ **do**
    compute $F_p(X(j))$
    **if** $F_p(Y) = F_p(X(j))$ **then** output "match" and halt
output "no match"
```

Error

This algorithm has one-sided error: it may output "match" when there is in fact no match. Following the analysis of fingerprinting in the previous section, a simple upper bound on Pr[Error] is $n\frac{\pi(m)}{\pi(T)}$ by the union bound (as we are making n attempts at matching the m-bit pattern). We can do better by observing that, in order for a false match to occur somewhere along the string, p must divide $|Y - X(j)|$ for some j, and therefore p must divide $\prod_j |Y - X(j)|$, which is an mn-bit number. So, the bound on the error can be improved to

$$\Pr[\text{Error}] \leq \frac{\pi(mn)}{\pi(T)}.$$

Thus, if we choose $T = cmn$ for a reasonable constant c, we will get a small error probability.

Running Time

To find the running time of the algorithm, we first note that p has only $O(\log(mn)) = O(\log n)$ bits, so we may reasonably assume that arithmetic mod p can be performed in constant time. First, the algorithm computes $F_p(Y)$; since Y is an m-bit number, this requires $O(m)$ time.

We also need to compute $F_p(X(j))$ for all $j = 1, \ldots, n-m+1$. Doing this by brute force would require $O(nm)$ time. Instead of that, we note that $X(j)$ and $X(j+1)$ differ in only the first and last bits, so we have the following relationship:

$$X(j+1) = 2(X(j) - 2^{m-1}x_j) + x_{j+m}.$$

The fingerprint of $X(j+1)$ can thus be computed as follows:

$$F_p(X(j+1)) = 2(F_p(X(j)) - 2^{m-1}x_j) + x_{j+m} \bmod p.$$

This involves a constant number of arithmetic operations mod p, and hence takes constant time. The loop iterates n times, so the total running time is $O(m+n)$, as claimed earlier.

Notes

This algorithm can be converted to a Las Vegas one by checking that a match is correct before outputting it. Conceivably this could require $O(mn)$ time (if we found a huge number of false matches), but we would have to be very unlucky. In fact the expected running time is $O(n+m)$.

Example

Searching in a string of DNA: take $n = 2^{12}$, $m = 2^8$, and T to be the machine word size, 2^{32}.

$$\Pr[\text{Error}] \leq \frac{\pi(mn)}{\pi(T)} \leq 1.26\frac{mn\ln T}{T\ln mn} = 1.26\frac{2^{20}}{2^{32}}\frac{32}{20} \approx 0.0005.$$

3.2.3 1-tape Turing machines

Consider the language $L_{PAL} = \{x \in \{0,1\}^* : x = x^R\}$. By a classical result from automata theory, we know that no 1-tape Turing machine can recognize L_{PAL} in better than $\Theta(n^2)$ time.

A theorem proved by Freivalds [Fre77] states that L_{PAL} can be recognized by a *probabilistic* 1-tape Turing Machine in $O(n \ln n)$ time with one-sided error (and bounded error probability). The idea of the proof is to view the input string x as a binary integer, compute the fingerprints of x and x^R modulo a small prime (with $O(\log n)$ bits), and compare them. With some technical work, one can perform the necessary arithmetic in $O(n \log n)$ time on a single tape.

This theorem demonstrates a provable gap between randomized and deterministic algorithms, albeit in a very restricted computational model.

3.3 Primality Testing

There are many situations in which the primality of a given integer must be determined. For example, fingerprinting requires a supply of prime numbers, as does the RSA cryptosystem (where the primes should typically have hundreds or thousands of bits).

A theoretical breakthrough in 2002, due to Agrawal, Kayal and Saxena [AKS02], has given us a deterministic polynomial time algorithm for primality testing. However, in practice randomized algorithms are more efficient and continue to be used. These algorithms date back to the 1970's and caused a surge in the study of applied number theory.

3.3.1 A simple deterministic algorithm

Given an odd integer n, we wish to determine whether n is prime or composite. Consider the following deterministic algorithm:

```
for $a = 2, 3, ..., \lfloor\sqrt{n}\rfloor$ do
    if $a|n$ then output "composite" and halt
output "prime"
```

This algorithm is obviously correct. However, because the for-loop has $O(\sqrt{n})$ iterations, the algorithm does not have running time polynomial in the number of input bits (which is $O(\log n)$). (Consider the case where n is an integer with hundreds or even thousands of bits; then $\sqrt{n}$ is an enormous number as well!) Other, more sophisticated algorithms based on prime number sieves are a bit more efficient but still suffer from the same drawback.

3.3.2 A simple randomized algorithm

The above trivial algorithm can be turned into a randomized, witness-searching algorithm by picking a at random, but this has a potentially huge error probability since n will in general have only few divisors. We need to be more intelligent in our definition of witnesses.

Our first randomized algorithm is based on the following classical theorem:

Theorem 3.2 (Fermat's Little Theorem) *If p is prime, then $a^{p-1} = 1 \bmod p$ for all $a \in \{1, ..., p-1\}$.*

In particular, for a given integer n, if there exists an $a \in \{1, ..., n-1\}$ such that $a^{n-1} \neq 1 \bmod n$, then surely n is composite. This fact suggests the following algorithm, known as "Fermat's Test":

```
pick $a \in \{1, ..., n-1\}$ uniformly at random
if $\gcd(a, n) \neq 1$ then output "composite" and halt
else if $a^{n-1} \neq 1 \bmod n$ then output "composite"
else output "prime"
```

Computing $\gcd(a, n)$ can be done in time $O(\log n)$ by Euclid's algorithm, and a^{n-1} can be computed in $O(\log^2 n)$ time by repeated squaring, so this algorithm runs in time polynomial in the input size $(\log n)$.

Error

Clearly the algorithm is always correct when n is prime. However, when n is composite it may make an error if it fails to find a "witness," i.e., a number $a \in \mathbb{Z}_n^*$ such that $a^{n-1} \neq 1 \bmod n$. Unfortunately, there are composite numbers, known as "Carmichael numbers," that have no witnesses. The first three CN's are 561, 1105, and 1729. (**Exercise**: Prove that 561 is a CN. Hint: $561 = 3 \times 11 \times 17$; use the Chinese Remainder Theorem.) These numbers are guaranteed to fool Fermat's Test.

However, it turns out that CN's are the *only* bad inputs for the algorithm, as we now show. In what follows, we use the notation $\mathbb{Z}_n$ to denote the additive group of integers mod n, and $\mathbb{Z}_n^*$ the multiplicative group of integers coprime to n (i.e., with $\gcd(a, n) = 1$).

Theorem 3.3 *If n is composite and not a Carmichael number, then* $\Pr[\text{Error in the Fermat test}] \leq \frac{1}{2}$.

Proof: Let $S_n = \{a \in \mathbb{Z}_n^* : a^{n-1} = 1 \bmod n\}$, i.e., the set of bad choices for a. Clearly S_n is a subgroup of $\mathbb{Z}_n^*$ (because it contains 1 and is closed under multiplication). Moreover, it is a *proper* subgroup since n is not a CN and therefore there is at least one witness $a \notin S_n$. By Lagrange's Theorem, the size of any subgroup must divide the size of the group so we may conclude that $|S_n| \leq \frac{1}{2}|\mathbb{Z}_n^*|$. ■

Fortunately, CN's are rare: there are only 255 of them less than 10^8, and only a little more than 20 million of them less than 10^{21}. For this reason, Fermat's Test actually performs quite well in practice. Indeed, even the simplified deterministic version which performs the test only with $a = 2$ is sometimes used to produce "industrial grade" primes. This simplified version makes only 22 errors in the first 10,000 integers. It has also been proved for this version that

$$\lim_{b \to \infty} \Pr[\text{Error on random } b\text{-bit number}] \to 0.$$

For values of b of 50 and 100, we get $\Pr[\text{Error}] \leq 10^{-6}$ and $\Pr[\text{Error}] \leq 10^{-13}$ respectively. However, it is much more desirable to have an algorithm that does not have disastrous performance on any input (especially if the numbers we are testing for primality are not random).

In the next section, we will develop a more sophisticated randomized algorithm of similar flavor that can successfully handle all inputs (including Carmichael numbers).

3.4 A randomized algorithm for primality testing

The algorithm we present in this section is due independently to Miller and Rabin (see the notes at the end of the section for details).

First observe that, if p is prime, the group $\mathbb{Z}_p^*$ is cyclic: $\mathbb{Z}_p^* = \{g, g^2, \cdots g^{p-1} = 1\}$ for some $g \in \mathbb{Z}_p^*$. (Actually this holds slightly more generally, for $n \in \{1,2,4\}$ and for $n = p^k$ or $n = 2p^k$ where p is an odd prime and k is a non-negative integer.) Note that then $\mathbb{Z}_p^* \cong \mathbb{Z}_{p-1}$. For example, here is the multiplication table for $\mathbb{Z}_7^*$, which has 3 and 5 as generators:

$\mathbb{Z}_7^*$	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

Definition 3.4 *a is a* **quadratic residue** *if $\exists\, x \in \mathbb{Z}_p^*$ such that $a = x^2 \mod p$. We say that x is a* **square root** *of a.*

Claim 3.5 *For a prime p,*
(i) $a = g^j$ is a quadratic residue iff j is even (i.e., exactly half of Z_p^* are quadratic residues)
(ii) each quadratic residue $a = g^j$ has exactly two square roots, namely $g^{\frac{j}{2}}$ and $g^{\frac{j}{2}+\frac{p-1}{2}}$

Proof: Easy exercise. (Hint: note that $p-1$ is even.) ■

Note that from the above table it can be seen that 2, 4, and 1 are quadratic residues in $\mathbb{Z}_7^*$. We obtain the following corollary, which will form the basis of our primality test:

Corollary 3.6 *If p is prime, then 1 has no non-trivial square roots in $\mathbb{Z}_p^*$, i.e., the only square roots of 1 in $\mathbb{Z}_p^*$ are ± 1.*

In $\mathbb{Z}_n^*$ for composite n, there may be non-trivial roots of 1: for example, in $\mathbb{Z}_{35}$, $6^2 = 1$.

The idea of the algorithm is to search for non-trivial square roots of 1. Specifically, assume that n is odd, and not a prime power. (We can detect perfect powers in polynomial time and exclude them: **Exercise!**). Then $n-1$ is even, and we can write $n - 1 = 2^r R$ with R odd. We search by computing $a^R, a^{2R}, a^{4R}, \cdots, a^{2^r R} = a^{n-1}$ (all mod n). Each term in this sequence is the square of the previous one, and the last term is 1 (otherwise we have failed the Fermat test and n is composite). Thus if the first 1 in the sequence is preceded by a number other than -1, we have found a non-trivial root and can declare that n is composite. More specifically the algorithm works as follows:

```
if $n > 2$ is even or a perfect power then output "composite"
compute $r, R$ s.t. $n - 1 = 2^r \cdot R$     [Note: $R$ is odd]
pick $a \in \{1, ..., n-1\}$ uniformly at random
if $\gcd(a, n) \neq 1$ then ouput "composite" and halt
compute $b_i = a^{2^i R} \mod n,\ i = 0, 1, \cdots r$
if $b_r [= a^{n-1}] \neq 1 \mod n$ then output "composite" and halt
else if $b_0 = 1 \mod n$ then output "prime" and halt
else let $j = \max\{i : b_i \neq 1\}$
if $b_j \neq -1$ then output "composite"
else output "prime"
```

For example, for the Carmichael number $n = 561$, we have $n-1 = 560 = 2^4 \times 35$. If $a = 2$ then the sequence computed by the algorithm is $a^{35} \bmod 561 = 263$, $a^{70} \bmod 561 = 166$, $a^{140} \bmod 561 = 67$, $a^{280} \bmod 561 = 1$, $a^{560} \bmod 561 = 1$. So the algorithm finds that 67 is a non-trivial square root of 1 and therefore concludes that 561 is not prime.

Notice that the output "composite" is always correct. However the algorithm may err when it outputs "prime". It remains to show that the error probability is bounded when n is composite; we will do this next.

The algorithm begins by testing to see if a randomly chosen a passes the test of Fermat's little theorem. If $a^{n-1} \neq 1 \bmod n$, then we know that n is composite, otherwise we continue by searching for a nontrivial square root of 1. We examine the sequence of descending square roots beginning at $a^{n-1} = 1$ until we reach an odd power of a:

$$1 = a^{n-1}, a^{(n-1)/2}, a^{(n-1)/4}, \ldots, a^R$$

There are three cases to consider:

1. The powers are all equal to 1.
2. The first power (in descending order) that is not 1 is -1.
3. The first power (in descending order) that is not 1 is a nontrivial root of 1.

In the first two cases we fail to find a witness for the compositeness of n, so we guess that n is prime. In the third case we have found that some power of a is a nontrivial square root of 1, so a is a witness that n is composite.

3.4.1 The likelihood of finding a witness

We now show that if n is composite, we are fairly likely to find a witness.

Claim 3.7 *If n is odd, composite, and not a prime power, then* $\Pr[a \text{ is a witness}] \geq \frac{1}{2}$.

To prove this claim we will use the following definition and lemma.

Definition 3.8 *Call $s = 2^i R$ a* ***bad*** *power if $\exists x \in \mathbb{Z}_n^*$ such that $x^s = -1 \bmod n$.*

Lemma 3.9 *For any bad power s, $S_n = \{x \in \mathbb{Z}_n^* : x^s = \pm 1 \bmod n\}$ is a proper subgroup of $\mathbb{Z}_n^*$.*

We will first use the lemma to prove claim 3.7, and then finish by proving the lemma.

Proof of Claim 3.7: Let $s^* = 2^{i^*} R$ be the largest bad power in the sequence $R, 2R, 2^2R, \ldots, 2^r R$. (We know s^* exists because R is odd, so $(-1)^R = -1$ and hence R at least is bad.)

Let S_n be the proper subgroup corresponding to s^*, as given by Lemma 3.9. Consider any non-witness a. One of the following cases must hold:

(i) $a^R = a^{2R} = a^{4R} = \ldots = a^{n-1} = 1 \bmod n$;

(ii) $a^{2^i R} = -1 \bmod n$, $\;a^{2^{i+1} R} = \ldots = a^{n-1} = 1 \bmod n$ (for some i).

In either case, we claim that $a \in S_n$. In case (i), $a^{s^*} = 1 \bmod n$, so $a \in S_n$. In case (ii), we know that $2^i R$ is a bad power, and since s^* is the largest bad power then $s^* \geq 2^i R$, implying $a^{s^*} = \pm 1 \bmod n$ and so $a \in S_n$.

Therefore, all non-witnesses must be elements of the proper subgroup S_n. Using Lagrange's Theorem just as we did in the analysis of the Fermat Test, we see that

$$\Pr[a \text{ is not a witness}] \leq \frac{|S_n|}{|\mathbb{Z}_n^*|} \leq \frac{1}{2}.$$

■

We now go back and provide the missing proof of the lemma.

Proof of Lemma 3.9:

S_n is clearly closed under multiplication and hence a subgroup, so we must only show that it is proper, i.e., that there is some element in $\mathbb{Z}_n^*$ but not in S_n. Since s is a bad power, we can fix an $x \in \mathbb{Z}_n^*$ such that $x^s = -1$. Since n is odd, composite, and not a prime power, we can find n_1 and n_2 such that n_1 and n_2 are odd, coprime, and $n = n_1 \cdot n_2$.

Since n_1 and n_2 are coprime, the Chinese Remainder theorem implies that there exists a unique $y \in \mathbb{Z}_n$ such that

$$\begin{aligned} y &= x \bmod n_1; \\ y &= 1 \bmod n_2. \end{aligned}$$

We claim that $y \in \mathbb{Z}_n^* \setminus S_n$.

Since $y = x \bmod n_1$ and $\gcd(x, n) = 1$, we know $\gcd(y, n_1) = \gcd(x, n_1) = 1$. Also, $\gcd(y, n_2) = 1$. Together these give $\gcd(y, n) = 1$. Therefore $y \in \mathbb{Z}_n^*$.

We also know that

$$\begin{aligned} y^s &= x^s \bmod n_1 \\ &= -1 \bmod n_1 \quad (*) \\ y^s &= 1 \bmod n_2 \quad (**) \end{aligned}$$

Suppose $y \in S_n$. Then by definition, $y^s = \pm 1 \bmod n$.

If $y^s = 1 \bmod n$, then $y^s = 1 \bmod n_1$ which contradicts $(*)$.

If $y^s = -1 \bmod n$, then $y^s = -1 \bmod n_2$ which contradicts $(**)$.

Therefore, y cannot be an element of S_n, so S_n must be a proper subgroup of $\mathbb{Z}_n^*$. ■

3.4.2 Notes and some background on primality testing

The above ideas are generally attributed to both Miller [M76] and Rabin [R76]. More accurately, the randomized algorithm is due to Rabin, while Miller gave a deterministic version that runs in polynomial time assuming the Extended Riemann Hypothesis (ERH): specifically, Miller proved under the ERH that a witness a of the type used in the algorithm is guaranteed to exist within the first $O((\log n)^2)$ values of a. Of course, proving the ERH would require a major breakthrough in Mathematics.

A tighter analysis of the above algorithm shows that the probability of finding a witness for a composite number is at least $\frac{3}{4}$, which is asymptotically tight.

Another famous primality testing algorithm, with essentially the same high-level properties and relying crucially on the subgroup trick but with a rather different type of witness, was developed by Solovay and Strassen around the same time as the Miller/Rabin algorithm [SS77]. Both of these algorithms, and variants on them, are used routinely today to certify massive primes having thousands of bits (required in applications such as the RSA cryptosystem).

In 2002, Agrawal, Kayal, and Saxena made a theoretical breakthrough by giving a deterministic polynomial time algorithm for primality testing [AKS02]. However, it is much less efficient in practice than the above randomized algorithms. This algorithm was inspired by the randomized polynomial time algorithm of Agrawal and Biswas in 1999 [AB99], which uses a generalization of the Fermat test to polynomials. We will sketch this algorithm in the next lecture.

The algorithm we have seen has a one-sided error: for prime n, the probability of error is 0, while for composite n the probability of error is at most $\frac{1}{2}$. Adleman and Huang [AH87] came up with a polynomial time randomized algorithm with one-sided error in the opposite direction, i.e., it is always correct on composites, but may err on primes with probability at most $\frac{1}{2}$.[1] Although the Adleman-Huang algorithm is not very efficient in practice, it is of theoretical interest to note that it can be combined with the Miller-Rabin algorithm above to create a stronger *Las Vegas* algorithm for primality testing:

```
repeat forever
    run Miller-Rabin on $n$
    if Miller-Rabin outputs "composite" then output "composite" and halt
    run Adleman-Huang on $n$
    if Adleman-Huang outputs "prime" then output "prime" and halt
```

If this algorithm ever terminates it must be correct since Miller-Rabin never makes an error when it outputs "composite", and likewise for Adleman-Huang and "prime". Also, since the probability of error in each component is at most $\frac{1}{2}$, the probability of iterating t times without terminating decreases exponentially with t.

References

[AH87] L.M. Adleman and A.M.-D. Huang, "Recognizing primes in random polynomial time," *Proceedings of the 19th ACM STOC*, 1987, pp. 462–469.

[AB99] M. Agrawal and S. Biswas, "Primality and identity testing via Chinese remaindering," *Proceedings of IEEE FOCS* 1999, pp. 202–209. Full version appeared in *Journal of the ACM* **50** (2003), pp. 429–443.

[AKS02] M. Agrawal, N. Kayal and N. Saxena, "PRIMES is in P," *Annals of Mathematics* **160** (2004), pp. 781–793.

[BM77] R.S. Boyer and J.S. Moore, "A fast string-searching algorithm," *Communications of the ACM*, 20(10):762–772, 1977.

[Fre77] R. Freivalds, "Probabilistic Machines Can Use Less Running Time," *IFIP Congress* 1977, pp. 839–842.

[1] More practically useful certificates of primality were developed by Goldwasser and Kilian [GK86]; to guarantee their existence one needs to appeal to an unproven assumption.

[GK86] S. Goldwasser and J. Kilian, "Almost all primes can be quickly certified," *Proceedings of the 18th ACM STOC*, 1986, pp. 316–329.

[KR81] R. Karp and M. Rabin, *Efficient randomized pattern-matching algorithms*, Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.

[KMP77] D. Knuth, J. Morris and V. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, 6(2):323-350, 1977.

[M76] G.L. Miller, "Riemann's hypothesis and tests for primality," *Journal of Computer and Systems Sciences* **13** (1976), pp. 300–317.

[R76] M.O. Rabin, "Probabilistic algorithms," in J.F. Traub (ed.), *Algorithms and Complexity, Recent Results and New Directions*, Academic Press, New York, 1976.

[SS77] R. Solovay and V. Strassen, "A fast Monte Carlo test for primality," *SIAM Journal on Computing* **6** (1977), pp. 84–85. See also *SIAM Journal on Computing* **7** (1978), p. 118.