Factoring & Primality

Lecturer: Dimitris Papadopoulos

In this lecture we will discuss the problem of integer factorization and primality testing, two problems that have been the focus of a great amount of research over the years. These problems started receiving attention in the mathematics community far before the appearance of computer science, however the emergence of the latter has given them additional importance. Gauss himself wrote in 1801 "The problem of distinguishing prime numbers from composite numbers, and of resolving the latter into their prime factors, is known to be one of the most important and useful in arithmetic. The dignity of the science itself seems to require that every possible means be explored for the solution of a problem so elegant and so celebrated."

1 Factorization Problems

The following result is known as the *Fundamental Theorem of Arithmetic*:

Theorem 1 *Every $n \in \mathbb{Z}$ with $n \neq 0$ can be written as:*

$$n = \pm \prod_{i=1}^{r} p_i^{e_i}$$

where p_i are distinct prime numbers and e_i are positive integers. Moreover, this representation is unique (except for term re-ordering).

This celebrated result (already known at Euclid's time) unfortunately does not tell us anything about computing the primes p_i or the powers e_i except for their existence. This naturally leads to the following problem statement known as the **integer factorization** problem:

Problem 1 *Given $n \in \mathbb{Z}$ with $n \neq 0$, compute primes p_i and positive integers e_i for $i = 1, ..., r$ s.t. $\prod_{i=1}^{r} p_i^{e_i} = n$.*

Clearly, if n is a prime number then the factorization described above is trivial. Namely $r = 1, p_1 = n, e_1 = 1$ and n itself is the correct output. This seems to imply that for a large class of integers (the primes) Problem 1 can be reduced to the following problem, known as **primality testing**:

Problem 2 *Given $n \in \mathbb{Z}$ with $n \neq 0$, decide whether n has any non-trivial factors.*

Another problem that is a simplified version of Problem 1 (in the case where n is composite) is instead of computing all the prime factors, to output one of them. More formally, the problem of **divisor computation** is:

Problem 3 *Given composite $n \in \mathbb{Z}$ with $n \neq 0$, compute $k \in \mathbb{Z}$ with $k \neq 1$ s.t. $\exists m \in \mathbb{Z}$ where $n = km$.*

Next we turn our attention to the hardness of these problems.

2 Hardness of Factoring Problems

We begin with the problem of primality testing. Let PRIMES be the problem of recognizing the language $L_1 = \{n \in \mathbb{Z} | x \text{ is prime}\}$. This has been a long standing problem and for many decades we only had feasible randomized algorithms. In the next section we will discuss the Miller-Rabin primality testing algorithm that is widely implemented in practice due to its practicality. This algorithm (and others of similar flavour) have deterministic versions under the unproven *Generalized Riemann Hypothesis*. In a very surprising result, Agrawal, Kayal, Saxena in 2002 gave the first deterministic algorithm that is not based on unproven assumptions and decides L_1 in polynomial time, proving that $L_1 \in P$. Let us now turn our attention to the more complicated factoring problems. The problem of computing one divisor is in the class FNP which is the function problem extension of the decision problem class NP. Whether or not it is computable by a deterministic polynomial algorithm (hence it belongs in FP) remains an open problem. The problem of integer factorization seems to be much harder but the following can be proven.

Lemma 1 *For the problems of* integer divisor computation (DIV) *and* integer factorization (FACT) *the following is true:*

$$FACT \approx_P DIV$$

(Proof in class)

The above states that there is a polynomial-time reduction from each of the problems to the other. This implies that in order to solve the integer factorization problem, it would suffice to have an algorithm for efficient computation of integer divisors. In the following sections we will discuss a few basic algorithms for primality testing and divisor computation. A final note regarding the hardness of integer factorization is that Shor in 1999 gave an algorithm for quantum computers that can solve this problem in polynomial time, proving that the problem is in BQP.

3 A General Algorithm

Intuitively, the simplest approach to all of the above problems is a trial-and-error approach where we try dividing with all the integers < 0. If any division succeeds (i.e., is perfect) then we know that n is composite and we have also found one of its divisors. The above can be written as:

Trial-and-error Division

On input non-zero integer n

1. $i = 2$
2. **while** $i < n$ **do**:
   if $i|n$ **output** i **and halt**
   $i = i + 1$

3. **output** "n is prime"

Clearly, the above procedure always finds a non-trivial divisor of n if one exists and if n is prime it reports that correctly too. Assuming that n is a k-bit number, then the cost of (naive) division is $O(k^2)$ but the loop runs for $n-2$ steps hence the overall complexity is $O(k^2 n)$ which is exponential to the input length. Of course the above version of the algorithm is "too naive". Every composite number n must have a divisor $i \leq \sqrt{n}$ (why?) therefore the iteration limit can be restricted to $n^{1/2}$ and the corresponding run-time would be $O(k^2 n^{1/2})$ which is still exponential to the input length.

Trial-and-error division is very impractical and cannot be used in practice to find divisors of more than a few bits length. However, it has the advantage that it both decides primality and finds a divisor if one exists. In the next sections we will discuss polynomial-time probabilistic algorithms for the problems of primality testing and factoring.

4 Algorithms for Primality Testing

The following result is known as *Fermat's Little Theorem*:

Theorem 2 *For any prime n, and $a \in \mathbb{Z}$ it holds that $a^{n-1} = 1 \mod n$.*

Can we use the above as a primality test? Here is an idea:

Fermat's Primality Test

On input non-zero integer n

1. Choose randomly $i \in [2, n-1]$
2. **if** $i^{n-1} \neq 1 \mod n$ **output** "composite"
3. **else output** "prime"

The above runs in time $O(k^3)$ (using exponentiation by repeated squaring) and by Theorem 3 always reports "composite" correctly if it finds a value i such that $i^{n-1} \neq 1 \mod n$. Unfortunately, Fermat's little theorem does not tell us anything for the other direction, i.e., it is not an "if and only if" statement. In practice there are a lot of composite numbers n for which there exist i s.t. $i^{n-1} = 1 \mod n$. Consider for example $n = 15$ and $i = 4$. Then $4^{14} = 256^3 \cdot 16 = 268435456 = 1 \mod 15$. How do we improve our algorithm's chances of success?

We can repeat the process for multiple trials, say k and output prime only if the relation holds for all of them. Hence, if we knew for example that the probability of failure ($i^{n-1} = 1 \mod n$) and n is composite) is upper bounded by $1/2$ then the overall failure probability would be upper bounded by 2^{-k}. We can prove the following:

Lemma 2 *If n is composite and $\exists i \in [1, n-1]$ s.t. $i^{n-1} \neq 1 \mod n$ then the probability the algorithm errs is at most 2^{-k}.*

Unfortunately, the above does not cover all cases. In fact, there exist composite numbers such that $i^{n-1} = 1 \mod n$ for all $i \in [1, n-1]$. These numbers are known as Carmichael numbers and, while very rare, we know that infinitely many of them exist so we cannot ignore them.

The above test can be abstracted as follows:

- Define an efficiently recognizable set $S_n \subseteq Z_n^*$.
- If n is prime, then $S_n = Z_n^*$.
- Otherwise, $|S_n| \leq c|Z_n^*|$ for some constant $0 < c < 1$.
- To test, generate k elements $a_1, ..., a_k \in Z_n^+$ and test if $a_i \in S_n$.
- If all of them are in S_n output "prime", otherwise "composite".

For Fermat's testing $S_n = \{a \in Z_n^+ | a^{n-1} = 1 \mod n\}$ and this fails with respect to the third constraint as we showed. The natural next step to take is to enforce some tighter restriction on the construction of S_n. The Miller-Rabin primality testing (which is very widely used) takes exactly this approach.

Lemma 3 *For any integer $n > 2$, $n - 1 = 2^t m$ for t, m positive integers where m is odd.*

(Proof in class)

Now, let S'_n be defined as follows:

$$S'_n = \{a \in \mathbb{Z}_n^+ | a^{2^t m} = 1 \wedge (a^m = 1 \vee \exists 0 \leq j < t \text{ s.t. } a^{2^j m} = -1\}$$

The above set is efficiently decidable (how?) and it is clear that $S'_n \subseteq S_n$. The following can be proven:

Lemma 4 *If n is a prime, then $S'_n = \mathbb{Z}_n^*$, otherwise $|S'_n| \leq 1/4|\mathbb{Z}_n^*|$.*

It should be clear from the above result that there are no "bad composite" cases for this test (similar to Carmichael numbers) hence we can write the following algorithm:

Miller-Rabin Primality Test

On input non-zero integer n

1. Compute t, m s.t. $n - 1 = 2^t m$
2. **for** $i = 1$ **to** k **do**:
   Generate randomly $a_i \in \mathbb{Z}_n^+$
   Check if $a_i \in S'_n$. If it is not, **output** "composite" **and halt**
3. **output** "prime"

Assuming that membership in S'_n can be tested in time $O(\log n^3)$ (how?) the above algorithm runs in time $O(k \log n^3)$. Also, by the previous lemma, the error probability is at most 4^{-k} in the case where n is composite only. Unfortunately, as in the case of Fermat's testing this algorithm does not produce a factor of n.

5 Pollard's Rho Algorithm for Integer Factorization

In this section we will be presenting one very popular randomized algorithm for factoring introduced by Pollard in 1973. On input a composite number n the algorithm outputs one of its divisors d. Recall that the problem of factoring can be efficiently reduced to the problem of divisor computation, we can use this algorithm for factorization. Also, the restriction that n must be known to be composite is not severe since there exist polynomial time algorithms to decide that.

The main idea of the algorithm is the following. Let $a, b < n$ be integers such that $a = b \mod d$. It follows that $a - b = kd$ for some $k \in \mathbb{Z}$. Also, by assumption $n = ld$ for some $l \in \mathbb{Z}$. From this it follows that $gcd((a-b), n) = gcd(kd, ld) = d$ and by computing this greatest common divisor we may find a divisor of n. The surprising fact about this algorithm is that we are using the fact that d must exist in order to compute it (without knowing its value).

Let us see again one way to reach the above. Randomly generate a number of values $a_1, ..., a_r < n^{1/2}$. Compute their pair-wise *gcd*. If for any of them, it is larger than 1, output it and halt. We say that values a_i, a_j with $i \neq j$ form a *collision* if $a_i = a_j \mod n$. The following result follows from the *birthday paradox*:

Lemma 5 *The expected number of necessary values a_i in order to produce a collision is $O(d^{1/2})$.*

This approach therefore immediately yields an algorithm for finding a divisor. Unfortunately, one needs to compute all the *gcd*'s involved which for r values is $O(r^2$ hence the overall complexity is $O((d^{1/2})^2) = O(d) = O(n^{1/2}$ which is as bad as the trial division approach. Moreover one needs to store all these values a_i which means that the algorithm requires exponential storage as well.

Pollard's rho algorithm is based on this approach but by using a clever trick known as *Floyd's Cycle Detection* manages to do so by storing at any time only two values.

Floyd's Cycle Detection

On input function f and point x_0

1. $i = 1$, $y_0 = x_0$
2. **while true**
3. $x_i = f(x_{i-1})$, $y_i = f(f(y_{i-1}))$
4. **if** $x_i = y_i$ **output** i **and halt**
5. $i = i + 1$

In the above the sequence x_i runs infinitely, hence if f takes values from a finite domain, it must repeat itself. Since $x_{i+1} = f(x_i)$ once one such repetition occurs all subsequent values must also agree. It can be shown that if t is the first index where such a collision occurs and l is the length of this cycle, the above algorithm runs at most for $t + l$ steps.

Moving on the actual factoring algorithm, our choice of function will be a function f from $\mathbb{Z}_n$ to itself such that $f(x) = f(y)$ implies that $x = y$. In fact, the original proposed function is $f(x) = x^2 + 1$. The algorithm then is as follows:

Pollard's rho Algorithm

On input integer n

1. Choose x_0 randomly from $\mathbb{Z}_n$
2. Set $i = 1$, $y_0 = x_0$
3. **while true**
4. $x_i = x_{i-1}^2 + 1 \mod n$, $y_i = (y_{i-1}^2 + 1)^2 + 1 \mod n$
5. **if** $d = gcd(x_i - y_i, n) > 1$ **output** d **and halt**
6. $i = i + 1$

The main idea here is that values x_i as we saw above, form an infinite sequence hence they must loop (possibly after some initial part). Now this sequence induces a similar sequence of values x'_i modulo d such that $x'_i = x_i \mod p$. Furthermore it must be (why?) that $x'_{i+1} = (x'^2_i + 1) \mod p$. Therefore, while we are not explicitly computing the sequence x'_i, its existence is implied. By the same approach as before, the sequence x'_i has a collision after an expected number of $O(d^{1/2})$.

By an analysis it follows that the algorithm always outputs a divisor of n its expected runtime is $\tilde{O}(n^{1/4})$ and it uses $\tilde{O}(1)$ bits of storage. However, it does not always produce a non-trivial factor of n. If the points on which the algorithm agrees satisfy the relation $x_i = y_i \mod n$ instead of $\mod p$ then it follows that $gcd(x_i - y_i, n) = n$ hence the output is the trivial divisor n. In this case the only known alternative is to re-run it with a different choice of x_0 and/or f. It should be noted that a formal analysis of the algorithm assumes that f is truly a random function and it still an open problem to prove why $x^2 + 1 \mod n$ behaves as such, making the whole analysis of the algorithm so far a heuristic argument.