

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.080 / 6.089 Great Ideas in Theoretical Computer Science  
Spring 2008

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.

## Lecture 15

Lecturer: Scott Aaronson

Scribe: Tiffany Wang

## 1 Administrivia

Midterms have been graded and the class average was 67. Grades will be normalized so that the average roughly corresponds to a B. The solutions will be posted on the website soon.

Pset4 will be handed out on Thursday.

## 2 Recap

### 2.1 Probabilistic Computation

We previously examined probabilistic computation methods and the different probabilistic complexity classes, as seen in Figure 1.

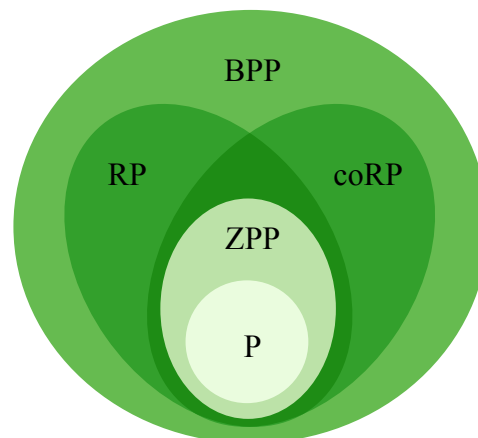


Figure by MIT OpenCourseWare.

**Figure 1.** Probabilistic Complexity Classes

*P*: *Polynomial time* - Problems that can be solved deterministically in polynomial time.

*ZPP*: *Zero-error Probabilistic Polynomial (Expected Polynomial) time* - Problems that can be solved efficiently but with 50% chance that the algorithm does not produce an answer and must be run again. If the algorithm *does* produce an answer it is guaranteed to be correct.

*RP*: *Randomized Polynomial time* - Problems for which if the answer is *NO*, the algorithm always outputs *NO*. Otherwise, if the answer is *YES*, the algorithm outputs *YES* at least 50% of the time. Hence there is an asymmetry between *YES* and *NO* outputs.

*coRP*: *Complement of RP* - These are problems for which there's a polynomial-time algorithm that always outputs *YES* if the answer is *YES* and outputs *NO* at least 50% of the time if the answer

is *NO*.

*BPP: Bounded-error Probabilistic Polynomial time* - Problems where if the answer is *YES*, the algorithm accepts with probability  $\geq \frac{2}{3}$ , and if the answer is *NO*, the algorithm accepts with probability  $\leq \frac{1}{3}$ .

## 2.2 Amplification and Chernoff Bound

The question that arises is whether the boundary values  $\frac{1}{3}$  and  $\frac{2}{3}$  have any particular significance. One of the nice things about using a probabilistic algorithm is that as long as there is a noticeable gap between the probability of accepting if the answer is *YES* and the probability of accepting if the answer is *NO*, that gap can be amplified by repeatedly running the algorithm.

For example, if you have an algorithm that outputs a wrong answer with  $\text{Pr} \leq \frac{1}{3}$ , then you can repeat the algorithm hundreds of times and just take the majority answer. The probability of obtaining a wrong answer becomes astronomically small (there's a much greater chance of an asteroid destroying your computer).

This notion of amplification can be proven using a tool known as the *Chernoff Bound*. The Chernoff Bound states that given a set of independent events, the number of events that will happen is heavily concentrated about the expected value of the number of occurring events.

So given an algorithm that outputs a wrong answer with  $\text{Pr} = \frac{1}{3}$ , repeating the algorithm 10,000 times would produce an expected number of 3333.3... wrong answers. The number of wrong answers will not be exactly the expected value, but the probability of getting a number far from the expected value (say 5,000) is very small.

## 2.3 P vs. BPP

There exists a fundamental question as to whether every probabilistic algorithm can be replaced by a deterministic one, or *derandomized*. In terms of complexity, the question is whether  $P=BPP$ , which is almost as deep a question as  $P=NP$ .

There is currently a very strong belief that derandomization is possible in general, but no one yet knows how to prove it.

## 3 Derandomization

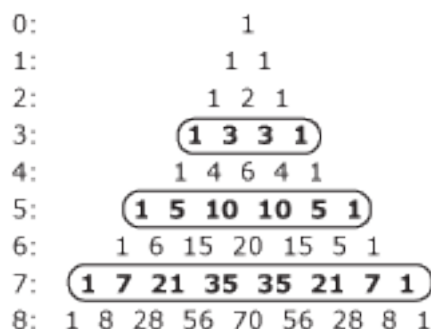
Although derandomization has yet to be proven in the general case, it *has* been proven for some spectacular special cases: cases where for decades, the only known efficient solutions came from randomized algorithms. Indeed, this has been one of the big success stories in theoretical computer science in the last 10 years.

### 3.1 AKS Primality Test

In 2002, Agrawal, Kayal, and Saxena of the Indian Institute of Technology Kanpur developed a deterministic polynomial-time algorithm for testing whether an integer is prime or composite, also known as the AKS primality test.

For several decades prior, there existed good algorithms to test primality, but all were probabilistic. The problem was first known to be in the class RP, and then later shown to be in the class ZPP. It was also shown that the problem was in the class P, but only assuming that the Generalized Riemann Hypothesis was true. The problem was also known to be solvable deterministically in  $n^{O(\log\log\log n)}$  time (which is *slightly* more than polynomial). Ultimately, it was nice to have the final answer and the discovery was an exciting thing to be alive for in the world of theoretical computer science.

The basic idea behind AKS is related to Pascal’s Triangle. As seen in Figure 2, in every prime-numbered row, the numbers in Pascal’s Triangle are all a multiple of the row number. On the other hand, in every composite-numbered row, the numbers are *not* all multiples of the row number.



**Figure 2.** Pascal’s Triangle and Prime Numbers

So to test the primality of an integer  $N$ , can we just check whether or not all the numbers in the  $N^{\text{th}}$  row of Pascal’s Triangle are multiples of  $N$ ? The problem is that there are exponentially many numbers to check, and checking all of them would be no more efficient than trial division.

Looking at the expression  $(x+a)^N$ , which has coefficients determined by the  $N^{\text{th}}$  row of Pascal’s Triangle, AKS noticed that the relationship  $(x+a)^N = x^N + a^N \pmod N$  holds if and only if  $N$  is prime. This is because if  $N$  is prime, then all the “middle” coefficients will be divisible by  $N$  (and therefore disappear when we reduce mod  $N$ ), while if  $N$  is composite then some middle coefficients will not be divisible by  $N$ . What this means is that the primality testing problem can be mapped to an instance of the polynomial identity testing problem: given two algebraic formulas, decide whether or not they represent the same polynomial.

In order to determine whether  $(x+a)^N = x^N + a^N \pmod N$ , one approach would be to plug in many random values of  $a$  and see if we get the same result each time. However, since the number of terms would still be exponential, we need to evaluate the expression not only mod  $N$ , but also mod a random polynomial:

$$(x+a)^N = x^N + a^N \pmod N, x^r - 1.$$

It turns out that this solution method works; on the other hand, it still depends on the use of randomness (the thing we're trying to eliminate).

The tour de force of the AKS paper was to show that if  $N$  is composite, then it is only necessary to try some small number of deterministically-chosen values of  $a$  and  $r$  until a pair is found such that the equation is not satisfied. This immediately leads to a method for distinguishing prime numbers from composite ones in deterministic polynomial time.

### 3.2 Trapped in a Maze

*Problem:* Given a maze (an undirected graph) with a given start vertex and end vertex, is the end vertex reachable or not?

*Proposed solution from the floor:* Depth-first search.

In a maze, this is the equivalent of wandering around the maze and leaving bread crumbs to mark paths that have already been explored. This solution runs in polynomial time, but the problem is that it requires breadcrumbs, or translated into complexity terms, a large amount of memory. The hope would be to solve the undirected connectivity problem in LOGSPACE: that is, to find a way out of the maze while remembering only  $O(\log n)$  bits at any one time. (Why  $O(\log n)$ ? That's the amount of information needed even to write down where you are; thus, it's essentially the best you can hope for.)

*Proposed solution from the floor:* Follow the right wall.

The trouble is that, if you were always following the right wall, it would be simple to create a maze that placed you in an infinite loop.

*Simple-minded solution:* Just wander around the maze randomly.

Now we're talking! Admittedly, in a *directed* graph it could take an exponential time for a random walk to reach the end vertex. For example, at each intersection of the graph shown in Figure 3, you advance forward with  $\text{Pr}=\frac{1}{2}$  and return to the starting point with  $\text{Pr}=\frac{1}{2}$ . The chance that you make  $n$  consecutive correct choices to advance all the way to the end is exponentially small, so it will take exponential time to reach the end vertex.

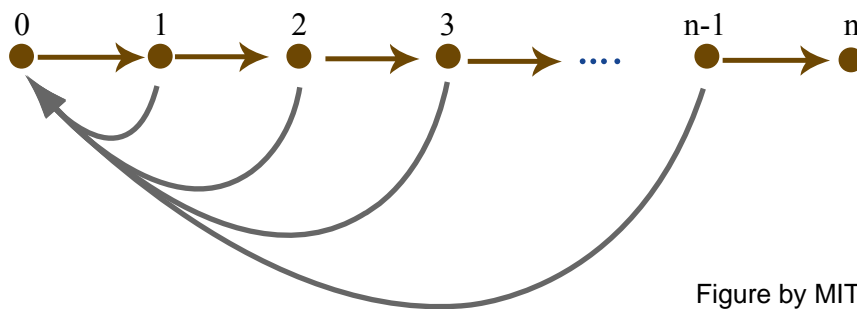


Figure by MIT OpenCourseWare.

**Figure 3.** Exponential Time Directed Graph

In 1979, Aleliunas et al. showed that randomly wandering through an undirected graph will get you out with high probability after  $O(n^3)$  steps. After  $O(n^3)$  steps, with high probability you will

have visited every vertex, regardless of the structure of the graph.

However, this still leaves the question of whether there's a *deterministic* algorithm to get out of a maze using only  $O(\log n)$  bits of memory. Finally, in 2005, Omer Reingold proved that by making *pseudorandom* path selections based on a somewhat complicated set of rules, the maze problem can be solved deterministically in LOGSPACE. At each step, the rule is a function of the outcome of the previous application of the rules.

## 4 New Unit: Cryptography

### 4.1 History

Cryptography is a 3,000-year old black art that has been completely transformed over the last few decades by ideas from theoretical computer science. Cryptography is perhaps the best example of a field in which the concepts of theoretical computer science have real practical applications: problems are designed to be hard, the worst case assumptions are the right assumptions, and computationally intractable problems are there because we put them there.

For more on the history of cryptography, a great reference is David Kahn's *The Codebreakers*, which was written before people even knew about the biggest cryptographic story of all: the breaking of the German naval code in World War II, by Alan Turing and others.

### 4.2 Caesar Cipher

One of the oldest cryptographic codes used in history is the "Caesar cipher." In this cryptosystem, a plaintext message is converted into ciphertext by simply adding 3 to each letter of the message, wrapping around to A after you reach Z. Thus A becomes D, Z becomes C, and DEMOCRITUS becomes GHPRFULWXV.

Clearly this encryption system can easily be broken by anyone who can subtract mod 26. As an amusing side note, just a couple years ago, the head of the Sicilian mafia was finally caught after 40 years because he was using the Caesar cipher to send messages to his subordinates.

### 4.3 Substitution Cipher

A slightly more complicated cryptographic encoding is to scramble the letters of a message according to a random rule which permutes all the letters of the alphabet. For example, substituting every A with an X and every S with a V.

This encoding can also be easily broken by performing a frequency analysis of the letters appearing in the ciphertext.

### 4.4 One-Time Pad

It was not until the 1920's that a "serious" cryptosystem was devised. Gilbert Sandford Vernam, an American businessman, proposed what is known today as the one-time pad.

Under the one-time pad cryptosystem, the plaintext message is represented by a binary string  $M$  which is then XOR-ed with a random binary key,  $K$ , of the same length. As seen in Figure 4,

the ciphertext  $C$  is equal to the bitwise sum of  $M$  and  $K$ , mod 2.

$$\begin{array}{r}
 M : 111010110001 \\
 \oplus K : 011011101011 \\
 \hline
 C : 100001011010
 \end{array}$$

**Figure 4.** One-time Pad Encryption

Assuming that the recipient is a trusted party who shares the knowledge of the key, the ciphertext can be decrypted by performing another XOR operation:  $C \oplus K = M \oplus K \oplus K = M$ . See Figure 5.

$$\begin{array}{r}
 C : 100001011010 \\
 \oplus K : 011011101011 \\
 \hline
 M : 111010110001
 \end{array}$$

**Figure 5.** One-Time Pad Decryption

To an eavesdropper who does not have knowledge of the key, the ciphertext appears to be nonsense since XOR-ing any string of bits with a random string just produces another random string. There is no way to guess what the ciphertext may be encoding because any binary key could have been used.

As a result of this, the one-time pad is a provably unbreakable cryptographic encoding, but only if used correctly. The problem with using the one-time pad is that it literally is a “one-time” encryption. If the same key is ever used to encrypt more than one message, then the cryptosystem is no longer secure. For example, if we sent another message  $M_2$  encrypted with the same key  $K$  to produce  $C_2$ , the eavesdropper could obtain a combination of the messages:  $C_1 \oplus C_2 = M_1 \oplus K \oplus M_2 \oplus K = M_1 \oplus M_2$ . If the eavesdropper had any idea of what either of the messages may have contained, the eavesdropper could learn about the other plaintext message, and indeed obtain the key  $K$ .

As a concrete example, Soviet spies during the Cold War used the one-time pad to encrypt their messages and occasionally slipped up and re-used keys. As a result, the NSA, through its VENONA project, was able to decipher some of the ciphertext and even gather enough information to catch Julius and Ethel Rosenberg.

#### 4.5 Shannon’s Theorem

As we saw, the one-time pad has the severe shortcoming that the number of messages that can be encrypted is limited by the amount of key available.

Is it possible to have a cryptographic code which is unbreakable (in the same absolute sense that the one-time pad is unbreakable), yet uses a key that is much smaller than the message?

In the 1940s, Claude Shannon proved that a perfectly secure cryptographic code requires the encryption key to be at least as long as the message that is sent.

*Proof:* Given an encryption function:  $e_k : \{0, 1\}_{plaintext}^n \rightarrow \{0, 1\}_{ciphertext}^m$ .

For all keys  $k$ ,  $e_k$  must be an injective function (provide a one-to-one mapping between plaintext and ciphertext). Every plaintext must map to a different ciphertext, otherwise there would be no way of decrypting the message.

This immediately implies that for a given ciphertext,  $C$ , that was encrypted with a key of  $r$  bits, the number of possible plaintexts that could have produced  $C$  is at most  $2^r$  (the number of possible keys). If  $r < n$ , then the number of possible plaintexts that could have generated  $C$  is smaller than the total number of possible plaintext messages. So if the adversary had unlimited computational power, the adversary could try all possible values of the key and rule out all plaintexts that could not have encrypted to  $C$ . The adversary would thus have learned something about the plaintext, making the cryptosystem insecure. Therefore the encryption key must be at least as long as the message for a perfectly secure cryptosystem.

The key loophole in Shannon's argument is the assumption that the adversary has unlimited computational power. For a practical cryptosystem, we can exploit computational complexity theory, and in particular the assumption that the adversary is a polynomial-time Turing machine that does not have unlimited computational power. More on this next time...