Meeting to talk about final projects on Wednesday, 11 March 2009, from 5pm to 7pm. Location: TBA. Includes food.

1 Overview of today’s lecture

- Randomized computation.
- Complexity classes: RP, coRP, BPP, ZPP.
- Basic properties of these complexity classes.

So far, we know that P is a computationally feasible class. We could try and expand this notion, and then study where the expanded notions lie in relation with P, NP, etc.

2 Examples of problems which have randomized algorithms

1. **Problem**: Find an $n$-bit prime.
   
   **Input**: $N \in \mathbb{N}, N > 3$ such that $2^{n-1} < N \leq 2^n$.
   
   **Output**: A prime $p$, such that $N \leq p < 2N$.

   A polynomial-time algorithm for this problem is as follows. This algorithm is randomized. No deterministic algorithm is known.

   ```
   1: loop \{n times\}
   2: Pick $k$ randomly and uniformly between $N$ (inclusive) and $2N$ (exclusive).
   3: if $k$ is prime then
   4:     return $k$
   5: else
   6:     continue loop.
   7: return a random value between $N$ (inclusive) and $2N$ (exclusive).
   ```

   A sketch of the proof of correctness of this algorithm is as follows.

   **Sketch of Proof** First we observe that we can always find such a prime. This is the following lemma, which we state without proof.

   **Lemma 2.1 (Bertrand’s Postulate)** If $n$ is a natural number greater than 3, then there exists a prime number $p$ such that $n \leq p < 2n$.

   Apart from Lemma 2.1 the algorithm depends on the Prime Number Theorem, which we state without proof.

   **Theorem 2.2 (Prime Number Theorem)** For any real number $x$, let $\pi(x)$ be the number of primes less than or equal to $x$. Then,

   $$\lim_{x \to \infty} \frac{\pi(x)}{x/\ln x} = 1.$$
In this context, the Prime Number Theorem implies that the number of primes between \( N \) and \( 2N \) is about \( \frac{2N}{\pi + 1} - \frac{N}{\pi} \), which is approximately \( \frac{N}{\pi} \). So the probability of \( k \) being prime is approximately \( \frac{1}{n} \). Since the algorithm is repeated \( n \) times, the probability of it not returning a prime is approximately

\[
\left( 1 - \frac{1}{n} \right)^n \approx \frac{1}{e}.
\]

We will see later that this error probability is small enough for our purposes.

2. **Problem:** Square-root modulo primes.
   **Input:** An \( n \)-bit long prime \( p \), an integer \( a \) such that \( 0 \leq a \leq p \).
   **Output:** An integer \( \alpha \) such that \( \alpha^2 \equiv a \pmod{p} \).

Berlekamp, and later Adleman, Manders and Miller, gave randomized polynomial-time algorithms to solve this problem. A deterministic polynomial-time algorithm is not known.

A randomized polynomial-time algorithm to solve this problem is as follows. First, \( \beta \) is chosen randomly and uniformly from \( [p - 1] \). If we can solve the equation \( \gamma^2 = \beta^2 \alpha \pmod{p} \) for \( \gamma \), then \( \alpha = \beta / \gamma \).

For this, \( \theta \) is picked randomly and uniformly from \( [p - 1] \), and the following equation can be solved, \( (x - \theta)^2 = \beta^2 \alpha \pmod{p} \). To do this, we use (without proof) the fact that \( \gcd(x^2 - 2x\theta + \theta^2 - \beta^2 \alpha, x^{p-2} - 1) \) is linear in \( x \) with probability \( 1/2 \).

If we find this \( \gcd q \) and if it is linear in \( x \), then it will be either \( x - \theta - \beta \alpha \) or \( x - \theta + \beta \alpha \), so we can just return \( (x - \theta - q) / \beta \).

3. **Problem:** Given \( kn \times n \) square matrices of integers \( M_1, M_2, \ldots, M_k \), do there exist integers \( r_1, r_2, \ldots, r_k \) such that \( \det(\sum_{i=0}^k r_i M_i) \neq 0 \)?

A randomized algorithm for this problem is as follows. Pick \( r_1, r_2, \ldots, r_k \) randomly and uniformly from \( \{1, 2, \ldots, 3n\} \) and check if \( \det(\sum_{i=0}^k r_i M_i) \neq 0 \). If so, output ‘yes’; otherwise output ‘no’.

The proof of the correctness of this algorithm is discussed in Section 3.

4. **Problem:** Equivalence of circuits.
   **Input:** Circuits \( C_1, C_2 \) over integer inputs \( x_1, x_2, \ldots, x_n \) with addition and multiplication gates and the constants \( \{-1, 0, 1\} \).
   **Output:** Is \( C_1 \) equivalent to \( C_2 \)? (Is the function computed by \( C_1 \) the same as the function computed by \( C_2 \)?)

A randomized algorithm for this problem is as follows. (Here, we assume that the size of the circuit is a polynomial in the number of inputs, to make estimations about the input size of our problem simpler.)

1. Pick a prime of size about \( 2^{O(n)} \) and call it \( p \).
2. Pick \( x_1, x_2, \ldots, x_n \) randomly and uniformly in \( \mathbb{Z}_p \).
3. {In the following if-statement, the output of each gate is computed in \( \mathbb{Z}_p \).}
4. if \( C_1(x_1, x_2, \ldots, x_n) = C_2(x_1, x_2, \ldots, x_n) \) then
5.   return ‘yes’
6. else
7.   return ‘no’

Observe that we can have a polynomial-sized circuit that computes \( 2^{2^n} \), as follows. (Here, ‘A’ denotes addition and ‘M’ denotes multiplication).
The number $2^n$ is too big for polynomial-time simulations, and it is clear that we can actually get a number of this size from a circuit of polynomial size. So we reduce modulo $p$, so as to restrict all the possible numbers in our computations to have at most $O(n)$ bits. This ensures that the algorithm does not exceed polynomial time.

3 Some proofs

To prove the algorithms for finding square-root modulo primes and for checking the equivalence of two circuits, we will use the following lemma. Recall that we have already used once in a previous lecture.

**Lemma 3.1 (Schwarz-Zippel Lemma)** Let $p(x_1, \ldots, x_n)$ be a not identically zero polynomial of total degree $d$ over any (possibly infinite) field $\mathbb{F}$. If $a_1, \ldots, a_n$ are chosen uniformly at random from any finite set $S \subset \mathbb{F}$, then

$$\Pr[p(a_1, \ldots, a_n) = 0] \leq \frac{d}{|S|}.$$  

To prove Item 2 (Square-root modulo primes), we have to calculate the probability that the algorithm makes an error. Note that $p(x_1, \ldots, x_k) = \det(\sum_{i=0}^k x_i M_i)$ is a polynomial of degree $d = n$ in the variables $x_i$. Suppose that the polynomial is not identically zero, otherwise the algorithm can never err.

If with the randomly chosen $r_i$, $\det(\sum_{i=0}^k r_i M_i)$ turns out to be non-zero, then there certainly exists an assignment to the $x_i$s such that $p(x_1, \ldots, x_i)$ is non-zero. On the other hand, if with the randomly chosen $r_i$ the quantity $p(r_1, \ldots, r_i)$ is zero, then there is some probability of error. This can be calculated by using Lemma 3.1. We have chosen the set $S$ to be $\{1, 2, \ldots, 3n\}$, and the $r_i$ have been chosen randomly and uniformly from $S$. Therefore,

$$\Pr[p(r_1, \ldots, r_k) = 0] \leq \frac{d}{|S|} = n \frac{1}{3n} = \frac{1}{3}.$$  

We will only sketch the proof of Item 4 (Circuit equivalence). For this we need to estimate the error probability. If for some choice of $x_1, x_2, \ldots, x_n$, we get that $C_1(x_1, \ldots, x_n) \neq C_2(x_1, \ldots, x_n)$ modulo $p$, then the circuits are certainly not equivalent. On the other hand, if $C_1(x_1, \ldots, x_n) = C_2(x_1, \ldots, x_n)$ modulo $p$, then there is some probability of error. We can also represent $C_1(x_1, \ldots, x_n)$ and $C_2(x_1, \ldots, x_n)$ as polynomials in $x_1, \ldots, x_n$. The following facts lead to the proof that the probability of error is sufficiently small.

- The degrees of the polynomials corresponding to the circuits may be quite large, but Lemma 3.1 still works because $|S| = |\mathbb{Z}_p| \geq 2^{\Omega(n)}$.
- The numbers appearing during the computation are no more than $n$ bits long after reduction modulo $p$. If the original probability of error is $\epsilon$, then we can repeat this algorithm poly$(n)$ times to decrease this to $\epsilon^{\text{poly}(n)}$, by using the following well-known result.

**Theorem 3.2 (Chinese Remainder Theorem)** Let $M, N$ be integers such that for each prime $p_i$ from $k$ distinct primes $p_1, p_2, \ldots, p_k$, $M \equiv N \pmod{p_i}$. Then $M \equiv N \pmod{p_1 p_2 \cdots p_k}$.

4 Complexity classes

4.1 Types of randomized algorithms

To start off the discussion of complexity classes, we first consider the kinds of errors that may occur in a randomized algorithm. For the purposes of the discussion below, we fix $\epsilon = \frac{1}{3}$. We will see later that this particular choice of $\epsilon$ is not special. Let $L$ be a language, and suppose that our algorithm is deciding membership of $x$ in $L$. Then we can have the following types of errors.
1. Two-sided error:
   \( x \in L \Rightarrow \text{probability of error is at most} \ \epsilon \), and
   \( x \notin L \Rightarrow \text{probability of error is at most} \ \epsilon \).
   
   The class of polynomial-time algorithms that behave in this manner is called BPP, which stands for Bounded-error Probabilistic Polynomial-time.

2. One-sided error. There are two types of one-sided error:
   
   (a) \( x \notin L \Rightarrow \text{no errors, and} \)
   \( x \in L \Rightarrow \text{probability of error is at most} \ \epsilon \).
   
   The class of polynomial-time algorithms that behave in this manner is called RP, which stands for Randomized Polynomial-time.
   
   (b) \( x \in L \Rightarrow \text{no errors, and} \)
   \( x \notin L \Rightarrow \text{probability of error is at most} \ \epsilon \).
   
   The class of polynomial-time algorithms that behave in this manner is called coRP, which stands for co-Randomized Polynomial-time.

3. Zero-sided error:
   \( x \in L \Rightarrow \text{no error,} \)
   \( x \notin L \Rightarrow \text{no error, but} \)
   may not halt on some inputs.
   
   Alternatively, we can say that the running time of the algorithm is a random variable with polynomial expectation. Or, we can say that the algorithm is permitted to return one of three values, 1 if it accepts, 0 if it rejects, and ? if it does not know (within some fixed time).

   The class of polynomial-time algorithms that behave in this manner is called ZPP, which stands for Zero-error Probabilistic Polynomial-time.

4.2 Models of randomized computation

A natural model for randomized computation is a Turing Machine \( M \) which has a special ‘coin-tossing’ state, the ‘\$’ state.

However, the preferred model for randomized computation is that of Two-input Turing Machines. In this case, \( x \) is the real input and \( y \) is an auxiliary input. The input \( x \) represents the actual input data, and the input \( y \) captures the randomness used for a particular instance of a randomized computation. A Two-input Turing Machine \( M \) takes in \( (x, y) \) and runs deterministically on \( (x, y) \). (For the cases of RP, coRP, ZPP and BPP, \( M \) must run in polynomial-time of the input, and therefore \( y \) must be a polynomial in the size of \( x \)).

4.3 New definitions for complexity classes

Using the language of two-input Turing Machines, we can redefine some of the complexity classes that we already know. Again, \( \epsilon = \frac{1}{3} \). For each of these complexity classes, the language \( L \) is in the class if there is a two-input Turing Machine \( M \) with the second input always a polynomial in the size of the first, such that certain results (defined in the following list) are true. In the following experiments, \( y \) is always chosen uniformly from all the available possibilities.

1. BPP

   (a) If \( x \in L \) then \( \Pr_y [M(x, y) \text{ accepts}] \geq 1 - \epsilon \).
   
   (b) If \( x \notin L \) then \( \Pr_y [M(x, y) \text{ accepts}] \leq \epsilon \).

2. NP
(a) If \( x \in L \) then \( \Pr_y[M(x, y) \text{ accepts}] > 0 \).
(b) If \( x \notin L \) then \( \Pr_y[M(x, y) \text{ accepts}] = 0 \).

3. \( \text{RP} \)
   (a) If \( x \in L \) then \( \Pr_y[M(x, y) \text{ accepts}] \geq 1 - \epsilon \).
   (b) If \( x \notin L \) then \( \Pr_y[M(x, y) \text{ accepts}] = 0 \).

4. \( \text{coRP} \)
   (a) If \( x \in L \) then \( \Pr_y[M(x, y) \text{ accepts}] = 1 \).
   (b) If \( x \notin L \) then \( \Pr_y[M(x, y) \text{ accepts}] \leq \epsilon \).

5. \( \text{ZPP} \)
   ZPP cannot be naturally defined with this notation, so we can give the following definition.
   \( \text{ZPP} = \text{RP} \cap \text{coRP} \).

5 Choice of error parameter

What is the ideal choice for the maximum permissible error? Is it on the order of \( 1/3 \), \( 1/n^3 \), \( 1/2^n \), or on the order of \( 1 - 1/n^5 \), \( 1 - 1/2^n \)? Let us only look at the class \( \text{RP} \) for now. This can be formalized by looking at the following result.

**Lemma 5.1 (Amplification Lemma)** Suppose an algorithm \( M \) errs with probability \( e(n) \), so that if \( x \in L \) then \( \Pr_y[M(x, y) \text{ accepts}] \geq 1 - e(n) \) and if \( x \notin L \) then \( \Pr_y[M(x, y) \text{ accepts}] = 0 \) (when \( y \) is chosen uniformly). Repeat \( M \) \( k(n) \) times, for some polynomial \( k \). The new algorithm makes errors with the following probabilities.

\[
\begin{align*}
  x \in L & \Rightarrow \Pr_y[M(x, y) \text{ accepts}] \geq 1 - (e(n))^{k(n)}, \\
  x \notin L & \Rightarrow \Pr_y[M(x, y) \text{ accepts}] = 0.
\end{align*}
\]

\( \text{RP} \) with an error probability of \( e(n) \) may be written as \( \text{RP}_{e(n)} \).

If we start with a constant error probability, we can make it as small as \( 1/2^{n^c} \) for any \( c \) in polynomially many iterations of the algorithm. This probability is small enough.

If we start with an error probability \( e(n) = 1 - 1/n^5 \), then

\[
(e(n))^{k(n)} = \left(1 - \frac{1}{n^5}\right)^{n^5 l(n)} \leq \left(\frac{1}{e}\right)^{l(n)},
\]

which is small enough if \( k(n) \) is a sufficiently large degree polynomial.

But if we start with an error probability \( e(n) = 1 - 1/2^n \), then we cannot make the error probability small enough after polynomially many iterations of the algorithm. In this case, \( \text{RP}_{e(n)} = \text{NP} \).

But \( \text{RP}_{1-1/poly(n)} = \text{RP}_{1-1/2^{o(l(n))}} \). So the class \( \text{RP} \) is robust with respect to large changes in the maximum permissible error probability.