Quantum Algorithms
Lecture 6
Probabilistic algorithms and the class BPP II

Zhejiang University

The algorithm

First steps

Assume that a number q is given.

Step 1. If q is even (and $q \neq 2$), then q is composite. If q is odd, we proceed to Step 2.

Step 2. Let $q - 1 = 2^k l$, where $k > 0$, and l is odd.

Step 3. We choose a random $a \in \{1, \dots, q-1\}$.

Step 4. We compute $a^l, a^{2l}, a^{4l}, \dots, a^{2^k l} = a^{q-1}$ modulo q.

Two tests

Test 1. If $a^{q-1} \neq 1$ (where modular arithmetic is assumed), then q is composite.

Test 2. If the sequence $a^l, a^{2l}, \ldots, a^{2^k l}$ (Step 4) contains a 1 that is preceded by anything except ± 1, then q is composite. In other words, if there exists j such that $a^{2^j l} \neq \pm 1$ but $a^{2^{j+1} l} = 1$, then q is composite.

In all other cases the algorithm says that "q is prime" (though in fact it is not guaranteed).

Primality test remarks

If algorithm says that "q is composite", then it is 100% true, that q is composite.

If algorithm says that "q is prime", then it actually may be prime, but also may be composite, in other words, there is high chance, that q is prime.

So we have one-sided error. If q is composite, then algorithm can still say, that it is prime. If q is prime, algorithm will always say that it is prime.

Theorem

If q is prime, then the algorithm always (with probability 1) gives the answer "prime".

If q is composite, then the answer "composite" is given with probability at least 1/2.

Probability remark

To get a probabilistic algorithm in sense of previous definitions, we repeat this test twice: the probability of an error (a composite number being undetected twice) is at most $1/4 < 1/2$.

Beginning of analysis

As we have seen, the algorithm always gives the answer "prime" for prime q.

Assume that q is composite (and odd). If $\gcd(a, q) > 1$ then Test 1 shows that q is composite. So, we may assume that a is uniformly distributed over the group $G = (Z/qZ)^*$. We consider two major cases.

Reminder: $(Z/qZ)^*$ - invertible elements, such a, that have $\gcd(a, q) = 1$. Example: $(Z/12Z)^* = \{1,5,7,11\}$.

Case A

$q = p^{\alpha}$, where p is an odd prime, and $\alpha > 1$.

We show that there is an invertible (mod q)-residue x such that $x^{q-1} \neq 1$, namely $x = (1 + p^{\alpha-1}) \, mod \, q$. Indeed, $x^{-1} = (1 - p^{\alpha-1}) \, mod \, q$, and

$x^{q-1} \equiv (1 + p^{\alpha-1})^{q-1} = 1 + (q - 1)p^{\alpha-1}$ + higher powers of p

$\equiv 1 - p^{\alpha-1} \neq 1 \, (mod \, q)$.

Case A

Therefore Test 1 detects the compositeness of q with probability $\geq 1/2$ (due to Lemma from previous lecture).

Lemma reminded: If $a^{q-1} \neq 1$ for some element $a \in (Z/qZ)^*$, then the Fermat test detects the compositeness of q with probability $\geq 1/2$.

Case B

q has at least two distinct prime factors.

Then $q = uv$, where u and v are odd numbers, $u, v > 1$, and $\gcd(u, v) = 1$. The Chinese remainder theorem says that the group $G = (Z/qZ)^*$ is isomorphic to the direct product $U \times V$, where $U = (Z/uZ)^*$ and $\mathrm{V} = (Z/vZ)^*$, and that each element $x \in G$ corresponds to the pair $((x \bmod u), (x \bmod v))$.

Chinese remainder theorem

Abstract terminology aside, Chinese remainder theorem says that the map $Z/qZ \to (Z/uZ) \times (Z/vZ)$ is one-to-one. In other words, for any u, v the system

$$x \equiv a1 \,(mod\, u),$$
$$x \equiv a2 \,(mod\, v)$$

has a unique, up to (mod q)-congruence, solution.

Note, that this also is true for subsets of mentioned groups, namely: $(Z/qZ)^* \subseteq (Z/qZ)$.

Case B

For any m we define the following subgroup: $G^{(m)} = \{x^m : x \in G\} = Im\,\phi_m$, where $\phi_m : x \to x^m$.

Note that $G^{(m)} = \{1\}$ if and only if $G_{(m)} = G$; this is yet another way to formulate the condition that $a^m = 1$ for all $a \in G$.

Reminder: $G_{(m)} = \{x \in G : x^m = 1\}$.

Case B

Also note that if a is uniformly distributed over G, then a^m is uniformly distributed over $G^{(m)}$.

Indeed, the map $\phi_m : x \to x^m$ is a group homomorphism; therefore the number or pre-images is the same for all elements of $G^{(m)}$. It is clear that $G^{(m)} \cong U^{(m)} \times V^{(m)}$.

Subcase B1

$U^{(q-1)} \neq \{1\}$ or $V^{(q-1)} \neq \{1\}$. This condition implies that $G^{(q-1)} \neq \{1\}$, so that Test 1 detects q being composite with probability at least $1/2$.

Subcase B2

$U^{(q-1)} = \{1\}$ and $V^{(q-1)} = \{1\}$. In this case Test 1 is always passed, so we have to study Test 2. Let us define two sequences of subgroups: $U^{(l)} \supseteq U^{(2l)} \supseteq \cdots \supseteq U^{(2^k l)} = \{1\}$, $V^{(l)} \supseteq V^{(2l)} \supseteq \cdots \supseteq V^{(2^k l)} = \{1\}$.

Note that $U^{(l)} \neq \{1\}$ and $V^{(l)} \neq \{1\}$. Specifically, both $U^{(l)}$ and $V^{(l)}$ contain the residues that correspond to -1. Indeed, both U and V contain -1, and $(-1)^l = -1$ since l is odd.

Reminder: $q - 1 = 2^k l$

Subcase B2

Going from right to left, we find the first place where one of the sets $U^{(m)}$, $V^{(m)}$ contains an element different from 1. In other words, we find $t = 2^s l$ such that $0 \leq s < k$, $U^{(2t)} = \{1\}$, $V^{(2t)} = \{1\}$, and either $U^{(t)} \neq \{1\}$ or $V^{(t)} \neq \{1\}$.

Subcase B2

We will prove that Test 2 shows (with probability at least 1/2) that q is composite. By our assumption $a^{2t} = 1$, so we need to know the probability of the event $a^t \neq \pm 1$. Let us consider two possibilities.

Subcase B2a

One of the sets $U^{(t)}$, $V^{(t)}$ equals $\{1\}$ (for example, let $U^{(t)} = \{1\}$). This means that for any a the pair $((a^t \bmod u), (a^t \bmod v))$ has 1 as the first component. Therefore $a^t \neq -1$, since −1 is represented by the pair (−1, −1).

Subcase B2a

At the same time, $V^{(t)} \neq \{1\}$; therefore the probability that a^t has 1 in the second component is at most 1/2 (by Lagrange's theorem). Thus Test 2 says "composite" with probability at least 1/2.

Subcase B2b

Both sets $U^{(t)}$ and $V^{(t)}$ contain at least two elements: $|U^{(t)}| = c \geq 2$, $|V^{(t)}| = d \geq 2$. In this case a^t has 1 in the first component with probability $1/c$ (there are c equiprobable possibilities) and has 1 in the second component with probability $1/d$.

Subcase B2b

These events are independent due to the Chinese remainder theorem, so the probability of the event $a^t = 1$ is $1/cd$. For similar reasons the probability of the event $a^t = -1$ is either 0 or $1/cd$. In any case the probability of the event $a^t = \pm 1$ is at most $2/cd \leq 1/2$.

Algorithm on input example

Suppose that $q = 161$

Step 1. q is odd number, so we proceed to Step 2.

Step 2. $q - 1 = 2^k l = \mathbf{2^5} * \mathbf{5} = 32 * 5 = 160$

Algorithm on input example

Suppose that $q = 161$, $2^k l = \mathbf{2^5 * 5}$

Step 3. Choose a random $a \in \{1, \dots, 160\}$. Let it be $a = 2$.

Step 4. We compute $2^5, 2^{10}, 2^{20}, 2^{40}, 2^{80}, 2^{160}$ modulo 161.

We get 32, 58, 144, 128, 123, 156.

Test 1. $2^{160} = 156 \neq 1 \ (mod\ 161)$, which means that 161 is composite.

Algorithm on input example

Suppose that $q = 161$, $2^k l = \mathbf{2^5 * 5}$

Step 3. Choose a random $a \in \{1, \dots, 160\}$. Let it be $a = 160$.

Step 4. We compute $160^5, 160^{10}, 160^{20}, 160^{40}, 160^{80}, 160^{160}$ modulo 161.

We get 160, 1, 1, 1, 1, 1.

Test 1. $160^{160} = 1 \ (mod\ 161)$.

Test 2. This test also fails to notice that q is composite, since $160 = -1 \ (mod\ 161)$.

BPP and circuit complexity

BPP ⊆ P/poly

A predicate F belongs to the class P/poly ("nonuniform P") if circuit complexity for F_n is $poly(n)$.

P ⊂ P/poly.

BPP ⊆ P/poly

Let $L(x)$ be a BPP-predicate, and M a probabilistic TM that decides $L(x)$ with probability at least $1 - \varepsilon$. By running M repeatedly, we can decrease the error probability. Recall that a polynomial number of repetitions leads to the exponential decrease. Therefore, we can construct a polynomial probabilistic TM M' that decides $L(x)$ with error probability less that $\varepsilon' < 1/2^n$ for inputs x of length n.

BPP ⊆ P/poly

The machine M' uses a random string r (one random bit for each step). For each input x of length n, the fraction of strings r that lead to an incorrect answer is less than $1/2^n$. Therefore the overall fraction of "bad" pairs (x, r) among all such pairs is less than $1/2^n$. [If one represents the set of all pairs (x, r) as a unit square, the "bad" subset has area $< 1/2^n$.]

BPP $\subseteq$ P/poly

It follows that there exists $r = r*$ such that the fraction of bad pairs (x, r) is less than $1/2^n$ among all pairs with $r = r*$. However, there are only 2^n such pairs (corresponding to 2^n possibilities for x). The only way the fraction of bad pairs can be smaller than $1/2^n$ is that there are no bad pairs at all!

BPP ⊆ P/poly

Thus we conclude that there is a particular string $r*$ that produces correct answers for all x of length n.

The machine M' can be transformed into a polynomial-size circuit with input (x, r). Then we fix the value of r (by setting $r = r*$) and obtain a polynomial-size circuit with input x that decides $L(x)$ for all n-bit strings x.

Nonconstructive proof

This is a typical nonconstructive existence proof: we know that a "good" string $r*$ exists (by probability counting) but have no means of finding it, apart from exhaustive search.

BPP vs PSPACE

It is clear that BPP $\subseteq$ PSPACE. Indeed, the algorithm that counts all values of the string r that lead to the answer "yes" runs in polynomial space. Note that the running time of this algorithm is $2^N poly(n)$, where $N = |r|$ is the number of random bits.

Pseudo-random bits

On the other hand, there is empirical evidence that probabilistic algorithms usually work well with pseudo-random bits instead of truly random ones. So attempts have been made to construct a mathematical theory of pseudo-random numbers.

Pseudo-random generator

A pseudo-random generator is a function g: $B^l \rightarrow B^L$ which transforms short truly random strings (of length l, which can be as small as $O(logL)$) into much longer pseudo-random strings (of length L). “Pseudo-random” means that any computational device with limited resources (say, any Boolean circuit of a given size N computing a function F: $B^L \rightarrow B$) is unable to distinguish between truly random and pseudo-random strings of length L.

Pseudo-random generator

We require that

$|\Pr[F(g(x)) = 1] - \Pr[F(y) = 1]| \leq \delta, \ x \in B^l, \ y \in B^L$

for some constant $\delta < 1/2$, where x and y are sampled from the uniform distributions. (In this definition the important parameters are l and N, while L should fit the number of input bits of the circuit).

Pseudo-random generator

It is easy to show that pseudo-random generators g: $B^{O(logL)} \to B^L$ exist: if we choose the function g randomly, it fulfills the above condition with high probability.

In this statement we can just pick a function in a way that all L bits in a result are random.

What we actually need is a sequence of efficiently computable pseudo-random generators g_L: $B^{l(L)} \to B^L$, where $l(L) = O(logL)$.

Pseudo-random generator

If such pseudo-random generators exist, we can use their output instead of truly random bits in any probabilistic algorithm. The definition of pseudo-randomness guarantees that this will work, provided the running time of the algorithm is limited by $c\sqrt{L}$ (for a suitable constant c) and the error probability ε is smaller than $1/2 - \delta$.

Probabilities

With pseudo-random bits the error probability will be $\varepsilon + \delta$, which is still less than 1/2. The estimate $c\sqrt{L}$ comes from the simulation of a Turing machine by Boolean circuits.

Pseudo-random generator

We decrease the number of needed genuine random bits from L to l. Then we can derandomize the algorithm by trying all 2^l possibilities. If $l = O(logL)$, the resulting computation has polynomial complexity.

Relation to circuit complexity

We do not know whether efficiently computable pseudo-random generators exist. The trouble is that the definition has to be satisfied for "any Boolean circuit of a given size"; this condition is extremely hard to deal with. But we may try to reduce this problem to a more fundamental one — obtaining lower bounds for the circuit complexity of Boolean functions.

Relation to circuit complexity

Even this idea, which sets the most difficult part of the problem aside, took many years to realize. Much work in this area was done in 1980’s and early 1990’s, but the results were not as strong as needed. In late 1990's there has been dramatic progress leading to more efficient constructions of pseudo-random generators and new derandomization techniques.

Relation to circuit complexity

It has been proved that BPP = P if there is an algorithm with running time $\exp(O(n))$ that computes a sequence of functions with circuit complexity $\exp(\Omega(n))$. There is also a work where pseudo-random generators are constructed from arbitrary hard functions in an optimal (up to a polynomial) way.

Derandomization

The process of taking a randomized algorithm and turning it into a deterministic algorithm. This is useful both for practical reasons (deterministic algorithms are more predictable, which makes them easier to debug and gives hard guarantees on running time) and theoretical reasons (if we can derandomize any randomized algorithm we could show results like P=RP, which would reduce the number of complexity classes that complexity theorists otherwise have to deal with).

Adleman's theorem

Short version: RP ⊆ P/poly. This means that given a machine $M(x, r)$ that outputs 1 at least half the time when $x \in L$ and never when $x \notin L$, there is a polynomial-sized string of advice p_n that depends only on the size of the input n and a machine M' such that $M'(x, p_{|x|})$ outputs 1 if and only if $x \in L$.

Derandomization

Derandomization is the process of removing randomness (or using as little of it as possible). It is not currently known if all algorithms can be derandomized without significantly increasing their running time. For instance, in computational complexity, it is unknown whether P = BPP, i.e., we do not know whether we can take an arbitrary randomized algorithm that runs in polynomial time with a small error probability and derandomize it to run in polynomial time without using randomness.

Algorithmic derandomization

Algorithmic derandomization techniques look at a particular randomized algorithm, and using the inherent properties of the problem, analyze the randomized algorithm better to come up with ways to remove randomness from that algorithm. We start with the original randomized algorithm for a particular problem, and improve it to derandomize it.

Complexity theoretic derandomization

Complexity theoretic derandomization techniques refer to a general strategy that can be used to remove randomness from a broad class of algorithms. Typically, this is done by pseudo-random generators, which produce random-looking bits.

Pseudo-random generators

Do such functions exist for all randomized algorithms? If such functions exist for every language in RP (with $l(n) = O(logn)$), RP=P.

RP – randomized polynomial-time. If the correct answer is NO, always returns NO. If the correct answer is YES, then returns YES with probability at least 1/2 (otherwise, returns NO).

For more information about derandomization and examples see:
http://web.cs.iastate.edu/~pavan/633/lec14.pdf

Freivalds' algorithm

A probabilistic randomized algorithm used to verify matrix multiplication.

Given three $n \times n$ matrices A, B, and C, a general problem is to verify whether $A \times B = C$.

Freivalds' algorithm

Input

Three $n \times n$ matrices A, B, and C.

Output

If $A \times B = C$, output "Yes"; Otherwise, output "No".

Procedure

1. Generate an $n \times 1$ random 0/1 vector r.
2. Compute $P = A \times (Br) - Cr$.
3. Output "Yes" if $P = (0,0,\ldots,0)^T$; Output "No", otherwise.

Error and runtime

If $A \times B = C$, then the algorithm always returns "Yes". If $A \times B \neq C$, then the probability that the algorithm returns "Yes" is less than or equal to one half. This is called one-sided error.

By iterating the algorithm k times and returning "Yes" only if all iterations yield "Yes", a runtime of $O(kn^2)$ and error probability of $\leq 1/2^k$ is achieved.

More details and example:
https://en.wikipedia.org/wiki/Freivalds%27_algorithm

Deterministic algorithm 2014

Result published in paper:

Korec I., Wiedermann J. (2014) Deterministic Verification of Integer Matrix Multiplication in Quadratic Time. In: Geffert V., Preneel B., Rovan B., Štuller J., Tjoa A.M. (eds) SOFSEM 2014: Theory and Practice of Computer Science. SOFSEM 2014. Lecture Notes in Computer Science, vol 8327. Springer, Cham. https://doi.org/10.1007/978-3-319-04298-5_33

Thank you for your attention!