38. Probabilistic Algorithms

1 Need for Probabilistic Algorithms

What is NP-hardness: reminder. NP-hardness of a problem means that, unless P = NP, it is not possible to have a feasible algorithm that always produces a solution to each instance of the problem.

Do we really need always? In practice, nothing works always. Even the best computers fail sometimes, with some small (but still positive) probability $\varepsilon > 0$.

From this viewpoint:

- instead of requiring that the algorithm *always* produces the desired result,
- it is sufficient to require that it produces the result in *almost all* cases – i.e., with probability $p \geq 1 - \varepsilon$, for some very small $\varepsilon > 0$.

Such algorithms are known as *probabilistic*.

2 Example of a Problem

Practical problem. Suppose that we want to check whether a given algorithm always produces the correct result. We already know that, in general, this is not possible – even a zero-checker is not possible. Let us see if we can have a probabilistic algorithm for solving this problem.

To be realistic, let us consider a problem with real-valued inputs. For simplicity, let us assume that the desired algorithm has only one input x. Usually, possible values of each input are bounded – e.g., velocities are bounded by the speed of light, etc. For simplicity, let us assume that possible values of the input x are limited by the interval $[0, 1]$. So:

- we have the desired function $f(x)$,
- we have the actual algorithm $g(x)$, and
- we want to check whether $f(x) = g(x)$ for all possible real numbers x from the interval $[0, 1]$.

Let us formulate this problem in precise terms. Of course, we cannot check that the desired equality is always true by simply trying all possible values

x – there are too many of them, many more than what a computer can test in reasonable time. We can only test the inequality for *some* of the values.

Which values should we choose? To answer this question, let us first think how can we describe the algorithms $f(x)$ and $g(x)$. The computer code can include computation of complex functions like $\exp(x)$, $\ln(x)$, etc. However, inside the computer, each such function is transformed into a sequence of elementary hardware supported operations, and these operations are usually addition and multiplication. Thus, what is usually computed is a polynomial with rational coefficients. This is definitely true in most compilers: functions like $\exp(x)$ or $\sin(x)$ are computed by computing the sum of the first few terms in their Taylor series.

So, we arrive at the following precise formulation of the problem.

Precise formulation of the problem.

- *given:* two polynomials $f(x)$ and $g(x)$ with rational coefficients;
- *check* whether $f(x) = g(x)$ for all x from the interval $[0, 1]$.

Comment. This problem would be trivial if both polynomials were given in their standard form $f(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots$: then we would just need to check that for the polynomials $f(x)$ and $g(x)$, the corresponding coefficients are equal to each other. However, the actual algorithms may correspond to more complex forms, such as $(x + a_1) \cdot (x + a_2) \cdot \ldots$ For such polynomials, checking equality is not so easy.

3 Analysis of the Problem and the Main Idea Behind the Probabilistic Algorithm

Reduction to zero-checking. Similarly to how we did it earlier, we can reduce this problem to checking whether a given polynomial is equal to 0: indeed, the values of the two polynomials are equal ($f(x) = g(x)$) if and only if the difference $d(x) = f(x) - g(x)$ is equal to 0. Thus, we get an equivalent formulation of the problem:

- *given:* a polynomial $d(x)$ with rational coefficients;
- *check* whether $d(x) = 0$ for all x from the interval $[0, 1]$.

A known fact about polynomials. It is known that a non-zero polynomial $d(x)$ of order n has no more than n zeros, i.e., no more than n values x_1, ..., x_n for which $d(x) = 0$.

- This is well-known fact for linear (1st order) equations – which usually have a single solution.
- This is a well-known fact for quadratic (2nd order) equations – which usually have two solutions (or one or none).

- This is also true for every n.

This fact leads to the following idea.

Main idea. Software engineering teaches us that to test a program, we need to test it on endpoint of the corresponding ranges, and also on *random* inputs.

For the purpose of such testing (and for other purposes), every programming language has a standard random number generator – that generates a number r uniformly distributed on the interval $[0, 1]$.

Uniform means that the probability to be within each subinterval is proportion (in this case, equal) to the width (= length) of this subinterval. For example:

- the probability to be in the interval $[0, 0.5]$ is equal to 0.5, since the width of this interval is $0.5 - 0 = 0.5$;
- the probability to be in each of the intervals $[0, 0.1]$, $[0.1, 0.2]$, $\ldots$, $[0.9, 1]$ is equal to 0.1, since the width of each of these intervals is

$$0.1 - 0 = 0.2 - 0.1 = \ldots = 1 - 0.9 = 0.1.$$

What can be say about the value $d(r)$?

- If the function $d(x)$ is always 0, then, of course, $d(r) = 0$.
- On the other hand, if $d(x)$ is not always 0, then $d(r)$ is equal to 0 only if r is equal to one of the n roots $x_1, \ldots, x_n$ – or, to be more precise, if it differs from one of the there roots by no more than the machine zero δ (= the small number indistinguishable by a given computer).

So, $d(r) = 0$ if r is within one of the n intervals $[x_i - \delta, x_i + \delta]$. Each of these intervals has width

$$(x_i + \delta) - (x_i - \delta) = x_i + \delta - x_i + \delta = 2\delta.$$

So, the overall width of these intervals is equal to $p_0 = 2n \cdot \delta$. Thus, the probability that $d(r) = 0$ is equal to p_0.

Let us explicitly describe this algorithm – and let us describe how it answers the question that we wanted to check:

- whether the new program $g(x)$ always produces the correct result,
- i.e., in other words, whether the algorithms $f(x)$ and $g(x)$ produce the same value for all possible inputs x.

4 First Probabilistic Algorithm for Solving the Above Problem

Algorithm. To check whether $f(x) = g(x)$ for all $x \in [0, 1]$, generate a random number r and check whether $f(r) = g(r)$.

What will happen if we apply this algorithm.

- If $f(x) = g(x)$ for all x, then $f(r) = g(r)$.
- If for some x, the value $f(x)$ and $g(x)$ are different, then:
  - we get $f(r) = g(r)$ with some small probability p_0 and
  - we get $f(r) \neq g(r)$ with the remaining probability $1 - p_0$.

So what can we conclude about the two functions.

- if $f(r) \neq g(r)$, then we can definitely conclude that the functions $f(x)$ and $g(x)$ are different;
- if $f(r) = g(r)$, then we conclude that most probably, the function $f(x)$ and $g(x)$ are identical (although, with some small probability p_0, they may be actually different).

5 How Can We Make the Conclusion More Reliable: Towards Second Probabilistic Algorithm

What if we want a more reliable answer? In some practical problems, the probability $p_0 = 2n \cdot \delta$ of the wrong answer may be still too high. How can we make the conclusion more reliable – i.e., how can we decrease the probability p_0 of the wrong answer?

Idea. What we are doing is testing the equality on one example. To increase the reliability of the answer, a natural idea is to test inequality on two examples – or, more generally, on $k \geq 2$ examples.

Thus, we arrive at the following algorithm.

Second algorithm. Let us pick a natural number k. To check whether $f(x) = g(x)$ for all $x \in [0, 1]$, we generate k random numbers $r_1, \ldots, r_k$ and check whether $f(r_i) = g(r_i)$ for all $i = 1, \ldots, k$. Then:

- if $f(r_i) \neq g(r_i)$ for some i, we definitely conclude that the functions $f(x)$ and $g(x)$ are different;
- if $f(r_i) = g(r_i)$ for all i, then we conclude that most probably, the functions $f(x)$ and $g(x)$ are equal (although there is some probability that the two functions are actually different).

What is now the probability of a mistake.

- If $f(r_i) \neq g(r_i)$, then, of course, the functions $f(x)$ and $g(x)$ are different.
- However, even if the functions $f(x)$ and $g(x)$ are actually different, it is possible that this algorithm will mistakenly conclude that these function are equal.

The conclusion if a mistake if $f(r_1) = g(r_1)$ *and* $f(r_2) = g(r_2)$, etc. Here:

- the probability of each such event is p_0, and
- these events are independent – since different random numbers are independent.

It is known that for independent events, the probability that all of them occur is equal to the product of the corresponding probabilities. For example, the probability that the coin falls heads twice is equal to $(1/2) \cdot (1/2) = 1/4$.

Thus, the probability that this algorithm made a mistake is equal to the product of k probabilities equal to p_0, i.e., to p_0^k.

What if we want to reach a probability P of the mistake: examples.

- If $p_0 = 1/2$ and we want to have the probability of a wrong answer smaller than $1/100$, then we need to select $k = 7$, then

$$(1/2)^7 = 1/2^7 = 1/128 < 1/100.$$

- If $p_0 = 1/2$ and we want the probability of a wrong answer to be smaller than $1/1000$, then we need to select $k = 10$, then

$$(1/2)^{10} = 1/2^{10} = 1/1024 < 1/1000.$$

Try it yourself. What if $p_0 = 1/3$, and we want the probability of the wrong answer to be $< 1/200$?

What if we want to reach a probability P of the mistake: general case. In general, we need to select k for which $p_0^k \le P$. If we take logarithm of both sides, we get an equivalent inequality $k \cdot \ln(p_0) \le \ln(P)$.

The probabilities are smaller than 1, so their logarithms are negative. If we divide both side of the inequality by a negative number $\ln(p_0)$, the sign of the inequality changes, i.e., we get an equivalent inequality $k \ge \dfrac{\ln(P)}{\ln(p_0)}$.

We want to have the fastest possible test, so we want to minimize k. Thus, k should be the smallest integer satisfying the above inequality. In Java, such an integer is called a *ceiling* of a number; in mathematics, the ceiling of a number x is usually denoted by $\lceil x \rceil$. So, we need to take

$$k = \left\lceil \frac{\ln(P)}{\ln(p_0)} \right\rceil.$$