

COS 330: Great Ideas in Theoretical Computer Science

Fall 2026

Lecture 7

Randomized Algorithms

Module: Randomization

1 Randomized Algorithms

Most of the algorithms we have seen so far are deterministic: they always do the same thing on the same input. A randomized algorithm is an ordinary algorithm with one extra instruction: it may flip a fair coin. This is enough to do a lot, for example, choosing a uniformly random element of $\{1, \dots, m\}$. We will use such operations freely, assuming each takes constant time.

We care about randomized algorithms for several reasons:

- Randomized algorithms are often simpler than deterministic ones.
- Randomized algorithms can be faster than deterministic ones, sometimes dramatically so.
- Some applications require randomness, for example cryptography and privacy.

1.1 What is random, and what is not

For the purposes of this lecture, we will treat the input as worst-case. That means the input to a randomized algorithm is *not* random. Randomness enters only through the algorithm's own coins. So the typical algorithm looks like this:

- The input is some fixed value x .
- The algorithm flips some fair coins.
- It performs some computation, which may depend on the input and the coin flips.
- It outputs an answer, which may be right or wrong.

This is a very different thing from *average-case analysis*, which assumes the input itself is random and shows that the algorithm does well on typical inputs. That regime is also interesting, but unfortunately we won't have time to cover it in this course. For a fixed input, then, the output of the algorithm and its running time are both random variables.

In practice, computers do not flip coins; they use *pseudorandom* number generators, deterministic procedures whose output looks random. We will ignore this and assume truly independent fair coins. Whether that assumption can be removed in general — whether every efficient randomized algorithm has an efficient deterministic counterpart — is one of the major open questions of the field. (We will briefly come back to this question later in the course).

1.2 What makes a randomized algorithm good?

Since randomized algorithms use randomness, their behavior is random. In particular:

- The output is a random variable, which means it may be wrong sometimes. In order for a randomized algorithm to be useful, we will often require something like the following:

$$\text{for every input } x: \quad \Pr_{\text{coins}} [\text{the algorithm is correct on } x] \geq 1 - \delta.$$

- The running time is a random variable, which means it may be slow sometimes. In order for a randomized algorithm to be useful, we will often require something like the following:

$$\text{for every input } x: \quad \Pr_{\text{coins}} [\text{the algorithm runs in time } T \text{ on } x] \geq 1 - \delta.$$

These aren't the only ways to measure correctness or running time. For example, you might recall that the randomized quicksort algorithm has a worst-case running time of $O(n^2)$, but an *expected running time* of $O(n \log n)$. In other words, let T_x be the random variable that is the running time of randomized quicksort on a fixed input x . Then, for every input x , we have $\mathbb{E}[T_x] = O(n \log n)$. It turns out you can convert this guarantee into one that looks like the previous one using one of the concentration inequalities we saw in the last lecture. **Exercise.** Assume that the expected running time of randomized quicksort is at most $3n \log n$. Show that with probability at least 90%, the running time is at most $30n \log n$.

1.3 Why accept any error at all?

An algorithm that is sometimes wrong can sound unacceptable. But the error probability is something we choose. With independent repetition it falls exponentially, so an error probability of 2^{-100} typically costs only about a hundred repetitions of a cheap procedure. That is far smaller than the chance that the computer itself misbehaves during the computation: deterministic algorithms are not error-free in practice either. And because the probability is over the algorithm's own coins rather than the input, a bad outcome is bad luck rather than a bad input — running again with fresh coins gives an independent chance, on the very same input.

2 Checking a Computation

Suppose someone hands you three $n \times n$ matrices A , B and C , and claims that $C = AB$. Perhaps C came from a source you don't trust, and so you want to verify this claim. The obvious approach is to compute AB and compare it with C entry by entry. The standard algorithm takes $\Theta(n^3)$ operations. Cleverer algorithms exist for matrix multiplication, but the asymptotically fastest known ones still take more than $n^{2.37}$ operations, and they are far too complicated to be used in practice. Note that reading the input already takes n^2 operations, so $\Omega(n^2)$ is a lower bound. But then, can we check the claim in $O(n^2)$ time?

2.1 A natural attempt

Let's first consider a natural idea and analyze it. Observe that entry (i, j) of AB is $\sum_l A_{il}B_{lj}$, which takes $O(n)$ operations to compute. So we could pick s entries uniformly at random, compute those entries of AB , and compare them with C .

This is very similar to the polling idea from the last lecture: if C is correct, we always accept, and if C is wrong, we catch it if we happen to sample an entry where C differs from AB . Suppose C differs from AB in exactly one entry. Each sampled entry hits that one entry with probability $1/n^2$, so by the union bound all s samples miss it with probability at least $1 - s/n^2$. To catch the error with probability $1/2$, we need $s \geq n^2/2$ samples, costing $\Theta(n^3)$ operations in total — no better than multiplying.

The problem is what we chose to sample. Polling estimates a fraction to within an additive error; here, even one wrong entry must be detected. Uniformly sampling entries gives that error very little chance to affect the result. Before repeating a random experiment, we should ask how likely it is to reveal the particular failure we need to detect. We need a different experiment, one in which even a single wrong entry has a substantial chance of being noticed.

2.2 Freivalds' algorithm

Given what we just saw, a successful algorithm can't test individual entries. So, here is a simple observation that combines multiple entries into one test. Suppose $v \in \mathbb{R}^n$ is a vector. Then, clearly, if $AB = C$, we have $ABv = Cv$. The interesting thing about this is that we can compute ABv in $O(n^2)$ time, by first computing Bv and then multiplying the result by A . This leads us to the following test:

Definition 2.1 (Freivalds' test). Choose $r \in \{0, 1\}^n$ uniformly at random, by flipping an independent fair coin for each coordinate. Compute $x = A(Br)$ and $y = Cr$. Accept if $x = y$, and reject otherwise.

The perhaps surprising fact is that this test is correct with constant probability:

Theorem 2.1. If $AB = C$, Freivalds' test always accepts. If $AB \neq C$, it accepts with probability at most $1/2$.

Proof. If $AB = C$ then $A(Br) = (AB)r = Cr$ for every r , so the test accepts.

Suppose $AB \neq C$, and let $D = AB - C$, which is not the zero matrix. The test accepts exactly when $Dr = 0$, so we need to show that at most half of the 2^n vectors $r \in \{0, 1\}^n$ satisfy $Dr = 0$. Since $D \neq 0$, some entry D_{ij} is nonzero. It is enough to show that at most half of all vectors r satisfy $(Dr)_i = 0$, because every r with $Dr = 0$ is among them.

Pair up the vectors in $\{0, 1\}^n$ by matching each r with the vector r' obtained by flipping its j -th coordinate, where j is the column of the nonzero entry D_{ij} . This splits $\{0, 1\}^n$ into 2^{n-1} pairs, and the two vectors in a pair agree everywhere except in coordinate j . Therefore

$$(Dr)_i - (Dr')_i = \sum_l D_{il}(r_l - r'_l) = D_{ij}(r_j - r'_j) = \pm D_{ij} \neq 0.$$

So $(Dr)_i$ and $(Dr')_i$ cannot both be zero: at most one vector in each pair satisfies $(Dr)_i = 0$. That is at most 2^{n-1} vectors, which is half of them. Since r is uniform, $\Pr[Dr = 0] \leq \Pr[(Dr)_i = 0] \leq 1/2$. ■

The pairing argument lets us analyze many random choices by focusing on just one of them. Fix all coordinates of r except r_j . Because the coordinates are independent, r_j is still a fair coin, and at most one of its two values can make $(Dr)_i = 0$. This works for every fixing of the other coordinates, so it also bounds the probability when all of them are random. When a random experiment looks complicated, it can help to reveal most of its choices and ask what uncertainty remains.

The test can still miss an error with probability as large as $1/2$. We can reduce this by repeating it independently. The rule for combining the results comes from its *one-sided* error: if $C = AB$ it always accepts, so even one rejection certifies that C is wrong. This gives the following guarantee:

Corollary 2.1. Run Freivalds' test k times with independent random vectors, and accept only if every run accepts. This takes $O(kn^2)$ time. If $AB = C$ it always accepts; if $AB \neq C$ it accepts with probability at most 2^{-k} .

Proof. If $AB = C$, every run accepts. If $AB \neq C$, each run accepts with probability at most $1/2$ by Theorem 2.1, and the runs are independent, so all k accept with probability at most $(1/2)^k$. ■

We can now tune the number of repetitions to make the error probability as small as we like. Suppose we want the error probability to be at most δ . Then we can take $k = \lceil \log_2(1/\delta) \rceil$. For example, suppose we want the error probability to be at most 10^{-9} (one in a billion). Then we can take $k = 30$, and the total cost is $O(kn^2) = O(30n^2) = O(n^2)$, which is still much cheaper than computing AB , for large enough n .

The improvement comes from fresh random choices. Reusing the same vector would repeat exactly the same computation, including any missed error. Independent repetition gives a fixed wrong matrix a new chance to be detected each time. Once one trial has a constant probability of detecting it, each additional trial reduces the remaining failure probability by a constant factor.

3 Finding Contaminated Samples

Let's look at a very different problem. Suppose a lab has n samples, and at most k of them are contaminated. Testing a sample is expensive, but the lab can combine small portions of several samples and test the mixture. Such a *pooled test* should be positive if at least one sample in the pool is contaminated, and negative if they are all clean. Can we identify every contaminated sample using far fewer than n tests?

To make the problem concrete, the set of contaminated samples is a fixed and arbitrary subset of at most k samples, we do not assume contamination is random. We also assume that each test independently reports the wrong answer with probability $1/10$, independently of how we choose the pools. A truly negative pool can be reported positive, and a truly positive pool can be reported negative. This is a simplified testing model: the error probability does not depend on the size or composition of the pool.

This example adds a source of randomness to the model from the beginning of the lecture: besides the algorithm's coins, there is noise in the observations. Our guarantee must hold for every fixed set of at most k contaminated samples, with probability taken over both the random pools and the test errors.

3.1 The natural attempts

The obvious approach is to test each sample separately. This already needs n tests, and with noise, a single test per sample is not enough to give a reliable answer for all of them. Repeating individual tests and taking majorities can reduce the error, but it only increases the number of tests. We want each test to do useful work for several samples at once.

Pooling gives us that possibility. If tests were reliable, a negative result would clear every sample in the pool. A positive result is less informative: it says that someone in the pool is contaminated, but not who. Testing all n samples together illustrates the problem. As soon as one is contaminated, we get a positive result and learn almost nothing about the others.

Noise creates another obstacle. We cannot declare a sample clean just because it appeared in one negative test: the result might be wrong. Instead, we will look for samples that appear in *many* negative tests. A contaminated sample can do so only through test errors. A clean sample has another way: it can land in a pool that contains no contaminated samples.

3.2 Choosing the pools

Let's choose each pool by including each sample independently with probability $p \in (0, 1)$. We will choose different pools independently as well. Leave p free until the analysis tells us what it should be.

Fix one clean sample i . If there are $s \leq k$ contaminated samples, the probability that a pool includes i and excludes all contaminated samples is

$$p(1-p)^s \geq p(1-p)^k.$$

If p is too small, the pool rarely includes i . If p is too large, it usually includes a contaminated sample as well. We want to include samples often enough to learn about them, while keeping the expected number of contaminated samples in a pool around one. Since there are at most k of them, a natural choice is

$$p = \frac{1}{k+1}.$$

Then the expected number of contaminated samples in a pool is $sp \leq k/(k+1) < 1$. The extra 1 also keeps $p < 1$ when $k = 1$: always including every sample would put us back where we started. We will show below that this choice gives a constant probability of excluding all contaminated samples, so a clean sample has a useful chance of appearing in a truly negative pool.

Fix a positive integer T , the number of tests we will perform. Prepare T pools, including each sample in each pool with probability $p = 1/(k+1)$, independently for every sample and every pool, and run one test on each pool. All the pools can be prepared before seeing any results. We will use their reported outcomes to classify the samples, and choose T from the error bound.

3.3 One sample, one test

We now need to turn the test results into an answer. Fix a sample i , and for each test $t \in \{1, \dots, T\}$ define the indicator random variable

$$X_{i,t} = \begin{cases} 1 & \text{if pool } t \text{ includes } i \text{ and its reported result is negative,} \\ 0 & \text{otherwise.} \end{cases}$$

Let $Z_i = \sum_{t=1}^T X_{i,t}$ be the number of negative tests containing i . For this fixed sample, the variables $X_{i,1}, \dots, X_{i,T}$ are independent: each uses the membership choices and the test error for a different pool. Write $q_i = \Pr[X_{i,t} = 1]$, which is the same for every test, and $\mu_i = \mathbb{E}[Z_i] = Tq_i$. We can now show the following:

Lemma 3.1. With $p = 1/(k+1)$,

$$\begin{aligned} \Pr[X_{i,t} = 1 \mid i \text{ is contaminated}] &= \frac{p}{10}, \\ \Pr[X_{i,t} = 1 \mid i \text{ is clean}] &\geq \frac{3p}{10}. \end{aligned}$$

Proof. If i is contaminated, any pool containing it is truly positive. For $X_{i,t} = 1$, the pool must include i and the test must report the wrong answer. These events are independent, so $q_i = p/10$.

Now suppose i is clean, and let $s \leq k$ be the number of contaminated samples. Conditional on including i , the probability of excluding all contaminated samples is $a = (1-p)^s$. This conditioning does not change any other sample's inclusion probability. With probability a the pool is truly negative and is reported negative with probability $9/10$; otherwise it is truly positive and is reported negative with probability $1/10$. Therefore

$$q_i = p \left(\frac{9}{10}a + \frac{1}{10}(1-a) \right) = p \left(\frac{1}{10} + \frac{4}{5}a \right).$$

For our choice of p , using $1+x \leq e^x$ gives

$$a \geq (1-p)^k = \left(1 + \frac{1}{k}\right)^{-k} \geq \frac{1}{e} > \frac{1}{4}.$$

Substituting $a \geq 1/4$ yields $q_i \geq p(1/10 + 1/5) = 3p/10$. ■

The count Z_i was chosen to distinguish the two possible statuses of sample i . Counting all negative tests would not do that: we need to count negative tests that actually include i . This is a useful way to choose a random variable for an algorithm: identify evidence whose probability changes between the cases we want to distinguish, then count it across independent trials.

There is a gap between the two expectations: $Tp/10$ for a contaminated sample, and at least $3Tp/10$ for a clean one. Put the threshold halfway between these two values, at $Tp/5$: declare sample i clean if $Z_i > Tp/5$, and contaminated otherwise. A contaminated sample would need unusually many test errors to cross this threshold. A clean sample would need unusually few negative results to fall below it. As we increase T , we expect both kinds of mistake to become less likely. The concentration inequalities from the last lecture let us turn this intuition into a guarantee.

3.4 From an expectation gap to a reliable decision

Let's first bound the chance of misclassifying one fixed sample. Recall the two Chernoff bounds from Lecture 6. For a sum X of independent Bernoulli random variables with mean $\mu > 0$,

$$\Pr[X \geq (1+\eta)\mu] \leq e^{-\eta^2\mu/(2+\eta)} \quad (\eta > 0),$$

and

$$\Pr[X \leq (1-\eta)\mu] \leq e^{-\eta^2\mu/2} \quad (0 < \eta < 1).$$

We use η for the deviation parameter to keep δ for the desired failure probability of the whole algorithm.

Lemma 3.2. For every sample i , the rule that declares it clean exactly when $Z_i > Tp/5$ is wrong with probability at most $e^{-Tp/60}$.

Proof. Suppose first that i is contaminated. By Lemma 3.1, $\mu_i = Tp/10$, so the classification is wrong only if $Z_i > Tp/5 = 2\mu_i$. Apply the upper-tail bound with $\eta = 1$:

$$\Pr[i \text{ is declared clean}] \leq \Pr[Z_i \geq 2\mu_i] \leq e^{-\mu_i/3} = e^{-Tp/30}.$$

Now suppose i is clean. We have $\mu_i \geq 3Tp/10$, so the threshold satisfies $Tp/5 \leq (2/3)\mu_i$. A wrong classification therefore requires a downward deviation of at least a third of the mean. The lower-tail bound with $\eta = 1/3$ gives

$$\Pr[i \text{ is declared contaminated}] = \Pr[Z_i \leq Tp/5] \leq \Pr[Z_i \leq (2/3)\mu_i] \leq e^{-\mu_i/18} \leq e^{-Tp/60}.$$

The first case has an even smaller bound, so $e^{-Tp/60}$ works in both cases. ■

The choice of inequality matters here. The difference between the threshold and either mean is at least $Tp/10$. Applying Hoeffding directly with this additive deviation would give $2e^{-Tp^2/50}$, requiring on the order of $1/p^2$ tests for constant error. Chernoff sees that the deviation is a constant fraction of the mean and gives an exponent proportional to Tp , requiring only on the order of $1/p$ tests. Since $p = 1/(k+1)$, this saves a factor on the order of k in the bound on the number of tests.

3.5 Getting every classification right

As with Freivalds' test, we can now choose the amount of repetition from the error bound. There is one extra step: our answer contains n classifications, and we need all of them to be correct. It is enough to make each one's error probability at most δ/n .

Theorem 3.1. Fix $0 < \delta < 1$. For every set of at most k contaminated samples, the random pools and classification rule identify that set exactly with probability at least $1 - \delta$, using

$$T = \left\lceil 60(k+1) \ln \frac{n}{\delta} \right\rceil = O\left(k \log \frac{n}{\delta}\right)$$

pooled tests. The probability is over the pool choices and the independent test errors.

Proof. Let F_i be the event that sample i is misclassified. By the law of total probability, splitting on the status of i ,

$$\Pr[F_i] = \Pr[F_i \mid i \text{ is contaminated}] \Pr[i \text{ is contaminated}] + \Pr[F_i \mid i \text{ is clean}] \Pr[i \text{ is clean}].$$

The status of i is fixed, not random, so exactly one term on the right is present; but Lemma 3.2 bounds *both* conditional probabilities by $e^{-Tp/60}$, so the bound holds regardless of which case is the truth. With $p = 1/(k+1)$,

$$\Pr[F_i] \leq e^{-Tp/60} \leq e^{-\ln(n/\delta)} = \frac{\delta}{n}.$$

By the union bound,

$$\Pr[\text{some sample is misclassified}] = \Pr\left[\bigcup_{i=1}^n F_i\right] \leq \sum_{i=1}^n \Pr[F_i] \leq n \cdot \frac{\delta}{n} = \delta. \quad \blacksquare$$

Notice where independence was needed. For one sample, its indicators across independent tests let us apply Chernoff. Classifications of different samples can be dependent, because the same test contributes to several counts. This causes no problem: the union bound does not require independence.

The distinction between one correct classification and an entirely correct answer also determined how many tests we needed. A constant error probability per sample would leave too many opportunities for a mistake. Working backward from a total failure probability of δ gave us a target of δ/n per sample, and the exponential tail bound turned that target into the $\log(n/\delta)$ factor in the number of tests.

We measure the cost of this algorithm by the number of tests it performs. When $k \log(n/\delta)$ is much smaller than n , the number of tests is sublinear in the number of samples. The constant 60 comes from convenient bounds; we have not tried to optimize it. The dependence on k is the important part: the algorithm benefits from contamination being sparse, without knowing which samples are contaminated.