1 Randomized algorithms

1.1 Introduction

A randomized algorithm is a deterministic algorithm that in addition to its input is also given a string of random bits. Usually, it is assumed that bits are uniformly distributed. The goal is to make our algorithm work with some probability over the choice of random bits. The randomized algorithm should “work” for all inputs I for most strings of random bits.

Sometimes it may be more relevant to look at average case analysis. In this case, we assume the probability distribution over all inputs, and we want our algorithm to behave well on most of the inputs. But the assumption of what the input distribution looks like may actually cause problems if this assumption is wrong.

The main question is how exactly randomization, i.e. adding a string of random bits, can actually help us come up with a better algorithm.

1.2 Paradigms of randomized algorithms

- **Foiling an adversary.** This is one of the cases where randomization is necessary. Let’s assume that someone is shaping inputs that our algorithm gets and trying to make it fail. For example, if we have a deterministic algorithm, then our clever adversary may find a set of inputs in which the algorithm fails/gives bad results. This is particularly relevant in cryptography. That is exactly why we use random keys when we perform encryption. If the keys are fixed and the adversary finds them, the adversary may easily perform the decryption. To counter this, we add some randomization to our algorithm. The adversary may know how the algorithm works, but not know what keys (random bits) are used.

Another good example is hashing. In hashing, we map a bigger set (the universe) to a smaller set (some array of a fixed length, for example). We try to avoid collisions – when more than one element of the universe will be mapped to the same element in the array. To counter this, we may use a random function, because if the input is not distributed randomly (if the adversary tries to use the same input many times), there could be a large number of collisions, which could lead our hash tables to explode. However, if the function is chosen randomly, the adversary won’t be able to guess the desired input. This example shows the difference between randomized algorithms and average-case algorithms. If the function is random-like, it will work well on average case inputs. However, there is no deterministic hash function that works on all inputs because the universe is much bigger than the set to which it is mapped. Randomization
doesn’t exclude collisions. It even won’t decrease the average case of collisions, but it will make the number of collisions not depend on the input. It will be impossible for the adversary to make an “attack” and fail our hash table (as long as they are not aware of the random bits we are using).

- **Abundance of witnesses.** Sometimes the task for the algorithm may be to determine whether a given input has some qualities. For example, a primality testing. We are given a number \( n \) and we need to determine whether it is prime or not. There are many deterministic algorithms that can solve this problem, but they are not fast enough. The Solovay-Strassen algorithm is a randomized algorithm that can solve this problem faster. The algorithm is based on the fact from number theory.

**Theorem 1** Let \( n \) be a prime number greater than 2. Then, for any integer \( a \):

\[
a^{n-1} \equiv \left(\frac{a}{n}\right)(\text{mod } n)
\]

Here, \( \left(\frac{a}{n}\right) \) is a Jacobi symbol that can be computed in \( O((\log n)^2) \) time. If \( n \) is not prime, this statement fails for at least half of possible values of ”\( a \)”s. So in order to test whether \( n \) is prime or not, we need to take a set of possible ”\( a \)”s and test this statement on each of them. If \( n \) is composite, there will be many cases in which this statement will fail. If we construct this set randomly, it will increase the chances of choosing those ”\( a \)”s that will be likely to fail if the \( n \) is composite. Each number ”\( a \)”s is considered as a witness for number \( n \). They are called the witnesses for the compositeness of \( n \). The accuracy of the algorithm depends on the number of ”\( a \)”s we pick.

- **Random sampling.** We are given a boolean formula and we want to approximate the number of satisfying assignments. Checking all possible inputs (there will be \( 2^n \) possible strings of bits of the length \( n \)) will be too complicated. As we need to find only the approximate number of satisfying assignments, we can use some facts from statistics: when we need to find/verify something about the population, we need only a sample of it. And the accuracy of our estimation will depend on the size and the randomness of this sample. So we can evaluate our boolean function only on a random sample of strings and then estimate the answer. The important thing is to make sure that the sampling is random. Without randomization (for example, let’s just pick the first \( k \) string of bits in lexicographical order) we may get a subset of inputs in which our function almost always fails (or behaves similarly). This will not be useful to us. When we use randomization over the sample of strings, however, the fraction of overall accepting inputs will be approximately equal to the fraction of accepting samples. Another important thing is to make sure that the sampling is done from the domain of the considered function (in case of the boolean function, the domain will be all binary strings of the length \( n \)).

Markov chains may be used to sample from complicated distributions. Markov chains are random processes in which we move from one state to another randomly. When
we walk over states randomly, the final distribution will converge to the sample distribution. It is possible to show that the convergence will be fast enough. By random sampling of complex distributions we can find what properties these distributions have.