CS271 Randomness & Computation **Fall 2022**

Lecture 1: August 25

Instructor: Alistair Sinclair

1.1 Introduction

There are two main flavors of research in randomness and computation. The first looks at randomness as a computational resource, like space and time. The second looks at the use of randomness to design algorithms. This course focuses on the second area; for the first area, see other graduate classes such as CS278.

The starting point for the course is the observation that, by giving a computer access to a string of random bits, one can frequently obtain algorithms that are far simpler and more elegant than their deterministic counterparts. Sometimes these algorithms can be "derandomized," although the resulting deterministic algorithms are often rather obscure. In certain restricted models of computation, one can even prove that randomized algorithms are more efficient than the best possible deterministic algorithm.

In this course we will look at both *randomized algorithms* and *random structures* (random graphs, random boolean formulas etc.). Random structures are of interest both as a means of understanding the behavior of (randomized or deterministic) algorithms on "typical" inputs, and as a mathematically clean framework in which to investigate various probabilistic techniques that are also of use in analyzing randomized algorithms.

1.1.1 Model of Computation

We first need to formalize the model of computation we will use. Our machine extends a standard model (e.g., a Turing Machine or random-access machine), with an additional input consisting of a stream of perfectly random bits (fair coin flips). This means that for a fixed input, the output is a random variable of the input bit stream, and we can talk about the probability of events such as Pr[output = 275] or Pr[algorithm halts].

In the above model, we can view a computation of a randomized algorithm as a path in a binary tree. At each step of the algorithm, we branch right or left with probability $\frac{1}{2}$ based on the random bit we receive. For decision problems, we can label each leaf of this tree with a "yes" or a "no" output. Obviously, we derive no benefit from randomization if all outputs agree.

We generally divide randomized algorithms into two classes: *Monte Carlo* algorithms and *Las Vegas* algorithms.

Monte Carlo algorithms always halt in finite time but may output the wrong answer, i.e., Pr[error] is larger than 0. These algorithms are further divided into those with *one-sided* and *two-sided* error.

One-sided error algorithms only make errors in one direction. For example, if the true answer is "yes", then Pr["yes"] $\geq \epsilon > 0$ while if the true answer is "no" then Pr["no"] = 1. In other words, a "yes" answer is guaranteed to be accurate, while a "no" answer is uncertain. We can increase our confidence in a "no" answer by running the algorithm many times (using independent random bits each time); we can then output "yes" if we see any "yes" answer after t repetitions, and output "no" otherwise. The probability of error in this

scheme is at most $(1-\epsilon)^t$, so if we want to make this smaller than any desired $\delta > 0$ it suffices to take the number of trials t to be $O(\log \frac{1}{\delta})$ (where the constant in the O depends on ϵ).

The class of decision problems solvable by a polynomial time Monte Carlo algorithm with one-sided error is known as RP ("Randomized Polynomial time").

Two-sided error algorithms make errors in both directions; thus, e.g., if the true answer is "yes" then $\Pr[\text{"yes"}] \geq \frac{1}{2} + \epsilon$, and if the true answer is "no" then $\Pr[\text{"no"}] \geq \frac{1}{2} + \epsilon$. In this case we can increase our confidence by running the algorithm many times and then taking the *majority vote*. The following standard calculation shows that the number of trials t required to ensure an error of at most δ is again $O(\log \frac{1}{\delta})$.

Proof: The probability that the majority vote algorithm yields an error is equal to the probability that we obtain at most $t/2$ correct outputs in t trials, which is bounded above by

$$\begin{aligned}
\sum_{i=0}^{\lfloor t/2 \rfloor} \binom{t}{i} \left(\frac{1}{2}+\epsilon\right)^i \left(\frac{1}{2}-\epsilon\right)^{t-i} &\leq \sum_{i=0}^{\lfloor t/2 \rfloor} \binom{t}{i} \left(\frac{1}{2}+\epsilon\right)^{t/2} \left(\frac{1}{2}-\epsilon\right)^{t/2} \\
&\leq \left(\frac{1}{4}-\epsilon^2\right)^{t/2} \sum_{i=0}^{\lfloor t/2 \rfloor} \binom{t}{i} \\
&\leq \left(\frac{1}{4}-\epsilon^2\right)^{t/2} 2^t \\
&= (1-4\epsilon^2)^{t/2}.
\end{aligned}$$

Taking $t = \frac{2\log \delta^{-1}}{\log(1-4\epsilon^2)^{-1}} = C \log \frac{1}{\delta}$, where C is a constant depending on ϵ, makes this at most δ. ■

The class of decision problems solvable by a polynomial time Monte Carlo algorithm with two-sided error is known as BPP ("Bounded-error Probabilistic Polynomial time"). It should be clear (**exercise!**) that the containments $\mathrm{P} \subseteq \mathrm{RP} \subseteq \mathrm{BPP}$ hold. The relationship between BPP and NP is unknown: on the one hand, note that while $\mathrm{RP} \subseteq \mathrm{NP}$ is obvious, $\mathrm{BPP} \subseteq \mathrm{NP}$ is not because of the two-sided error allowed in BPP; however, it is known that $\mathrm{BPP} \subseteq \mathrm{NP}^{\mathrm{NP}}$, the second level of the polynomial hierarchy (i.e., every problem in BPP can be solved by an NP machine with an additional NP oracle) [Lau83,Sip83]. In the other direction, it is very unlikely that $\mathrm{NP} \subseteq \mathrm{BPP}$ because this would imply that every problem in NP has polynomial size circuits (since this property is known to hold for BPP [Adl78]), which in turn would imply that the polynomial hierarchy collapses.

In a **Las Vegas algorithm**, the output is always correct but the running time may be unbounded. However, the *expected* running time is required to be bounded. Equivalently (**exercise!**), we require the running time to be bounded but allow the algorithm to output either a correct answer or a special symbol "?", so that the probability of outputting "?" is at most 1/2.

Note that we can always turn a Las Vegas algorithm into a Monte Carlo algorithm by running it for a fixed amount of time and outputting an arbitrary answer if it fails to halt. This works because we can bound the probability that the algorithm will run past a fixed limit (using Markov's inequality and the fact that the expectation is bounded). The reverse is apparently not true because of the strict "correct output" requirement of Las Vegas algorithms. Thus, there is no general scheme for converting a Monte Carlo algorithm into a Las Vegas one.

The class of decision problems solvable by a polynomial time Las Vegas algorithm is known as ZPP ("Zero-error Probabilistic Polynomial time").

Exercise: Show that $\mathrm{ZPP} = \mathrm{RP} \cap \text{co-RP}$. [co-RP is the class of languages whose complement is in RP.]

In our model, every outcome has probability $a/2^l$ where $0 \leq a \leq 2^l$ is an integer and l is the running time of the algorithm. However, we shall feel free to write such statements as "with probability 2/3 do..." or "pick

$x \in S$ u.a.r." The justification for this is that we can clearly approximate such branching probabilities to arbitrary accuracy by a relatively small number of random coin flips, and absorb the small errors into the error probability of the algorithm. The only caveat is that we must not design algorithms that rely for their operation on branching with very precise probabilities. If in doubt, we should always return to the basic model.

Random Number Generators: Truly random strings of bits are hard to come by; numerous sources based on physical processes have been proposed, including shot noise, output from quantum devices, decay of radioactive nuclei etc. In practice, we use *pseudorandom generators*, which are actually deterministic but output long sequences of "seemingly random" numbers. The most common example is the Linear Congruential Generator, which takes a seed x_0 and emits further numbers from the sequence

$$x_{t+1} = (ax_t + c) \bmod m,$$

where a, c, m are positive integers, with m typically chosen as something like $2^{31}-1$. For a detailed discussion of practical generators, see [Knuth97], and for the theoretical background see [Luby96] or [Gold99]. See also [Hayes93] for a discussion of potential pitfalls when using pseudo-random generators.

It is an open question whether randomness truly helps, from a complexity theoretic perspective. Indeed, the question whether P = RP or P = BPP (i.e., whether every polynomial time randomized algorithm can be "de-randomized" with at most a polynomial loss of efficiency) is among the most important open problems in theoretical CS. We will barely touch on this question in this class, other than to quote the following remarkable theorem due to Impagliazzo and Wigderson [IW97]:

Theorem 1.1 *If* SAT *(or any other problem solvable in time* $2^{O(n)}$*) cannot be solved by circuits of size* $2^{o(n)}$*, then* BPP = P*.*

(Other versions of this theorem are also available, in which both the hardness assumption on SAT and the derandomization conclusion are weakened.) Thus we are faced with choosing which of two cherished beliefs to jettison: on the one hand, that NP-complete problems like SAT are truly hard; and on the other hand, that randomization can lead to dramatically more efficient algorithms. Most people in complexity theory prefer to retain the first and drop the second, leading to the conjecture that BPP = P. However, this does not imply that studying randomized algorithms is fruitless: even if it is eventually proved that every randomized algorithm can be derandomized with a polynomial slow-down, the resulting (black-box) deterministic algorithms would owe their existence to the original randomized ones and are unlikely to be either practical or enlightening.

We also point out a philosophical dilemma. Suppose we can prove that randomness *does* help, so that there is (say) a randomized algorithm that runs in polynomial time and solves a problem for which no deterministic polynomial time algorithm exists (i.e., BPP $\neq$ P). Then, if we were to implement this algorithm using a pseudorandom number generator, we would know that the resulting algorithm cannot work efficiently since what we have really implemented is a purely deterministic algorithm for the same problem!

For most of the rest of the class, we will ignore these issues, and concentrate on designing algorithms with the aid of a perfect random number generator, because thinking of algorithms in this way has proven to be a very useful practical tool and has given rise to many important algorithmic and mathematical ideas.

1.2 Checking matrix multiplication

This is one of the first published uses of randomization, and is due to Freivalds [Frei77]. Suppose we have an untrustworthy oracle for multiplying matrices, and we want to check whether the oracle was correct on a certain input.

Input: Three $n \times n$ matrices, A, B, C.

Output: "Yes" if $AB = C$. "No" otherwise.

A trivial deterministic algorithm would be to run a standard multiplication algorithm and compare each entry of AB with each entry of C. Matrix multiplication requires $O(n^3)$ time naively, or $O(n^{2.376})$ time using the (essentially) best-known algorithm [CW87]. (See [VW12,LeG14,AVW20] for more recent relatively small numerical improvements in the exponent.)

Using randomization, however, we can answer the question in $O(n^2)$ time, with extremely small probability of error. The idea is to pick a random n-dimensional vector, multiply both sides by it, and see if they're equal. Formally:

pick a vector $r = (r_1, \cdots, r_n)$ such that each r_i is i.i.d. uniform from a finite set S, with $|S| \geq 2$
if $(AB)r \neq Cr$ **then**
    output *no*
else
    output *yes* (with one-sided error)
endif

This algorithm runs in $O(n^2)$ time because matrix multiplication is associative, so $(AB)r$ can be computed as $A(Br)$, thus requiring only three matrix-vector multiplications for the algorithm.

The algorithm also has one-sided error. If we find an r such that $(AB)r \neq Cr$, then certainly the answer is *no*. If it outputs *yes*, we could have been unlucky, but we now show that this happens with probability $\leq \frac{1}{2}$.

Claim 1.2 *If $AB \neq C$, then* $\Pr[(AB)r = Cr] \leq \frac{1}{|S|}$.

Proof: Define $D = AB - C$. Assuming $D \neq 0$, without loss of generality $d_{11} \neq 0$. (If $d_{11} = 0$, just permute the rows and columns of the matrix.) We need to show that $\Pr[Dr = 0] \leq \frac{1}{|S|}$.

If $Dr = 0$, then

$$(Dr)_1 = \sum_{i=1}^{n} d_{1i} r_i = 0$$

and so

$$r_1 = -\frac{1}{d_{11}}(d_{12} r_2 + \cdots + d_{1n} r_n).$$

Thus, for each choice of the values $r_2, \ldots, r_n$, there is only one value for r_1 that could have caused $Dr = 0$. Since r_1 is chosen uniformly from S, $\Pr[Dr = 0]$ is at most $\frac{1}{|S|}$. ■

To decrease the probability of error, repeat the algorithm t times (independently). If we get *no* at least once, output *no*; otherwise, output *yes*. Since each trial is independent, $\Pr[error] \leq |S|^{-t} \leq 2^{-t}$. The meta-algorithm still has run-time $O(n^2)$. (Another way to reduce the probability of error is to increase $|S|$, though this is less practical.)

The above analysis illustrates the simple but powerful "principle of deferred decisions": we first fixed the choices for $r_2, \ldots, r_n$, and then considered the effect of the random choice of r_1 *given* those choices.

We have demonstrated **searching for witness**, a basic technique for using randomization to check whether a predicate $P(r)$ is true for all values r:

1. Pick a random value r from some suitable set
2. If $P(r)$ is false then output "no" else output "yes"

This algorithmic template outputs "no" if and only if it finds a value r (a "witness") for which $P(r)$ is false. Searching for witnesses works only when the density of witnesses is large (as a fraction of the size of the set of possible witnesses) whenever $P(r)$ is false. In our algorithm above, the density is at least $(1 - \frac{1}{|S|}) \geq \frac{1}{2}$.

1.3 Checking associativity

We now give a more sophisticated application of the witness-searching paradigm, due to Rajagopalan and Schulman [RS96].

Input: A binary operator $\circ$ on a set X of size n.

Output: "Yes" if $\circ$ is associative (i.e., $\forall i, j, k \in X : (i \circ j) \circ k = i \circ (j \circ k)$); "No" otherwise.

The naive deterministic algorithm runs in time $O(n^3)$ by simply checking every possible triple i, j, k.

As with matrix multiplication, we could try searching for a witness by picking a triple uniformly at random. Unfortunately, for this problem there are examples of non-associative operators with only a constant number of witness triples, so we would need to pick $O(n^3)$ triples before we find a witness with good probability.

Exercise: Show there exists a family of binary operators (one for each n) such that the number of "non-associative" triples is at most some constant (independent of n).

It turns out we can design a randomized $O(n^2)$ time algorithm for this problem, and the trick is to pick random *sets* of elements from X instead of individual triples. We write $\mathcal{X} = 2^X$ to denote the set of subsets of X.

We can think of an element $R \in \mathcal{X}$ as a binary vector of length $|X|$, with the i^{th} bit r_i denoting the presence or absence of that element of X. We alternately write R as $\sum_{i \in X} r_i \cdot i$.

We define the following operations on $\mathcal{X}$:

$$R + S = \sum_{i \in X} (r_i + s_i) \cdot i$$

$$R \circ S = \sum_{i,j \in X} (r_i s_j) \cdot (i \circ j)$$

$$\alpha R = \sum_i (\alpha r_i) \cdot i \quad \text{for a scalar } \alpha,$$

where operations on the coefficients are performed mod 2. (If X were a group, this would be the "group algebra" of X. But we don't know that X is a group because we don't know that $\circ$ is associative.)

Our algorithm is as follows:

pick R,S,T independently and u.a.r. from $\mathcal{X}$
if $(R \circ S) \circ T \neq R \circ (S \circ T)$ **then**
    output *no*
else
    output *yes* (with one-sided error)
endif

Note that this algorithm runs in $O(n^2)$ time. Calculating $R \circ S$ takes $O(n^2)$ time since R and S are both vectors of length n. The result U is also an element in $\mathcal{X}$, so we can then calculate $U \circ T$ in $O(n^2)$ time as well.

To see that the algorithm is correct, we first make a simple but important observation:

Exercise: Show that $\circ$ is associative over $X \Leftrightarrow \circ$ is associative over $\mathcal{X}$.

You are encouraged to verify this statement (the proof is not hard).

The above Exercise ensures that the algorithm has only one-sided error; i.e., it can err only in the case that $\circ$ is not associative (by failing to find a witness triple R, S, T). The key fact is that this is not terribly likely to happen:

Claim 1.3 *If $\circ$ is not associative, then at least 1/8 of the possible triples R, S, T are witnesses, i.e.,*

$$\Pr[(R \circ S) \circ T = R \circ (S \circ T)] \leq 7/8.$$

Proof: Assume $\circ$ is not associative, so there exists at least one triple (i^*, j^*, k^*) in X such that $(i^* \circ j^*) \circ k^* \neq i^* \circ (j^* \circ k^*)$.

Take the space $\mathcal{X}^3$ representing all possible triples (R, S, T) that our algorithm may choose. We will partition $\mathcal{X}^3$ into disjoint groups of eight triples, such that each group contains at least one witness.

To do this, let R_0, S_0, T_0 be any sets such that $i^* \notin R_0$, $j^* \notin S_0$, $k^* \notin T_0$, and define $R_1 = R_0 \cup \{i^*\}$, $S_1 = S_0 \cup \{j^*\}$, $T_1 = T_0 \cup \{k^*\}$. With this notation, the groups of eight alluded to above are $\Psi = \{\{R_a, S_b, T_c\} : a, b, c \in \{0, 1\}\}$.

With some abuse of notation, define $f(i, j, k) = (i \circ j) \circ k - i \circ (j \circ k)$. (Here f should be thought of as a function from $\mathcal{X}^3$ to $\mathcal{X}$, with $\mathcal{X}$ viewed as a vector space as indicated earlier, and we are using i to denote the singleton subset $\{i\}$.) We extend f to the whole of $\mathcal{X}^3$ by

$$f(R, S, T) = \sum_{i \in R, j \in S, k \in T} f(i, j, k) \quad = (R \circ S) \circ T - R \circ (S \circ T).$$

Now consider any group from Ψ. Using the inclusion-exclusion principle, we can write $f(i^*, j^*, k^*)$ in terms of the eight elements of this group as

$$\begin{aligned} f(i^*, j^*, k^*) \quad = \quad & f(R_1, S_1, T_1) - f(R_1, S_1, T_0) - f(R_1, S_0, T_1) - f(R_0, S_1, T_1) + \\ & f(R_1, S_0, T_0) + f(R_0, S_1, T_0) + f(R_0, S_0, T_1) - f(R_0, S_0, T_0). \end{aligned}$$

Since $f(i^*, j^*, k^*)$ is non-zero, at least one of the terms on the right hand side must be non-zero also. Thus at least one in every eight triples from $\mathcal{X}^3$ are witnesses to the non-associativity of $\circ$, as claimed. [Note that the sum $f(R_1, S_1, T_1)$ contains the term $f(i^*, j^*, k^*)$, which is by definition nonzero. This does not mean that $f(R_1, S_1, T_1)$ is necessarily nonzero, however, since there could be other witness triples that cancel $f(i^*, j^*, k^*)$.] ■

References

[AVW20] J. Alman and V. Vassilevska Williams, "A refined laser method and faster matrix multiplication." *Proceedings of ACM-SIAM SODA*, 2020.

[Adl78] L. Adleman. "Two theorems on random polynomial time." *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1978, pp. 75–83.

[CW87] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions." *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*, 1987, pp. 1–6.

[Frei77] R. Freivalds, "Probabilistic machines can use less running time." *Proceedings of IFIP*, 1977, pp. 839–842.

[Gold99] O. Goldreich, *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Springer-Verlag, Berlin, 1999.

[Hayes93] B. Hayes, "The Science of Computing: The Wheel of Fortune." *American Scientist* **81**, 1993, pp. 114–118.

[IW97] R. Impagliazzo and A. Wigderson, "P = BPP unless E has subexponential circuits: Derandomizing the XOR lemma." *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*, 1997, pp. 220–229.

[Knuth97] D.E. Knuth, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms.* Addison-Wesley, Reading, MA, 1997.

[Lau83] C. Lautemann, "BPP and the polynomial hierarchy." *Information Processing Letters* **17**, 1983, pp. 215–217.

[LeG14] F. Le Gall, "Powers of tensors and fast matrix multiplication." *Proceedings of ISSAC*, 2014, pp. 296–303.

[Luby96] M. Luby, *Pseudorandomness and Cryptographic Applications.* Princeton University Press, Princeton, NJ, 1996.

[RS96] S. Rajagopalan and L. Schulman, "Verifying identities." *Proceedings of 37th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1996, pp. 612–616.

[Sip83] M. Sipser, "A complexity-theoretic approach to randomness." *Proceedings of the 15th ACM Symposium on Theory of Computing (STOC)*, 1983, pp. 330–335.

[VW12] V. Vassilevska Williams, "Multiplying matrices faster than Coppersmith-Winograd." *Proceedings of ACM STOC*, 2012, pp. 887–898.