ℙROBABILISTIC 𝕋HINKING

Get high on probability.

Aryaman Singhal, Kumar Saurav, Lavesh Mangal

April 2023

Abstract

The field of randomized algorithms is hard and diverse, characterized by simple but hard to analyze algorithms that achieve some very unexpected results. This article is our attempt to synthesize the essence of our exposure to randomized algorithms while also providing jumping points for independent exploration. Much of this article is based on the book by Motwani and Raghavan, the course CS648A taught by Surender Baswana and a few disparate mathematical tools that the mentors have picked up from different places.

"Just see this."

1 Randomized Algorithms

An **algorithm** is a finite sequence of rigorous instructions, typically used to solve a class of specific problems or to perform a computation. If this is your first introduction to randomized algorithms, all the algorithms that you will have studied until now would have the following two features:

1. For any given input $\mathcal{I}$ to the algorithm, the running time of the algorithm is fixed.
2. The algorithm computes a correct answer for each possible input $\mathcal{I}$ to the algorithm.

Algorithms that have the above two properties are called **deterministic algorithms**. Examples of deterministic algorithms that you may be familiar with are binary search, merge-sort, depth-first-search in a graph, Kruskal and Prim's minimum spanning tree algorithms, computing articulation points or bridges in a graph, primality testing by checking up to $\sqrt{n}$ etc.

For each of these algorithms the way that an algorithm $\mathcal{A}$ proceeds for a given input $\mathcal{I}$ is *determined*, hence the name.

What randomized algorithms do is get rid of one of the two properties above. An algorithm that always gives the correct answer but whose runtime is a random variable is called a **Las Vegas algorithm**.
An algorithm that gives the correct answer with some probability is called a **Monte Carlo algorithm**.
These algorithms are named after world centres for gambling to represent the probabilistic risk that we take in the randomized paradigm.

2 Paradigms

Note: This section is copied verbatim from the preface to *Randomized Algorithms* by Rajeev Motwani and Prabhakar Raghavan and the chapter numbers herein refer to those.
A handful of general principles lies at the heart of almost all randomized algorithms, despite the multitude of areas in which they find application. We briefly survey these here, with pointers to chapters in which examples of these principles may be found. The following summary draws heavily from ideas in the survey paper by Karp.

1. **Foiling an adversary:** The classical adversary argument for a deterministic algorithm establishes a lower bound on the running time of the algorithm by constructing an input on which the algorithm fares poorly. The input thus constructed may be different for each deterministic algorithm. A randomized algorithm can be viewed as a probability distribution on a set of deterministic algorithms. While the adversary may be able to construct an input that foils one (or a small fraction) of the deterministic algorithms in the set, it is difficult to devise a single input that is likely to defeat a randomly chosen algorithm. While this paradigm underlies the success of any randomized algorithm, the most direct examples appear in Chapter 2 (in game tree evaluation), Chapter 7 (in efficient proof verification), and Chapter 13 (in online algorithms).

2. **Random sampling:** The idea that a random sample from a population is representative of the population as a whole is a pervasive theme in randomized algorithms. Examples of this paradigm arise in almost all the chapters, most notably in Chapters 3 (selection algorithms), 8 (data structures), 9 (geometric algorithms), 10 (graph algorithms), and 11 (approximate counting).

3. **Abundance of witnesses:** Often, an algorithm is required to determine whether an input (say, a number x) has a certain property (for example, "is x prime?"). It does so by finding a witness that x has the property. For many problems, the difficulty with doing this deterministically is that the witness lies in a search space that is too large to be searched exhaustively. However, by establishing that the space contains a large number

of witnesses, it often suffices to choose an element at random from the space. The randomly chosen item is likely to be a witness; further, independent repetitions of the process reduce the probability that a witness is not found on any of the repetitions. The most striking examples of this phenomenon occur in number theory (Chapter 14).

4. **Fingerprinting and hashing:** A long string may be represented by a short fingerprint using a random mapping. In some pattern-matching applications, it can be shown that two strings are likely to be identical if their fingerprints are identical; comparing the short fingerprints is considerably faster than comparing the strings themselves (Chapter 7). This is also the idea behind hashing, whereby a small set S of elements drawn from a large universe is mapped into a smaller universe with a guarantee that distinct elements in S are likely to have distinct images. This leads to efficient schemes for deciding membership in S (Chapters 7 and 8) and has a variety of further applications in generating pseudo-random numbers (for example, two-point sampling in Chapter 3 and pairwise independence in Chapter 12) and complexity theory (for instance, algebraic identities and efficient proof verification in Chapter 7). Random re-ordering. A striking use of randomization in a number of problems in data structuring and computational geometry involves randomly re-ordering the input data, followed by the application of a relatively naive algorithm. After the re-ordering step, the input is unlikely to be in one of the orderings that is pathological for the naive algorithm. (Chapters 8 and 9).

5. **Load balancing:** For problems involving choice between a number of resources, such as communication links in a network of processors, randomization can be used to "spread" the load evenly among the resources, as demonstrated in Chapter 4. This is particularly useful in a parallel or distributed environment where resource utilization decisions have to be made locally at a large number of sites without reference to the global impact of these decisions. Rapidly mixing Markov chains. For a variety of problems involving counting the number of combinatorial objects with a given property, we have approximation algorithms based on randomly sampling an appropriately defined population. Such sampling is often difficult because it may require computing the size of the sample space, which is precisely the problem we would like to solve via sampling. In some cases, the sampling can be achieved by defining a Markov chain on the elements of the population and showing that a short random walk using this Markov chain is likely to sample the population uniformly (Chapter 1 1).

6. **Isolation and symmetry breaking:** In parallel computation, when solving a problem with many feasible solutions it is important to ensure that the different processors are working toward finding the same solution. This requires isolating a specific solution out of the space of all feasible solutions without actually knowing any single element of the solution space.

A clever randomized strategy achieves isolation, by implicitly choosing a random ordering on the feasible solutions· and then requiring the processors to focus on finding the solution of the lowest rank. In distributed computation, it is often necessary for a collection of processors to break a deadlock and arrive at a consensus. Randomization is a powerful tool in such deadlock-avoidance, as shown in Chapter 12.

7. **Probabilistic methods and existence proofs:** It is possible to establish that an object with certain properties exists by arguing that a randomly chosen object has the properties with positive probability. Such an argument gives no clue as to how to find such an object. Sometimes, the method is used to guarantee the existence of an algorithm for solving a problem; we thus know that the . algorithm exists, but have no idea what it looks like or how to construct it. This raises the issue of non-uniformity in algorithms (Chapters 2 and 5).

3 Overview of Mathematical Preliminaries

Fairly often the design and analysis of randomized algorithms comes down to showing the following:

1. For Las Vegas Algorithms, that the expected running time is small. One of the ways this is often achieved is by showing that certain random variables associated with the running time (say the number of comparisons of two reals in a sorting algorithm) have small expected value and that it takes values close to the expected value with high probability(concentration of the running time).

2. For Monte Carlo algorithms, that the probability that the algorithm outputs an incorrect answer is very small, usually inverse polynomial in the input size.

The following mathematical tools often come in handy in the analysis of such algorithms:

1. **Linearity of Expectation:** For a collection of random variables $X_1, X_2 \ldots X_n$ which **need not be independent**:

$$\mathbb{E}[X_1 + X_2 \ldots X_n] = \mathbb{E}X_1 + \mathbb{E}X_2 \ldots \mathbb{E}X_n$$

It is worth re-emphasizing that the variables need not be independent. What this allows us to do, is to take a local view of a large process with lots of dependencies. Consider the following riddle from Brainstellar.

2. **Union Bound:** The following result which is easy to prove is often used in establishing bounds. For a collection of events $\{A_i : i \in \{1, 2, \ldots n\}\}$,

$$\mathbb{P}\left(\bigcup_{i=1}^{n} A_i\right) \leq \sum_{i=1}^{n} \mathbb{P}(A_i)$$

3. **Chernoff Bound:** Useful for proving concentration inequalities. For independent Bernoulli random variables $X_1, X_2 \dots X_n$, let $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathbb{E}X$. Then, for $\delta > 0$:

$$\mathbb{P}(X \geq (1+\delta)\mu) < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}$$

And for $0 < \delta < 1$:

$$\mathbb{P}(X \leq (1-\delta)\mu) < \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu}$$

There are a bunch of variations of the above results, for example, Hoeffding's inequality, method of bounded differences and some approximate inequalities that get rid of the denominators. Mainly the idea is that the probability of deviating by more than $\delta\mu$ is roughly inverse exponential in the deviation.

4. **A few approximations and inequalities:**
   (a) $e^{\frac{x}{1+x}} \leq 1 + x \leq e^{x}$
   (b) $\binom{n}{k} < \frac{1}{e}\left(\frac{en}{k}\right)^{k}$
   (c) More bounds

4 Examples of randomized algorithms

The prototypical examples of randomized algorithms are those of randomized QuickSort and the algorithm for finding a min-cut of a graph. Please see Chapter-1 of the book by Motwani and Raghavan for descriptions of the above

5 Probabilistic Method

See sections 3.3 and 2.1.2 of the following handouts.

6 Parting shots

There are a lot of points for further exploration here. If you are interested in distributed system design, several algorithms used for computation there are randomized because of the decentralized nature of networks.

If you enjoy the probabilistic method, then you might find the fields of graph theory and Ramsey theory fascinating. There are some very interesting results relating random walks on graphs to the theory of electric circuits.

If you're interested in exploring more, the mentors would be happy to direct you toward good resources and/or professors that you might enjoy working with. In biblical terms, "*Ask and you shall receive*".