Chapter 1

Introduction to Randomized Algorithms

By Sariel Har-Peled, April 26, 2022[1]

People tell me it's a sin
To know and feel too much within.
I still believe she was my twin, but I lost the ring.
She was born in spring, but I was born too late.
Blame it on a simple twist of fate.

A little twist of fate, Bob Dylan

1.1. What are randomized algorithms?

Randomized algorithms are algorithms that makes random decision during their execution. Specifically, they are allowed to use variables, such that their value is taken from some random distribution. It is not immediately clear why adding the ability to use randomness helps an algorithm. But it turns out that the benefits are quite substantial. Before listing them, let start with an example.

1.1.1. The benefits of unpredictability

Consider the following game. The adversary has a equilateral triangle, with three coins on the vertices of the triangle (which are, numbered by, I don't known, 1,2,3). Initially, the adversary set each of the three coins to be either heads or tails, as she sees fit.

At each round of the game, the player can ask to flip certain coins (say, flip coins at vertex 1 and 3). If after the flips all three coins have the same side up, then the game stop. Otherwise, the adversary is allowed to rotate the board by $0, 120$ or -120 degrees, as she seems fit. And the game continues from this point on. To make things interesting, the player does not see the board at all, and does not know the initial configuration of the coins.

A randomized algorithm. The randomized algorithm in this case is easy – the player randomly chooses a number among $1, 2, 3$ at every round. Since, at every point in time, there are two coins that have the same side up, and the other coin is the other side up, a random choice hits the lonely coin, and thus finishes the game, with probability $1/3$ at each step. In particular, the number of iterations of the game till it terminates behaves like a geometric variable with geometric distribution with probability $1/3$ (and thus the expected number of rounds is 3). Clearly, the probability that the game continues for more than i rounds, when the player uses this random algorithm, is $(2/3)^i$. In particular, it vanishes to zero relatively quickly.

A deterministic algorithm. The surprise here is that there is no deterministic algorithm that can generate a winning sequence. Indeed, if the player uses a deterministic algorithm, then the adversary can simulate the algorithm herself, and know at every stage what coin the player would ask to flip (it is easy to verify that flipping two coins in a step is equivalent to flipping the other coin – so we can restrict ourselves to a single coin flip at each step). In particular, the adversary can rotate the board in

the end of the round, such that the player (in the next round) flips one of the two coins that are in the same state. Namely, the player never wins.

The shocker. One can play the same game with a board of size 4 (i.e., a square), where at each stage the player can flip one or two coins, and the adversary can rotate the board by $0, 90, 180, 270$ degrees after each round. Surprisingly, there is a deterministic winning strategy for this case. The interested reader can think what it is (this is one of these brain teasers that are not immediate, and might take you 15 minutes to solve, or longer [or much longer]).

The unfair game of the analysis of algorithms. The underlying problem with analyzing algorithm is the inherent unfairness of worst case analysis. We are given a problem, we propose an algorithm, then an all-powerful adversary chooses the worst input for our algorithm. Using randomness gives the player (i.e., the algorithm designer) some power to fight the adversary by being unpredictable.

1.1.2. Back to randomized algorithms

(A) **Best.** There are cases where only randomized algorithms are known or possible, especially for games. For example, consider the 3 coins example given above.

(B) **Speed.** In some cases randomized algorithms are considerably faster than any deterministic algorithm.

(C) **Simplicity.** Even if a randomized algorithm is not faster, often it is considerably simpler than its deterministic counterpart.

(D) **Derandomization.** Some deterministic algorithms arises from derandomizing the randomized algorithms, and this are the only algorithm we know for these problems (i.e., discrepancy).

(E) **Adversary arguments and lower bounds.** The standard worst case analysis relies on the idea that the adversary can select the input on which the algorithm performs worst. Inherently, the adversary is more powerful than the algorithm, since the algorithm is completely predictable. By using a randomized algorithm, we can make the algorithm unpredictable and break the adversary lower bound.

Namely, randomness makes the algorithm vs. adversary game a more balanced game, by giving the algorithm additional power against the adversary.

1.1.3. Randomized vs average-case analysis

Randomized algorithms are not the same as ***average-case analysis***. In average case analysis, one assumes that is given some distribution on the input, and one tries to analyze an algorithm execution on such an input.

On the other hand, randomized algorithms do not assume random inputs – inputs can be arbitrary. As such, randomized algorithm analysis is more widely applicable, and more general.

While there is a lot of average case analysis in the literature, the problem that it is hard to find distribution on inputs that are meaningful in comparison to real world inputs. In particular, for numerous cases, the average case analysis exposes structure that does not exist in real world input.

1.2. Examples of randomized algorithms

1.2.1. 2SAT

The input is a 2SAT formula. That is a 2CNF boolean formula – that is, the formula is a conjunction of clauses, where each clause is made out of two literals, which are ored together. A literal here is either a variable or its negation. For example, the input formula might be

$$F = (x_1 \vee \overline{x_2}) \wedge (\overline{x_2} \vee x_3) \wedge \cdots \wedge (\overline{x_1} \vee x_{17}).$$

(Here, $\vee$ is a boolean or, and $\wedge$ is a boolean and.) Assume that F is using n variables (say $x_1, \ldots, x_n \in \{0,1\}$), and m clauses. The task at hand is to compute a satisfying assignment for F. That is, determine what values has to be assigned to $x_1, \ldots, x_n$.

This problem can be solved in linear time (i.e., $O(n+m)$) by a somewhat careful and somewhat clever usage of directed graphs and strong connected components. Here, we present a much simpler randomized algorithm – we will present some intuition why this algorithm works. We hopefully will provide a more detailed formal proof later in the course.

1.2.1.1. The algorithm

The algorithm starts with an arbitrary assignment to the variables of F. If F evaluates to TRUE, then the algorithm is done. Otherwise, there must be a clause, say $C_i = \ell_i \vee \ell_i'$, that is not satisfied. The algorithm randomly chooses (with equal probability) one of the literals of C_i, and flip the value assigned to variable in this literal. Thus, if the algorithm chosen $\ell_i' = \overline{x_{17}}$, then the algorithm would flip the value of x_{17}. The algorithm continues to the next iteration.

Claim 1.2.1 (Proof later in the course). *If F has a satisfying assignment, then the above algorithm performs $O(n^2)$ iterations in expectation, till it finds a satisfying assignment. Thus, the expected running time of this algorithm is $O(n^2 m)$.*

1.2.1.2. Intuition

Fix a specific satisfying assignment Ξ to F. Assume X_i is the number of variables in the assignment in the beginning of the ith iteration that agree with Ξ. If $X_i = n$, then the algorithm found Ξ, and it is done. Otherwise, X_i changes by exactly one at each iteration. That is $X_{i+1} = X_i + 1$ or $X_{i+1} = X_i - 1$. If both variables of C_i are assigned the "wrong" value (i.e., the negation of their value in Ξ), then $X_{i+1} = X_i + 1$. The other option is that one of the variables of C_i is assigned the wrong value. The probability that the algorithm guess the right variable to flip is $1/2$. Thus, we have $X_{i+1} = X_i + 1$ with probability $1/2$, and $X_{i+1} = X_i - 1$ with probability $1/2$.

Thus, the execution of the algorithm is a random process. Starting with X_1 being some value in the range $[\![0 : n]\!] = \{0, \ldots, n\}$, the question is how long do we have to run this process till $X_i = n$? It turns out that the answer is $O(n^2)$, because essentially this process is related to the random walk on the line, described next.

1.2.2. Walk on the grid

1.2.2.1. Walk on the line

Let $\mathbb{Z}$ denote the set of all integer numbers. Consider the random process, that starts at time zero, with the "player" being at position $X_0 = 0$. In the ith step of the game, the player randomly choose with

probability half to go left – that is, to move to $X_{i-1} - 1$, or with equal probability to the right (i.e., $X_i = X_{-1} + 1$). The sequence $\mathcal{X} = X_0, X_1, \ldots$ is a *random walk* on the integers. A natural question is how many times would the walk visit the origin, in the infinite walk $\mathcal{X}$?

Well, the probability of the random walk at time $2n$ to be in the origin is exactly $\alpha_n = \binom{2n}{n}/2^{2n}$. Indeed, there are 2^{2n} random walks of length $2n$. For the walk to be in the origin at time $2n$, the walk has to be balanced – equal number of steps have to be taken to the left and to the right. The number of binary sequences of length $2n$ that have exactly n 0s and n 1s is $\binom{2n}{n}$.

Exercise 1.2.2. Prove that $\binom{2n}{n} = \Theta(2^{2n}/\sqrt{n})$. (An easy proof follows from using Stirling's formula, but there is also a not too difficult direct elementary proof).

As such, we have that $c_-/\sqrt{n} \leq \alpha_n \leq c_+/\sqrt{n}$, where c_-, c_+ are two constants. Thus, the expected number of times the random walk visits the origin is

$$\sum_{n=1}^{\infty} \alpha_n \geq c_- \sum_{n=1}^{\infty} \frac{1}{\sqrt{n}} = +\infty.$$

(Why the last argument is valid would be explained in following lectures.)

Namely, the random walk visits the origin infinite number of times.

1.2.2.2. Walk on the two dimensional grid

The same question can be asked when the underlying set is $\mathbb{Z} \times \mathbb{Z}$ – that is the two dimensional integer grid. Here, the walk starts at the origin $X_0 = (0, 0)$, and in the ith step, the walk moves with (equal) probability to one of the four adjacent locations. That is, if $X_{i-1} = (x_{i-1}, y_{i-1})$, then X_i is one of the following four locations with equal probability:

$$(x_i - 1, y_i), \quad (x_i + 1, y_i), \quad (x_i, y_i - 1), \quad \text{and} \quad (x_i, y_i + 1).$$

As before, one can ask what is the number of times this random walk visits the origin. Let β_n be the probability of being in the origin at time $2n$.

Exercise 1.2.3. Prove that $\beta_n = \alpha_n^2 = \Theta(1/n)$. (There is a nifty trick to prove this. See if you can figure it out.)

Arguing as above, we have that the expected number of times the walk visits the origin is $\sum_{n=1}^{\infty} \Theta(1/n) = +\infty$. Namely, the walk visit the origin infinite number of times.

1.2.2.3. Walk on the two dimensional grid

The same question can be asked when the underlying set is $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ – as before the walk starts at the origin, and at each step the walk goes to one of the six adjacent cells. It turns out that the probability of being in the origin at time (say) $6n$ is $\Theta(1/n^{3/2})$ (the proof is not clean or easy in this case), and as such, the expected number of times this walk visits the origin $\sum_{n=1}^{\infty} \Theta(1/n^{3/2}) = O(1)$. Surprise!

1.2.3. RSA and primality testing

Oversimplifying the basic idea, RSA works as follows. Compute two huge random primes p and q, release $n = pq$ as the public key. Given n one can encrypt a message, but to decrypt it, one needs both p and q. So, we rely here on the computational hardness of factoring.

Using RSA thus boils down to computing large prime numbers. Fortunately, the following is known (and somewhat surprisingly, is not difficult to prove):

Theorem 1.2.4. *The range $[\![n]\!] = \{1, \ldots, n\}$ contains $\Theta(n/\log n)$ prime numbers.*

As a number n can be written using $O(\log n)$ digits, that essentially means that a random number with t digits has probability $\approx 1/t$ to be a prime number. Namely, primes are quite common.

Fortunately, one can test quickly whether or not a random number is a prime.

Theorem 1.2.5. *Given a positive integer n, it can be written using $T = \lceil \log_{10} n \rceil$ digits. Furthermore, one can decide in $O(T^4) = O(\log^4 n)$ randomized time if n is prime. More precisely, if n is not prime, the algorithm would return "not prime" with probability half, if it is prime, it would return "prime".*

A natural way to decide if a number n with t bits is prime, is to run the above algorithm (say), $10t$ times. If any of the runs returns that the number is not prime, then we return "not prime". Otherwise, we return the number of is a prime. The probability that a composite number would be reported as prime is $1/2^{10t} \leq 1/10^t$, which is a tiny number, for t, say, larger than 512.

This gives us an efficient way to pick random prime numbers – the time to compute such a number is polynomial in the number of bits its uses. Now, we can deploy RSA as computing large random prime numbers is the main technical difficulty in computing it.

1.2.4. Min cut

In the most basic version of the min-cut problem, you are given an undirected graph G with n vertices and m edges, and the task is to compute the minimum number of edges one has to delete so the graph becomes disconnected.

Consider the following algorithm – it randomly assigns the edges of G weights from the range $[0, 1]$. It then computes the MST T of this graph (according to the random weights on the edges). Let e be the heaviest edge in T. Removing it breaks T into two subtrees, with two disjoint sets of vertices S and T. Let (S, T) denote the set of all the edges in G that have one end point in S and one in T. The algorithm outputs the edges of (S, T) as the candidate to be the minimum cut.

The following result is quite surprising.

Theorem 1.2.6. *The above algorithm always outputs a cut, and it outputs a min-cut with probability $\geq 2/n^2$.*

In particular, it turns out that if you run the above algorithm $O(n^2 \log n)$ times, and returns the smallest cut computed, then with probability $\geq 1 - 1/n^{O(1)}$, the returned cut is the minimum cut! This algorithm has running time (roughly) $O(n^4)$ – it can be made faster, but this is already pretty good.