Predictably uncertain: glimpses of randomized algorithms

Neeldhara Misra

STATUTORY WARNING
This document is a draft version.

We are out on a leisurely and somewhat random walk that visits a few classic examples in the very rich literature of randomized algorithms. At a very high level, there are two distinct ways in which the flavor of randomness creeps into the design and analysis of algorithms: one is when the algorithm itself decides to make random decisions, and the other is when we want to understand the expected behavior of an algorithm in a "typical situation". The former scenario broadly falls under the umbrella of randomized algorithms, while the latter is referred to as the probabilistic analysis of algorithms. We will see one example of how the probabilistic analysis of a deterministic algorithm morphs into a proposal for introducing randomness in the algorithm itself. We will also see how to derandomize a randomized algorithm, by replacing the work done by the "random engine" by a deterministic program, typically at the expense of additional time. Somewhere on the way, we will pause briefly to look at the complexity classes associated with randomized algorithms, and point out some of the intriguing questions that lurk there.

1 Random Walks to Solve SAT

The question of satisfiability is a special case of the general class of constraint satisfaction problems. A number of real-life situations can be modeled as instances of constraint satisfaction: for instance, the problem of planning the semester time table is rife with constraints — you wouldn't want to assign the same lecture hall to two courses at the same time, you wouldn't want a linear algebra class to be scheduled at the same time that a probability class is scheduled, and so on — the central question being if there is, at all, some assignment of timings to classes in such a way that all constraints are met. In general, constraint satisfaction problems are not easy at all (as you might imagine, if you have ever faced a situation similar to the task of determining the course schedule!) Even the restriction to questions of satisfiability (where the variables are typically boolean and there is a fixed notion of how a constraint is to be satisfied) retain a fair amount of modeling power, and continue to be hard. The question of satisfiability for 2CNF formulas is a further specialization, and is restrictive enough to be manageable, while still being interesting.

We will restrict ourselves to the question of whether a 2CNF formula admits a satisfying assignment. We note that the problem admits a number of deterministic polynomial-time algorithms, but the randomized approach that we describe, apart from being a neat case study, is also easier to implement (as is often the case with randomized algorithms) and even admits generalizations to 3CNF formulas when combined with greedy heuristics — note that no deterministic polynomial time algorithms are known or expected for 3CNF satisfiability. Before describing the algorithm, for the sake of completeness, we introduce the terminology involved. The next couple of paragraphs can be safely skipped by readers who are familiar with the question.

Let $X = \{x_1, \ldots, x_n\}$ be a fixed finite set, which we will refer to as the *variables*. A *literal* is either a variable or the negation of a variable, denoted by $\bar{x}$. The class of propositional formulas over X is defined recursively as follows:

every literal is a propositional formula. If ϕ is a propositional formula, then $\overline{\phi}$ is a propositional formula. Finally, if α and β are propositional formulas, then so are $\alpha \vee \beta$ and $\alpha \wedge \beta$. The formula $(x_1 \vee \ldots \vee x_n)$ is called the disjunction of $x_1, \ldots, x_n$, and the formula $(x_1 \wedge \ldots \wedge x_n)$ is called the conjunction of $x_1, \ldots, x_n$.

An assignment is a function $f : V \to \{0, 1\}$, and $f(\phi)$ is again defined recursively: if ϕ is x, then $f(\phi) = f(x)$, and if ϕ is $\overline{x}$, then $f(\phi) = 1 - f(x)$. If $\phi = \overline{\alpha}$, then $f(\phi) = 1 - f(\alpha)$. If $\phi = (\alpha \wedge \beta)$, then $f(\phi) = \min\{f(\alpha), f(\beta)\}$ and finally, if $\phi = (\alpha \vee \beta)$, then $f(\phi) = \max\{f(\alpha), f(\beta)\}$. An assignment f is said to be *satisfying* for a formula ϕ if $f(\phi)$ evaluates to one.

A 2CNF formula over a X involves conjunctions of clauses, where a clause is a disjunction of at most two literals. Note that $f(\phi)$ evaluates to 1 if and only if every clause in ϕ has a literal l such that $f(l) = 1$.

Let ϕ be a 2CNF formula over the variables $X := \{x_1, \ldots, x_n\}$. The randomized algorithm begins — appropriately enough — with a random assignment $f : X \to \{0, 1\}$. If it's a lucky day, f is already satisfying for ϕ — there is nothing left to do.

On the other hand, if f does not satisfy ϕ, then there is at least one clause that is dissatisfied. Pick such a clause (if there are many, choose one arbitrarily). Further, randomly choose one of the variables involved in the clause — say x — and flip the assignment that f made to x. The new assignment now satisfies the clause that we picked out, and potentially some more, while possibly unsettling some other clauses. For example, in the formula:

$$(x \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{z} \vee \overline{y}),$$

the assignment $(x = 1, y = 0, z = 0)$ fails to satisfy the second clause, and when we flip, say, x from 1 to 0, the second clause comes to life while the first clause becomes unsatisfied. Of course, we might have chosen z for the flip, in which case everyone would have been happy. There are algorithms that try to be locally greedy with the flip, but we'll keep things simple and pick one of the two variables to flip with equal probability.

If the input formula indeed had a satisfying assignment, we claim that expected number of flips the algorithm makes before it hits a satisfying assignment is $O(n^2)$.

To see this, begin by fixing some satisfying assignment to the variables, say τ. Let f_t denote the assignment made by the random variable after t flips, and let $k(t)$ denote the number of variables for which the random assignment agrees with τ, that is, $f_t(x) = \tau(x)$. Notice that when $k(t) = n$, the algorithm is done.

The value of $k(0)$ (the number of variables "assigned correctly" with respect to τ in the initial assignment) can be anything between 0 and n. But as flips are made, notice that the value of $k(t+1)$ is either one more or one less than $k(t)$. Indeed, if x is the variable that was chosen for the t^{th} flip, then either $f_t(x) = \tau(x)$ and $f_{t+1} \neq \tau(x)$ (in which case $k(t+1) = k(t) - 1$), or $f_t(x) \neq \tau(x)$ and $f_{t+1} = \tau(x)$ (in which case $k(t+1) = k(t) + 1$).

This is beginning to look remarkably like a random walk on the line. What is the probability that $k(t+1) = k(t)+1$? Notice that in a clause that is not satisfied by f_t, at least one of the variables is such that $f_t(x) \neq \tau(x)$, since we know that τ is a satisfying assignment, and in particular, satisfies the clause in question. So with probability at least one-half, we move forward along the line. While we cannot model a transition probability of "at least one-half" in a standard Markov chain, let's model our situation with forward transition probabilities of exactly one-half, after all, that would be the worst-case scenario for the algorithm.

So let's consider the Markov chain along a path, with states $\{0, 1, \ldots, n\}$, where the the transition probabilities from i to $(i+1)$ and from $(i+1)$ to i are $1/2$ each, for $1 \leqslant i < n$. Note that the transition probability from 0 to 1 is 1, and the probability of staying at n is also 1. (Why?)

It should be clear that the expected number of flips made by the algorithm before it finds a satisfying assignment is bounded by the expected time for us to go from the starting state s to state n. The algorithm might actually finish faster in many ways (maybe forward transition probabilities were much better than one-half at many states, maybe the starting assignment was a lucky one, maybe the algorithm just hit a different satisfying assignment on the way) — what we are proposing is a bound on the expected number of flips in the worst-case scenario.

Let E_i denote the expected number of time steps taken to go from state i to state n. Notice that $E_n = 0$ and $E_0 = E_1 + 1$. Further,

$$E_i = p_{i,i-1}(E_{i-1} + 1) + p_{i,i+1}(E_{i+1} + 1) = \frac{E_{i-1} + E_{i+1}}{2} + 1.$$

Solving the above, we arrive at the following:

$$E_i = \sum_{j=i}^{n} (2j + 1) \leqslant n(n-1) + n = n^2.$$

It is easy to verify the first closed form by induction. Let's begin with a suitable expression for E_{i+1}:

$$E_i = \frac{E_{i-1} + E_{i+1}}{2} + 1 \Rightarrow E_{i+1} = 2E_i - E_{i-1} - 2$$

Applying the induction hypothesis, we have:

$$E_{i+1} = 2\sum_{j=i}^{n}(2j+1) - \sum_{j=i-1}^{n}(2j+1) - 2 = \sum_{j=i+1}^{n}(2j+1) + (2i+1) - (2(i-1)+1) - 2 = \sum_{j=i+1}^{n}(2j+1).$$

Therefore, as we were able to see a couple of steps before, the expected number of time steps needed to arrive at n starting from any point $0 \leqslant i \leqslant n$ is bounded by n^2. This is also the expected number of flips performed by the algorithm before it arrives at a satisfying assignment.

Notice that all our careful modeling breaks down if the input formula happens to be one that is not satisfiable, in which case there is no τ that we can work with, and as for the algorithm — it would end up flipping forever. In practice, one would truncate the algorithm after $2n^2$ flips, the reasoning being that if the algorithm hasn't found a satisfying assignment yet, chances are that the formula was unsatisfiable to begin with. The exact chances can be determined with the help of Markov's inequality.

To this end, it's easier to think of the error probability — what is the probability that the formula was satisfiable, but the algorithm needed more than $2n^2$ flips to arrive at the assignment? Well, when the formula is satisfiable we know that the expected running time of the algorithm is bounded by n^2. So we simply apply Markov's inequality on X, the running time of the algorithm:

$$P(X > 2n^2) < E(X)/2n^2 \Rightarrow P(X > 2n^2) < 1/2.$$

Thus the probability that we answer correctly when we truncate is at least one-half. We can amplify this success probability by simply allowing for more flips, and the trade-off between time and confidence is quite clear: to achieve a $(11/c)$ probability of success, we would need cn^2 flips, so the longer we spend flipping, we find the chances of the formula being satisfiable are vanishing.

Randomized algorithms that always terminate with the right answer are called Las Vegas algorithms. The time spent by a Las Vegas algorithm is potentially unbounded, although some definitions demand that the expected running time should be finite. In contrast, a Monte Carlo algorithm spends a fixed amount of time on every input, but terminates with result that is potentially incorrect. Monte Carlo algorithms are required to have bounded error probabilities. Using the Markov inequality like we did above, a Las Vegas algorithm can typically be converted into a Monte Carlo algorithm by early termination (whenever the algorithm structure allows for it).

2 From Probabilistic Analysis to Randomized Algorithms: QuickSort

Quicksort is a divide and conquer sorting algorithm developed by Tony Hoare. Given a list of numbers, quicksort picks an element (called the pivot) and partitions the list based on whether they are bigger or smaller than the pivot. Each part is sorted recursively, and the recursively sorted lists are easily merged with the chosen element. The base case of the recursion is easy because singleton elements don't need any sorting.

Quicksort can be implemented as a completely deterministic algorithm — the choice of the pivot at every stage can be, for example, the first element of the input list (or the left-most element of the array in which the list is stored). The running time would then hinge on how the pivots partition the list. If we keep getting splits into roughly equal-sized sub-lists, then the depth of recursion is roughly proportional to $\log n$. On the other hand, if the splits are severely uneven, then the depth of recursion can be as bad as $n - 1$ — consider quicksort running on a fully sorted list, and observe that it will go the full nine yards ($n - 1$ recursive calls).

Given this kind of variance in the running time, one might want to analyze the average case complexity of quicksort - if one is given a random permutation, what is the expected running time of quicksort? It turns out (and will follow from what we prove) that the the average case complexity of quicksort is $O(n \log n)$.

The fact that this algorithm works well on most inputs may not be superbly useful if the inputs we are faced with are coming from a distribution that not sufficiently random (for instance, perhaps the lists we are dealing with are always mostly sorted). What might be a good workaround to ensure that the algorithm's average-case complexity is maintained, irrespective of the input distribution? Well — here's a simple trick: before running quicksort as we would above, we randomly shuffle all the elements of the input! Now, well, things are as good as random. (Of course, if you know a *lot* about your input distribution, you would probably be better off trying exploit that more directly.)

In other words, we are adding randomization into the algorithm itself. Incidentally, we could either shuffle the input first, or we could simply choose pivots uniformly at random - a moment's reflection reveals that these procedures are equivalent. For now, we will think of randomized quicksort as the algorithm that picks pivots as random, as this is an useful perspective for the analysis. We will now bound the worst-case expected running time of the algorithm, and we remark that this is better than an average-case bound because we are no longer assuming any special properties of the input. It is amusing in that making the algorithm probabilistic is giving us more control over the running time. This is a classic example of how the probabilistic analysis of a deterministic algorithm might "inspire" a dose of randomization in the algorithm itself. Notice that the randomized version of quicksort is a Las Vegas algorithm.

To kickstart the analysis, let's assume no two elements in the array are equal (such elements can be preprocessed if they exist). We will try to express the quantity we care about (the total number of comparisons) as a sum of simpler random variables, and then just analyze the simpler ones.

Define random variable X_{ij} to be 1 if the algorithm does compare the i^{th} smallest and j^{th} smallest elements in the course of sorting, and 0 if it does not. Let X denote the total number of comparisons made by the algorithm. Since we never compare the same pair of elements twice, we have

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}.$$

and therefore,

$$E[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}].$$

Let us consider one of these X_{ij}'s for $i < j$. Denote the i^{th} smallest element in the array by e_i and the j^{th} smallest element by e_j. If the pivot we choose is between e_i and e_j, then the algorithm never compares e_i against e_j because

they fall out into different sub-problems. If the pivot is e_i, then the algorithm certainly compares e_i with e_j, to determine which sub-problem to send e_j to. Similarly, a comparison between e_i and e_j is certain if e_j is the pivot. If the pivot is either more than e_j or less than e_i, then both e_i and e_j are transferred to the same sub-problem, and we have to choose a new pivot down here.

Think of every number as a ball in an urn, where the balls corresponding to e_i and e_j are black, and the balls corresponding to all e_k where $i < k < j$ are white, and all remaining balls are green. We pick a ball at random, and if we get a black ball then X_{ij} is one, if we pick a white ball then X_{ij} is zero, and if we pick a green ball then we start over (possibly after discarding some balls). At every stage, conditioned on the event that the process ends at this stage, the probability that X_{ij} is 1 is exactly $(2/(j-i+1))$, and therefore, the overall probability that $X_{ij} = 1$ is also $(2/(j-i+1))$. So we have:

$$E[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} 2/(j-i+1) = 2\left(\sum_{i=1}^{n} \frac{1}{2} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-i+1}\right).$$

The quantity $1 + 1/2 + 1/3 + \ldots + 1/n$, denoted H_n, is called the "nth harmonic number" and is in the range $[\ln n, 1 + \ln n]$ (this can be seen by considering the integral of $f(x) = 1/x$). Therefore,

$$E[X] < 2n(H_n 1) \leqslant 2n \ln n.$$

Now that we have analyzed quicksort, let's turn to a very brief primer on randomized complexity classes.

3 Randomized Complexity Classes: in a nutshell

We begin by recalling the following basic definition: A *language* L is a set of finite strings over some fixed alphabet Σ, that is, $L \subseteq \Sigma^*$. An algorithm is said to *recognize* L if on input $x \in L$, the algorithm outputs Yes, and on input $x \notin L$, it outputs No. To begin with, we have the following fundamental classification:

Polynomial time (P). A language L lies in P if there is a polynomial-time algorithm that recognizes L.

Non-deterministic Polynomial time (NP). The class NP consists of all languages L which have witnesses that can be recognized by polynomial time. More formally, $L \in$ NP implies that there is some polynomial time-computable predicate $\mathcal{P}$, and a polynomial p such that

$$x \in L \iff \exists y \mathcal{P}(x, y),$$

and the length of y (the "witness" for membership of x in L) is at most $p(|x|)$.

We now introduce classes which are defined based on the kind of randomized algorithms that the languages admit.

Randomized Polynomial time (RP). The class RP consists of all languages L that have a polynomial-time randomized algorithm A with the following behavior:

- If $x \notin L$, then A always rejects x (with probability 1).
- If $x \in L$, then A accepts x in L with probability at least $1/2$.

Note that RP admits Monte Carlo algorithms with one-sided error (where the error is in acceptance). Also, observe that the probability of success can always be amplified with repeated runs. Since the error is one-sided, we can repeat the algorithm t times independently: we reject the string x if any of the t runs reject, and accept x otherwise. Since the runs were independent, we have

$$\Pr[\text{ algorithm makes a mistake t times }] \leqslant \frac{1}{2^t}.$$

Thus, we can repeat the algorithm a polynomial number of times and make the error exponentially small.

The Complement of RP (co-RP). The class co-RP consists of all languages L for which $\overline{L}$ is in the class RP (where $\overline{L} = \Sigma^* \setminus L$). Equivalently, coRP consists of all languages L that have a polynomial-time randomized algorithm A with the following behavior:

- If $x \in L$, then A always accepts x (with probability 1).
- If $x \notin L$, then A rejects x in L with probability at least 1/2.

We now turn to the class that captures languages with Las Vegas style randomized algorithms.

Zero-error Probabilistic Polynomial time (ZPP). A language L is in ZPP if there is an algorithm A that recognizes L (with no error) and runs in expected poly-time.

Recall our remark earlier about how Las Vegas algorithms can be terminated early to obtain Monte Carlo algorithms. It is not very hard to imagine that the other way works too, that given two Monte Carlo style algorithms that err in opposite ways (that is, the first algorithm always accepts correctly and the other always rejects correctly), they can be combined into a Las Vegas style algorithm. These comments are formalized in the following theorem:

Theorem. ZPP = RP ∩ co-RP

A formal proof is not very difficult, and we invite the reader to turn the suggestions made so far into a formal proof. Finally, we have the complexity class that captures two-sided error.

Bounded-error Probabilistic Polynomial time (BPP). The class BPP consists of all languages L that have a polynomial-time randomized algorithm A with the following behavior:

- If $x \in L$, then A accepts x in L with probability at least 3/4.
- If $x \notin L$, then A accepts x in L with probability at most 1/4.

Hence, the error probability in either case is at most 1/4.

Again, we can use amplification to decrease the error. Note that both RP and co-RP are subsets of BPP. An important open question in complexity theory is whether BPP ⊆ NP or not. For readers keen on exploring more complexity classes, an excellent resource online is the Complexity Zoo.

4 Derandomizing MaxCut, with some help from Chimpanzees

Recall that a *cut* in a graph is a partition of its vertex set into two parts. The size of a cut is the number of edges that cross the cut, that is, the number of edges that have one endpoint in each part of the cut. We know, by an application of the probabilistic method, that any graph with m edges admits a cut of size at least (m/2). This can be turned into a naive randomized algorithm for finding a cut (A, B) of size (m/2) — for every vertex, toss a fair coin and place the vertex in A if the coin shows up heads, and place the vertex in B if the coin shows up tails. The expected size of the cut obtained is, sure enough, m/2, but what if we would like to be absolutely sure about ending up with a cut of size m/2?

Fortunately, some randomized procedures can be *derandomized*, which is to say, the random component can be substituted with a deterministic procedure. The substitution typically comes with a price tag — the time spent

by the derandomized version of a randomized algorithm is usually more than the original randomized algorithm, and it is also common that the algorithm itself becomes somewhat more involved.

Derandomization is a vast theme, and several methods and procedures have been developed over time. As a case in point, we will derandomize the max cut algorithm above, and this will demonstrate what is called the method of conditional probabilities. Let us say that we are dealing with a graph on the vertex set $[n]$. Before we begin a formal description, we present the following explanation which involves solving the max cut problem with the help of a chimpanzee (Jukna attributes this brilliant description to Maurice Cochand). Imagine building a complete binary tree of height n, with 2^n leaves, where each leaf corresponds to one of the possible cuts of the graph. Leaves are, quite naturally, close to the sky. At each level i, going upwards via the left branch implies dumping the vertex i into partition A, and going upwards via the right branch implies dumping the vertex i into partition B.

In order to motivate the chimpanzee for this fascinating problem, we attach at every leaf of the tree a black box containing a number of bananas equal to the size of the cut corresponding to that leaf. We then invite the chimpanzee to go up in oder to bring down one of the black boxes, making him most clear the potential benefit of the operation. We repeat this experiment many times, with many different trees corresponding to as many graphs G, having different number of vertices and edges. The chimpanzee never looked at the graph (although he was allowed to do it), did not even care about the number of vertices. He moved up quickly along the tree, and always brought back a box having a number of bananas at least equal to half the edges!

We asked him for his secret (he definitely had one, this was more than luck!). For a number of bananas we do not dare to mentioned here, he gave the following answer: "Embracingly simple," he said. "At every junction I do the same: because of the weight, the branch supporting the subtree having the biggest number of bananas is not as steep as the other one, there I go!"

To apply the method of conditional expectation, we first model the random experiment as a sequence of small random steps. In this case it is natural to consider each step to be the choice of color for a particular vertex (so there are n steps). Next, replace the random choice at each step by a deterministic choice, so as to maintain the expected size of the cut as we go along. We are going to use conditional expectations (conditioned on the choices made until a certain point) to formalize this intuition.

Let the random variable X be the number of edges that cross the cut. Introduce n random variables $Y_1, \ldots, Y_n$, where Y_i is 1 if the vertex i belongs to A and is 0 otherwise. Let's say that we have made assignments for the first i vertices.The chimpanzee's strategy amounts to comparing the quantity:

$$E[X|Y_{i+1} = 0, Y_i = a_i, \ldots, Y_1 = a_1]$$

against

$$E[X|Y_{i+1} = 1, Y_i = a_i, \ldots, Y_1 = a_1].$$

The setting of Y_{i+1} is determined by whatever quantity is greater. It is easily checked that if we maintain this at every stage, then the final cut we end up with does have size at least $(m/2)$.

Notice that what the algorithm is doing at step i is the following: it places vertex i in A and counts how many edges incident on i are crossing the cut built so far, and then it places vertex i in B and counts how many edges incident on i are crossing the cut built so far. It places i according to whatever leads to the bigger cut. It is an easy exercise to check that the running time of the algorithm is now $O(n + m)$ (what was it before we derandomized it?).

5 Communicating sparingly, but communicating enough

Let's say that Alice maintains a large database of information, and her friend, Bob, maintains a backup. Over time, they would like perform periodic consistency checks to ensure that the data is indeed safe. However, communication is rather expensive, and transmitting all the data is quite out of budget.

Let Alice's data be the sequence $a = a_0, a_1, \ldots, a_{n-1}$, and let Bob's sequence be $b = b_0, b_1, \ldots, b_{n-1}$. Now, any procedure that hopes for complete accuracy cannot afford to save even one bit, because a wicked adversary can mess up just that bit. However, with a little randomization thrown in the mix, we can perform sanity checks that detect inconsistencies with high probability, while transmitting only $O(\log n)$ bits.

We begin by thinking of the data as polynomials over $\mathbb{F}_p$, where p is a prime between n^2 and $2n^2$. (Our proposal is reasonable, because it is known that there always exists a prime number between any number and its double.) We are considering the polynomials:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots a_{n-1} x^{n-1},$$

and

$$B(x) = b_0 + b_1 x + b_2 x^2 + \ldots b_{n-1} x^{n-1}.$$

In order to check if $a = b$, Alice picks a random number r in $\mathbb{F}_p$ and sends Bob the numbers r and $A(r)$. Bob evaluates $B(r)$ and if $B(r) = A(r)$, sends a "1" back to Alice. If $B(r) \neq A(r)$, then Bob sends a "0", hinting at an inconsistency. Notice that the number of bits transmitted is:

$$1 + 2\log p \leqslant 1 + 2\log 2n^2 = 2(1 + \log n) = O(\log n).$$

If the data was consistent, then this strategy just works, because the polynomials are exactly the same. However, if the data was inconsistent, then chances are that the polynomials may still evaluate to the same value, causing an error. What is the probability that for a randomly chosen element $r \in \mathbb{F}_p$, $A(r) = B(r)$? or equivalently, $A(r) - B(r) = 0$? A bound on this probability can be obtained thanks to the Schwarz-Zippel lemma, which is known for multivariate polynomials, but we will just state the one-variable case here for simplicity:

If $f(x)$ be a non-zero polynomial of total degree $d \geqslant 0$ over a field F. Let S be a finite subset of F and let r be selected randomly from S. Then,

$$P[f(x) = 0] \leqslant d/|S|.$$

In our setting, $d = (n-1)$ and $|S| = |\mathbb{F}| \geqslant n^2$. Therefore, the error probability is at most:

$$P[A(r) - B(r) = 0] \leqslant d/|S \leqslant \frac{n-1}{|\mathbb{F}|} \leqslant (n-1)/n^2 \leqslant 1/n.$$

Therefore, the probability of detecting an inconsistency if there is one is at least $(11/n)$, and as we have discussed before, one could repeat the transmission to amplify the probability of success.

6 Further Reading

These notes are heavily borrowed from sections of *Randomized Algorithms*, the definitive text on the subject by Motwani and Raghavan, and the chapters on Randomized Algorithms and Derandomization in Jukna's excellent book, *Extremal Combinatorics*. The section on quicksort is an adaptation of these notes.