Lectures on Pseudorandomness

(Lecture Notes)

JAYALAL SARMA

Department of Computer Science and Engineering
Indian Institute of Technology Madras (IITM)
Chennai, India

Last updated on : February 6, 2021

Table of Contents

Todo list

Part I

General Introduction and Tools

LECTURE 1

Power of Randomization

Lecturer : Jayalal Sarma

Randomized algorithms play a very powerful role in algorithm design. We will concentrate on the randomized algorithms for decision problems in this course. So all of our computational problems can be abstractly represented as given a string $x \in \Sigma^*$ in an alphabet, does x have property $\mathcal{P}$ or not?

Informally, a randomized algorithm running in time t is an algorithm that that on input x is allowed to perform at most t instances of random experiment of tossing unbiased coins during its computation and uses the outcome of the experiment in the computation, but however, provides a guarantee that the answer of the algorithm is the *correct* answer for the input x in a good fraction of the possible outcomes of the experiment. [1]

1.1 Randomness helps in Matrix Multiplication Verification

Consider the task of multiplying two $n \times n$ matrices over the field $\mathbb{F}_2$. The trivial algorithmic solution to the problem takes $O(n^3)$ time and the trivial lower bound for the problem is $\Omega(n^2)$. It has been a long standing question which is the right complexity bound for this important problem (improvement to which will lead to improvements even in practice !). The exponent of matrix multiplication is the smallest constant ω such that two $n \times n$ matrices may be multiplied by performing $O(n^{\omega+\epsilon})$ for every $\epsilon > 0$.

Indeed, one of the basic ideas that we learn in algorithms courses for demonstrating the power of divide and conquer is the Strassen's multiplication which gives a running time bound of $O(n^{2.,73})$ which went through a sequence of improvements and to the current best of $O(n^{2.31})$.

The question that we address now is something closely related - that of verifying whether a given multiplicaiton is correct.

PROBLEM 1.1.1. *Given three matrices $A, B, C \in \mathbb{F}_2^{n \times n}$, check whether $AB = C$ or not.*

Indeed, the trivial method would be to mutliply the two matrices and check if the result is equal to C. But then this requires, $O(n^{2.31} + n^2)$ time using the best matrix multiplication algorithm that we know currently. Since we just require verification, it is conceivable that we might be able to do better if we are allowed to make a small some error in the process. We show that this indeed possible to be done in $O(n^2)$ with error probability at most 2^{-k} for any constant k (independent of

[1]Notice that if the guarantee for the algorithm is not saying *"strictly more than half fraction of the coin tosses"*, then essentially the algorithm is useless since we can always replace it with a random experiment of tossing an unbiased coin and returning the answer to be YES if we get the heads and NO if we get tails. Note that we have at least a $\frac{1}{2}$ probability of success $\frac{1}{2}$.

n).

Trivial Approach using randomization: A natural first cut attempt is to choose an entry $(i, j) \in [n] \times [n]$ uniformly at random from the n^2 entries of C and checking if :

$$\sum_{k=1}^{n} A_{ik} B_{kj} = C_{ij}$$

This runs in time $O(n)$. And we can choose constant k more entries to amplify the success probability. However, the probability of correctness is very small. Suppose, in the worst case input, there was only one $(i, j) \in n \times n$ where there was an error in the multiplication. The probability that we will choose that particular (i, j) for verification is as small as $\frac{1}{n^2}$. Amplifying this to a success probability of $\frac{1}{2^k}$ takes more than $\Omega(n^{1+\epsilon})$ iterations and hence the overall algorithm will take $\Omega(n^{2+\epsilon})$ time which is beyond what we can afford to spend time on.

Freivalds' Approach: The idea is to check a randomly chosen "linear combination" of entries rather than a single entry of the matrix C. If we choose this to be a random linear combination of rows of the matrix C, then the combinatorics helps to achieve a much better probability of error. We now formally write down the algorithm and analyse it.

Algorithm 1.1 : Frievald's Algorithm for Verification of Matrix Multiplication - $\mathcal{F}(A, B, C)$

```
1: Choose a vector $r \in \mathbb{F}_2^n$ uniformly at random.
2: If [$A(Br) = Cr$] then output YES else output NO.
```

The computation of $(A(Br))$ and Cr are done using $O(n^2)$ time algorithms since computing a linear transformation result Ax for an $n \times n$ matrix can be done in $O(n^2)$ time. Now we argue correctness guarantees. If the given matrices indeed satisfy $AB = C$, then no matter which $r \in \mathbb{F}_2^n$ algorithm chooses in step 1, $ABr = Cr$ and hence it always will output YES. The error can happen only when $AB \neq C$ and the algorithm ends up choosing an unfortunate r such that $ABr = Cr$. The following claim upper bounds the probability of this .

CLAIM 1.1.2. *For any $A, B, C \in \mathbb{F}_2^n$ such that $AB \neq C$,*

$$\Pr[\mathcal{F}(A, B, C) \text{ outputs YES}] \leq \frac{1}{2}$$

Proof. Let $A, B, C \in \mathbb{F}_2^n$ such that $AB \neq C$. We need to analyze the probability that $ABr = Cr$ for $r \in \mathbb{F}_2^n$ chosen uniformly at random. If $AB \neq C$, then $D = AB - C$ is a non-zero matrix. Thus

$$\Pr_{r \in \mathbb{F}_2^n}[ABr \neq Cr] = \Pr_{r \in \mathbb{F}_2^n}[Dr \neq 0]$$

Imagine that D was all 1s matrix. Now the above probability is exactly the number of vectors $r \in \mathbb{F}_2^n$ with an odd number of 1s in it. By an obvious bijection, this is also the number of subsets of $[n]$ with odd size. The latter is exactly 2^{n-1} and hence the probability of such a vector r being chosen from $\mathbb{F}_2^n$ is $\frac{1}{2}$.

Now we formalize and generalize this. Let p be the vector Dr. Since $D \neq 0$, there must be an entry $D_{ij} \neq 0$. Define, $A = \{j : D_{ij} \neq 0\}$. We know that $A \neq \phi$. $i, j \in [n]$. Thus :

$$p_i = \sum_{k=1}^{n} D_{ik} r_k = \sum_{k \in A} r_k$$

$$\text{Note that: } \Pr_{r \in \mathbb{F}_2^n}[Dr = 0] \leq \Pr_{r \in \mathbb{F}_2^n}[p_i = 0]$$

Notice that the latter probability depends only on r_k where $k \in A$. The fraction of assignments of the bits $\{r_k : k \in A\}$ which makes $p_i = 0$ is exactly $\frac{1}{2}$ since the number of even sized subsets of A and number of odd sized subsets of A are exactly the same. $\square$

REMARK 1.1.3. Informally, if we run the above algorithm $\mathcal{F}(A, B, C)$, and the algorithm outputs NO, then we can trust the answer and conclude that indeed $AB \neq C$. But a YES answer from the algorithm cannot be trusted - it could be because of the unfortunate choice of $r \in \mathbb{F}_2^n$ that came as the outcome of the experiment.

Why are we interested in the above algorithm even though it gives a success probability bound of only $\frac{1}{2}$? The reason is that, it is a one-sided error algorithm and hence still much better than a coin toss outcome because the algorithm does not make an error when $AB = C$. In fact, such algorithms can be repeated in a natural way - run k times, and if any of them says $AB \neq C$ output NO. This reduces the error probability in exponentially in k - since each of the trials should give an error (with probability $\frac{1}{2}$) and hence the error probability bound is at most $\frac{1}{2^k}$.

1.2 Algorithms for Polynomial Identity Testing Problem

Now we will do an example problem where there is an efficient (polynomial time in the input size) randomized algorithm for solving the problem but a deterministic algorithm for solving the problem is not known. The problem is easy to state algorithmic question on polynomials. Fix $\mathbb{F}$ to be the field where the coefficients are chosen from. *Given a polynomial $p \in \mathbb{F}[x_1, x_2, \ldots x_n]$, test if it is identically zero.* That is, do all the terms cancel out and become the zero polynomial.

This problem has its roots in the simple high school arithmetic. Suppose we are given a polynomial in a complicated form where the monomials may repeat with arbitrary coefficients etc. We want to find out if the coefficient of the monomials cancel out to zero. This in effect is testing whether the polynomial is the zero polynomial, and equivalenty it is testing if the polynomials evaluates to zero on all substitutions of the variable from the underlying field $\mathbb{F}$.

How are we given the polynomial? This indeed is going to have effect on the complexity of the problem. Let us start with the high school arithmetic again. Suppose we are given it in the monomial form (though some monomials may repeat) along with their coefficients. To solve the problem, it suffices to check, for each monomial whether the coefficient in its various appearences is adding up to zero. Given the explicit representation at the input, this is very easy to do by simply going over the input for each monomial. Hence this can be done in time polynomial in the input.

What if the polynomial is not given that explicity. What is the most implicit form that we can think of? A black box which evaluates the polynomial. That is, we have an oracle p when given input a returns $p(a)$, the value of polynomial at a.

Assume that we are also given an upper bound on the degree of the polynomial $deg(p) \leq d$. Indeed, we do not have access to the actual polynomial except through the blackbox. We have to use some property of the degree d polynomials. The most obvious one is the number of points in which they can evaluate to zero. Based on this thought, the following deterministic algorithm solves the problem.

Algorithm 1.2 A deterministic algorithm for univariate polynomial identity testing

```
1: Choose $d+1$ different points $a_1, \ldots, a_{d+1}$.
2: Call the oracle $d+1$ times to evaluate $p(a_1), \ldots, p(a_{d+1})$.
3: If all calls returned 0 accept else reject.
```

If p were really the zero polynomial then all calls will return 0 and we will definitely accept. If p were not 0, then at most d calls can return 0 since a polynomial with degree at most d has at most d roots. Hence if $p \neq 0$, then our algorithm will definitely reject.

Multivariate PIT: Now let us think about the problem when p is a multivariate polynomial. The previous assertion that a degree d polynomial has at most d roots no longer holds. To see this, consider the degree 2 polynomial $p(x_1, x_2) = x_1 x_2$. This has an infinite number of roots $x_1 = 0, x_2 \in \mathbb{F}$, where $\mathbb{F}$ is the (possibly infinite) field over which p is defined.

We can work around this problem by considering a finite subset of the field, say $S = \{0, \ldots, 10\}$. The polynomial p has 19 zeroes. So if x_1, x_2 is chosen uniformly at random from S there is at most $19/100$ chance that we will get a false result. As can be seen from the above example, by making the size of S arbitrarily large, we can make the error probability arbitrarily small. But then the disadvantage is that we will need more random bits in order to choose an element at random from the set $|S|$, and the running time of our algorithm will also increase.

Generalizing this strategy that we will follow is as follows: If the total degree of the polynomial is $\leq d$, and if $S \subseteq \mathbb{F}$, such that $|S| \geq 2d$, instead of picking elements arbitrarily, we pick elements uniformly at random from S. Indeed, there may be many choices for the values which may lead to zero. But how many?

LEMMA 1.2.1 (**Schwartz-Zippel Lemma**). *Let $p(x_1, x_2, \cdots, x_n)$ be a non-zero polynomial over a field $\mathbb{F}$. Let $S \subseteq \mathbb{F}$*

$$\Pr_{\overline{a} \in S^n}[p(\overline{a}) = 0] \leq \frac{d}{|S|}$$

Proof. (By induction on n) For $n = 1$: For a univariate polynomial p of degree d, there are $\leq d$ roots. Now in the worst case the set S that we picked has all d roots. Thus for a random choice of substitution for the variable from S, the probability that it is a zero of the polynomial p is at most $\frac{d}{|S|}$.

For $n > 1$, write the polynomial p as a univariate polynomial in x_1 with coefficients as polynomials in the variables $p(x_2, \ldots, x_n)$.

$$\sum_{j=0}^{d} x_1^j p_j(x_2, x_3, \ldots, x_n)$$

For example: $x_1x_2^2 + x_1^2x_2x_3 + x_3^2 = (x_2x_3)x_1^2 + (x_2^2)x_1 + x_1^0(x_3^2)$.

We need to analyze the probability that we will choose a zero of the polynomial (even though the polynomial is not identically zero). For a choice of the variables as $(a_1, a_2, \cdots, a_n) \in S^n$, we ask the question : how can $p(a_1, a_2, \cdots, a_n)$ be zero? It could be because of two reasons:

1. $\forall j \ : \ 1 \leq j \leq n, \ \ p_j(a_2, a_3, \ldots, a_n) = 0$.
2. Some coefficients $p_j(a_2, a_3, \ldots, a_n) = 0$ are non-zero, but the resulting univariate polynomial in x_1 evaluates to zero upon substituting $x_1 = a_1$.

Now we are ready to calculate $Pr[p(a_1, a_2, \ldots, a_n) = 0]$. For a random choice of $(a_1, \ldots, a_n)$. Let A denote the event that the polynomial $p(a_1, \ldots, a_n) = 0$. Let B denote the event that $\forall j \ : \ 1 \leq j \leq n, \ \ p_j(a_2, a_3, \cdots, a_n) = 0$. Note that, $Pr[A] = Pr[A \wedge B] + Pr[A \wedge \overline{B}]$.

We calculate both the terms separately: $Pr[A \wedge B] = Pr[B].Pr[A|B] = Pr[B]$ where the last equality is because $B \Rightarrow A$. Let ℓ be the highest power of x_1 in $p(x)$. That is $p_\ell \neq 0$. Since the event B insists that for all j, $p_j(a_2, a_3, \ldots, a_n) = 0$, we have that $Pr[B] \leq Pr[p_\ell(a_2, a_3, \ldots, a_n) \neq 0]$. By induction hypothesis, since this polynomial has only $n-1$ variables and has degree at most $\frac{d-\ell}{S}$. Thus, $Pr[B] \leq \frac{d-\ell}{S}$.
To calculate the other term,

$$Pr[A \cap \overline{B}] \quad = \quad Pr[\overline{B}].Pr[A|\overline{B}] \leq Pr[A|\overline{B}] \leq \frac{\ell}{|S|}$$

where the last inequality holds because the degree of the non-zero univariate polynomial after substituting for $a_2, \ldots, a_n$ is at most ℓ and hence the base case applies. ∎

This suggests the following efficient algorithm for solving PIT. Given d and a blackbox evaluating the polynomial p of degree at most d.

Algorithm 1.3 : Schwartz-Zippel Algorithm for Multivariate PIT

```
1: Choose $S \subseteq \mathbb{F}$ of size $\geq 4d$.
2: Choose $(a_1, a_2, \ldots, a_n) \in_R S^n$.
3: Evaluate $p(a_1, a_2, \ldots a_n)$ by querying the blackbox.
4: If it evaluates to 0 accept else reject.
```

The algorithm runs in time $\mathsf{poly}(n)$. The following Lemma states the error probability and follows from the Schwartz-Zippel Lemma that we saw before.

LEMMA 1.2.2. *There is a randomized polynomial time algorithm A, which, given a black box access to a polynomial p of degree d (d is also given in unary), answers whether the polynomial is identically zero or not, correctly with probability at least $\frac{3}{4}$.*

Notice that in fact the lemma is weak in the sense that it ignores the fact that when the polynomial is identically zero then the success probability of the algorithm is actually 1 !. In other words, is it is a one-sided error randomized algorithm.

LECTURE 2

The Derandomization Problem

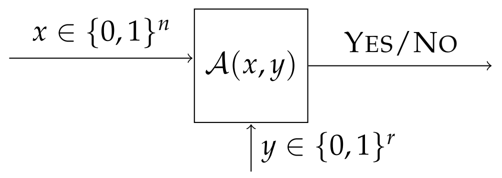
Lecturer : Jayalal Sarma

In the previous lecture, we talked about randomized algorithms for problems for which we do not know deterministic algorithms with similar complexity resource bounds. Indeed, we are not happy about randomized algorithms as such since these algorithms require perfect unbiased coin toss experiments to be performed and we do not have them in practice. Indeed, the fact that they can output erroneous answers, even though with low probability makes them useless in critical practical applications.

How do convert them to deterministic algorithms without causing much overhead?. One possible way is to look at each algorithm and use inherent properties of the problem to analyze the randomized algorithm better to come up with ways to remove randomness from that algorithm. Here, we start with the original randomized algorithm for a particular problem, and improve it to derandomize it and the techniques are usually very algorithm specific. We will do some examples of this kind later in the course.

2.1 Abstract Model and some Approaches

From now on, we will be concentrating only on abstract models of these randomized algorithms. We fix some notations first. A randomized algorithm $\mathcal{A}$ on input x runs in time $t(n)$ (where $n = |x|$) and let $y \in \{0,1\}^{r(n)}$ be the concatenation of the unbiased coin toss experiement that the algorithm does during its execution. Notice that $r(n) \leq t(n)$ (we drop the n when it is not required explicitly). If the algorithm runs in polynomial time $t(n) \leq n^c$ for a constant c independent of n.



The guarantee we have is there is an $\epsilon \in (0, \frac{1}{2}]$.

$$\forall x \in \{0,1\}^n, \quad \Pr_{y \in \{0,1\}^r}[A(x,y) \text{ is correct.}] \geq \frac{1}{2} + \epsilon$$

2.1.1 Trivial Derandomization Approach

The trivial approach to obtain an equivalent deterministic algorithm is run over all possible outcomes of the experiment and check the answer from the algorithm for each of them. Whichever answer comes as majority - report that as the final answer.

If the running time of the randomized algorithm $\mathcal{A}$ is $t(n)$, then the running time of the new algorithm (which is deterministic) is $t(n)2^{r(n)}$. To argue correctness, if the actual answer for input

Algorithm 2.4 ($\mathcal{A}'$) : input $x \in \{0,1\}^n$, where success prob. $\frac{1}{2} + \epsilon$ for $\mathcal{A}$

```
1: $count \leftarrow 0$.
2: for each $y \in \{0,1\}^r$ do
3:     Check if $\mathcal{A}(x,y)$ accepts, if so increment $count$
4: end for
5: If $[count > 2^{r-1}]$ then output YES else output NO.
```

$x \in \{0,1\}^n$ is YES, then the fraction of $y \in \{0,1\}^r$ which makes $\mathcal{A}(x,y)$ accept is strictly more than $\frac{1}{2}$ and hence the algorithm will output YES. If the actual answer for input $x \in \{0,1\}^n$ is NO, then then the fraction of $y \in \{0,1\}^r$ which makes $\mathcal{A}(x,y)$ accept is strictly less than $\frac{1}{2}$ and hence the algorithm will output NO.

REMARK 2.1.1. Note that the algorithm $\mathcal{A}$ will run in $\mathsf{poly}(n)$ time if the original randomized algorithm was running in $t \leq \mathsf{poly}(n)$ time and was using $r \leq O(\log n)$ random bits.

2.2 Pseudorandomness : An Informal Overview

Ideally, we would like to replace the randomized algorithm with a deterministic one as done in the previous section. However, we know how to do this trivially only when the randomized algorithm uses $O(\log n)$ random bits.

We outline two "out of the box" thoughts related to our target of derandomization of randomized algorithms.

Fooling the Algorithm with Pseudorandom bits : PRGs The first one is about using $y \in \{0,1\}^r$ as not independent random bits. But use *dependent* random bits instead. Indeed, the analysis for the error bound for the algorithm $\mathcal{A}$ now may fail since it may assume total independence between the bits of y in its mathematical argument. However, sometimes, it is possible that same analysis (or even a better analysis) may work even when the bits of the y are dependent in a limited way [2] But this may be specific to the algorithm and sometimes to the problem itself. We would ideally want a more abstract strategy which would work for randomized algorithms in general, modelled by what we described in the previous section. But even if we made it work with some dependent randombits, how do we produce this distribtion of y' with the desired limited dependence among them? Construction of the methods which can produced limited dependence thus becomes important.

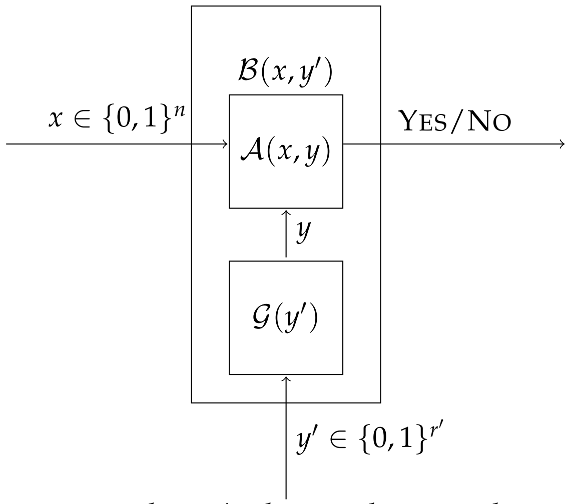
Taking a more abstract view point, informally, we would like to have a box (formally an algorithm G) which takes in pure random bit string of length $y' \in \{0,1\}^{r'}$ and produces a string $y \in \{0,1\}^r$ such that the distribution of y "looks" pseudo-random for the resource limited algorithm $\mathcal{A}$. The idea is that we will run the algorithm $\mathcal{A}$ with the random string provided the $G(y')$ - the output of G on input $y' \in \{0,1\}^{r'}$ chosen uniformly at random.

[2] At one extreme, if we had an algorithm and an analysis which works wen all the bits of the y are the same (which is an example of extreme dependence) we dont require the random bit at all - we can directly simulate the algorithm deterministically the trivial way.

Indeed, since we are not providing pure random bits to $\mathcal{A}$. Hence we should expect its correctness guarantees to not hold good anymore. That is, it will deteriorate a bit. Can we guarantee that it does not deteriorate too much? This can be done in two ways (1) rework or reanalyse the algorithm $\mathcal{A}$ and argue that it is still having success probability greater than $\frac{1}{2}$ (which is enough for trivial derandomization) (2) use only resource bounds of $\mathcal{A}$ to argue that the change of y to $G(y')$ will not affect the success probability much. For this, the generator has to satisfy certain properties.

A function $G : \{0.1\}^{r'} \to \{0,1\}^r$ is said to be a Peudo-Random Generator (PRG) for complexity measure[3] s and error parameter $\delta \in [0,1]$ if, for any algorithm $\mathcal{A}$ which runs in time $t \leq s$ (or having complexity measure bounded by s): For any x,

$$\left| \Pr_{y \in \{0,1\}^r} [\mathcal{A}(x,y) \text{ Accepts}] - \Pr_{y' \in \{0,1\}^{r'}} [\mathcal{A}(x, G(y')) \text{ Accepts}] \right| \leq \delta$$



Connecting to the informal description, δ is the quantity by which the success probability deteriorates because of the use of the pseudorandom generator output, instead of pure random bits. Hence if we ensure that $\delta < \epsilon$, even after the use of the pseudorandom generator output, we still will have a randomized algorithm $\mathcal{B}$ with the following guarantee $\forall x \in \{0,1\}^n$:

$$\Pr_{y \in \{0,1\}^r} [\mathcal{B}(x,y) \text{ is correct.}] \geq \frac{1}{2} + \epsilon - \delta > \frac{1}{2}$$

We will end the description by asking the question - *what parameters determine how good our pseudrandom generator is?*. As per the above discussion it is:

- The relative values of r and r'. This leads to the definition of the *stretch* of the psuedorandom generator. We would ideally want an exponential stretch function so that with $r' \in O(\log n)$ we can produce y for $\mathcal{A}$ which is of length $r = O(n^c)$ for constant c.

- The value of s. This determines how powerful an algorithm can the pseudorandom generator manage to fool. The larger the s the better. Ideally we want s to be covering all polynomial time running time bounds.

- The value of δ. This determines the quantity by which the success probability of the algorithm deteriorates after plugging in the output of the pseudorandom generator instead of the y from the pure random bits. Ideally, we want $\epsilon - \delta > 0$. The smaller the δ, the better.

- Running time of the generator itself. Notice that we need $\mathcal{G}$ to be explicit polynomial time algorithm, which runs in time $\mathsf{poly}(n)$. That is, if $r' \in O(\log n)$ (which is what ideally we

[3]We will make it more precise when it comes to the section where we handles these objects. We are leaving at the above description at a less precise level.

would want, so that the trivial derandomization runs in $\mathsf{poly}(n)$ time), then technically, the generator can run for exponential time in terms of its input size[4].

The main part of the game is in describing the generator algorithms (or functions from $\{0,1\}^{r'} \to \{0,1\}^{r}$). However, it is not even clear whether such functions exists for the range of parameters that we care about. Indeed, this is the kind of flavour that we will have.

- We can prove that the functions that we are looking for exist, with a non-constructive argument. This is done by - what is termed as the *Probablistic method*.
- Explicit descriptions of the functions, which are are required for the algorithms with required runtime bounds for $\mathcal{G}$ are not known. In fact, if we have such descriptions, then a complete derandomization of all randomized algorithms is possible, which will be a big achievement.

Refining Randomness : Randomness Extractors - Here is a completely different idea about supplying dependent randomness. We do not have source of pure random bits to supply for the randomized algorithm $\mathcal{A}$. But we may have impure radom bit sources. An imaginative question is *can we invest a few pure random bits in order to purify/extract and hence improve the impurity in the given random source?*. This, at first sounds crazy and leads to the following questions.

- How do we define *impure random bit sources*. They define distributions which are not uniform. There is the notion of entropy which can tell us how uniform the source is.
- How do we define *how good the output is*. Again, one could have used entropy here too naturally. However, noticing the fact that we would like to finally apply it our algorithms like $\mathcal{A}$, a different measure of "purity" is used which is the notion of statistical distance[5] to uniform distribution.

The above discussion leads to the definiton of a randomness extractor, which is a function $\mathcal{E} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ such that when X is a distribution on $\{0,1\}^n$ with entropy at least k, then the distribution of the output $\mathcal{E}(X, U_d)$ is ϵ-close to U_m where U_d and U_m are uniform distributions on the set $\{0,1\}^d$ and $\{0,1\}^m$ respectively.

Again, how do we determine how good is our extractor function? We want extractors which works on highly biased distributions (the smallest k possible) using fewest number of pure random bits (the smallest d possible) and produces output distributions which are closest to uniform distributions (ϵ must be smallest) - and still run in time polynomial in n.

Similar to pseudorandom generator functions, it is unclear apriori whether such functions even exist for the range or parameters we care about. A similar situation arises, where by using probablistic method, we can prove that such objects (functions) exists, but at the same time, we do not know how to construct them deterministically (equivalently describe the algorithm for $\mathcal{E}$).

REMARK 2.2.1 ((Informal) - **Psuedo-random Objects**). The common flavour that we observed about the previous sections is that there are mathematical objects which we would like to describe

[4]This marks the difference between the pseudorandom generators studied in cryptography and derandomization.

[5]Informally, this is the sum of the difference (in absolute value) between the probability values assigned to points in the sample space.

(by providing algorithms to compute those functions) and we do know that the function that we seek exists. This situation is a common phenomenon in many objects. In fact, in most situations, it is not just the existance of the objects that is argued, but also that if the object that is of interest is chosen at random (with appropriately set up experiments), the object of interest shows up at the outcome with high probability. Thus, there is a randomized algorithm to explicitly construct the object, and now we have to derandomize them !. However, notice that in such situations, we can think of deranodomizations which just depends on those algorithms which chooses the object at random.

Other Contexts and Psuedorandom Objects We now describe a totally unrelated context in which the required mathematical functions display such a psuedorandom behaviour and an explicit construction is being sought for. The context is that of coding theory.

Coding theory had its inception in the late 1940's with the theory of reliable communication over a channel in the presence of noise - an area that started with the pioneering work of Claude Shannon and Richard Hamming. The former addressed and answered the fundamental questions about the possibility of the use of codes for reliable communication and the later develped some basic combinatorial constructions of error correcting codes that laid the foundations for the work later.

Theoretical computer scientists have a major role to play in the algorithmic aspects of coding theory research, and coding theory has proved to be instrumental in several interesting results in theoretical computer science as well. There has been several surprising applications of codes and the associated mathematical objects, in areas like algorithms, complexity theory and cryptography. Some part of the course will aim to discuss of those applications. However, we do not intend to be exhaustive.

The channel is not harmless in the real world. It introduces errors in the transmission. Depending on the application the error may be in the physical storage media (communication over time) or in the physical channel (communication over space). Some of the 0s gets flipped to 1s and vice versa, and some bits may get dropped too. For the purposes of this course we will study only model (Shannon studied several interesting variants), namely what are called Binary Symmetric Channels. In this model, each bit gets flipped with a probability p. That is, a 1 gets flipped to a 0 with probability and 0 gets flipped to 1 with probability p.

$$\begin{tikzcd} 0 \arrow[r, "1-p"] \arrow[dr, "p"'] & 0 \\ 1 \arrow[r, "1-p"] \arrow[ur, "p"] & 1 \end{tikzcd}$$

What is the natural strategy to cope up with errors in transmission? Create redundancy. For example, if Alice wants to send a bit 0 to Bob, she will do it five times, and send 11111 and ask Bob to take the majority of the bits as the bit that was sent. In this simple looking example we have all the essence. The string that was sent will be called the *codeword* and the original bit to be sent is called the *message*. There are only two codewords 00000 and 11111 in the above example. If we define the notion of distance as the hamming distance, then the majority decoding mechanism described above can also be seen as choosing the codeword that is closest to the received word. This natural strategy of decoding is called *nearest neighbor decoding* or *maximum likelyhood decoding*.

Now let us observe facts about guarantees. Clearly if the channel is such that it will not corrupt more than 2 bits in a sequence of 5 bits, then Bob will be able to decode the message bit correctly. But the channel may actually flip more number of bits but with relatively lower probability. Thus if we increase the number of copies we make of the original message, with high probability (over the errors) introduced by the channel we are going to be able to decode the bit correctly.

To fix some notations, we denote $E : \{0,1\}^k \to \{0,1\}^n$ as the encoding function where k is the message length (in general) and n is the length of the codeword (which we will call the *block length*. Let $m \in \{0,1\}^k$ be a message, and $E(m) \in \{0,1\}^n$ is the transmitted word. The channel corrupts the message and let $y \in \{0,1\}^n$ is the received word. The error introduced by the channel could also be thought of as a string $\eta \in \{0,1\}^n$ where the η_i determines whether $y_i = (E(m))_i$ or not.

We want the following guarantee for any $m \in \{0,1\}^k$ as translating the above intuition:

$$Pr_\eta(D(E(m) + \eta) = m) \geq 1 - o(1)$$

where the $o(1)$ term is exponentially small depending on n and hence on k (since c is a constant).

Although the above statement is written in terms of a probability over choice of the channel error vector, a natural combinatorial guarantee that we would want is an encoding and decoding scheme such that if the error string η has weight at most $t < \frac{d}{2}$ the decoder retrieves the message correctly. That is, the encoder-decoder pair is guaranteed to get the message across the channel, if the number of corrputions by the chaneel is limited a number t. Indeed, the relative redundant information we sent should be minimised (which is the ration of k and n called the rate of the code).

Shannons theorem essentially states that under sutiable choice of the parameters there is a pair of encoding-decoding functions that can achieve this high confidence decoding of the original message. We will state the theorem formally only later. But again, the spirit of the theorem is that there does exist good encoding and decoding schemes with respect to the parameters we usually care about (which we make precise later). The area of algorithmic coding theory essentially attempts to address the question of constructing coding schemes for which there is an efficient decoding.

We conclude the lecture by stating that the three mathematical objects that we stated in this lecture do have some interconnections among themselves and also the pseudorandom objects that we are going to state in the next lecture too.

Part II

Exercise & Problem Sets

Chapter 3

Exercises

3.1 Exercises

3.2 Curiosity Drive

Here we list down all the ”out of curious” questions that we discussed (sometimes even not discussed) in the class (and hence in this document).

Chapter 4

Problem Sets

Bibliography

[Beck, 1978] Beck, J. (1978). On 3-chromatic hypergraphs. *Discrete Mathematics*, 24(2):127 – 137.

[Erdos, 1963] Erdos, P. (1963). On a Combinatorial Problem. *Nordisk Matematisk Tidskrift*, 11(1):5–10.

[Gabber and Galil, 1981] Gabber, O. and Galil, Z. (1981). Explicit constructions of linear-sized superconcentrators. *J. Comput. System Sci.*, 22(3):407–420. Special issued dedicated to Michael Machtey.

[Margulis, 1973] Margulis, G. A. (1973). Explicit constructions of expanders. *Problemy Peredači Informacii*, 9(4):71–80.

[Radhakrishnan and Srinivasan, 2000] Radhakrishnan, J. and Srinivasan, A. (2000). Improved bounds and algorithms for hypergraph 2-coloring. *Random Structures & Algorithms*, 16(1):4–32.

[Schmidt, 1964] Schmidt, W. M. (1964). Ein kombinatorisches Problem von P. Erdos und A. Hajnal. *Acta Mathematica Academiae Scientiarum Hungarica*, 15(3):373–374.