An Invitation to Computational Complexity Theory Research

Dr. Nutan Limaye

ITU-Copenhagen

E-mail: nuli@itu.dk

My area of research is Computational Complexity Theory. In this article, I would like to give a brief introduction to this topic. My attempt is to explain this area to a younger version of me. Say when I was 20 years old. Back then, I did not know anything about Complexity Theory. I was interested in puzzle solving. I liked to think about the design and analysis of algorithms. And I really enjoyed mathematical thinking. If you share some of these traits with me, then read on!

Before I start, here are some things you may want to know. Firstly, there are many really good articles and books about this topic. I will point you to many resources throughout. So, this article will only serve as a gateway into the rich world of complexity theory. You are encouraged to follow all and any rabbit holes and find your path to explore this wonderland.

Secondly, it will be good to have some familiarity with algorithmic thinking. It is a bonus if you have done some coding in the past. Some familiarity with basic mathematical concepts such as probability theory and graph theory is also useful.

1. What is complexity theory?

Complexity theory attempts to understand the boundary of computation, namely what can be computed efficiently and what cannot be computed efficiently. It does so by creating mathematical abstractions of computation and by reasoning about them. The notion of efficiency depends on the context. Typically, the context determines the resources we are working with. For example, we may want time-efficient or space-efficient algorithms. In some situations, we may want energy efficiency or efficiency of communication.

But what exactly do I mean by efficiency? Well, this can be made quite precise and mathematical. Let me illustrate with an example.

Primes. You are given a number N in binary notation. So, the length of the input is $n = \lceil \log N \rceil$[1]. For example, if the input is 14, then it is written down as 1110. Now, you are asked to check whether N is a prime number or not. A typical algorithm, which you may have coded in your favorite programming language, will do the following. For each number r in the range $[2, \lceil \sqrt{N} \rceil]$ check whether r divides N. If some r divides N then output that N is not a prime. If no such number is found, then declare that N is a prime number. As any number N must have at least one factor less than $\lceil \sqrt{N} \rceil$, the algorithm is correct. Now, let us analyze the running time of this algorithm.

It is easy to see that its running time is bounded by $O(\sqrt{N} \cdot \log N)$[2]. This is because it does a division computation for each number in a range of size $O(\sqrt{N})$. Assuming that each division step takes time $O(\log N)$, the bound follows. Is that efficient? To answer this question, let us first define the notion of efficiency[3] .

> We say that an algorithm is efficient if it takes number of steps which is at most a polynomial function of the length of its input. In this case the algorithm is said to be a *polynomial time algorithm.*
> We will say that a problem is *hard* if it does not have a polynomial time algorithm.

Here, the length of the input is $n = \lceil \log N \rceil$. The running time of the algorithm is $O(\sqrt{N} \cdot \log N)$. The running time is equal to $O(2^{n/2})$ when expressed as a function of n. So the running time of the algorithm is *exponential* in the length of the input.

In fact, the problem of designing a poly(n) running time[4] algorithm algorithm for this problem was open for many decades. In 2002 three Indian scientists Manindra Agrawal (IIT Kanpur), Neeraj Kayal (MSRI, Bangalore)[5] and Nitin Saxena (IIT Kanpur) gave a polynomial time algorithm for this problem [1].

P and NP. Several important problems can be solved in polynomial time (in the length of their inputs). Some well-known examples are multiplying or adding integers, Gaussian elimination, determinant computation, matrix multiplication, and finding a maximum flow in a network [6]. The class of all problems that can be solved in polynomial time is called P.

Which problems are in P and which are not? Can we categorise problems that are not in P? Can we prove that a certain problem is not in P? All these are very interesting and deep questions.

For example, consider the following problem. Given two numbers N and k (in binary), check

[1] All the logarithms are base 2, unless stated otherwise.
[2] Let $F, G : \mathbb{N} \to \mathbb{N}$. We say that $G(n)$ is in $O(F(n))$, if there exists a constant c and a number n_0 such that $G(n) \leqslant c \cdot F(n)$ for all inputs of length $n \geqslant n_0$. Informally, the growth rate of $G(n)$ is slower than the growth rate of $F(n)$ for large enough n.
[3] This is the most accepted notion of efficiency. Here are some papers that consider other refined notions of efficiency.[37].
[4] We will use poly(n) to denote the class of functions $\bigcup_{c \leqslant 0} \{n^c\}$.
[5] In fact, Neeraj Kayal is originally from Assam!

if N has a prime factor bounded by k. We will call this problem Factor. It is not known whether Factor is in P or not. It is widely believed that Factor is not in P, i.e., it is a hard problem.

The hardness of Factor has implications for cryptography. Today many online transactions, password verifications, and credit card payments depend on secure cryptography protocols. At the heart of the security guarantees of these crypto-systems, lies the assumption that certain problems such as Factor are hard. See for instance Section 2 in [9] for more information about this problem and its relation to cryptography.

We would like to prove that Factor is hard. But unfortunately, we do not have the techniques to prove this result yet. What we can say about this problem is the following. Given a number $m \leqslant k$, we can check whether m divides N or not in P. That is, it is *easy to verify*.

Here is another example of a problem that is easy to verify. You are given n positive numbers $a_1, a_2, \ldots, a_n$ and a target value T. Check whether there exists a subset of these numbers that add to exactly T. That is, is there a subset $S \subseteq [n]$ such that $(\sum_{i \in S} a_i) = T$, where $[n]$ stands for the set $\{1, 2, \ldots, n\}$. This problem is known as the Subset Sum problem. This problem is also not known to have polynomial time algorithms. However, like Factor, even this is easy to verify. Given a subset of numbers, checking whether they add to T or not requires integer additions, which can be done in polynomial time.

There are many problems which have this feature. The class of problems that are easy to verify is called NP. It is easy to see that any problem that can be solved in P is in NP, i.e., P $\subseteq$ NP. This is because there is nothing left to verify once the problem is solved. So these problems are trivially easy to verify. However, whether NP $\subseteq$ P or not is a long-standing open problem.

A million dollar problem, literally! This problem is famously known as the P vs. NP problem. This is one of the seven named *Prize Problems* established by the Clay Mathematics Institute (CMI). Here I quote the description given by CMI about these Prize Problems [5].

> *"In order to celebrate mathematics in the new millennium, The Clay Mathematics Institute of Cambridge, Massachusetts (CMI) established seven Prize Problems. The Prizes were conceived to record some of the most difficult problems with which mathematicians were grappling at the turn of the second millennium; to elevate in the consciousness of the general public the fact that in mathematics, the frontier is still open and abounds in important unsolved problems; to emphasize the importance of working towards a solution of the deepest, most difficult problems; and to recognize achievement in mathematics of historical magnitude."*

The P vs. NP problem is at the center stage of computational complexity theory. The problem seems to be notoriously hard to resolve. If we cannot solve the problem we are trying to solve, how should we proceed next? We could do many things. We could simplify the problem and try to resolve a simpler version of the problem. We could reformulate the problem differently and try to attack the reformulation. We could identify a bottleneck for resolving the problem using the existing techniques and develop a theory for explaining the failed attempts. We could imagine a world where

the problem is resolved in a specific way, say P $\neq$ NP (a widely held belief), and develop methods to cope with computational problems in that world.

When a fundamental problem you are trying to tackle is rich enough, every natural sub-problem arising from it is likely to inherit its richness. As it turns out, all these and many more themes have been explored in complexity theory, each giving rise to profound insights into the P vs. NP problem.

2. What is the history of complexity theory?

The history of complexity theory starts before physical computers came into existence. This may seem counter-intuitive today. How can one think about computing without a computer? But that is what is surprising about this area of computer science. It involves abstract reasoning about computation without actually ever using a computer! The prerequisites for this are mathematical reasoning, logical thinking, and a robust definition of computation.

All of these came together in the foundational work of Alan Turing. He developed an abstract model of computation, which was later named after him, a Turing machine. This appeared in his 1937 paper titled "On computable numbers, with an application to the Entscheidungsproblem" [32].

Consequently, the foundations of complexity theory were laid by early works of scientists such as Alan Turing, von Neumann, Kurt Gödel, and David Hilbert. For more information about that era of computer science, please take a look at these references [15, 21, 29, 31].

By the 1950s, computers were being used for basic arithmetic.[6] Some problems were efficiently solvable on these computers. But some required brute-force computation. For example, Factor and Subset Sum problems listed above can be solved in exponential time using a brute-force algorithm. That is an algorithm that enumerates all possible solutions for the given problem and verifies whether any of them is a valid solution. For example, consider the following brute-force algorithm for the Factor problem. For every number i between 2 and k, check whether i is a factor of N. For the Subset Sum problem, enumerate every subset $S \subseteq [n]$ and check whether the elements indexed by S sum to T or not. These are brute-force algorithms. In the worst case, these algorithms can take exponential time.

In the early 1970s, Cook in the US and Levin in the Soviet Union simultaneously and independently formalized this notion of bute-force search. They defined the concept of NP problems. They gave a list of problems in NP that were practically relevant, but the only known algorithm for them was a brute-force algorithm.

There is another fundamental definition in their work. This is the notion of NP-completeness. That is slightly more tricky to describe. Informally speaking, it allows us to compare problems that are in NP and show that they are computationally equivalent to each other up to simple algorithmic manipulations.

[6] There was a team of *human computers*, primarily consisting of women, who developed fast computational skills and later learned to code using early programming languages such as Fortran [28].

Some of these notions were studied by several predecessors of Cook and Levin under different guises [29]. For example, the genesis of some central computational themes can be found in the works of Kolmogorov, Shannon, Hartmanis and Strean, Blum, Smale, Edmonds, and Rabin. Cook and Levin's works formalized many of these themes. Their work is an important cornerstone in the history of complexity theory.

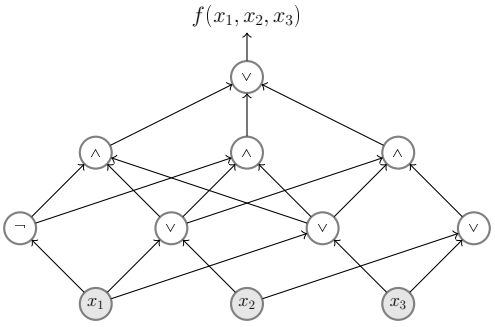
It may surprise some readers today to know that when Cook and Levin wrote their papers, hardly any universities in the world had Computer Science departments. Computer scientists were employed either in the Electrical Engineering departments or in the Mathematics departments till the late 70s. Computer Science (and consequently Complexity Theory) is a relatively new field. I would argue that the history of Complexity Theory is being written today as we speak. I hope some of the readers will become important personalities in this story!

3. What are the famous and classical questions in this field?

In this section I would like to describe some important themes studied in complexity theory. The list below is neither exhaustive nor representative. You may check textbooks and lecture notes from the references for more details on each topic and for exploring other topics.

3.1. ***The circuit complexity versions of the P vs. NP question***

Circuits are models of computation of Boolean functions. You may have heard about them in the context of microchips and electronics. They consist of AND (denoted as $\wedge$), OR (denoted as $\vee$) and NOT (denoted as $\neg$) gates. The gates are connected in an acyclic graph structure. There are some designated nodes for inputs and they are labeled with bits $\{0,1\}$. There is a designated node for the output. Each gate computes a Boolean function of its inputs. An $\wedge$ gate computes the logical AND of its inputs. That is, it outputs 1 if and only if all its input bits are 1. Similarly, an $\vee$ gate computes the logical OR of its inputs. It outputs 1 if and only if at least one of its input bits is 1. Finally, the $\neg$ gate outputs the negation of its input bit. The values propagate from the input nodes to the output node. See for instance the figure below.



Here the circuit takes inputs from $\{0,1\}^3$ and outputs a value from $\{0,1\}$. The depth of the circuit is the length of the longest input to output gate path. The size of the circuit is the number of operators used in it. So here, the size is 8 and the depth is 3.

We have already seen one notion of efficiency. We said that a function is efficiently computable if it has a polynomial time algorithm. But based on the circuit model, we could define another notion of efficiency. We will say that a function is efficiently computable if it has a polynomial-size circuit. The notion of the size of a circuit is a bit delicate. Let me define it formally.

> A circuit C_n with n inputs is said to compute a function $f_n : \{0,1\}^n \to \{0,1\}$ if for every input $x \in \{0,1\}^n$, $C_n(x) = f_n(x)$. A function family $\mathcal{F} = \{f_n\}_{n \geqslant 0}$ is said to be computed by a family of circuits $\mathcal{C} = \{C_n\}_{n \geqslant 0}$, if for every n, C_n computes f_n. We say that $\mathcal{F}$ has polynomial size circuits if there is a circuit family $\mathcal{C}$ such that for every n, C_n computes f_n and the size of C_n is bounded by poly(n).

If the definition above seems too formal, here is an informal description. Note that a circuit comes with some designated number of input gates. So it can only compute a function that depends on that many input bits. However, we would like to compute functions on n bits as $n \to \infty$. So we say that functions are computed by a family of circuits rather than just one circuit.

After this brief technical digression, let me bring the reader back to our main theme. The class of functions that have polynomial-size circuits is called P/poly in the literature. Can we show that there exists a function f in NP such that f is not in P/poly? Clearly, this is a variant of the P vs. NP question. Moreover, it is an interesting variant. This is because it is known that proving NP is not contained in P/poly is sufficient to prove that NP is not contained in P as well. This gives a different route for attacking the P vs. NP question.

The circuit model is quite structured. Therefore, one hopes that proving such results will be easier than attacking the P vs. NP question. However, unfortunately, nothing better is known for this P/poly vs. NP question as well. Over the years, researchers have explored this question in different restricted settings. Here is one such setting.

Constant depth circuits. Consider a class of functions computable by circuits that have polynomial size but fixed constant depth. That is, the depth is a fixed constant (say, 10 or 1000) independent of the input length. This class is known as AC^0 and is well-studied. We can ask the corresponding AC^0 vs. NP question. Interestingly, this question is fully resolved! That is, we know that there exists a function in NP that is provably not in AC^0 [2, 7, 13, 26, 30].

This result (established in a body of work by many researchers in a large set of papers in the 1980s) was a landmark breakthrough in complexity theory. It took a step closer to resolving the P vs. NP question.

The next breakthrough in this line of work is by Ryan Williams in 2010 [36]. We do not have enough vocabulary to discuss the result here. But what is worth noting is that this result managed to develop a new proof technique to prove a version of the P vs. NP question.

Barriers. Here is a natural question arising from the above successes. Can the proof strategy used to resolve AC^0 vs. NP question be extended to prove the P/poly vs. NP question or the P vs. NP question? To answer this question positively, one needs to find a way to apply the proof strategy in a more general setting. We have not made much progress in this endeavor after almost three decades of initial breakthroughs.

What if the answer to the above question is no? What if the proof strategy used to resolve the AC^0 vs. NP question is not strong enough to prove other results? If that is the case, can we mathematically prove the limitations of this proof strategy?

To do that, one needs to formalize what one means by 'this proof strategy'. Razborov and Rudich precisely did that in the 1990s [27]. They developed a meta-theory to talk about proof techniques that prove theorems of interest. They compared the powers of these proof techniques. They established a framework to argue about the limitations of proof strategies, which gave rise to a rich theory of meta-reasoning.

Algebraic Models. In another line of work, Leslie Valiant introduced algebraic circuits in the late 1970s [33]. These are circuits that have $\times$ and $+$ gates. The inputs are variables or constants (not necessarily just 0 and 1), and the output is a polynomial. The number of inputs is the parameter of interest. The size of the circuit is the number of $+, \times$ operators in it. We say that an n-variate polynomial $P_n(x_1, \ldots, x_n)$ is computed efficiently by a circuit with n input gates if the circuit has size $\text{poly}(n)$. Here too, as $n \to \infty$, we will say that a family of polynomials $\{P_n\}_{n \geqslant 0}$ is computed by a circuit family $\{C_n\}_{n \geqslant 0}$ as before.

Consider the following polynomial.

$$P(x_1, \ldots, x_n) = \sum_{S \subseteq [n]} \prod_{i \in S} x_i$$

It sums over all possible subsets S of the index set of the variables, i.e. $[n]$, and for each such set, it adds a monomial obtained by multiplying the variables from S with a coefficient 1. So here, the size of the circuit is $O(2^n)$. But consider the following alternative representation.

$$P(x_1, \ldots, x_n) = \prod_{i \in [n]} (1 + x_i).$$

It is not hard to see that this computes the same polynomial. However, this has a size of only $O(n)$.

In general, a polynomial may seem quite complicated on the face of it. But it may have an efficient representation. Here, by efficient, we again mean a circuit of size $\text{poly}(n)$.

Does every polynomial have an efficient circuit computing it? This is a version of the P vs. NP question in the algebraic setting. It is called the VP vs. VNP question. It is known that if NP is not contained in P/poly, then VNP is not contained in VP [4]. Thus, proving VNP $\neq$ VP is formally easier than proving NP $\neq$ P/poly.

Overall, the circuit model allows us to attack the P vs. NP question in many different ways. We may not be close to resolving the P vs. NP question yet, but certainly, this area is taking us forward in the right direction.

3.2. ***Approximation algorithms and hardness of approximation***

Consider a social network, e.g., Facebook (FB). Let us say we assign a node for every profile on FB. We link two nodes if the two profiles are friends with each other on FB. This collection of nodes and links is called a graph. You can imagine that this is a graph with millions of nodes and links. We will use $G = (V, E)$ to denote the graph. Here, V is the collection of nodes, and $E \subseteq V \times V$ is a collection of links. The links are undirected.

MaxCut*:* ***the problem statement*** For a given $X \subseteq V$, a cut C_X is defined to be the set $\{(u, v) \in E \mid u \in X \text{ and } v \in V \backslash X \text{ or } v \in X \text{ and } u \in V \backslash X\}$. That is, the cut corresponding to $X \subseteq V$ is a subset of the links of G that have one node in X and the other in $V \backslash X$. The size of the cut C_X is the number of links in it, i.e., $|C_X|$.

Given a graph $G = (V, E)$ and a parameter k, check whether the graph has a cut of size at least k. This is the MaxCut problem. This problem is an NP-complete problem.

A brute-force algorithm for this will just go over every $X \subseteq V$ and compute the size of C_X. Finally, it will output 'yes' if there is an X for which C_X has at least k links and output 'no' otherwise. Given a set X, checking whether C_X has a size at least k or not will take time $\text{poly}(n)$. As there are 2^n possibilities for X, where n is the size of V, it is easy to see that the brute-force algorithm will take time $O(2^n \, \text{poly}(n))$.

Can we design an algorithm that runs in time $\text{poly}(n)$? Well, if we do, then, in particular, we will prove that P = NP, thereby resolving the P vs. NP question. Given all the above discussion, designing a polynomial time algorithm for the problem may be difficult! One natural question to ask is, can we answer the question approximately?

Approximation algorithms Let us first define the notion of approximation.

For a given G, let $\text{OPT}(G)$ denote the size of the maximum cut. For $\alpha \geqslant 1$, we say that a cut C is an α-approximation if

$$|C| \geqslant \frac{\text{OPT}(G)}{\alpha}.$$

Here, α is called the approximation ratio. We would like it to be as close to 1 as possible.

We may not be able to compute the best solution efficiently, but how about designing an efficient algorithm for an α-approximation? In fact, in this case, we can very easily compute a 2-approximation as follows.

Arrange the vertices in some arbitrary order $\{v_1, \ldots, v_n\}$.

Let $A = \{v_1\}, B = \varnothing$.

For every $2 \leqslant i \leqslant n$

    If v_i has more neighbours in A than in B,
    then add v_i to B
    else add it to A.

Final output is (A, B).

Analysis of the algorithm. The algorithm terminates after these steps. It is not hard to see that the total time taken is only poly(n). To see why this gives a 2-approximation, first observe that OPT(G) is at most $|E|$. We now need to show that the size of the cut given by (A, B) is at least $|E|/2$. This is slightly technical.

For each link in the graph, let the node with the higher number be the owner of that link. For example, if (v_i, v_j) is a link in the graph and $i < j$, then v_j is the owner of (v_i, v_j). Suppose a vertex v_i is an owner of r_i many links. Then we know that $\sum_{i=1}^{n} r_i = |E|$. Suppose v_i is added to B by the algorithm, then at least half of its neighbors among the vertices $\{v_1, \ldots, v_{i-1}\}$ are in A. This means that when v_i is added to B, at least $r_i/2$ links are added to the cut. Similarly if v_i is added to A. Hence, the cut constructed by the algorithm contains at least $|E|/2$ links.

Improving the approximation ratio. The next obvious question is: can we get a better approximation ratio than 2? This is a well-studied question. The best-known approximation ratio for the problem is around 1.139 by Goemans and Williamson [8]. They achieve this using a randomized strategy. The analysis of this strategy uses some really clever techniques. If you wish to read more about this, you may look up lecture notes by Venkatesh Guruswami here [11].

It is also known that no better than (17/16)-approximation can be obtained unless P equals NP. Such results are called *hardness of approximation* results. Typically, they show that an NP-complete problem is hard to approximate beyond a certain approximation ratio under some complexity-theoretic assumptions (e.g. P is not equal to NP).

In the early 2000s, Subhash Khot introduced a new complexity-theoretic assumption called *The Unique Games Conjecture* (UGC) [18]. The conjecture says that a certain computational problem, which he called the Unique Game, cannot be approximated efficiently. In the last two decades, this conjecture has been a driving force behind many hardness of approximation results. Assuming UGC and that P is not equal to NP, we can show that some NP-complete problems are not just hard to solve, but are also hard to approximate beyond a certain approximation ratio. Specifically, Khot, Kindler, Mossel, and O'Donnell showed that the approximation ratio for MaxCut achieved by Goemans and Williamson's algorithm is the best we can get assuming UGC [17].

3.3. ***Randomness in computation and derandomization***

A randomized algorithm is an algorithm that has access to random bits. Randomized algorithms are ubiquitous in many real-world applications including large-scale data analysis, disease prediction, image processing algorithms, weather prediction, and natural language processing.

As complexity theorists, we would like to understand the role of randomness in computation. How does randomness affect computation? Suppose we have a problem that is not known to be in P, for example, the Factor problem. Can we design efficient randomized algorithms for such problems? Before I go any further, let me formalize what I mean by *efficient randomized algorithms*.

> A randomised algorithm $\mathcal{A}$ is said to compute a function $f : \{0,1\}^n \to \{0,1\}$ efficiently if it has the following properties. During its computation, it tosses $\text{poly}(n)$ many random coins and for every x, it has the following behavior.
>
> If $f(x) = 1$ then it outputs 1 with probability at least $1 - p$.
>
> If $f(x) = 0$ then it outputs 0 with probability at least $1 - p$.
>
> It halts in $\text{poly}(n)$ steps.

Informally, the algorithm runs in polynomial time using at most polynomially many random bits. Moreover, it may have a small probability of error. It may output either a false-negative[7] with probability p, or it may output a false-positive[8] with probability p. (Here, think of p as a real number between $(0, 1/2)$. Typically, we want it to be as close to 0 as possible.)

The class of problems that have efficient randomized algorithms is called BPP. We would like to understand the relationship between P, NP, and BPP. Overall, we would like to understand how randomness affects computation. Is it possible to remove randomness from computation? Can it be done efficiently? [16]

Here is a relevant observation in this context. We can always apply a brute-force strategy to remove randomness from the algorithm. Suppose we have a randomized algorithm $\mathcal{A}$ that uses r random bits. Then we can design another algorithm $\mathcal{A}'$ as follows. For every $u \in \{0,1\}^r$, $\mathcal{A}'$ simulates $\mathcal{A}$ with u as its random bits. After all these simulations, $\mathcal{A}'$ outputs whatever majority of the simulations did.

$\mathcal{A}'$ is correct and not randomized. However, if the original algorithm used $\text{poly}(n)$ bits of randomness then $\mathcal{A}'$ will run for exponentially many steps. So, we can remove randomness from computation at the expense of time efficiency.

The process of removing randomness from computation is called *derandomization*. It is widely believed that efficient derandomization is possible, i.e., BPP equals P. The above naive brute-force strategy does not give an efficient derandomization. Moreover, looking at all the real-world applications BPP = P conjecture sounds counter-intuitive. Then why do experts believe that BPP = P?

[7] the true answer is 1, but the algorithm outputs 0

[8] the actual answer is 0, but the algorithm outputs 1

One reason to believe in efficient derandomization is evidence-based. For many problems, we first tend to obtain efficient (and often simple) randomized algorithms. Over time, a conglomeration of clever ideas leads to efficient deterministic algorithms for them. This has famously happened for the Primes problem we discussed earlier.

The other reason to believe in BPP = P comes from cryptography [22]. Let us assume that Factor is not solvable in P. (This is not known, but we believe it to be true.) As we discussed before, if Factor is not in P, we have provable safety guarantees for our cryptographic systems. The safety of the online banking systems, credit card payments, etc. depends on this assumption.

Under this assumption, complexity theorists have developed *pseudorandom generators* (PRG). PRGs are devices that assume that hard functions exist (e.g. assume that Factor is a hard function), and under this assumption, they *stretch* a small string of random bits into a very long string of random bits. The existence of such PRGs further implies that randomized algorithms can be efficiently derandomized.

Safe crypto-systems $\Leftarrow$ Factor is not in P $\Rightarrow$ Pseudorandom generators $\Rightarrow$ BPP = P.

3.4. ***Other important themes***

Here, I would like to give you a short list of a few more topics studied in complexity theory and some references for each.

Communication complexity. *[19, 20, 12]* Suppose we wish to compute a function on input bits that are split among two or more players. In this case, the players need to communicate with each other to compute the function. Communication Complexity studies the number of bits required to compute functions in this setting.

Coding theory. *[34]* Suppose we are trying to communicate between multiple parties in the presence of errors in the communication channel. In coding theory, we study how best to communicate in the presence of such errors.

Proof complexity. *[25]* A proof system is a set of axioms and rules that prove theorems. Peano arithmetic and trigonometry are examples of such proof systems. In proof complexity, we study the strengths and weaknesses of different proof systems. Specifically, we try to understand the computational limits of theorem proving.

Quantum complexity. *[23]* This area of complexity theory studies quantum computation. Here, the laws of quantum physics govern the rules of the computation. The theory of quantum computers precedes physical quantum computers. Just like Alan Turing created an abstract model

of computation before a physical computer was built, today, complexity theorists study quantum computation. A lot is known about this model.

For example, we know that Factor has a polynomial time algorithm on a quantum computer. In particular, this implies that if we can build sufficiently powerful quantum computers, then our classical cryptographic systems will be rendered unsafe!

Today, many leading tech giants such as IBM, Google, and Microsoft are developing quantum computers [14, 10]. Along with this, cryptographers are designing new crypto-systems that will be safe even if we can build quantum computers.

3.5. ***Closing remarks***

At this point, I would like to remind the reader that the list above is not comprehensive. You may wonder why I chose the above topics and not some other topics. Good question! Here are my reasons.

Firstly, it shows my personal biases. These are the topics that fascinate me the most and they are the topics that got me deeply interested in complexity theory.

Secondly, these are central to complexity theory. Any complexity theory text is incomplete without the mention of a good number of the above topics.

Finally, I chose these themes to demonstrate the importance of asking the right questions. The central goal of complexity theory is to understand computation. (Depending on your area of research, it may be something else.) The above questions are important because they are exactly aligned with this central goal. In my view, the success of your research career depends on whether you can ask the right questions or not. To quote Einstein –

> *If I had an hour to solve a problem and my life depended on the solution, I would spend the first 55 minutes determining the proper question to ask... for once I know the proper question, I could solve the problem in less than five minutes.*

4. How can I start working in this area?

If you are thinking about working in complexity theory, here are some pointers to get you started.

Prerequisites. The main prerequisites for working in complexity theory are algorithm design, randomized algorithms, and discrete mathematics. It will be great if you have also gone through the Toolkit for CS course offered by Ryan O'Donnell [24]. (All the course material and YouTube videos are available online.)

Texts. I highly recommend reading Mathematics and Computation, a textbook by Avi Wigderson [35]. He is one of the most renowned researchers in complexity theory. He has put

together a fantastic collection of topics for beginners in this book. (It also contains many advanced topics, but you may skip those the first time.)

Many standard complexity theory courses taught across the world also refer to Introduction to Complexity Theory: A Modern Approach, by Arora and Barak [3]. The textbook has a nice set of problems that you may be able to attempt on your own.

References

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Ann. of Math. (2)*, 160(2):781–793, 2004. doi:10.4007/annals.2004.160.781.

[2] Miklós Ajtai. $\sum^1_1$-formulae on finite structures. *Ann. Pure Appl. Log.*, 24(1):1–48, 1983. doi:10.1016/0168-0072(83)90038-6.

[3] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[4] Peter Bürgisser. Cook's versus Valiant's hypothesis. *Theoretical Computer Science*, 235(1):71–88, 2000. URL: https://www.sciencedirect.com/science/article/pii/S0304397599001838, doi:https://doi.org/10.1016/S0304-3975(99)00183-8.

[5] CMI. The millennium prize problems, May 2000. URL: https://www.claymath.org/millennium-problems/millennium-prize-problems.

[6] INFO 4220 Course Blog, Mar 2015. URL: https://blogs.cornell.edu/info4220/2015/03/10/the-origin-of-the-study-of-network-flow/.

[7] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Math. Syst. Theory*, 17(1):13–27, 1984. doi:10.1007/BF01744431.

[8] Michel X. Goemans and David P. Williamson. .879-approximation algorithms for MAX CUT and MAX 2sat. In Frank Thomson Leighton and Michael T. Goodrich, editors, *STOC'94*, pages 422–431. ACM, 1994. doi:10.1145/195058.195216.

[9] Shafi Goldwasser and Mihir Bellare. Lecture notes on cryptography, 2008. URL: https://cseweb.ucsd.edu/~mihir/papers/gb.

[10] Google. Quantum AI. URL: https://quantumai.google/hardware.

[11] Venkatesan Guruswami and Vibhor Rastogi. The PCP theorem and hardness of approximation,, Autumn 2005. URL: https://courses.cs.washington.edu/courses/cse533/05au/lecture17.pdf.

[12] Prahladh Harsha, Meena Mahajan, and Jaikumar Radhakrishnan. Communication complexity, Monsoon 2011. URL: https://www.tifr.res.in/~prahladh/teaching/2011-12/comm/.

[13] J Hastad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, page 6–20, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/12130.12132.

[14] IBM. IBM quantum system one. URL: https://research.ibm.com/interactive/system-one/.

[15] David S. Johnson. A brief history of NP-completeness, 1954-2012. 2012.

[16] Gill Kalai, Dec 2009. URL: https://gilkalai.wordpress.com/2009/12/06/four-derandomization-problems/.

[17] S. Khot, G. Kindler, E. Mossel, and R. O'Donnell. Optimal inapproximability results for max-cut and other 2-variable CSPs? In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 146–154, 2004. doi:10.1109/FOCS.2004.49.

[18] Subhash Khot. On the power of unique 2-prover 1-round games. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 767–775. Association for Computing Machinery, 2002. doi:10.1145/509907.510017.

[19] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997. doi:10.2277/052102983X.

[20] Troy Lee. 16:198:671 Communication Complexity, 2010. A course offered at Rutgers University (Spring 2010). URL: http://www.research.rutgers.edu/~troyjlee/cc.html.

[21] Richard Lipton and Ken Regan, Apr 2009. URL: https://rjlipton.wpcomstaging.com/the-gdel-letter/.

[22] N. Nisan and A. Wigderson. Hardness vs. randomness-a survey. In *[1989] Proceedings. Structure in Complexity Theory Fourth Annual Conference*, pages 54–, 1989. doi:10.1109/SCT.1989.41802.

[23] Ryan O'Donnell. Quantum computation and quantum information, Fall 2018. URL: http://www.cs.cmu.edu/~odonnell/quantum18/.

[24] Ryan O'Donnell. CS theory toolkit, Summer 2022. URL: https://www.youtube.com/playlist?list=PLm3J0oaFux3ZYpFLwwrlv_EHH9wtH6pnX.

[25] Toniann Pitassi. Propositional proof complexity, Winter 2017. URL: https://www.cs.toronto.edu/~toni/Courses/ProofComp2017/CS2429.html.

[26] A. A. Razborov. On the method of approximations. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, page 167–176, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/73007.73023.

[27] Alexander A. Razborov and Steven Rudich. Natural proofs. *J. Comput. Syst. Sci.*, 55(1):24–35, 1997. doi:10.1006/jcss.1997.1494.

[28] Margot Lee Shetterly. Katherine Johnson (1918-2020). *Nature*, 579(7798):341–342, 2020.

[29] Michael Sipser. The history and status of the P versus NP question. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, page 603–618. Association for Computing Machinery, 1992. doi:10.1145/129712.129771.

[30] R. Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. STOC '87, page 77–82, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/28395.28404.

[31] B. A. Trakhtenbrot. A survey of Russian approaches to perebor (brute-force searches) algorithms. *IEEE Annals of the History of Computing*, 6(4):384–400, oct 1984. doi:10.1109/MAHC.1984.10036.

[32] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. URL: http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf.

[33] Leslie G. Valiant. Completeness classes in algebra. In Michael J. Fischer, Richard A. DeMillo, Nancy A. Lynch, Walter A. Burkhard, and Alfred V. Aho, editors, *Proceedings of the 11h Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1979, Atlanta, Georgia, USA*, pages 249–261. ACM, 1979. doi:10.1145/800135.804419.

[34] Atri Rudra Venkatesan Guruswami and Madhu Sudan, 2022. URL: https://cse.buffalo.edu/faculty/atri/courses/coding-theory/book/index.html.

[35] Avi Wigderson. *Mathematics and Computation*. Princeton University Press, 2019. URL: https://www.math.ias.edu/avi/book.

[36] Ryan Williams. Nonuniform ACC circuit lower bounds. *J. ACM*, 61(1):2:1–2:32, 2014. doi:10.1145/2559903.

[37] Virginia Vassilevska Williams and R. Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *J. ACM*, 65(5):27:1–27:38, 2018.

Dr. Nutan Limaye is an Associate Professor at the IT University of Copenhagen in Denmark. Prior to this she was an Associate Professor at the Indian Institute of Technology (IIT) Bombay. She did her PhD at the Institute of Mathematical Sciences (IMSc) Chennai, followed by a stint at TIFR Mumbai, before joining IIT Bombay. Her research interests are in theoretical aspects of Computer Science. Specifically she is interested in Algorithms and Complexity Theory. Her work has won international acclaim and has been featured extensively in the media, for example in Quanta Magazine.

This article was invited by the Editors of Ganit Bikash and we thank Dr. Limaye for her contribution.