DIMACS REU Research Overview

Sam Hopkins

June 16, 2011

This is a nontechnical introduction to one potential research direction this summer. It is drawn from lecture notes I wrote for a small presentation at DIMACS. The research is with my mentor, Prof. Eric Allender. We are interested in the relationship between randomness and computational power.

Randomness Here are two strings:

$$000000000000000000000000000000000000000000000000000000$$
$$011000100000101001011101010100101010010101111101011 1.$$

Classical probability theory says that eight coin flips have an equal chance of resulting in the first string as they do in resulting in the second, but intuition says that the first is in some sense far more uniform than the second; the second is more irregular than the first.

In the mid-20th century, mathematicians began to attempt to formalize these intuitions. A number of theories of randomness and string complexity resulted. Of interest here is one that has come to be called either *algorithmic information theory* or *Kolmogorov complexity theory* after Russian mathematician Andrey Kolmogorov. The key insight is that what makes the first string above appear uniform is that it follows an easily-described pattern, and what makes the second irregular is that it fails to follow a pattern.

So, how do we talk mathematically about the existence of patterns? A second key insight is this: where there are patterns there are descriptions. That is to say: if I wanted to give you all the information you need to reproduce the first string—I want to describe that string to you—all I have to write is 0^{52}. But to give you all the information you need to reproduce the second string, I need to tell you *every bit* of the string! There is no shorter way to describe it. Here, then, is as technical a definition as we will give: "x is random" means that x admits no description shorter than itself. We will denote with R the set of random strings.

Computational Complexity Computational complexity theory analyzes how difficult computational problems are to solve. What is a computational problem? We will define by example. Here are a couple computational problems.

1. Given a sequence of length n (e.g. 0011010100 or 000100000), does it contain a 1?

2. Given $n \in \mathbb{N}$ (e.g. 2, 5683), what is its prime factorization?

How hard are these problems? We talk about hardness of a problem in terms of how much time—that is, how many steps—is required by an algorithm that solves the problem. We think of it as a function of the size of the input. That is, "P has time-complexity $f(n)$" means that, for all n, no input of size n requires more than $f(n)$ steps to get an answer. For example, problem 1 above has time complexity $\mathcal{O}(n)$, since we must look at every element of the sequence to solve it. The complexity of problem 2 we leave as an exercise.

We can also get an interesting analysis of complexity if we change our notion of *algorithm* somewhat. We have heretofore relied on an entirely intuitive notion of algorithm, and we will continue to do so in part, but we want to allow our algorithms to have access to a little more information when they do their computation.

Definition. (Warning: nonstandard terminology.) Let an *algorithm*$_f$, where $f : \mathbb{N} \to \mathbb{N}$, function as follows. On input x with size n, an algorithm$_f$ may consult a lookup table containing entries for each $n \in \mathbb{N}$ and retrieve the entry corresponding to the size of x. We call this entry the *advice string* for inputs of size n. The size of the advice string must be bounded by $f(n)$.

Now we are in a position to define a particular complexity class that we will require later on to state a conjecture.

Definition. We denote by $\mathsf{P/poly}$ the class of computational problems solvable by some algorithm$_f$ with time complexity $\mathcal{O}(n^k)$ for some $k \in \mathbb{N}$ and with $f \in \mathcal{O}(n^i)$ for some $i \in \mathbb{N}$.

We need one more notion from complexity theory. We have thus far spoken only of the difficulty of a computational problem in isolation, but we might also want to talk about the difficulty of such a problem with respect to another problem. Again, we will define by example. Consider the following two computational problems:

$$A = \text{Given a sequence in } \mathbb{N}, \text{ is it monotone?}$$
$$B = \text{Given a sequence in } \mathbb{N}, \text{ is it nondecreasing?}$$

Each of these obviously has time complexity $\mathcal{O}(n)$. However, suppose I already have all the answers to A. Then I can easily find answers to B. Namely, given an input x, I query my answers for A. If I find that x is not monotone, then I know that x is also not decreasing. If x is monotone, I can look at the first and last elements of x to find whether it is deceasing. Thus, I can easily transform answers to A into answers to B. We write $B \leq^{\mathcal{O}(1)} A$, where the superscript means that the transformation of answers from A to B is of constant difficulty (though we will leave this notion of difficulty ill-defined).

Now we can state an almost-conjecture that I may tackle this summer.

Almost-Conjecture 1.

$$\{A : A \text{ is a computational problem and } A \leq^{poly} R\} \subset \mathsf{P/poly}$$

where the poly in the superscript says that the translation of answers from R to A is polynomially-difficult.