So if you've dug around on the internet or heard of the P vs NP problem before coming into this course (we don't expect this from you by the way), you might know that P vs NP is somehow connected to cryptography or even hacking! Cool stuff indeed. Question 5 was designed by nobody other than Daniel in order to ~~satisfy his sadism~~ hopefully give you an idea on why P vs NP is a relevant problem for cryptography.

As you might know (again from sources outside of this course), computers suck at being random! Your computer can't generate *truly random* numbers. It can only generate what are called *pseudo-random* numbers, that is, numbers that look "random" enough (even though they are generated by some deterministic algorithm). Generating truly random numbers is important for cryptography though! Just like you want your passwords to be truly random (e.g. "123456" is a really bad password), we need good random number generators to make sure information is kept private.

In your homework, we defined a number generator G to be a poly-time computable function

$$G : \{0,1\}^n \to \{0,1\}^{n+1}$$

where n is a variable. This is a really restrictive definition, only used for the sake of simplicity on the assignment. In general, a **number generator** G is a poly-time computable function

$$G : \{0,1\}^n \to \{0,1\}^{l(n)}$$

where $l : \mathbb{N} \to \mathbb{N}$ is a function such that $l(n) > n$ for all n. I like to think of G as a machine: you give it a n-bit input from $\{0,1\}^n$ as a "random seed" (if you've played games like Minecraft this should look familiar!), and it will do some computation (which takes polynomial time) and give you a $l(n)$-bit output (with $l(n) > n$).[1]

Number generators (from the above definition) need not be "random", whatever that means. For example, the following number generator doesn't look very random...

$$G : \{0,1\}^n \to \{0,1\}^{2n}, G(x) = xx.$$

In other words, G takes in the random seed x and just outputs x appended to itself. If you were given this machine G, and you've tried feeding it some inputs (say $0101, 0, 011, 0111$), after some time you might start to realize a pattern with G's outputs (in our case it would be $01010101, 00, 011011, 01110111$): it just duplicates whatever you gave it! After some time, you start to realize that this doesn't seem like a "random" number generator.

Now imagine a game: I have a *truly random* number generator T, which outputs strings from $\{0,1\}^{2n}$ with uniform probability.[2] I'll also have another *truly random* number generator B which outputs strings from $\{0,1\}^n$ with uniform probability. $G : \{0,1\}^n \to \{0,1\}^{2n}$ is defined as before. Here's how the game will work: I will flip a coin. If I get heads, I will write whatever T outputs (which is a string in $\{0,1\}^{2n}$) on a piece of paper. If I get tails, I will take whatever B outputs, say $x \in \{0,1\}^n$, then compute $G(x)$ and write $G(x)$ on a piece of paper. After I've written the string, I will pass the paper to you. Your job is now to guess whether I got heads or tails during my coin flip process. In other words, you want to try your best to answer the question "did I generate this $2n$-bit string using a truly random process, or did I plug some number through $G(x)$?"

To answer that question, you could try a strategy like this:

- If the $2n$-bit string isn't of the form xx for some n-bit string x, then you can guarantee that I landed heads, as G's output is always just some string duplicated.

[1] In practice, $l(n)$ is usually orders of magnitude greater than n: in the Minecraft analogy, given a random seed of say 128 bits length, we will generate a complete Minecraft world out of it, usually taking up at least a few megabytes!

[2] Formally this would be a *random variable* with uniform distribution over $\{0,1\}^{2n}$, if you have taken stats before!

- If the $2n$-bit string is of the form xx, then you will *guess* that I landed tails. You can't guarantee that the string is generated by G, since there is always a chance that the truly random process of T just happened to generate a string of the form xx. But the chance of this happening, compared to just me landing tails, is pretty small, so your best bet would be that I've used G to generate this string.

Here's a table to illustrate the possible outcomes of this game, if you use the above strategy:

Actual value of coin flip	String is of the form xx	String is not of the form xx
Heads	Guess Tails	Guess Heads
Tails	Guess Tails	(This never happens)

Again, if the string is not xx, then your guess that I've landed heads is guaranteed to be correct. On the other hand, if the string is of the form xx, then there is a possibility that you are wrong (corresponding to the "Heads, String is of the form xx" cell). The probability of this case happening is

$$\begin{aligned} P(\text{String is } xx \mid \text{Heads})P(\text{Heads}) &= \frac{1}{2} \cdot \frac{\text{\# of strings in } \{0,1\}^{2n} \text{ of the form } xx}{\text{\# of strings in } \{0,1\}^{2n}} \\ &= \frac{1}{2} \cdot \frac{2^n}{2^{2n}} = \frac{1}{2^{n+1}}. \end{aligned}$$

So the probability that you guess wrong is $\frac{1}{2^{n+1}}$, which is really small when n is large. Effectively, you are extremely likely to guess correctly using the strategy described above.

The reason why you are able to guess correctly is precisely because $G : \{0,1\}^n \to \{0,1\}^{l(n)}$ sucks as a random number generator! You can easily predict, with good probability, which strings look like outputs from G, compared to a truly random output. This is where the idea of a pseudo-random number generator comes in. If G were a "random" number generator, then you wouldn't be able to distinguish the outputs of G from the outputs of T as easily, and hence you wouldn't be so successful in the above game.[3]

To formalize this procedure, instead of presenting the string on the paper to a human, we will present this string to a Turing machine opponent (which we assume runs in poly-time, see footnote 3). This Turing machine, instead of guessing "heads" or "tails", will instead accept or reject the string to indicate its guess. Given such a Turing machine D (for *Distinguisher*), we define $p_D(n)$ to be the probability that it guesses heads for an input of size n, given that the actual value of the coin flip is tails; we define $r_D(A)$ to be the probability that it guesses heads for an input of size N, given that the actual value of the coin flip is heads. The "rate of success"

[3]There is still a way to win most of the time! You could go through *every string* $x \in \{0,1\}^n$, calculate $G(x)$, and see if there exists any $x \in \{0,1\}^n$ such that $G(x)$ matches the string written on the paper. The table would then look like

Actual value of coin flip	String is of the form $G(x)$	String is not of the form $G(x)$
Heads	Guess Tails	Guess Heads
Tails	Guess Tails	(This never happens)

so the only case where you fail is the top left cell, and the probability of that occurring is

$$P(\text{String is } G(x) \mid \text{Heads}) \cdot P(\text{Heads}) = \frac{1}{2} \cdot \frac{\text{number of outputs of } G}{2^{l(n)}} \leq \frac{1}{2} \cdot \frac{2^n}{2^{l(n)}} = \frac{2^n}{2^{l(n)+1}}.$$

(The number of outputs of G is $\leq 2^n$, since G is a function with domain $\{0,1\}^n$, so G's range could consist of at most 2^n numbers.)

This is a *computationally infeasible* strategy though: you'd have to compute G on all 2^n strings, which takes exponential time and hence is practically useless on large input sizes. That's why we only restrict ourselves to *poly-time strategies*, i.e. you're restricted to doing a poly-time computation before guessing. In the problem set, we've added the restriction that any distinguisher D must be poly-time for this reason.

of this Turing machine D would therefore be

$$P(\text{Win}) = P(\text{Accept} \mid \text{Heads})P(\text{Heads}) + P(\text{Reject} \mid \text{Tails})P(\text{Tails}) = \frac{1}{2}r_D(n) + \frac{1}{2}(1 - p_D(n))$$
$$= \frac{1}{2} + \frac{1}{2}(r_D(n) - p_D(n)).$$

So if $r_D(n) - p_D(n) > 0$, D will guess correctly more than half of the time. If $r_D(n) - p_D(n) < 0$ though, D guesses correctly less than half of the time. But then we can just create D', which is D but swapping the accept and reject states, so that D' will guess correctly more than half of the time. Just like how consistently guessing wrong on a multiple choice quiz takes "skill", if D can consistently give a wrong answer, then in some sense we can still distinguish between G and a truly random number generator!

Thus, if there is a Turing machine D such that $r_D(n) - p_D(n)$ is much greater than 0 or $r_D(n) - p_D(n)$ is much smaller than 0, then we can tell that G isn't truly using this guessing game. So if G were a really good random number generator (in this case we call it a **pseudo-random number generator**), ideally we want $|r_D(n) - p_D(n)|$ to be really small for all D, so that the chance of guessing correctly for any Turing machine is really close to $\frac{1}{2}$! The condition that "for every poly-time Turing machine D and every $c > 0$, for large enough n we have $|p_D(n) - r_D(n)| \leq \frac{1}{n^c}$" is an attempt to capture this concept in mathematical terms: as n gets large, we want the probability of success to be as close to $\frac{1}{2}$ as possible, so we want to make $|r_D(n) - p_D(n)| = |p_D(n) - r_D(n)|$ as small as possible: smaller than $\frac{1}{n^c}$, for any $c > 0$.

Some bad news: we actually don't even know if pseudo-random number generators exist at all! To prove that a number generator is pseudo-random, you'd have to go through every single Turing machine D and show that $|p_D(n) - r_D(n)| \leq \frac{1}{n^c}$ for large enough n. But showing that a number generator G is *not* pseudo-random is easier: you can just find a poly-time D such that the inequality $|p_D(n) - r_D(n)| \leq \frac{1}{n^c}$ won't always be satisfied for some $c > 0$, *no matter how large n gets*. As an example, I will show you that the not-so-random number generator $G : \{0,1\}^n \to \{0,1\}^{2n}$, $G(x) = x$ is not pseudo-random. You may want to imitate a proof like this for part (a).

Example: $G : \{0,1\}^n \to \{0,1\}^{2n}$, $G(x) = xx$ is not pseudo-random.
Proof. Let D be the Turing machine that does the following:

$D(w)$: If the left half of w is equal to the right half of w:
  Reject
Accept

Then D will reject any output from G, and since the string is generated by G when we flip tails, we have $p_D(n) = 0$ for all n. On the other hand, the probability that D will accept a randomly generated string from $\{0,1\}$ is $1 - \frac{1}{2^n}$ as we have calculated from before, so $r_D(n) = 1 - \frac{1}{2^n}$. We have

$$|p_D(n) - r_D(n)| = 1 - \frac{1}{2^n}.$$

Now if $n > 2$, then $\frac{1}{2^n} < \frac{1}{2}$, so we have $|p_D(n) - r_D(n)| > \frac{1}{2}$. Therefore if we let $c = 1$, for $n \geq 2$ we have

$$\frac{1}{n^c} \leq \frac{1}{2} < |p_D(n) - r_D(n)|.$$

So no matter how large n gets, we can't guarantee $|p_D(n) - r_D(n)| \leq \frac{1}{n^c}$. Thus G is not pseudo-random per definition. ■

(b) asks you to prove that if P = NP, then there are no pseudo-random number generators at all! So if someone manages to prove P = NP, then all of this cryptography talk has gone to waste, and you can come back and complain to us that we've assigned you this question in the first

place. After you've withdrawn all the money from your bank account, of course, in preparation for the impeding global financial crisis resulting from the obsolescence of encryption methods.

Unfortunately, a good majority of computer scientists believe that $P \neq NP$, so you'll still have to do this problem either way. Sorry.

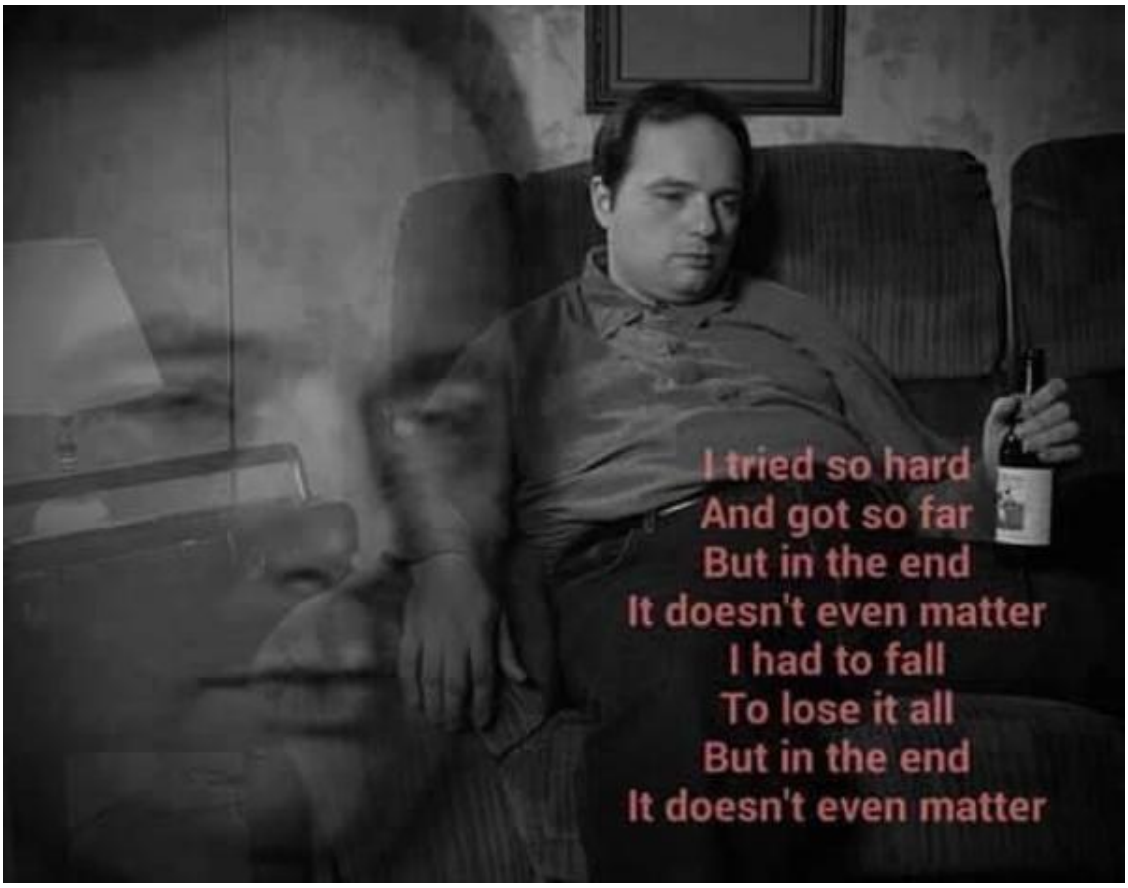

Figure 1: Cryptographers, after $P = NP$ is proven, probably.

I'm not getting paid to write this. Send me some sushi coupons or something.

- Paul