Problem 8: Who Doesn't Want to be a Millionaire

By Nalin Ranjan

This problem was not attempted by many. Though all of the necessary background was there, it would have been difficult to understand within the competition period if you hadn't had prior exposure.

(i) One possible way is to emulate the nested loop

```
for x_1 in range(n):
    for x_2 in range(x_1 + 1, n):
        ...
        for x_k in range(x_{k-1} + 1, n):
            read(tape[x_k])
        for x_k in reversed(range(x_{k-1} + 1, n)):
            read(tape[x_k])
```

For example, for $k = 4$ this sequence of nested loops is

```
for x_1 in range(0, n):
    for x_2 in range(x_1 + 1, n):
        for x_3 in range(x_2 + 1, n):
            for x_4 in range(x_3 + 1, n):
                read(tape[x_4])
            for x_4 in reversed(range(x_3 + 1, n)):
                read(tape[x_4])
```

The parameters for a Turing machine simulating this loop could be as follows:

- Set of symbols Γ: It suffices to have three symbols, with $\Gamma = \{0, 1, b\}$. Here b denotes the blank symbol.
- Set of states Q: We have a set of states called STARTLOOP_i for every $1 \leq i \leq k$, which help us start the iteration of the ith nested loop. We also have a set of states called ENDLOOP_i that will help when a loop finishes and control is returned to the parent loop. Finally, we also have a HALT state.
- The initial state q_0 of the machine is STARTLOOP_1.
- The transition function, which is defined as a function $\delta : (Q \setminus \{\text{HALT}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ can be defined as follows:

- If $i \leq n-1$, then $(\text{STARTLOOP}_i, 1) \to (\text{STARTLOOP}_{i+1}, 0, R)$
- $(\text{STARTLOOP}_n, 1) \to (\text{STARTLOOP}_n, 1, R)$
- For all $2 \leq i \leq n$, $(\text{STARTLOOP}_i, b) \to (\text{ENDLOOP}_i, b, L)$
- $(\text{ENDLOOP}_i, 1) \to (\text{ENDLOOP}_i, 1, L)$
- For all $2 \leq i \leq n$, $(\text{ENDLOOP}_i, 0) \to (\text{STARTLOOP}_{i-1}, 1, R)$
- All other (pairs of) inputs to the function lead to the HALT state. In particular, if the state is STARTLOOP_1 and the symbol that is read is the blank symbol b, we halt.

The number of steps in this computation is dominated by a term proportional to

$$\sum_{x_1=0}^{n} \sum_{x_2=x_1+1}^{n} \cdots \sum_{x_k=x_{k+1}+1}^{n} 1 = \sum_{x_1<x_2<\cdots<x_k} 1 = \Theta(n^k)$$

There is a nice Turing machine simulator at `http://www.morphett.info/turing` where you can simulate this Turing machine with the code attached in `turing.txt`, which is for the case $k = 4$. You can verify that this halts in roughly $\Theta(n^4)$ via a doubling hypothesis. For example, we can see that on an input of 16 ones it halts in ~ 6400 steps, while on an input of 32 ones it halts in ~ 94000 steps, which is roughly sixteen times more steps.

Scoring Notes: A common mistake was to depict a turing machine that terminated in $\Theta(n^k)$ steps, but had a number of states dependent on n. Such a Turing machine would halt in $\Theta(n^k)$ steps only for one value of n. For a function $f(n)$ to be time-constructible, we must show the existence of a single Turing machine M that halts after $f(n)$ steps on an input of n ones, *for every* n large enough. As such, a correct answer can't have properties of the Turing machine depending on n. We awarded four points if your solution ran into this problem, but was otherwise correct.

We also deducted one point from submissions that did not explicitly showcase Γ, Q, q_0, F, and δ. The reason for this is because actually describing these five parameters explicitly requires some attention, and doesn't follow completely trivially from a description of what the Turing machine does.

(ii) The following proof does the trick:

Suppose that P = NP. Then there exists an algorithm running in polynomially many steps, say in $O(n^k)$ steps that solves 3-SAT. Then to solve any problem $L \in$ NP, we may simply reduce L to an instance of 3-SAT (indeed, Cook's Theorem guarantees this is possible). Let $p(n)$ be an upper bound for the number of verification steps to verify any problem in NP. By the assumption, $p(n) = \mathcal{O}(n^a)$. By Cook's theorem, a reduction to 3-SAT can be done with

only a polynomial number of steps of additional computation — indeed, with $\mathcal{O}(p(n)^4)$ additional steps — and increases the input size at most polynomially — indeed, to some size that is $\mathcal{O}(p(n)^4)$. We may solve this modified instance of 3-SAT in $\mathcal{O}(p(n)^{4k})$ steps, as the size of the modified instance is $\mathcal{O}(p(n)^4)$. Altogether, this reduction will solve the problem in $\mathcal{O}(p(n)^{4k} + p(n)^3) = \mathcal{O}(n^{\max(4ka,4a)})$ time. But this means that every problem in NP must be solvable in $\mathcal{O}(n^{\max(4ka,4a)})$ time, so this must imply that

$$\text{DTIME}(n^{\max(4ka,4a)}) = \text{DTIME}(n^{\max(4ka,4a)+1})$$

which contradicts the Time-Hierarchy Theorem. Thus, P $\neq$ NP. $\square$

Of course, the premise is false, so we'll have to find some other way to solve P vs. NP...

Plaudits:

- Shout-out to Christopher Ye '21 and Byron Chin '21, who scored an impressive 14/15 on this problem and were the only team to reproduce our proof in part (ii).
- Allison Qi '24 and Michael Tang '24, who, despite not having a completely correct solution for this problem, had the right ideas in their proofs. As freshmen without the same exposure to the theory of computation as your upperclassmates, it's very impressive that you were able to parse through all the information we threw at you and propose reasonable solutions to both parts!