(1) For those who like numerics: Prove the final \( t' = O(t \log n) \) estimate in the \( k \)-tapes-to-2 simulation. Define \( J_1 \) to be the 4-step path \( 0 \to 1 \to 0 \to -1 \to 0 \). For \( k \geq 2 \), define \( J_k \) to be \( J_{k-1} \to 1 \to 2 \to \cdots \to 2^k - 1 \to 2^k - 2 \to \cdots \to 1 \to 0 \to -1 \to \cdots \to -2 \to -3 \to -2 \to -1 \to 0 \to 1 \to 0 \to -1 \to 0 \).

Note that \( J_{k-1} \) appears twice, and you should substitute it into this definition recursively to make one long path. Or you can picture the process iteratively going forward:

\[
J_2 = 0 \to 1 \to 0 \to -1 \to 0 \to 1 \to 2 \to 3 \to 2 \to 1 \to 0 \to -1 \to -2 \to -3 \to -2 \to -1 \to 0 \to 1 \to 0 \to -1 \to 0.
\]

\[
J_3 = 0 \to 1 \to 0 \to -1 \to 0 \to 1 \to 2 \to 3 \to 2 \to 1 \to 0 \to -1 \to -2 \to -3 \to -2 \to -1 \to 0 \to 1 \to 0 \to -1 \to 0 \to 1 \to 2 \to 3 \to 2 \to 1 \to 0 \to -1 \to -2 \to -3 \to -2 \to -1 \to 0 \to 1 \to 0 \to -1 \to 0.
\]

What makes this a little tricky is that you don’t immediately get \( t' \) as a function of \( t \). Instead you get both as functions of \( k \): \( t' \) is the number of steps in \( J_k \), and \( t \) is (proportional to) the number of times the subsequence \( 0 \to 1 \to 0 \to -1 \to 0 \) appears. Then combine the equations so \( k \) goes away and you can estimate \( t' \) in terms of \( t \). Be precise enough so that you can give the constant factor in the “\( O \).”

You are welcome to simplify by dropping the negative-number portions so everything is halved—this should not change the constant in the \( O \)-notation. Finish by summarizing how this allows a Turing machine \( M \) running in time \( t(n) \) first to be simulated by an oblivious two-worktape TM \( M' \) running in time \( t'(n) = O(t(n) \log t(n)) \), and then by Boolean circuits \( C_n \) of size \( O(t(n) \log t(n)) \).

**Answer:** As a function of \( k \), the number \( t \) is proportional to \( 2^k \). The number \( t' \) as a function of \( k \) obeys the recurrence relation

\[
t'(k) = 2t'(k - 1) + 2 \cdot 2^k,
\]

because you count \( t'(k - 1) \) steps through the \( (k - 1) \)st jag at the halfway point, then up through the middle part of the \( k \)th jag is an exact repeat, but the outer part of the jag adds \( 2^k \) steps at each end. Expressed in terms of \( t \), this recurrence is

\[
t'(k) = 2t'(k - 1) + \Theta(t).
\]

This is the same recurrence as for mergesort and other two-branch “divide-and-conquer” recursions where the recursion steps down by half. The solution is \( t' = \Theta(t \log t) \) and that is what we needed to prove. To be more precise, if we write \( N = 2^k \) in place of \( t \) then we can re-index the recursion as

\[
t'(N) = 2t'(\frac{N}{2}) + 2N.
\]

Try \( t'(N) = aN \log_2 N \); then we solve

\[
\begin{align*}
aN \log_2 N &= 2a \frac{N}{2} \log_2 (\frac{N}{2}) + 2N \\
aN \log_2 N &= aN (\log_2 N - 1) + 2N \\
0 &= -aN + 2N \\
a &= 2.
\end{align*}
\]
So the constant in the $O(t \log t)$ is a reasonable one, just 2. This is the running time of the oblivious machine whose main head follows the jags—and whose head on the second tape (which is used just to copy blocks between the upper and lower tracks) follows a similar pattern. The predictable monotony of the pattern saves needing a whole row of circuitry for each step in the simulation of a general TM in section 3 of ALR chapter 27, so the circuit size is $O(t'(n)) = O(t(n) \log t(n))$.

(2) For those who like (explaining away) apparent paradoxes: Given any real number $c \geq 1$, let $\mathsf{DQ}_c$ abbreviate $\mathsf{DTIME}[\tilde{O}(n^c)]$. The case $c = 1$ is deterministic quasilinear time, which the “scholia” to the Cook-Levin theorem called $\mathsf{DQL}$.

- Show that whenever $c < d$, $\mathsf{DQ}_c$ is properly contained in $\mathsf{DQ}_d$. (This has essentially the same proof as (a) of problem (2) on the homework’s written part.) This means there is a language $A_d$ in $\mathsf{DQ}_d$ that is not in $\mathsf{DQ}_c$.
- From the fact that there are uncountably many real numbers, conclude that there are uncountably many different complexity classes of the form $\mathsf{DQ}_d$.
- But wait—there are only countably many decidable languages in all. How can there be uncountably many languages $A_d$??

If this doesn’t perplex you, don’t choose this. If this does perplex you but you resolve it, fine. If this perplexes you and you stay perplexed, you can contact me for a hint.

Answer: The main answer key includes the calculation in the first part, and it does follow that there are uncountably many complexity classes $\mathsf{DQ}_d$. For every pair $c, d$ we also get a language $A_d$ in $\mathsf{DQ}_d$ that is not in $\mathsf{DQ}_c$, and maybe it is more helpful to call it $A_{c,d}$ since $c$ is involved in the proof too. The paradox would come in if $A_{c,d}$ were unique to either $c$ or $d$, let alone being unique to the pair. There are two levels of answer to this, one special to complexity theory and the other not:

1. When we did the proof, we actually used the existence of rational numbers $q, r$ such that $c < q < r < d$. We did this because the function $t_2(n) = n^d$ is not “reasonable” when the $d$ is an uncomputable number—in particular, it is not fully time constructible. Whereas, $r$ being rational allows us to take $t_2(n) = n^r$ instead, which complies with this requirement and allows the actual construction of the diagonalizing machine and its language $A_r$ (or rather, $A_{q,r}$) to go through.

But this means we can do all the diagonalizing for all these uncountably many classes with languages of the form $A_{q,r}$ where $q$ and $r$ are rational numbers. There are only countably many pairs of rational numbers, so there is no contradiction to having only countably many languages.

2. Consider just the real numbers by themselves. For every real number $c$, define $\mathsf{DK}_c$ to be the set of rational numbers $p$ such that $p < c$. Then $\mathsf{DK}_c$ uniquely determines $c$, so there are uncountably many subsets of the rational numbers having this form. The subsets $\mathsf{DK}_c$ are analogous to the classes $\mathsf{DQ}_c$; the fact that there are uncountably many of them does not contradict the fact that there are only countably many rational numbers total. (The letters $\mathsf{DK}$ here stand for “Dedekind Cut” after the mathematician Richard Dedekind, who defined and studied them.)
This does illustrate the need for the time-constructibility requirement—there are other reasons, too.

(3) Show that the following problem is complete for co-NP under \( \leq_P \):

\text{ALL}_{\text{REG},n}

\text{INSTANCE: A regular expression } \alpha \text{ using only 0, 1, +, \cdot (no Kleene star), and a number } n.

\text{QUESTION: } \text{Does } \alpha \text{ match every string in } \{0, 1\}^n? 

For a hint, take a 3DNF formula \( \psi(x_1, \ldots, x_n) = T_1 \lor \cdots \lor T_m \). Then \( \psi \) is a tautology if and only if every assignment \( a \in \{0, 1\}^n \) satisfies one of the terms. (Or if you prefer, think of a 3CNF formula \( \phi(x_1, \ldots, x_n) = C_1 \land \cdots \land C_m \) and note that \( \phi \) is unsatisfiable if and only if every assignment \( a \in \{0, 1\}^n \) “unsatisfies” one of the clauses.) Show how to create \( \alpha \) from \( \psi \) (or from \( \phi \)) and give analysis to show that the reduction from 3TAUT (or from the complement of 3SAT) is correct. (For membership in co-NP, note that powering abbreviations like \( (0 + 1)^{17} \) are disallowed, or you can require \( n \) to be given as \( 0^n \) in unary notation.)

\text{Answer: } \text{The problem belongs to co-NP because the complementary problem belongs to NP: if there is a string } y \in \{0, 1\}^n \text{ that is not matched by } \alpha, \text{ then we can guess } y \text{ and verify that it does not match. The latter verification could be tricky, but there is a way we have discussed in lecture: We can quickly convert } \alpha \text{ into an equivalent NFA } N_\alpha \text{ and then simulate } N_\alpha \text{ on } y \text{ directly (not converting it to a DFA) by keeping track of possible states at each step. The latter simulation runs in polynomial time. For the reduction, let any 3DNF formula } \psi \text{ as above be given. For each term } T_j \text{ we create a “concatenation term” } \alpha_j \text{ of } n \text{ “components” as follows: Let the literals in } T_j \text{ be } \ell_{i_1} \land \ell_{i_2} \land \ell_{i_3} \text{ with } i_1 < i_2 < i_3. \text{ For all } i \notin \{i_1, i_2, i_3\}, \text{ use } (0 + 1) \text{ as the } i\text{th component. If } \ell_{i_1} = x_{i_1} \text{ then make component } i_1 \text{ be 1, else if } \ell_{i_1} = \bar{x}_{i_1} \text{ then make it be 0. Do likewise for } i_2 \text{ and } i_3. \text{ For example, if } n = 6 \text{ and } T_j = x_1 \land x_3 \land x_4 \text{ then } \alpha_j = 1(0 + 1)01(0 + 1)(0 + 1). \text{ Making } \alpha \text{ be the + of all the } \alpha_j \text{ completes the construction. For correctness, all one needs to say is that truth assignments in } \{0, 1\}^n \text{ are the same as binary strings } x \text{ that may-or-may-not match } \alpha, \text{ and } x \text{ matches } \alpha \text{ iff it matches some } \alpha_j, \text{ which is iff it satisfies } T_j, \text{ which means } x \text{ satisfies } \psi \text{ since } \psi \text{ is in DNF.}

The upshot is that even regular expressions having a simple structure and no *s can be difficult to analyze.

(4) Show that the following problem is NP-complete:

\text{BINARY LINEAR EQUATIONS}

\text{INSTANCE: A set } E_1, \ldots, E_m \text{ where each } E_j \text{ is a linear equation in three of the variables } x_1, \ldots, x_n.

\text{QUESTION: } \text{Is there a solution to the equations in which each } x_i \text{ is 1 or 0?}

Here is where using a reduction from EXACTLY ONE 3SAT (rather than from vanilla 3SAT) may come in especially handy. Note that the equations can have nonzero constant terms—this is the case when solving linear equations in the form \( Ax = b \) where \( A \) is an \( m \times n \) matrix, \( x = (x_1, \ldots, x_n) \) is the \( n \)-vector being solved for, and \( b = (b_1, \ldots, b_m) \) is the \( m \)-vector of constants in the equations. “Wait a second: \( Ax = b \) is solvable in polynomial time by methods taught in MTH309. Why doesn’t that prove NP = P?” Explain...
Answer: The problem is in NP because if we guess a solution \((a_1, \ldots, a_n)\) we can simply verify each little equation. For the reduction, if we have a clause \((\ell_i \lor \ell_j \lor \ell_k)\) then we want to rig a linear equation involving the variables \(x_i, x_j, x_k\) that holds when exactly one of the literals is true. If the literals are all positive, then the equation can be simply \(x_i + x_j + x_k = 1\). If they are all negative, then we want \(x_i + x_j + x_k = 2\) instead, because we need exactly one variable to be false to make its negated literal true, so the other two have to be true and that adds up to 2. The most interesting case may be a clause like \((x_i \lor \bar{x}_j \lor x_k)\) with one variable negated. Then we get \(x_i - x_j + x_k = 0\) with no constant term. For \((x_i \lor \bar{x}_j \lor \bar{x}_k)\) we get \(x_i + 1 - x_j + 1 - x_k = 1\), which simplifies to \(x_j + x_k - x_i = 1\). Then a binary solution to the equations is the same as an assignment that satisfies each clause exactly once.

The reason this does not imply \(P = NP\) is that the polynomial-time algorithm for solving linear system might only give you a fractional solution, not an integer one. For example, \(x_1 + x_2 = 2\) and \(x_1 - x_2 = 1\) have only the solution \((\frac{3}{2}, \frac{1}{2})\). We can force solutions to be binary by adding the equations \(x_1^2 - x_1 = 0\) through \(x_n^2 - x_n = 0\). But those extra equations make the whole system quadratic, not linear. It may seem like coming tantalizingly close to making \(P = NP\), but when you work with really big systems of equations, it seems less so.

(5) Let us revisit the palindrome language, specifically its subset \(P = \{v10^r1w : v, w \in \{0,1\}^*, w = v^R\}\) from the HW4 presentation option (D). Let \(M\) be a deterministic Turing machine with a read-only input tape such that \(L(M) = P\), \(M\) runs in time \(t(n)\), and \(M\) runs in space \(s(n)\). Prove that

\[
t(n)s(n) = \Omega(n^2),
\]

where \(n = 3r + 2\) is the whole length of the input as before. Note that there are two ways of doing no worse than this: One is to copy \(v\) to a tape, go to the end of the input tape, and compare \(v\) against \(w^R\) while going right-to-left on the input tape. This uses linear time but also linear space. At the other extreme is the idea of comparing bits of \(v\) and \(w\) one at a time, each time taking the \(k\)-th bit of \(v\) from the front and the \(k\)-th bit of \(w\) from the end. Without changing any input tape characters, this can be managed by calculating \(r\) and maintaining \(r, k,\) and a counter \(j\) going from 1 to \(k\), each in binary notation on a separate tape. This uses \(O(\log n)\) space but quadratic time, giving \(t(n)s(n) = O(n^2\log n)\). We can make the time-space product exactly quadratic by comparing \(v\) in log \(n\)-sized chunks at a time, rather than single bits. There are ways in-between, such as using \(O(\sqrt{n})\) space to compare \(\sqrt{n}\)-sized chunks at a time, but this takes order-\(n^{1.5}\) time, giving \(t(n)s(n) = \Omega(n^2)\) again.

To prove \(t(n)s(n) = \Omega(n^2)\) in general, the trick is that once an input length \(n\) is given, we can (wlog.) mark off the \(s(n)\) cells that \(M\) is allowed to use in advance. Then we can consider every possible contents \(u \in \Gamma^{s(n)}\) of these cells. For any fixed \(n\), we can imagine \(M\) instead as a single-tape TM \(M'\) with state set \(Q' = Q \times \Gamma^{s(n)}\) allowing for every possible “state” of the worktapes too. Redo the calculation of presentation option (D) with \(Q'\) in place of \(Q\). You may freely use the answer key of (D). You may use other sources as well, but should make your presentation follow this outline and make the calculation look like that of (D).

Answer: In brief, we get the same inequality \(|Q'|^k \geq 2^r\) as in (D) on the HW4 presentations. Substituting gives

\[
|Q|^k|\Gamma|^{ks(n)} \geq 2^r.
\]
Taking logs of both sides gives $k \log |Q| + ks(n) \log |\Gamma| \geq r$. Now we have that the time is at least $kr$, i.e., $t(n) \geq kr$, so $k \leq \frac{t(n)}{r}$. Substituting again thus gives us

$$\frac{t(n)}{r} \log |Q| + \frac{t(n)s(n)}{r} \geq r,$$

so $t(n) \log |Q| + t(n)s(n) \log |\Gamma| \geq r^2$. Since $\log |Q|$ and $\log |\Gamma|$ are just constants, this implies $t(n)s(n) = \Omega(r^2)$, which $= \Omega(n^2)$ per the terms of the problem.

The point is that this is true for any Turing machine, not just a 1-tape Turing machine.

(6) With reference to (5), now suppose we allow Turing machines $M''$ a source of randomness. Specifically, we can allow them to compute hash functions $h_\rho : \{0,1\}^r \rightarrow \{0,1\}^\ell$ from random seeds $\rho$ of length $O(\ell)$ such that for all distinct $v, w \in \{0,1\}^r$,

$$\Pr[ h_\rho(v) = h_\rho(w) ] = \frac{1}{2^\ell}.$$  

Suppose this holds for all $\ell \leq r$ and that $h_\rho(v)$ can always be computed in $O(r)$ time and $O(\ell)$ space. Show that now we can arrange for $M''$ to decide whether a given string $v10^r1w$ belongs to $P$ while beating the time-space tradeoff in (5), on pain of a slight $\frac{1}{2^\ell}$ chance of erroneously accepting when $v10^r1w \not\in P$.

[This problem can be done independent of (5) but makes a nice combo with it. This is reminiscent of how many password systems are implemented. The system does not store your whole password $w$ but rather a hash of it, $h(w)$. There may be a chance that an intruder can gain access by hitting on a string $v \neq w$ such that $h(v) = h(w)$, but by choosing $\ell$ wisely—maybe $\ell > r$, especially if your password is short—it can be made minuscule.]

Answer: Again in brief, we can get such a family of hash functions $h$ for any desired range $\{0,1\}^\ell$ by taking random $\ell \times r$ matrices $H$ and using arithmetic modulo 2. (There is also a more succinct way using just $O(r + \ell)$ random bits rather than $r\ell$ bits.) One helpful fact is that computing $Hv = v'$ uses each bit of the matrix $H$ exactly once, so we can generate the (pseudo-)random bits on the fly without needing space to store $H$ at all. We can thus compute $v' = Hv$ and $w' = Hw$ in $O(\ell)$ space and store them in $O(\ell)$ space. This does take $O(r\ell)$ time (again, there are ways to get $O(r + \ell)$ time) but if $\ell = 3 \log n$, say, then this is still quasi-linear time—with logarithmic space—and the chance of erroneously accepting when $w \neq v$ is only about $2^{-\ell} = 1/n^3$. Thus we can beat the time-space tradeoff handily, at the expense of only a relatively small loss of certainty.