1 \( \text{MIN} - \text{FORMULA} \in NP_{\text{SAT}} \)

This is very similar to a problem you had in problem set 5, but with a slight oracular twist. As before, it is easy to see how, given a formula \( \phi \), we can non-deterministically guess a formula \( \psi \) such that \( |\psi| < |\phi| \). However, we’d like to show that there exists some \( \psi \) that is equivalent to \( \phi \), and it is not obvious that equivalence is in NP.

Let \( \text{EQUIV} = \{\langle \phi, \psi \rangle \mid \phi \text{ and } \psi \text{ are equivalent formulae} \} \), and let \( \overline{\text{EQUIV}} \) be its complement. \( \overline{\text{EQUIV}} \) is in NP, because we can non-deterministically guess the assignment that yields a different answer for both formulae. Since SAT is NP-complete, by the Cook-Levin theorem we can in polynomial time reduce \( \overline{\text{EQUIV}} \) to SAT using some TM \( R \). This gives us a way to solve \( \text{MIN} - \text{FORMULA} \) in non-deterministic polynomial time with a SAT oracle:

On input \( \phi \):

1. Guess formula \( \psi \) where \( |\psi| < |\phi| \)
2. Use \( R \) on \( \langle \phi, \psi \rangle \) to produce SAT input \( \varphi \)
3. Run the SAT oracle on \( \varphi \)
4. Accept if it rejects, otherwise reject

2 \( P^{\text{TQBF}} = NP^{\text{TQBF}} \)

First, we should note that we already know that \( P^{\text{TQBF}} \subseteq NP^{\text{TQBF}} \), because an NTM with a TQBF oracle can simply not take any non-deterministic steps and it will simulate a TM with a TQBF oracle. So it remains to show that \( NP^{\text{TQBF}} \subseteq P^{\text{TQBF}} \), since that would imply that they are equal.

2.1 \( NP^{TQBF} \subseteq \text{NPSPACE} \)

Let \( A \) be any language in \( NP^{TQBF} \). Then there is some NTM \( M \) that decides it using a polynomial amount of time and some polynomial number of calls to TQBF. We can simulate \( M \) with an NTM using polynomial space but without a TQBF oracle by simply solving TQBF whenever \( M \) would have called the oracle. This takes at most polynomial space for each call, and since there are only a polynomial number of such calls, even if we did not reuse space we could simulate \( M \) in polynomial space. Thus, \( A \in \text{NPSPACE} \), so \( NP^{TQBF} \subseteq \text{NPSPACE} \).

2.2 \( \text{NPSPACE} = \text{PSPACE} \)

Here we can just use Savitch’s Theorem.
2.3 \( PSPACE \subseteq P^{TQBF} \)

Now say we have some TM \( M \) that uses polynomial space to decide a language \( A \in PSPACE \). We can simulate \( M \) in polynomial time with a TQBF oracle to show the desired result. Remember that TQBF is \( PSPACE \)-complete, so we can reduce \( A \) to TQBF in polynomial time. Then we can run our TQBF oracle on the reduced input and get our answer.

Putting it all together: 
\[
NP^{TQBF} \subseteq NPSPACE = PSPACE \subseteq P^{TQBF} \Rightarrow NP^{TQBF} \subseteq P^{TQBF}
\]

3 Probabilistic Algorithms

3.1 Probabilistic Turing Machines

A probabilistic TM is a non-deterministic TM where each non-deterministic step (called a ”coin-flip step”) has two legal next moves that occur with equal probability \( 1/2 \).

On input \( w \), a probabilistic TM \( M \) will go down one branch of computation in the tree of possible computations formed by the different coin flips. But just like NTMs, when we analyze the space and time complexity of a probabilistic TM, we look at the branches with the highest space and time complexity respectively; in other words, we look at the worst case.

We define the probability that \( M \) goes down branch \( b \) as \( \Pr[b] = 2^{-k} \), where \( k \) is the number of coin-flip steps on branch \( b \). We define the probability that \( M \) accepts \( w \) as

\[
Pr[M \text{ accepts } w] = \sum_{b \text{ is an accepting branch}} \Pr[b]
\]

This is just the probability that \( M \) on \( w \) takes some accepting branch as it computes and makes its coin flips. A probabilistic TM will either accept or reject, so we define probability of rejection similarly: \( \Pr[M \text{ rejects } w] = 1 - \Pr[M \text{ accepts } w] \).

But how do we rate how well a probabilistic TM does on an input? That’s where error probability comes in.

3.2 Error Probability

A probabilistic TM must still accept strings in its language and reject strings out of its language, but it is allowed to be wrong with some probability. A probabilistic TM decides language \( A \) with error probability \( 0 \leq \epsilon < 1/2 \) if for every string \( w \in A \), it accepts with probability at least \( 1 - \epsilon \), and for every string \( w \not\in A \), it rejects with probability \( 1 - \epsilon \).

The error probability does not need to be a constant; it can also be a function of \( n \), such as \( 1/n \) or \( 2^{-n} \).

3.3 The Complexity Class BPP

BPP is the set of languages decidable by probabilistic TMs running in polynomial time with error probability \( 1/3 \). Essentially, it’s the set of languages we can decide efficiently using randomness, but with a small amount of error. You might find the value \( 1/3 \) to be a little strange; in reality, we could have chosen any positive constant less than \( 1/2 \), due to the following lemma.
3.4 Amplification Lemma

The basic idea here is that if we have a machine with some error probability, say 1/3, then we can run it a few times and take the majority vote, and we’re more likely to get the right answer than we were before. You can think of it like an unfair coin that gives heads 2/3 of the time: if you flip the coin 1000 times and take the majority answer, you’d be pretty surprised if you decided tails.

If we are given a probabilistic TM $M_1$ with error probability $\epsilon < 1/2$, then for any polynomial $p(n)$ we can make a probabilistic TM $M_2$ running in polynomial time with error probability $2^{-p(n)}$ like so:

On input $x$:

1. Run $M_1$ on $x$, $2k$ times (see below)
2. If most runs of $M_1$ accept, accept. Otherwise, reject

The tricky part of this lemma is picking $k$ so that we get the error probability we want. The approach we’ll take is to figure out what the probability of $M_2$ being wrong is (in terms of $k$), then setting $k$ so that we keep that probability below $2^{-p(n)}$.

First, we’ll calculate the probability we’ll get a bad sequence $S$ of $2k$ runs of $M_1$ on $x$, i.e. one where there are more wrong answers than right answers. Say the number of correct answers in the sequence is $c$ and the number of wrong answers is $w$, so $c + w = 2k$. Then we have $\Pr[\text{bad sequence } S] \leq \epsilon_x^c (1 - \epsilon_x)^w$, where $\epsilon_x$ is the probability $M_1$ is wrong on $x$.

We can rewrite this as $\Pr[\text{bad sequence } S] \leq \epsilon_x^c (1 - \epsilon_x)^w$, because this is a bad sequence so $c \leq w$. We know $\epsilon_x \leq \epsilon$ and $\epsilon_x (1 - \epsilon_x) \leq \epsilon (1 - \epsilon)$, so $\epsilon_x^c (1 - \epsilon_x)^w \leq \epsilon^c (1 - \epsilon)^w$, which means $\Pr[\text{bad sequence } S] \leq \epsilon^c (1 - \epsilon)^w$. Since $k \leq w$ and $\epsilon < 1 - \epsilon$, $\epsilon^w (1 - \epsilon)^w \leq (1 - \epsilon)^k$, so we finally have $\Pr[\text{bad sequence } S] \leq \epsilon^k (1 - \epsilon)^k$.

We know that $\Pr[\text{M_2 is wrong on x}] = \sum_{\text{bad sequences } S} \Pr[S]$. No more than all of the $2^{2k}$ possible sequences could be bad, so we also know that $\Pr[\text{M_2 is wrong on x}] \leq 2^{2k} \epsilon^k (1 - \epsilon)^k = (4\epsilon(1 - \epsilon))^k$.

Now, say we want the probability to at most $2^{-p(n)}$. Then we may set $(4\epsilon(1 - \epsilon))^k \leq 2^{-p(n)}$. \(\Rightarrow k \log_2 (4\epsilon(1 - \epsilon)) \geq -p(n)\) (Taking log of both sides flips the sign because $(4\epsilon(1 - \epsilon)) < 1)

$\Rightarrow k \geq p(n)/(\log_2 (4\epsilon(1 - \epsilon)))$

So we can choose $k$ so that $M_2$ runs in polynomial time and is wrong with probability at most $2^{-p(n)}$.

4 Branching Program Practice

4.1 XOR Program

Given two Boolean variables $x_1$ and $x_2$, create a read-once branching program that computes $x_1 \oplus x_2$, which is 1 when $x_1 \neq x_2$ and is 0 otherwise. Remember, read-once means that no path through the branching program can have the same variable twice.

Solution:
4.2 OR Program

Given three Boolean variables $x_1$, $x_2$ and $x_3$, create a read-once branching program that computes $x_1 \lor x_2 \lor x_3$.

Solution: