PSPACE

(from Sipser Definition 8.6) $PSPACE = \bigcup_k SPACE(n^k)$. $NPSPACE = \bigcup_k NSPACE(n^k)$.
Since $NSPACE(f(n)) \subseteq SPACE(f^2(n))$ (from last worksheet; "Savitch's Theorem"), $PSPACE = NPSPACE$.
(from Sipser Definition 8.8) A language B is PSPACE-hard if every A in PSPACE is polynomial time reducible to B. If B is also in PSPACE, it is PSPACE-complete.

a It might seem more natural to use polynomial-*space* reductions now that we're discussing space complexity. But prove that if we were to modify the above definition to use poly-space reductions, then *every* language in PSPACE (except two edge cases) would be PSPACE-complete. (Then fill in the analogous claim for time complexity, i.e. for what X would we say "Using poly-time reductions, every language in X (except two edge cases) is X-complete.")

Solution: First, the exceptions: for *all* our different kinds of reductions that have a requirement looking like $f(w) \in A \leftrightarrow w \in B$, $\emptyset$ and Σ^* each reduce to themselves but nothing else reduces to them. This is because e.g. for any $w \in B$, there is *no* possible function which will give us $f(w) \in \emptyset$.

Now if we use poly-space reductions above, then *every* language in PSPACE (other than our two exceptions) would be PSPACE-complete . Proof: Let L be in PSPACE (with $L \neq \emptyset, L \neq \Sigma^*$). We will show it is PSPACE-complete by reducing an arbitrary $A \in PSPACE$ to L. Let M be a poly-space decider for A. Let w be in L and s not be in L. Then the reduction is computed by the following TM: "On input r, run M on r. If it accepts *return* w, otherwise *return* s." *(Make sure you see which specific sentence of this proof would fall apart if we had failed to exclude our two exceptions.)*

b Show that PSPACE is closed under star. ($L^* = \{x_1 x_2 \cdots x_k \mid k \geq 0$ and each $x_i \in L\}$. Note that $|x_i|$ is not necessarily 1.)

Solution:

Let L be in PSPACE. Then it is decided by some $O(n^k)$-space TM M_L. Now we decide L^* as follows:

$R =$ "On input w:

- For each way of dividing w into non-empty substrings:
  - Run M_L on each substring. *Accept* if all are accepted.
- *Reject*"

This successfully decides L^* since there is a way of dividing w into substrings in L exactly when $w \in L^*$, and R tries all ways so it will find it if it exists. The number of ways of dividing w is exponentially large, but no information other than the $O(n)$-sized loop counter needs to be stored in between each loop iteration, so this runs in the same space bound as M_L.

c Show that NP $\subseteq$ PSPACE. Then show that any PSPACE-hard language is also NP-hard.

Solution:

Let L be in NP. Then for some k there is a n^k-time verifier for L. Now we decide L in poly-space by running this verifier on our input using every certificate of length up to n^k sequentially. Since we only need to store one certificate on the tape at a time, this takes n^k space, so $L \in PSPACE$.

Now we will show that PSPACE-hard languages are also NP-hard. Let A be PSPACE-hard and let B be in NP. It suffices to now show that $B \leq_P A$. Since $NP \subseteq PSPACE$, $B \in PSPACE$. So since A is PSPACE-hard, a poly-time reduction from B to A exists.

d We will now consider a harder version of SAT by adding quantifiers to our formulas. We will work with "fully quantified" formulas, meaning that there are no "free" variables, i.e. every variable must be in the scope of a corresponding quantifier (so e.g. $\forall x_1 \exists x_2, [x_1 \wedge x_2]$ is fine but $\forall x_1, [x_1 \wedge x_2]$ is not).

i For simplicity we will also only work with formulas in "prenex normal form", i.e. where all quantifiers appear at the beginning of the formula (and the whole formula is in their scope) - for example, $\forall x_1 \exists x_2, [x_1 \wedge x_2]$, but not $\forall x_1, [x_1 \wedge (\exists x_2, x_2)]$ or $\exists x_2, [(\forall x_1, x_1) \wedge x_2]$. Describe how to convert any fully quantified formula that is *not* in prenex normal form into one that is.

Solution: First, ensure all variables have distinct names by providing fresh names to any duplicates (or simply to all of them), e.g. convert $(\forall x, P(x)) \wedge (\exists x, Q(x))$ into $(\forall v_1, P(v_1)) \wedge (\exists v_2, Q(v_2))$. After that, we can just pull all quantifiers to the front while maintaining their current left-to-right order, and flipping $\forall$ to $\exists$ and vice versa if there were an odd number of enclosing negations. For example, $Q \wedge \forall x, \neg[(\exists y, P) \vee (\forall z, R)]$ becomes $\forall x, \forall y, \exists z, (Q \wedge \neg[P \vee R])$. (To see that this works, the main thing we would need to justify is that individual quantifiers can always 'bubble up' by one level repeatedly, e.g. $(Q \wedge \forall x P) \equiv \forall x (Q \wedge P)$ as long as Q does not use "x" as a variable, and that bubbling up through a negation flips the quantifier while $\wedge$ and $\vee$ do not.)

ii Define $TQBF = \{\langle \phi \rangle \mid \phi$ is a True fully quantified Boolean formula (in prenex normal form)$\}$. Prove that $TQBF \in PSPACE$.

Solution: See Sipser Theorem 8.9 (p339)

e We will define a two-player game played using (prenex normal form) quantified boolean formulas. For each quantifier in order, if it is a $\forall$ then Player A chooses the value of the variable, and if it is a $\exists$ then Player E chooses the value of the variable. Once all variables have been assigned in this way, we see whether the unquantified part of the formula is true or false using this variable assignment. Player A wins on False, Player E on True. For example, if $\phi = \forall x_1 \exists x_2, [x_1 \wedge x_2]$, then Player A would go first and might choose $x_1 = F$, Player E would go next and might choose $x_2 = T$, and then Player A would win because $F \wedge T$ is False. Let $FORMULA\text{–}GAME = \{\langle \phi \rangle \mid$ Player E has a winning strategy in the formula game associated with $\phi\}$. (A player has a winning strategy if they will win so long as they play optimally.)

Prove that $TQBF \leq_P FORMULA\text{–}GAME$. (And then, though we won't prove this, $TQBF$ is PSPACE-complete so $FORMULA\text{–}GAME$ is too.)

Solution: See Sipser Theorem 8.11 (p343)

f A *ladder* is a sequence of strings $s_1, s_2, \cdots, s_k$, wherein every string differs from the preceding one by exactly one character. For example, the following is a ladder of English words, starting with "head" and ending with "free": head, hear, near, fear, bear, beer, deer, deed, feed, feet, fret, free. Let $LADDER_{DFA} = \{\langle M, s, t \rangle \mid M$ is a DFA and there are strings in $L(M)$ which can be used to construct a ladder that starts with s and ends with $t\}$. Show that $LADDER_{DFA}$ is in PSPACE. *(Hint: PSPACE = NPSPACE)*

Solution:

Note that there are $|\Sigma|^n$ strings of length n. A minimal ladder will have no repeats, so if a ladder exists at all then there exists a ladder of length at most $|\Sigma|^n$. We can thus decide $LADDER_{DFA}$ with an NTM by repeatedly guessing a next word and then checking if the DFA accepts that word, rejecting if the DFA ever rejects or if we make more than $|\Sigma|^n$ guesses, and accepting if we make it to the target word. Checking if a DFA accepts a string can be done in poly space, and the only storage needed between checks is a linear-sized loop counter ($\log(|\Sigma|^n) = O(n)$).

g Using our normal method of counting time and space complexity, there's not much point in talking about sub-linear complexity, since it takes n time and space just to read the whole input. Yet there are

other models where we can talk about sub-linear complexity - for example, we normally consider binary search on real computers to take $O(log(n))$ time. What are two key features of that example which allow us to get sub-linear complexity? Design a new way of computing space complexity (which may involve using a slightly different model than a normal TM) which will allow us to meaningfully study sub-linear space complexity.

Solution:

The two key features in the example are

- Preconditions on the input: in particular, binary search assumes that the input list is already sorted. This assumption means it is not necessary to read the whole input.
- A RAM (Random Access Memory) model of computation. This allows us to repeatedly jump to the middle of some range in a single computation step, rather than scanning linearly over the whole list.

Note that neither of these would be sufficient on its own - random access doesn't help you efficiently scan an *unsorted* list, and knowing a list is sorted doesn't help you if you still have to linearly scan it.

For the new way of computing space complexity, see R2TMs and WSPACE on this week's homework.