Problem 6.3.3a  Show that \([\log n]\) is fully space constructible.

Solution.  Let the input tape store \(n\) 1s: \(B11...111B\). Let the work tape look like \(BBB\). We will implement the conversion from unary to binary - with the least significant digit at the left end. The first instruction is

\[\delta(s, B, B) = (q_1, B, 0, R, S).\]

The rest of the instructions perform the addition. Unfortunately, every time we add a 1 from the input, we must restart the addition from the leftmost cell in the work-tape, which means we will have to see the \(B\) at the left of the partial sum before moving back to the right. Since \([\log(2^k)] = \log 2^k = k\), this would exceed by 1 the space we are restricted to live in. We can solve the problem by adding two symbols to the tape: \(a\) and \(b\), used, respectively, to "simulate" 0 and 1. They will be used only in the leftmost cell position and will be used to stop the leftward movement of the head so it does not need an extra cell. Instructions:

\[
\begin{align*}
\delta(s, B, B) &= (q_1, B, a, R, S) \\
\delta(q_1, 1, a) &= (q_1, 1, b, R, S); \text{ adding 1 to leftmost 0} \\
\delta(q_1, 1, b) &= (q_2, 1, a, R, R); \text{ adding 1 to leftmost 1 - carry} \\
\delta(q_1, B, a) &= (b, B, 0, S, S); \text{ done} \\
\delta(q_1, B, b) &= (h, B, 1, S, S); \text{ done} \\
\delta(q_2, 1, B) &= (q_3, 1, 1, S, L); \text{ new 1 and move back to beginning} \\
\delta(q_2, B, B) &= (q_3, B, 1, S, L); \text{ new 1 and move back to beginning} \\
\delta(q_2, 1, 0) &= (q_3, 1, 1, S, L); \text{ new 1 and move back to beginning} \\
\delta(q_2, B, 0) &= (q_3, B, 1, S, L); \text{ new 1 and move back to beginning} \\
\delta(q_2, 1, 1) &= (q_2, 1, 0, S, R); \text{ still a carry} \\
\delta(q_2, B, 1) &= (q_2, B, 0, S, R); \text{ still a carry} \\
\delta(q_3, 1, 0) &= (q_3, 1, 0, S, L); \text{ keep moving left} \\
\delta(q_3, 1, 1) &= (q_3, 1, 1, S, L); \text{ keep moving left} \\
\delta(q_3, 1, a) &= (q_1, 1, b, R, S); \text{ adding 1 to leftmost 0} \\
\delta(q_3, 1, b) &= (q_2, 1, a, R, R); \text{ adding 1 to leftmost 1 and carry} \\
\delta(q_3, B, 0) &= (q_3, B, 0, S, L); \text{ keep moving left} \\
\delta(q_3, B, 1) &= (q_3, B, 1, S, L); \text{ keep moving left} \\
\delta(q_3, B, a) &= (q_1, B, a, S, S); \text{ adding nothing to leftmost 0} \\
\delta(q_3, B, b) &= (q_1, B, b, S, S); \text{ adding nothing to leftmost 1}
\end{align*}
\]
Problem 6.3.4a. Show that if \( t(n) \) is fully time-constructible, \( \text{DTIME}(t(n)) \subseteq \text{DSPACE}(t(n)) \).

Solution. We must show that, for every language in \( \text{DTIME}(t(n)) \) there exists a Turing Machine \( M' \) that decides that language in space \( \leq t(n) \). The naive approach would go as follows: since \( t(n) \) is fully time constructible, we can set up a TM that, on the work-tape, will mark off - always moving to the right from the beginning cell - \( t(n) \) cells. Mark the next cell to the right with a special symbol. Then simulate the original machine on this space. If you encounter the special symbol, reject.

This almost works, except that Theorem 6.8 points out that \( \text{Space}_M(x) \leq \text{Time}_M(x) + 1 \): there is that cell on the worktape where the read-write head rested before any computing activity took place. If you check the process above, you will notice we have "seen" \( t(n) + 1 \) cells, and not \( t(n) \) before using another one for the special symbol. We can solve the problem by doing two things: at the end of the marking phase, we mark with the special symbol the last cell reached by the TM that had the task of realizing the function \( t(n) \). Furthermore, we take all instructions of the original TM deciding the language in time \( t(n) \) that lead to the halt state: \( \delta(q_1, a_1, \ldots, a_k) = (h, a_{i_1}, \ldots, a_{i_k}, m_1, \ldots, m_k) \), where each \( m_i \in \{L, S, R\} \) and replace them by the instructions \( \delta(q_1, a_1, \ldots, a_k) = (h, a_{i_1}, \ldots, a_{i_k}, S, \ldots, S) \), making sure that only \( t(n) - 1 \) instructions result in movement. After all, we don’t need any further action after reaching the halt state. We simulate this variant of the machine, rejecting if we encounter the special symbol, and we are done: every accepting computation uses no more than \( t(n) \) space for each string \( x, |x| = n \), every string that requires more space is rejected.

Problem 6.4.1a. Construct a multi-tape NTM to accept, in time \( t(n) = 2n \), the language:

\[
L_1 = \{a^i b a^{i+2} b \ldots b a^k \mid i_1, i_2, \ldots i_k \geq 0, k \geq 3, i_r = i_s = i_t \text{ for some } 1 \leq r < s < t \leq k\}.
\] (2)

Note: Example 6.22, followed by quite a few people, is not a good example. That algorithm uses a single tape, and the time is, quite clearly, not linear. Furthermore, we have to figure out how to choose, nondeterministically, exactly three blocks.

Solution. On tape 1 store the input; tape 2 is blank. We look at the following table of transitions:

\[
\begin{align*}
\delta(q_0, B, B) &= (q_0, B, B, R, S); \text{ skip this block} \\
\delta(q_0, B, B) &= (q_1, B, B, R, S); \text{ first block chosen - copy} \\
\delta(q_0, a, B) &= (q_0, a, B, R, S); \text{ skip this block} \\
\delta(q_0, b, B) &= (q_1, b, B, R, S); \text{ first block chosen - copy} \\
\delta(q_0, B, B) &= \text{FAIL}; \text{ reached end - nothing picked} \\
\delta(q_1, a, B) &= (q_1, a, a, R, R); \text{ copy first block} \\
\delta(q_1, b, B) &= (q_2, b, B, R, S); \text{ skip next block} \\
\delta(q_2, a, B) &= (q_2, a, B, R, S); \text{ keep skipping} \\
\delta(q_2, b, B) &= (q_2, b, B, R, S); \text{ keep skipping} \\
\delta(q_2, B, B) &= \text{FAIL}; \text{ reached end - only one block picked}
\end{align*}
\] (3)
\(\delta(q_1, b, B) = (q_3, b, B, R, L); \) compare blocks

\(\delta(q_3, a, a) = (q_3, a, a, R, L); \) keep comparing

\(\delta(q_3, b, B) = (q_4, b, B, R, S); \) successful - skip next block

\(\delta(q_3, b, B) = (q_5, b, B, R, R); \) successful - choose next block

\(\delta(q_3, a, B) = \text{FAIL}; \) blocks do not match

\(\delta(q_3, b, a) = \text{FAIL}; \) blocks do not match

Etc. Notice that \(q_4\) can either continue skipping until it runs out of input - and fails, or turns into a \(q_5\) at some \(b\) farther down the input. \(q_5\), similar to \(q_3\), performs the attempted block match, this time from left to right. If it succeeds, \(M\) goes into an accepting state, if it fails (see the failures of \(q_3\)) the acceptance fails.

This solution executes in time \(t(n) = 2n\), counting the rewind time for the input. On the other hand, it introduces a number of paths that will fail, because it does not guarantee that each path will contain exactly three blocks by the time the input scan has reached the end. I don’t believe there is a way to guarantee the three blocks unless you provide the machine with more ”self-awareness” (how many blocks are left from this point on?): use another tape to store \(k - 3\); erase a 1 on this tape every time you make a choice. If you reach the blank before being done, you check the state you are in and force the ”deterministic choice” - since there are only three possible situations to take care of, you can ensure that all possible triples will be ”visited” by creating enough states. A problem with this approach is that the ”worst case” has the input \(bbb \ldots bbb\) and \(k = n + 1\). If you assume that only non-pathological situations arise, then you can do it. Another observation is that one can run this algorithm during the ”rewind phase”, thus needing a total \(n\) time.

**Problem 6.4.4.** Show that the complexity class \(NP\) is closed under union, intersection, concatenation and Kleene closure.

**Solution.** Recall the definitions:

\[ NTIME(t(n)) = \{ L(M) \mid M \text{ is an NTM with time bound } t(n) \}, \]

\[ NP = \bigcup_{c > 0} NTIME(n^c). \]  \hspace{1cm} (5)

Note: see p. 294, Examples 6.13 and 6.14. These are the non-deterministic variants of the same problem. p. 233, Example 5.9 gives the template for union and intersection.

Assume \(L_1\) and \(L_2\) are two languages in \(NP\), with one-working-tape (and one input tape) NTMs \(M_1\) and \(M_2\), polynomial bounds \(n^{c_1}\) and \(n^{c_2}\), and start states \(s_1\) and \(s_2\), respectively.

1. Let \(x \in L_1 \cap L_2\). Consider a Turing machine with all the states and transitions of \(M_1\) and \(M_2\), and an additional state \(s\) and a transition \(\delta(s, B, B) = \{(s_1, B, B, S, S), (s_2, B, B, S, S)\}\). Then at least one of the machines will decide in time \(t(n) = \max(n^{c_1}, n^{c_2}) + 1\), which is polynomial.

2. Let \(x \in L_1 \cap L_2\). \(x\) must be accepted by both machines. If one constructs the non-deterministic product machine, whose halt state is just \((h_{M_1}, h_{M_2})\), its time bound would be \(\min(n^{c_1}, n^{c_2})\),
and would be thus polynomial. The problem with this approach is that, since the two machines are different, running both for the minimum time may not work, and running both for the maximum time may introduce spurious acceptances. If use of the original time-bounds and determination of a joint polynomial time-bound were not essential, the product approach would work since you would run until the end of the input. Since, in this case, membership is decidable, one can run the machines in sequence: create a machine $M$ with just an input tape. Have it copy the input to the input tape of $M_1$, execute until acceptance or rejection of $x$ (time bound $n^{c_1}$) and, if it accepts, have $M$ copy the saved input string onto the input tape of $M_2$ move to the start state of $M_2$. Accept if it halts in the halt state of $M_2$ (in time $n^{c_2} + n^{c_1} + c \cdot n$, where $c\cdot n$ accounts for the two copies of the input and the transitions needed to connect, reject otherwise.

3. Let $x \in L_1 \cdot L_2$. It would be tempting to think that one could let the non-deterministic nature of the TMs take care of the problem of splitting the string $x$: unfortunately, the halt state in each machine cannot be reached if the input is non-empty. This requires that we introduce a first part of the machine which will, non-deterministically (with time-bound $n$) split the string into two substrings, $x_1$ and $x_2$. We check each partition is succession. The total time bound is $|x_1|^{c_1} + |x_2|^{c_2} + a \cdot n$, for some constant $a$.

4. Let $x \in L^*$. Example 6.14, essentially, simulates non-determinism, and manages, via dynamic programming, to reduce an apparent exponential combinatorial explosion to (only) a polynomial one. Since we now have the real thing, we can non-deterministically break the string into all its possible partitions ($x_1 \ldots x_k$, maximum $n$ of them) and check each for membership in $L$ in time $|x_1|^{c}$. An upper bound for the non-deterministic time: $n \cdot n^{c}$.

**Problem 6.5.3.** Show that the class of context-sensitive languages is closed under union, intersection, concatenation and Kleene closure.

**Solution.** By Theorem 6.3.5 we know that the class of context-sensitive languages is exactly the class $NSPACE(n)$.

Assume $L_1$ and $L_2$ are two languages in $NSPACE(n)$, with one-working-tape (and one input tape) NTMs $M_1$ and $M_2$, and start states $s_1$ and $s_2$, respectively. What we will do is use a "mixed method": some proofs are easier using grammars, some are easier using the space bound. For the grammar-based proofs, we assume $L_1$ has a CSG $G_1$ with start symbol $S_1$, and $L_2$ has a CSG with start symbol $S_2$.

1. Let $x \in L_1 \cup L_2$. Consider the grammar $G$ which is the union of $G_1$ and $G_2$ augmented by the symbol $S$ and by the rule $S \rightarrow S_1 \mid S_2$. The new grammar satisfies the rules for being context sensitive (right hand side at least as long as left hand side of every rule). If either $S_1 \rightarrow \epsilon$ or $S_2 \rightarrow \epsilon$, remove the rule and add $S \rightarrow \epsilon$.

2. Let $x \in L_1 \cdot L_2$. Consider the grammar $G$, union of $G_1$ and $G_2$, augmented by the symbol $S$, with the added rule $S \rightarrow S_1 S_2$. Same additional comments as in the case before.

3. Let $x \in L^*_1$. $G = G_1 \cup \{S\} \cup \{S \rightarrow SS_1 \mid \epsilon\}$. Same comments as above for the possible removal of an $\epsilon$-rule from $G_1$. 

4
4. Let $x \in L_1 \cap L_2$. This is where the grammar approach is harder: the modifications needed for the "intersection grammar" are less than obvious. How do we construct a TM that will be able to accept $x \in L_1 \cap L_2$ in space exactly $|x|$? There are two possible approaches: run the machines $M_1$ and $M_2$ sequentially. The machine that starts one and moves you to the second on acceptance by the first needs only a few extra states, but no extra space, so you have your decision in $NSPACE(n)$. Another possibility is to construct a machine where the two working tapes of $M_1$ and $M_2$ are made into a single tape (if you think this creates problems with the input tape, have two copies of the input tape, one for each machine: tapes 1 and 2 are used by machine $M_1$, tapes 3 and 4 by machine $M_2$. The joint machine has the product set of instructions). We have a three or four tape machine. Align the tapes at the left, and think of this as a multi-tape machine: coalescing the tapes - with symbols for head positions on all of them - gives you a single work-tape with $n$ cells.

**Problem 6.5.4.** Find a recursive language $L$ that is not context-sensitive.

**Solution.** Since most of the people who provided (an incomplete) solution had a reference to http://www.mathreference.com/lan-tm.univ.html, I will leave you with that... except for the observation that the Turing enumerability of CSGs, crucial to the diagonalization proof submitted, requires some kind of reference to an appropriate result in our textbook...

For those who cannot find such a reference (or cannot come up with a proof of their own), here it is, a variant of the treatment in Hopcroft and Ullman’s 1979 book:

Note that CSGs have terminal, nonterminal, start, bracket symbols and rules, so we need to come up with a unary representations for all of them, separated by appropriate numbers of 1s. This implies that we can encode any CSG as a binary string (we have done that before, anyway, for TMs and other items). We know that CSLs are recursive sets (both Corollary 6.37 and a different proof presented in class). We can also construct a TM that, given a binary string as input, can decide whether it is a string that defines a CSG and provides an NTM that implements that CSG, and decides the corresponding CSL.

At this point, we can take all the binary strings in lexicographic order and, passing them through this TM, decide whether they represent CSGs or not. This gives us an enumeration of all those binary strings that are the programs for the UTM that simulates the NTMs that implement CSGs. The rest of the proof proceeds, as in the reference, by diagonalization.