# Theory of Computation: Assignment 10 Solutions

Arjun Chandrasekhar

1. For this problem, we will note that because $L_1 \in \text{P}$ and $L_2 \in \text{TIME}(n^c)$, there exist machines $M_1$ and $M_2$ which decide $L_1$ and $L_2$ respectively in time $O(n^c)$.

   (a) To decide $\overline{L_1}$, we run $M_1$ If it accepts, we reject; otherwise we reject. Because $M_1$ runs in $O(n^c)$, so will our algorithm. Thus, $\overline{L} \in \text{TIME}(n^c)$.

   (b) To decide $L_1 \cup L_2$ we run $M_1$ and $M_2$, and accept if either machine accepts. Because both machines run in time $O(n^c)$, the overall runtime will be $O(n^c) + O(n^c) = O(n^c)$. Thus, $L_1 \cup L_2 \in \text{TIME}(n^c)$.

   (c) To decide $L_1 \circ L_2$, we will use the following algorithm

      **1.** For each possible way of splitting up $w = xy$ do the following:
         **a.** Run $M_1$ on $x$ and $M_2$ on $y$
         **b.** If both machines accept, then accept $w$. Otherwise move on to the next split
      **2.** If all splits failed, reject $w$

   There are $O(n)$ possible splits to try. Each split takes $O(n^c)$ to test, because both $M_1$ and $M_2$ run in $O(n^c)$. Thus, the overall runtime is $O(n) \cdot O(n^c) = O(n^{c+1})$. Thus, $L_1 \circ L_2 \in \text{TIME}(n^{c+1})$

2. To show that to show that P is closed under complement, union, and concatenation, we would need to show that if $L_1, L_2 \in \text{P}$ then $\overline{L_1}, L_1 \cup L_2$, and $L_1 \circ L_2$ are in P. We note that if $L_1, L_2 \in \text{P}$, then $L_1, L_2 \in \text{TIME}(n^c)$ for some $c$. As problem 1 shows, this implies that $\overline{L_1}$ and $L_1 \cup L_2$ are in $\text{TIME}(n^c) \subseteq \text{P}$, and $L_1 \circ L_2 \in \text{TIME}(n^{c+1}) \subseteq \text{P}$. Thus, $\overline{L_1}, L_1 \cup L_2$, and $L_1 \circ L_2$ are all in P, which establishes closure under those three operations.

3. (a) The algorithm has in $O(k)$ loop iterations. However, because $k$ is represented in binary, the value of $k$ is exponential in the length of the input $|k|$. Thus, the algorithm really runs in $O(2^{|k|})$, which is clearly not polynomial.

   (b) Following the hint, we compute $\text{lcm}(a, b) = \dfrac{a \cdot b}{\gcd(a, b)}$. We can use the Euclidean algorithm to compute $\gcd(a, b)$ in polynomial time. We then accept if $k = \text{lcm}(a, b)$, and reject otherwise. The Euclidean algorithm runs in polynomial time. Multiplication, division, integer comparison can also be calculated in polynomial time. Thus, the overall runtime is polynomial.

4. Following the hint, we will use dynamic programming. We will create a 1-D array of size $B + 1$. Each element $A[i]$ will tell us whether there is a combination that adds up to $i$. At the end, $A[B]$ will tell us whether there is a combination that adds up to $B$.

   Formally, we'll use the following algorithm:

   **1.** $A \leftarrow$ array of length $B + 1$. Initialize $A[0]$ to TRUE, all other elements to FALSE
   **2.** For $i = 1, \dots B$ do the following:
      **a.** For $j = 1 \dots n$ do the following:
         **i.** If $i \geq x_j$ and $A[i - x_j] = \text{TRUE}$, set $A[i]$ to TRUE
   **3.** If $A[B] = \text{TRUE}$, accept $\langle B, x_1, \dots, x_n \rangle$. Otherwise, reject

The algorithm has $O(B)$ outer loop iterations; because the input is unary, this is $O(|\langle B \rangle|)$. Each outer loop iteration has $O(n)$ inner loop iterations. Thus the overall runtime is polynomial.

5. **Approach 1:** We'll give a greedy algorithm for checking if $G$ is 2-colorable.

   **1.** Pick any vertex and color it red

   **2.** Repeat the following until the graph is colored or we reach a contraadiction

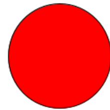
   **a.** Color the neighbors of all red vertices green, and color the neighbors of all greeen vertices red.

   **b.** If there is a vertex that is a neighbor of both a red and green vertex, then we can't color it. Immediately reject $\langle G \rangle$.

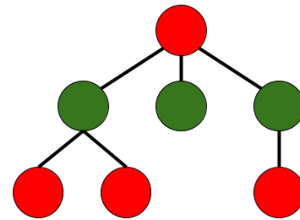
   **3.** If we were able to color the graph, accept $\langle G \rangle$

   The following picture illustrates the algorithm

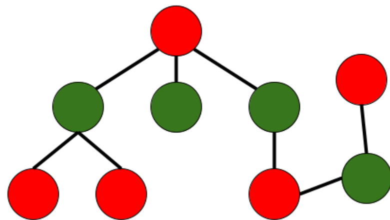
1. Color a vertex red



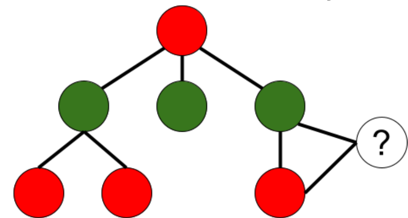
2. Color its neighbors green (and their neighbors red, and so on)



3. Continue until the graph is colored…



4. ...or until we reach an impasse




Every vertex and every edge is visited exactly once by the algorithm, thus the overall runtime is $O(m+n)$ where $m$ is the number of vertices and $n$ is the number of edges. Thus, 2-COLORING $\in$ P.

**Approach 2:** We will convert the graph into a 2-SAT formula $F$ such that $F$ is satisfiable if and only if $G$ is 2-colorable.

Consider an edge $(u, v)$. We want one of these two vertices to be red, and one of them to be green. Now consider the formula $(u \vee v) \wedge (\neg u \vee \neg v)$. This formula is true if $u$ and $v$ have opposite truth values.

To convert $G$ into a formula $F$, we create a variable for every vertex $u$. We go through every edge $(u, v)$ and add $(u \vee v) \wedge (\neg u \vee \neg v)$ to the formula. Then we check if $F$ is satisfiable. If it is, we color all of the TRUE vertices red, and all the FALSE vertices green. The way we constructed our formula ensures that adjacent vertices have opposite truth values, and thus opposite colors. Conversely, if $G$ is 2-colorable, we set all of the red variables to TRUE and the green variables to FALSE. Connected vertices must opposite colors, so every $(u \vee v) \wedge (\neg u \vee \neg v)$ sub-formula will be satisfied.

Converting $G$ into a formula $F$ will take $O(m)$ steps, where $m$ is the number of edges. Then 2-SAT can be solved in polynomial time. Thus, 2-COLORING $\in$ P