CS 38 Algorithms **April 14, 2023**

Problem Set 1

Codename: Jambul **Late Submission:** No

1 Problem 1

1.1 Part a

Proof. Consider the functions

$$f(x) = \begin{cases} x^{2x-1} & x \text{ even} \\ x^{2x} & x \text{ odd} \end{cases}, g(x) = \begin{cases} x^{2x} & x \text{ even} \\ x^{2x-1} & x \text{ odd} \end{cases},$$

for $x \in \mathbb{Z}_+$. Note that, for any x,

$$f(x+1) - f(x) = (x+1)^{2(x+1)-1} - x^{2x} = (x+1)^{2x+1} - x^{2x} > 0,$$
$$(x+1)^{2(x+1)} - x^{2x-1} = (x+1)^{2x+2} - x^{2x-1} > 0,$$

since $x > 0$. Thus both functions are strictly increasing over $\mathbb{Z}_+$.

Now suppose that there exists $c > 0$ such that $f(x) \leq c \cdot g(x)$ for all $x \in \mathbb{Z}_+$. But

$$\frac{f(x)}{g(x)} = \begin{cases} x^{-1} & x \text{ even} \\ x & x \text{ odd} \end{cases},$$

so if such a c does exist, we may simply set $x = 2c + 1$ to see that

$$\frac{f(x)}{g(x)} = 2c + 1 \not\leq c,$$

so indeed $f \notin \mathcal{O}(g)$. Similarly, if we suppose there exists $c > 0$ such that $g(x) \leq c \cdot f(x)$ for all x, we may argue that since

$$\frac{g(x)}{f(x)} = \begin{cases} x & x \text{ even} \\ x^{-1} & x \text{ odd} \end{cases},$$

taking $x = 2c$ would yield

$$\frac{g(x)}{f(x)} = 2c \not\leq c,$$

so again $g \notin \mathcal{O}(f)$, as desired. $\square$

1.2 Part b

Proof. Take $g(x) = \max\{f_1(x), ..., f_k(x)\}, x \in \mathbb{Z}_+$. Then clearly we have that $f_i \leq g$ for every $1 \leq i \leq k$, confirming that there exists at least one function satisfying property (i).

Suppose now that h is another function such that $f_i \in \mathcal{O}(h)$ for every $1 \leq i \leq k$ and such that $g \notin \mathcal{O}(h)$. Then we may simply set $g(x) = \min\{g(x), h(x)\}$ for all $x \in \mathbb{Z}_+$ so $g \in \mathcal{O}(h)$.

We can then repeat this process for all such $h : \mathbb{Z}_+ \to \mathbb{Z}_+$ where $g \notin \mathcal{O}(h)$ until no such h exists. This algorithm is guaranteed to terminate, since we will only ever be lowering the values of g, and these values are bounded below by 0. Thus there exists a g with properties (i) and (ii). $\square$

1.3 Part c

Proof. We will show that there exists such a g in a similar fashion to the way in which we showed (b). Define

$$g(n) = \max\{f_1(n), ..., f_n(n)\}.$$

Then, intuitively, as we let $n \to \infty$, we will have that $g(n)$ will always bound the countably infinite set of functions $F = \{f_1, f_2, ...\}$. Indeed, if $f_k \in \mathcal{O}(\max\{f_1, ..., f_{n^*}\})$ for some $n^* \in \mathbb{Z}_+$ and all $k > n^*$, then it will always be the case that $f_k \in \mathcal{O}(g), k > n^*$, and of course, by construction $f_1, ..., f_{n^*} \in \mathcal{O}(g)$. If this is not the case (e.g., if $f_i \in \mathcal{O}(f_{i+1})) \ \forall \ i \geq 1$), then we will nonetheless still have that as $n \to \infty, g \geq f_1(n), ..., f_n(n)$, and so multiplication of g by a constant will allow g to bound all of $f_1, ..., f_n$ at all points in $\mathbb{Z}_+$ (and since we can do this as $n \to \infty$, we have that this applies to all of F).

In this way, then, we have that for g defined above, $f_i \leq c \cdot g$ for all $i \geq 1$ and some $c > 0$; IOW, $f_i \in \mathcal{O}(g) \ \forall \ i \geq 1$. □

2 Problem 2

Collaborated with: Phalsa

Algorithm 1. (Membership in Context-Free Grammar). We will use dynamic programming (DP) to solve the problem.

We will denote our input string by $w = (w_1, ..., w_n)$ so w is of length n.

```
1. If $z = \epsilon$ : return True.
2. Create an empty $n \times n$ table $T$. We will index the table as though it is a Cartesian plane with origin at
   the lowermost, leftmost cell. In this way, the lowermost, rightmost cell is $T(n, 1)$, and the uppermost,
   leftmost cell is $T(1, n)$. We will only use the lower triangle of this table (diagonal included).
3. For $i = 1$ to $n$:
   (a) For $A \to a \in R$:
       i. If $a = w_i$: Add $A$ to $T(i, 1)$
4. For $r = 2$ to $n$:
   (a) For $c = 1$ to $n - r + 1$:
       i. For $k = r - 1$ to 1 by $-1$:
          A. Denote $X$ the Cartesian product of the sets of variables at $T(c, k)$ and $T(c+k, r-c-k)$
          B. For $A \to BC \in R$:
             • If $BC \in X$ : Add $A$ to $T(r, c)$
5. If $S \in T(1, n)$ : return True
6. Else: return False
```

Proof of correctness. Denote our input string by $w = (w_1, ..., w_n)$. Denote by $w_{(c,r)}$ the partition $w_c, ..., w_{c+r-1}$, so the partition starts at the cth character in w and is of length r. We propose that Algorithm 1 yields a table T with entries $T(c, r)$ (indexed as described in the algorithm) corresponding to the variables in V that can be used to generate $w_{(c,r)}$. We will prove this claim by induction on r and c.

We will first prove the base case for the algorithm. Consider entries of the form $T(c, 1)$, so $r = 1$ and $1 \leq c \leq n$. Such an entry will correspond to the variables that can be used to produce the single-character string w_c. Thus we populate the lowermost row of the table ($r = 1$) in this exact way: By checking all rules $A \to a$ and if a is w_c, we add it to $T(c, 1)$. Then clearly $T(c, 1)$ holds the variables that can produce $w_{(c,1)}$.

Now assume the proposition holds for some $2 \leq r < n$ and any arbitrary $1 \leq c \leq n-r+1$ (the upper bound on c comes from the fact that if we wish for our substring to be of length r, the latest character it can start at is $n-r+1$). $T(c, r)$ must correspond to the variables that can be used to produce $w_{(c,r)} = (w_c, ..., w_{c+r-1})$. If we know how to obtain the set of variables that can be used to produce $(w_i, ..., w_{i+k})$, but we want to produce $(w_i, ..., w_{i+k+l})$, then we can first take those variables that produce $(w_i, ..., w_{i+k})$ and add any variable that can produce $(w_{i+k+1}, ..., w_{i+k+l})$. This is the intuition behind the algorithm.

Building on this idea, to fill out cell $T(c, r + 1)$, we can iterate down through all cells $T(c, r - k + 1), 1 \leq k \leq r$. By the induction hypothesis, each of these cells will hold the variables that can be used to produce $w_{(c,r-k)} = (w_c, ..., w_{c+r-k-1})$. We therefore require the variables that can be used to produce $w_{(c+r-k,k)} = (w_{c+r-k}, ..., w_{c+r-1})$. By construction, these are held in cell $T(c + r - k, k)$.

Then, our desired set of variables will be the pairs of variables produced from the Cartesian product between sets $T(c, r - k + 1)$ and $T(c + r - k, k)$. Denote this set X. But since we cannot include pairs of variables, we must again loop through all rules in R, and if $A \to BC$ with $BC \in X$, then we know we can use A to get BC, and BC to get $w_{(c,r+1)}$. Thus we then add A to $T(c, r+1)$, and again $T(c, r+1)$ will have all the variables in V that can be used to produce $w_{(c,r+1)}$.

Finally, it is easy to check whether the starting variable S is in $T(1, n)$. If it is, then we know that S can be used to produce the string starting at w_1 and of length n—i.e., w.

Complexity. The algorithm will populate $1 + ... + n \in \mathcal{O}(n^2)$ cells in the table. At worst, we will need to iterate back down all the rows below the current cell we are at. This will require $n - 1$ steps. Thus the complexity of this algorithm is $\mathcal{O}(n^3)$. This is reflected in the three nested `for` loops in the algorithm above, which each have individual runtimes of $\mathcal{O}(n)$, yielding a total runtime for the algorithm of $\mathcal{O}(n^3)$.

3 Problem 3

Algorithm 2. (Shortest Common Supersequence). We will use DP to solve the problem. Denote $x_{(i)}, y_{(j)}$ the partitions of x and y up to (and including) characters x_i and y_j, respectively, and $f(x_{(i)}, y_{(j)})$ is the length of the SCS of $x_{(i)}$ and $y_{(j)}$. Then

$$f(x_{(i)}, y_{(j)}) = \begin{cases} \max(i,j), & i = 0 \text{ or } j = 0 \\ f(x_{(i-1)}, y_{(j-1)}) + 1, & x_i = y_j \\ \min(f(x_{(i-1)}, y_{(j)}), f(x_{(i)}, y_{(j-1)})) + 1, & \text{else} \end{cases}.$$

This is the guiding function for our solution below.

```
1. Create an empty $m \times n$ table $T$.
2. For $i = 0$ to $m$:
    (a) For $j = 0$ to $n$:
        i. If $i = 0$ or $j = 0$: $T(i,j) = \max\{i,j\}$
        ii. Else if $x_i = y_j$: $T(i,j) = T(i-1, j-1) + 1$
        iii. Else: $T(i,j) = \min\{T(i-1,j), T(i,j-1)\} + 1$
3. Create an empty output string $r$ and set indices $i = m, j = n$.
4. While $i! = 0$ and $j! = 0$:
    (a) If $x_i = y_j$:
        i. Add $x_i$ to $r$. Subtract one from $i$ and $j$.
    (b) Else if $T(i, j-1) < T(i-1, j)$:
        i. Add $y_j$ to $r$. Subtract one only from $j$.
    (c) Else:
        i. Add $x_i$ to $r$. Subtract one only from $i$.
5. If $i > 0$: Reverse the substring $x_1, ..., x_i$ and add it to $r$.
6. Else if $j > 0$ : Reverse the substring $y_1, ..., y_j$ and add it to $r$.
7. Reverse the string $r$ and return.
```

Proof of correctness. Denoting the input strings by $x = (x_1, ..., x_m)$ and $y = (y_1, ..., y_n)$, we claim that the algorithm provided above will populate the empty $m \times n$ table T such that $T(i,j)$ contains the length of the shortest common supersequence between $x_{(i)}$ and $y_{(j)}$. In particular, $T(m,n)$ We will prove the claim by induction on i and j.

For our base case, consider entries with $i = 0$ and $j = 0$. In such cases, we are asking for the length of the shortest common supersequence of the empty string ϵ and some substring of either x or y. Clearly, then, this will be the length of the substring. Since either $i = 0$ or $j = 0$, this will simply be $\max\{i,j\}$.

Now suppose that the claim holds for some $0 < i < m$ and any arbitrary $0 \le j \le n$. Then we will show by induction on j^* that the claim holds for all cells of the form $T(i+1, j^*), j \le j^* \le n$. For the base case, consider $T(i+1, j+1)$. This cell should hold the length of the SCS of $x_{(i+1)}$ and $y_{(j+1)}$. If $x_{i+1} = y_{j+1}$, then this is simply the length of the SCS of $x_{(i)}$ and $y_{(j)}$ plus one (for the character x_{i+1} at the end). Otherwise, we can set this length to be the minimum of $T(i+1, j)$ and $T(i, j+1)$ plus one, since these entries (again by the IH) will hold the lengths of the SCS of $x_{(i+1)}$ and $y_{(j)}$ and that of $x_{(i)}$ and $y_{(j+1)}$, respectively, so all we must append to the shorter of these entries is the last missing character—either y_{j+1} or x_{i+1}, respectively.

Thus the base case holds. Now assume that it holds for some $j < j^* < n$. Then we can apply the same subroutine as above—replacing the base case $T(i+1, j+1)$ by $T(i+1, j^*+1)$—to populate cell $T(i+1, j^*+1)$ with the length of the SCS of $x_{(i+1)}$ and y_{j^*+1}. So by induction, we have that all entries of the form $T(i+1, j^*+1)$ will adhere to the claim, and thus we have shown that any cell $T(i, j), 0 \leq i \leq m, 0 \leq j \leq n$, will contain the length of the SCS of $x_{(i)}$ and $y_{(j)}$.

Finally, we must use this populated table to reconstruct a particular instance of the SCS of x and y. Denote our output string by r. We can do this by first starting at the lowermost, rightmost cell in the table, $T(i = m, j = n)$. If, from this cell, $T(i, j-1) < T(i-1, j)$—that is, the length of the SCS of $x_{(i)}$ and $y_{(j-1)}$ is less than that of $x_{(i-1)}$ and $y_{(j)}$—then we must add y_j to r, since this will result in the supersequence with the shorter length, as we already know (and now assume) that the cells in the table have been populated correctly. Otherwise, for the same reason, we add x_i to r. To then move to the cell we have just referenced, we subtract one from j in the former case and one from i in the latter case. (Of course, if the lengths stored at the two cells are equal, we may choose an arbitrary one to move to.) Again, since we assume any entry $T(i, j)$ contains the (optimally minimized) length of the SCS of $x)(i)$ and $y_{(j)}$, we can be certain that this routine will reconstruct the shortest possible supersequence that resulted in the entry at $T(m, n)$ by simply tracing back through the steps employed by the algorithm to populate the table originally.

Complexity. The algorithm will first visit each cell in the $m \times n$ table once, and filling each cell is done in constant time, so this routine runs in $\mathcal{O}(mn)$. The `while` loop that follows will traverse $\mathcal{O}(m + n)$ cells. Thus the overall runtime of the algorithm is $\mathcal{O}(mn)$.