University of Houston

Midterm 2 Review

# COSC 3320
# Algorithms and Data Structures

**Note**

**Read the Academic Honesty policy.**

## Exercise 1: Longest Common Subsequence (20 Points)

Let $A$ and $B$ be two arrays of lengths $m$ and $n$, respectively. A *common subsequence* of $A$ and $B$ is any subsequence present in both arrays. For example, given

$$A = [3, 17, 9, 16, 9, 16, 0, 1, 6, 9]$$
$$B = [19, 10, 18, 15, 17, 7, 3, 9]$$

the sequence [17, 9] is a common subsequence of $A$ and $B$. Give a dynamic programming algorithm to find the *length* of the *longest* common subsequence (LCS) of arrays $A$ and $B$.

1. State the subproblems. Clearly explain your notation.
2. Give a recursive formulation to solve the subproblems.
3. Give pseudocode for your algorithm.
4. State the runtime of your algorithm. Justify your answer.
5. Explain how to modify your solution to output the actual LCS.

**Solution.**

1. Let $\text{LCS}(i, j)$ denote the longest common subsequence on $A[0 : i]$ and $B[0 : j]$.
2. We have base cases:
$$\text{LCS}(0, j) = \text{LCS}(i, 0) = 0$$
and
$$\text{LCS}(i, j) = \begin{cases} 1 + \text{LCS}(i-1, j-1) & \text{if } A[i-1] = B[j-1] \\ \max(\text{LCS}(i-1, j), \text{LCS}(i, j-1)) & \text{otherwise} \end{cases}$$
3. Assume `cache` is a lookup table with $\texttt{cache}[0, j] = \texttt{cache}[i, 0] = 0$ for all $0 \leq i \leq m$ and $0 \leq j \leq n$.
```
def LONGEST-COMMON-SUBSEQUENCE(A, B):
    cache = LOOKUP-TABLE(m + 1, n + 1)
    ▷ Initialize Base Cases
    for i in 0 ... m + 1:
        cache[i, 0] = 0
    for j in 0 ... n + 1:
        cache[0, j] = 0

    def LCS(i, j):
        if (i, j) ∉ cache:
            if A[i − 1] == B[j − 1]:
                cache[i, j] = 1 + LCS(i − 1, j − 1)
            else:
                discard-ai = LCS(i − 1, j)
                discard-bj = LCS(i, j − 1)
                cache[i, j] = max(discard-ai, discard-bj)
        return cache[i, j]

    return LCS(m, n)
```
4. The runtime is $\mathcal{O}(mn)$, since there are $(m+1) \times (n+1)$ subproblems, each requiring $\mathcal{O}(1)$ time.
5. Begin with an empty array `result` and set $i, j = m, n$. Perform traceback on $\text{LCS}(i, j)$. If $A[i-1] == B[j-1]$, then append $A[i-1]$ to `result` and continue the traceback on $\text{LCS}(i-2, j-2)$. Otherwise, continue the traceback on the the maximum of $LCS(i-2, j-1)$, $LCS(i-1, j-2)$. In pseudocode:
```
result = []
```

```
i, j = (m, n)
while i ≥ 0 and j ≥ 0:
    if A[i − 1] == B[j − 1]:
        result.APPEND(A[i − 1])
        i, j = i − 1, j − 1
    else if LCS(i − 1, j) ≥ LCS(i, j − 1):
        i = i − 1
    else:
        j = j − 1
```

□

## Exercise 2: Coin Change (20 Points)

Given coins with denominations $C_1, C_2, \ldots, C_n$ and a target value $t$, find the minimum number of coins required to add up to $t$.

1. Consider the following greedy algorithm: find the coin with the greatest denomination less than or equal to $t$. Say that coin has denomination $C$. Take one such coin and repeat on $t-C$.

   Show that this algorithm does not, in general, output the optimal value.

2. Give a dynamic programming algorithm to find the minimum number of coins required to add up to $t$. If no combination of coins has sum $t$, output $\infty$. What is the runtime of this algorithm?

**Solution.**

1. Take denominations 1, 2, 5, and 7 and $t = 10$. Our greedy algorithm uses three coins:

   $$10 = 7 \times 1 + 2 \times 1 + 1 \times 1$$

   but the optimal value is 2.

   $$10 = 5 \times 2$$

2. Let $\textsc{Min-Coins}(t)$ denote the minimum number of coins needed to make $t$. Observe that the only way to make $t$ is to first make one of

   $$\begin{gathered} t - C_1 \\ t - C_2 \\ \vdots \\ t - C_n \end{gathered}$$

   and then add the corresponding coin to make $t$. In particular, if the optimal choice were to make $t - C_i$ for some particular $i$, then we would have

   $$\textsc{Min-Coins}(t) = \textsc{Min-Coins}(t - C_i) + 1$$

   since, *by definition*, the optimal way to make $t - C_i$ is $\textsc{Min-Coins}(t - C_i)$. Thus, we have

   $$\textsc{Min-Coins}(t) = 1 + \min_{1 \le i \le n} (\textsc{Min-Coins}(t - C_i))$$

   with base cases $\textsc{Min-Coins}(0) = 0$ and $\textsc{Min-Coins}(x) = \infty$ for $x < 0$.

   There are $t$ subproblems, each of which takes $\mathcal{O}(n)$ time, for a runtime of $\mathcal{O}(nt)$.

$\square$

**Exercise 3: Word Formation (20 Points)**

Given a dictionary of words $D$ and a string $s$, design a dynamic programming algorithm to determine if the string $s$ can be broken into a sequence of words from $D$. What is the runtime of your algorithm? Assume that checking if a string is in the dictionary takes $\mathcal{O}(1)$ time.

**Solution.** Let $\textsc{Can-Be-Formed}(i)$ be `True` if $s[0:i]$ can be broken up into a sequence of words from $D$ and `False` otherwise. Further, let $\textsc{in-dict}(s)$ denote whether $s$ is in $D$, i.e., $\textsc{in-dict}(s)$ is `True` if $s$ is in $D$ and `False` otherwise.

Now, consider some index $k$ such that $\textsc{in-dict}(s[k:i])$ is `True`. Notice that, if $\textsc{Can-Be-Formed}(i)$ is `True`, then

$$S[0:i] = \underbrace{s_0\, s_1\, \ldots\, s_{k-1}}_{\substack{\text{Can be broken into}\\\text{sequence of words}}}\ \underbrace{s_k\, \ldots s_{i-1}}_{\text{in } D}$$

Thus, if this is true for *any* $k$, then we must have that $\textsc{Can-Be-Formed}(i)$ is `True`. This yields the recursive formulation:

$$\textsc{Can-Be-Formed}(i) = \bigvee_{0 \le k < i} \left(\textsc{Can-Be-Formed}(k) \wedge \textsc{in-dict}(s[k:i])\right)$$

or, in a more functional style:

$$\textsc{Can-Be-Formed}(i) = \operatorname*{any}_{0 \le k < i} \left(\textsc{Can-Be-Formed}(k) \wedge \textsc{in-dict}(s[k:i])\right)$$

with base case $\textsc{Can-Be-Formed}(0) =$ `True`. □