# CSCI 3110 Assignment 8 Solutions

1. You are given a string of n characters $s[1 \ldots n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like "itwasthebestoftimes..." ). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function $\textsc{dict}(\cdot)$ : for any string $w$,

$$\textsc{dict}(w) = \begin{cases} true & \text{if } w \text{ is a valid word} \\ false & \text{otherwise.} \end{cases}$$

   (a) (10 pts) Give a dynamic programming algorithm that determines whether the string $s[\cdot]$ can be reconstituted as a sequence of valid words. The running time should be at most $\mathrm{O}(n^2)$, assuming calls to dict take unit time. Start by describing the problem as an array $d[\cdot]$ and then provide a recurrence for $d[i]$ in terms of $d[j]$ where $j < i$. Then determine the dynamic programming order that solves this recurrence efficiently and give pseudocode that does this.

   **ANSWER:** We first need to describe our problem so let

$$d[i] = \begin{cases} true & \text{if } s[1 \ldots i] \text{ is a valid sequence of words} \\ false & \text{otherwise.} \end{cases}$$

   Then we want to determine $d[n]$.

   Now we need to describe our subproblems. $s[1 \ldots i]$ is a valid sequence of words if, and only if, $s[1 \ldots j]$ is a valid sequence of words and $s[j+1 \ldots i]$ is a valid word. Thus $d[i]$ is given by the following recurrence:

$$d[i] = \begin{cases} false & \text{if } i \leq 0 \\ \max_{1 \leq j < i} \left( d[j] \wedge \textsc{dict}(s[j+1 \ldots i]) \right) & \text{otherwise.} \end{cases}$$

   To solve this recurrence efficiently we observe that $d[i]$ depends only on values $d[j]$ where $j < i$. Thus we should solve $d[1], d[2], \ldots d[n]$:

```
ValidSentence(s, n)
 1 ▷ d[1...n] is an array of boolean values
 2 for i ← 1 to n
 3    do d[i] ← false
 4       for j ← 1 to i − 1
```

```
5          do if d[j] ∧ DICT(s[j + 1 . . . i])
6                d[i] ← true
```

(b) (Bonus: 5 pts) In the event that the string is valid, make your algorithm output the corresponding sequence of words. Use a choice array to record the optimal choices made by your algorithm and then use a backtracking algorithm to output the sequence of words.

We first need to make a small alteration to our algorithm to record the choices made. We keep a choice array $c[i]$ that records what subproblem $d[j]$ we used. The algorithm now becomes:

```
VALIDSENTENCE(s, n, d, c)
 1 ▷ d[1 . . . n] is an array of boolean values
 2 ▷ c[1 . . . n] is an array of integers
 3 for i ← 1 to n
 4    do d[i] ← false
 5       c[i] ← 0
 6       for j ← 1 to i − 1
 7          do if d[j] ∧ DICT(s[j + 1 . . . i])
 8                d[i] ← true
 9                c[i] ← j
```

To output the sequence of words we follow these choices backwards beginning from $c[n]$ (assuming that it is true).

```
OUTPUTSENTENCE(s, n, d, c)
1 i = n
2 while i > 0
3    do print s[([c[i] + 1) . . . i]
4       i = c[i]
```

Note that this prints the words in reverse order but we could easily print them in the correct order using a stack.

2. Consider the following game. A "dealer" produces a sequence $s_1 \dots s_n$ of "cards", face up, where each card $s_i$ has a value $v_i$. Then two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect cards of largest total value. (For example, you can think of the cards as bills of different denominations.) Consider the score from the perspective of the first player, that is, the sum of the cards the first player has chosen minus the sum of the cards the second player has chosen. Assume $n$ is even.

(a) (5 pts) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the natural greedy

strategy is suboptimal. Why does the greedy algorithm fail on your sequence and what is the optimal strategy on your sequence?

**ANSWER:** Consider the sequence $2, 100, 1, 1$. If the first player is greedy and takes the first card with a value of 2 then the second player can take the card with a value of 100 and win. The optimal strategy for the first player is to take the last card with a value of 1. Then the second player will take either the 2 or the remaining 1 and the first player can take 100.

(b) (10 pts) Describe an $O(n^2)$ algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in $O(n^2)$ time some information, and then the first player should be able to make each move optimally in $O(1)$ time by looking up the precomputed information. Assume that the second player will always make an optimal move: after every move the first player makes the second player will make the move that reduces the final score by as much as possible. Describe the problem, Give a recurrence for the subproblems, and then explain the dynamic programming order. You do not need to provide pseudocode but explain how Player 1 chooses what move to make from your precomputed information.

**ANSWER:** What information does a player need to know to make an optimal move? The maximum score they can achieve on a subproblem $s_i \dots s_j$! Let $d[i, j]$ be the score for $s_i \dots s_j$ if both players make optimal moves. Now consider what an optimal move looks like. Player 1 attempts to maximize the score, while Player 2 attempts to minimize it. Player 1 has two choices: take the first or the last card. Then Player 2 has their 2 choices. To play optimally, Player 1 should choose the maximum score he can achieve after assuming that Player 2 will minimize this value. This provides the following recurrence:

$$d[i,j] = \begin{cases} 0 & \text{if } i \leq j \\ \max(s_i + \min\left(d[i-1, j-1] - s_j, d[i-2, j] - s_{i+1}\right), & \\ \qquad s_j + \min\left(d[i-1, j-1] - s_i, d[i, j-2] - s_{j-1}\right)) & \text{otherwise.} \end{cases}$$

To compute these values, we observe that $d[i, j]$ relies on values of $d[x, y]$ where $i \leq x \leq y \leq j$. Our algorithm simply computes these values in order of increasing $j$. This takes $O(n^2)$ time because there are $O(n^2)$ values to compute, each of which takes constant time. We store a matrix $c[i, j]$ which is the value that Player 1 should take. Using this choice matrix, Player 1 can play optimally.

3. **Time and space complexity of dynamic programming.** Our dynamic programming algorithm for computing the sequence alignment edit distance between strings of length $m$ and $n$ creates a table of size $nm$ and therefore needs $O(nm)$ time and space. In practice, it will run out of space long before it runs out of time. How can this space requirement be reduced?

(a) (5 pts) Show that if we just want to compute the value of the edit distance (rather

than the optimal sequence of edits), then only $O(n)$ space is needed, because only a small portion of the table needs to be maintained at any given time.

**ANSWER:** Any edit distance value $D[i, j]$ relies only on values $D[\cdot, j-1]$ or $D[\cdot, j]$. Thus we can simply store the current and previous columns of the matrix rather than the entire matrix. This requires $2n = O(n)$ space.

(b) (5 pts) Now suppose that we also want the optimal sequence of edits. As we saw earlier, this problem can be recast in terms of a corresponding grid-shaped dag, in which the goal is to find the optimal path from node $(0, 0)$ to node $(n, m)$. It will be convenient to work with this formulation, and while we're talking about convenience, we might as well also assume that $m$ is a power of 2. Lets start with a small addition to the edit distance algorithm that will turn out to be very useful. The optimal path in the dag must pass through an intermediate node $(k, m/2)$ for some $k$; show how the algorithm can be modified to also return this value $k$.

**ANSWER:** Each optimal path to a node $(i, j)$ will pass through only one intermediate node $(k, m/2)$. So we create an extra value $k[i, j]$: the intermediate node $(k, m/2)$ on the optimal path to $(i, j)$. $k[x, m/2] = x$ for all $1 \leq x \leq n$. For other nodes $k[i, j]$ is set to the value of $k[i-1, j]$, $k[i-1, j-1]$, or $k[i, j-1]$ depending on which subproblem we use to get to $(i.j)$. The value of $k$ that we are looking for is $k[n, m]$.

(c) (Bonus: 5 pts) Now consider a recursive scheme:

FIND-PATH$((0, 0) \rightarrow (n, m))$
1 compute the value $k$ above
2 FIND-PATH$((0, 0) \rightarrow (k, m/2))$
3 FIND-PATH$((k, m/2) \rightarrow (n, m))$
4 concatenate these two paths, with $k$ in the middle

Show that this scheme can be made to run in $O(nm)$ time and $O(n)$ space.

**ANSWER:** Lines 1 and 4 take $O(nm)$ time, so this is a recursive algorithm with recurrence $T(nm) = T(k \cdot m/2)T(n-k) \cdot m/2) + O(nm)$. This is not an easy recurrence to analyze. Each level $i$ of the recursion has $2^i$ recursive calls. Also, every pair of recursive calls shares $(k+1) \cdot m/2 + (n-(k-1)) \cdot m/2 = O(nm/2)$ of the table. Thus each level of the recurrence throws away about half of the possible table space. The running time is $O(nm) + O(nm/2) + O(nm/4) + \ldots = O(nm)$. This uses $O(n)$ space as discussed in part (a) because we only need to store the previous two columns of values rather than the entire matrix. We also end up using $O(n+m)$ space for the path, which is a mistake in the original question (Perhaps the authors of the book assumed $m = O(n)$? I did not see this assumption written anywhere).