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 \(\text{dict}(\cdot)\) : for any string \(w\),
\[
\text{dict}(w) = \begin{cases} 
\text{true} & \text{if } w \text{ is a valid word} \\
\text{false} & \text{otherwise.}
\end{cases}
\]

(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 \(O(n^2)\), assuming calls to \text{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.

(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.

2. Consider the following game. A “dealer” produces a sequence \(s_1\ldots s_n\) 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?

(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.

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.

   (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. Let's 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 $k$.

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

   ```
   FIND-PATH((0,0) → (n,m))
   1 compute the value $k$ above
   2 FIND-PATH((0,0) → (k,m/2))
   3 FIND-PATH((k,m/2) → (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.