1. **Breaking up is hard to do**

   A certain string-processing language offers a primitive operation which splits a string into two pieces. Since this operation involves copying the original string, it takes \(n\) units of time for a string of length \(n\), regardless of the location of the cut. Suppose, now, that you want to break a string into many pieces. The order in which the breaks are made can affect the total running time. For example, if you want to cut a 20-character string at positions 3 and 10, then making the first cut at position 3 incurs a total cost of \(20 + 17 = 37\), while doing position 10 first has a better cost of \(20 + 10 = 30\).

   Give a dynamic programming algorithm that, given the locations of \(m\) cuts in a string of length \(n\), finds the minimum cost of breaking the string into \(m+1\) pieces.

2. **Longest common subsequence**

   Given two strings \(x = x_1 x_2 \cdots x_n\) and \(y = y_1 y_2 \cdots y_n\) we wish to find the length of their longest common subsequence. That is, the largest \(k\) for which there are indices \(i_1 < i_2 < \cdots < i_k\) and \(j_1 < j_2 < \cdots < j_k\) with \(x_{i_1} x_{i_2} \cdots x_{i_k} = y_{j_1} y_{j_2} \cdots y_{j_k}\). Show how to do this in time \(O(mn)\).

3. **Time and space complexity of dynamic programming**

   Our dynamic programming algorithm for computing the edit distance between strings of length \(m\) and \(n\) creates a table of size \(n \times m\) and therefore needs \(O(mn)\) 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) 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) 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) Now consider a recursive scheme:

   ```
   procedure find-path ((0,0) \rightarrow (n,m))
   compute the value \(k\) above
   find-path((0,0) \rightarrow (k, m/2))
   find-path((k, m/2) \rightarrow (n,m))
   concatenate these two paths, with \(k\) in the middle.
   ```

   Show that this scheme can be made to run in \(O(mn)\) time and \(O(n)\) space.
4. **Sequence alignment** When a new gene is discovered a standard approach to understanding its function is to look through a database of known genes and find close matches. The closeness of two genes is measured by the extent to which they are aligned. To formalize this, think of a gene as being a long string over an alphabet $\Sigma = \{A, C, G, T\}$. Consider two genes (strings) $x = ATGCC$ and $y = TACGCA$. An alignment of $x$ and $y$ is a way of matching up these two strings by writing them in columns, for instance:

```
  - A T - G C C
  T A - C G C A
```

Here the “-” indicates a “gap”. The characters of each string must appear in order, and each column must contain a character from at least one of the strings. The score of an alignment is specified by a scoring matrix $\delta$ of size $(|\Sigma| + 1) \times (|\Sigma| + 1)$, where the extra row and column are to accommodate gaps. For instance the preceding alignment has the following score:

$$
\delta(-, T) + \delta(A, A) + \delta(T, -) + \delta(-, C) + \delta(G, G) + \delta(C, C) + \delta(C, A)
$$

Give a dynamic programming algorithm that takes as input two strings $x[1 \cdots n]$ and $y[1 \cdots m]$ and a scoring matrix $\delta$ and returns the highest-scoring alignment. The running time should be $O(mn)$.

5. **Alignment with gap penalties** The alignment algorithm of the previous exercise helps to identify DNA sequences that are close to one another. The discrepancies between these closely matched sequences are often caused by errors in DNA replication. However, a closer look at the biological replication process reveals that the scoring function we considered earlier has a qualitative problem: nature often inserts or removes entire substrings of nucleotides (creating long gaps), rather than editing just a position at a time. Therefore, the penalty for a gap of length 10 should not be 10 times the penalty for a gap of length 1, but something significantly smaller.

Repeat the previous exercise but use a modified scoring function in which the penalty for a gap of length $k$ is $c_0 + c_1 k$, where $c_0 > c_1$ are given constants.