Problem 1 (CLRS 15.1-3). (1 point) Consider a modification of the rod-cutting problem in which, in addition to a price $p_i$ for each rod, each cut incurs a fixed cost of $c$. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

Solution: We can modify BOTTOM-UP-CUT-ROD algorithm from section 15.1 as follows:

```java
BOTTOM-UP-CUT-ROD(p, n, c)
1 let r[0..n] be a new array
2 r[0] = 0
3 for j = 1 to n
4     q = -∞
5     for i = 1 to j - 1
6         q = max(q, p[i] + r[j - i] - c)
7     r[j] = max(q, p[j])
8 return r[n]
```

We need to account for cost $c$ on every iteration of the loop in lines 5-6 but the last one, when $i = j$ (no cuts). We make the loop run to $j - 1$ instead of $j$, make sure $c$ is subtracted from the candidate revenue in line 6, then pick the greater of current best revenue $q$ and $p[j]$ (no cuts) in line 7.

Problem 2 (CLRS 15.4-5). (1 point) Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of $n$ numbers.

Solution: We observe that the longest monotonically increasing subsequence (LIS) of sequence $S$ is a longest common subsequence (LCS) of $S$ and sorted $S$. For a sequence of length $n$, sorting can be done in $O(n \lg n)$ time, and finding the LCS – in $O(n^2)$ time. Utilizing the LCS-LENGTH and PRINT-LCS procedures from section 15.4, we can find the LIS as follows:

```java
LIS(S)
1 S' = SORT(S)
2 c, b = LCS-LENGTH(S, S')
3 PRINT-LCS(b, S, S.length, S.length)
```

Problem 3 (CLRS 15.4-6). (2 points) Give an $O(n \lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of $n$ numbers. (Hint: Observe that the last element of a candidate subsequence of length $i$ is at least as large as the last element of a candidate subsequence of length $i - 1$. Maintain candidate subsequences by linking them through the input sequence.)
Solution: We build our algorithm around two key insights:

- We can keep track of candidate subsequences of $X$ using array $M$ such that $M[j] = k$ indicates $X[k]$ being the smallest value for which there is a monotonically increasing subsequence of length $j$ ending with $X[k]$. For example, for $X = \langle 2, 3, 13, 11, 5, 7 \rangle$, we have $M = \langle \text{NIL}, 0, 1, 4, 5 \rangle$.

- With $L$ representing the length of the longest increasing subsequence found so far, sequence $X[M[1]], X[M[2]], ... X[M[L]]$ is monotonically increasing. This allows us to search in this sequence in $O(\lg n)$ time.

Our algorithm makes a single pass through $X$, maintaining array $M$ by locating the largest $j \leq L$ such that $X[M[j]] \leq X[i]$ for the current $i$. We also maintain array $P$, in which $P[k] = q$ indicates $X[q]$ being the predecessor of $X[k]$ in the longest monotonically increasing subsequence ending with $X[k]$; we use this array to reconstruct the solution. The following pseudocode is based on pseudocode from https://en.wikipedia.org/wiki/Longest_increasing_subsequence#Efficient_algorithms:

```plaintext
COMPUTE-LIS(X, n)
1 let P[0..n] be a new array
2 let M[0..n+1] be a new array
3 L = 0
4 for i = 0 to n
5     lo = 1
6     hi = L
7     while lo \leq hi
8         mid = lo + \lfloor (hi - lo)/2 \rfloor
9         if X[M[mid]] \leq X[i]
10            lo = mid + 1
11        else
12            hi = mid - 1
13     newL = lo
14     P[i] = M[newL - 1]
15     M[newL] = i
16     L = max(L, newL)
17 return P, M, L

PRINT-LIS(X, P, M, L)
1 let S[0..L] be a new array
2 k = M[L]
3 for i = L - 1 to 0
4     S[i] = X[k]
5     k = P[k]
```

**Problem 4 (CLRS 15-4).** (3 points) Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text
is a sequence of \( n \) words of lengths \( l_1, l_2, \ldots, l_n \), measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of \( M \) characters each. Our criterion of "neatness" is as follows. If a given line contains words \( i \) through \( j \), where \( i \leq j \), and we leave exactly one space between words, the number of extra space characters at the end of the line is \( M - j + i - \sum_{k=i}^{j} l_k \), which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of \( n \) words neatly on a printer. Analyze the running time and space requirements of your algorithm.

**Solution:** If we define \( \text{extras}[i, j] = M - j + i - \sum_{k=i}^{j} l_k \) to be the number of extra space characters at the end of the line containing words \( i \) through \( j \), we can define the following line cost function for the line containing words \( i \) through \( j \):

\[
\text{lc}[i, j] = \begin{cases} 
\infty & \text{if } \text{extras}[i, j] < 0, \\
0 & \text{if } j = n \text{ and } \text{extras}[i, j] \geq 0, \\
(\text{extras}[i, j])^3 & \text{otherwise.}
\end{cases}
\]

- Negative \( \text{extras}[i, j] \) indicates that words \( i \) through \( j \) don't fit; this should never occur in a correct solution and so we assign this case an infinite cost.
- We don't need to minimize \( \text{extras}[i, j] \) in the last line, so any non-negative value is an acceptable solution.
- The remaining case specifies the cost function according to the problem statement.

We note that the problem exhibits optimal substructure: If an arrangement of words \( 1, \ldots, j \) with the last line containing words \( i, \ldots, j \) is optimal, the preceding lines contain an optimal arrangement of words \( 1, \ldots, i - 1 \).

Let us define \( c[j] \) to be the cost of an optimal arrangement of words \( 1, \ldots, j \). Following the optimal substructure argument above, \( c[j] = c[i-1] + \text{lc}[i, j] \). Enumerating all possible choices for \( i \) (the first word on the last line for a given subproblem) gives the following recursive definition for \( c[j] \):

\[
c[j] = \begin{cases} 
0 & \text{if } j = 0, \\
\min_{1 \leq i \leq j}(c[i-1] + \text{lc}[i, j]) & \text{otherwise.}
\end{cases}
\]

To be able to reconstruct the actual solution, we also record the arrangement in array \( p \) such that \( p[j] = k \) indicates that \( c[j] \) ended up picking \( c[k-1] + \text{lc}[k, j] \) for the optimal solution. This way, the last line of the final arrangement contains words \( p[n], \ldots, n \), the line before last – words \( p[p[n]], \ldots, p[n] - 1 \), and so on.

Implementing each of the steps above as a separate procedure gives:
**COMPUTE-EXTRAS**(l, n, M)

1. let extras[1..n, 1..n] be a new array
2. for i = 1 to n
   3. // One-word line, so just width minus word length.
   4. extras[i, i] = M − l[i]
   5. for j = i + 1 to n
      6. // Previous minus new word length minus space between words.
      7. extras[i, j] = extras[i, j − 1] − l[j] − 1
3. return extras

**COMPUTE-LINE-COST**(extras, n)

1. let lc[1..n, 1..n] be a new array
2. for i = 1 to n
   3. for j = i to n
      4. if extras[i, j] < 0
         5. // Words don’t fit.
         6. lc[i, j] = ∞
      7. elseif j = n and extras[i, j] ≥ 0
         8. // Last line and at least zero trailing spaces.
         9. lc[i, j] = 0
   10. else
       11. // Normal cost function.
       12. lc[i, j] = (extras[i, j])³
3. return lc

**COMPUTE-COST**(lc, n)

1. let c[1..n] be a new array
2. c[0] = 0
3. for j = 1 to n
   4. c[j] = ∞
   5. for i = 1 to j
      6. if c[i − 1] + lc[i, j] < c[j]
         7. c[j] = c[i − 1] + lc[i, j]
         8. p[j] = i
3. return c, p

**PRINT-LINES**(p, j)

1. i = p[j]
2. if i = 1
   3. k = 1
   4. else
      5. k = PRINT-LINES(p, i − 1) + 1
3. // Print words i through j on line k.
4. PRINT(k, i, j)
5. return k
The algorithm runs in $\Theta(n^2)$ time and requires $\Theta(n^2)$ space. Both characteristics can be improved to $\Theta(nM)$ by observing that at most $\lceil M/2 \rceil$ words can fit on a line (each word being at least one character long plus spaces between words), and only computing and storing extras and $lc$ for $j - i + 1 \leq \lceil M/2 \rceil$.

**Problem 5 (CLRS 15-5).** (4 points) See CLRS 15-5 for full problem statement.

(a) Given two sequences $x[1..m]$ and $y[1..n]$ and set of transformation-operation costs, the edit distance from $x$ to $y$ is the cost of the least expensive operation sequence that transforms $x$ to $y$. Describe a dynamic-programming algorithm that finds the edit distance from $x$ to $y$ and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

**Solution:**

Let us define $X_i = x[1..i]$ and $Y_j = y[1..j]$ to be prefixes of sequences $x$ and $y$, $X_i \rightarrow Y_j$ to be a problem of determining the cost of the least expensive operation sequence that transforms $X_i$ to $Y_j$, and $c[i, j]$ to be that cost.

We observe that the problem exhibits optimal substructure: an optimal solution to $X_p \rightarrow Y_q$ includes optimal solutions to $X_i < p \rightarrow Y_j < q$.

We now consider different possibilities for the last operation in the optimal solution to $X_i \rightarrow Y_j$:

- **Copy.** Then $x[i] = y[j]$, the remaining subproblem is $X_{i-1} \rightarrow Y_{j-1}$ and $c[i, j] = c[i - 1, j - 1] + \text{cost(copy)}$.

- **Replace.** Then $x[i] \neq y[j]$, the remaining subproblem is $X_{i-1} \rightarrow Y_{j-1}$ and $c[i, j] = c[i - 1, j - 1] + \text{cost(replace)}$.

- **Delete.** Then we have no restrictions on $x[i]$ and $y[j]$, the remaining subproblem is $X_{i-1} \rightarrow Y_j$ and $c[i, j] = c[i - 1, j] + \text{cost(delete)}$.

- **Insert.** Then we have no restrictions on $x[i]$ and $y[j]$, the remaining subproblem is $X_i \rightarrow Y_{j-1}$ and $c[i, j] = c[i, j - 1] + \text{cost(insert)}$.

- **Twiddle.** Then $x[i] = y[j - 1]$ and $x[i - 1] = y[j]$ for $i, j \geq 2$, the remaining subproblem is $X_{i-2} \rightarrow Y_{j-2}$ and $c[i, j] = c[i - 2, j - 2] + \text{cost(twiddle)}$.

- **Kill.** This must be the final operation, so the current problem must be $X_m \rightarrow Y_n$. We can kill the string starting from any $0 \leq i < m$ and so $c[i, j] = c[m, n] = \min_{0 \leq i < m} (c[i, n]) + \text{cost(kill)}$. 

**Proof:**
We can now provide the following recursive definition for $c[i, j]$:

$$c[i, j] = \min \begin{cases} 
    c[i - 1, j - 1] + \text{cost}(\text{copy}) & \text{if } x[i] = y[j], \\
    c[i - 1, j - 1] + \text{cost}(\text{replace}) & \text{if } x[i] \neq y[j], \\
    c[i - 1, j] + \text{cost}(\text{delete}) & \text{in all cases}, \\
    c[i, j - 1] + \text{cost}(\text{insert}) & \text{in all cases}, \\
    c[i - 2, j - 2] + \text{cost}(\text{twiddle}) & \text{if } i, j \geq 2, x[i] = y[j - 1], x[i - 1] = y[j], \\
    \min_{0 \leq i < m}(c[i, n]) + \text{cost}(\text{kill}) & \text{if } i = m, j = n. 
\end{cases}$$

In addition, we note that $X_0 \rightarrow Y_j$ – conversion from an empty string – can be viewed as a sequence of $j$ inserts, and $X_i \rightarrow Y_0$ – conversion to an empty string – as a sequence of $i$ deletes.

We can now convert the definition above into the following dynamic programming algorithm for computing edit distance and an optimal operation sequence:
**COMPUTE-EDIT-DISTANCE**($x, y, m, n$)

1. let $c[0..m, 0..n]$ be a new array
2. let $op[0..m, 0..n]$ be a new array

// Converting to an empty string.
3. for $i = 0$ to $m$
4.     $c[i, 0] = i \cdot \text{cost}(\text{DELETE})$
5.     $op[i, 0] = \text{DELETE}$

// Converting from an empty string.
6. for $j = 0$ to $n$
7.     $c[0, j] = j \cdot \text{cost}(\text{INSERT})$
8.     $op[0, j] = \text{INSERT}$

// All cases other than KILL.
9. for $i = 1$ to $m$
10. for $j = 1$ to $n$
11.     $c[i, j] = \infty$
12.     if $x[i] = y[j]$
13.         $c[i, j] = c[i - 1, j - 1] + \text{cost}(\text{COPY})$
14.         $op[i, j] = \text{COPY}$
15.     if $x[i] \neq y[j]$ and $c[i - 1, j - 1] + \text{cost}(\text{REPLACE}) < c[i, j]$
16.         $c[i, j] = c[i - 1, j - 1] + \text{cost}(\text{REPLACE})$
17.         $op[i, j] = \text{REPLACE} + y[j]$ 
18.     if $c[i - 1, j] + \text{cost}(\text{DELETE}) < c[i, j]$
19.         $c[i, j] = c[i - 1, j] + \text{cost}(\text{DELETE})$
20.         $op[i, j] = \text{DELETE}$
21.     if $c[i, j - 1] + \text{cost}(\text{INSERT}) < c[i, j]$
22.         $c[i, j] = c[i, j - 1] + \text{cost}(\text{INSERT})$
23.         $op[i, j] = \text{INSERT}$
24.     if $i \geq 2$ and $j \geq 2$ and $x[i] = y[j - 1]$ and $x[i - 1] = y[j]$
25.         and $c[i - 2, j - 2] + \text{cost}(\text{TWIDDLE}) < c[i, j]$
26.         $c[i, j] = c[i - 2, j - 2] + \text{cost}(\text{TWIDDLE})$
27.         $op[i, j] = \text{TWIDDLE}$

// KILL.
28. for $i = 0$ to $m - 1$
29.     if $c[i, n] + \text{cost}(\text{KILL}) < c[m, n]$
30.         $c[m, n] = c[i, n] + \text{cost}(\text{KILL})$
31.         $op[m, n] = \text{KILL} + i$

return $c, op$

The algorithm fills an $m \times n$ table, spending constant time on each cell, so the running time and space are both $\Theta(mn)$. 

7
We can reconstruct the actual sequence of operations using the following procedure:

**PRINT-OPERATIONS**(\( op, i, j \))

1. if \( i = 0 \) and \( j = 0 \) then return.
2. if \( op[i, j] = \text{COPY} \) or \( op[i, j] = \text{REPLACE} \) then
   3. \( i' = i - 1 \)
   4. \( j' = j - 1 \)
3. elseif \( op[i, j] = \text{DELETE} \) then
   4. \( i' = i - 1 \)
   5. \( j' = j \)
4. elseif \( op[i, j] = \text{INSERT} \) then
   5. \( i' = i \)
   6. \( j' = j - 1 \)
5. elseif \( op[i, j] = \text{TWIDDLE} \) then
   6. \( i' = i - 2 \)
   7. \( j' = j - 2 \)
6. else \( // \text{KILL} \) then
   7. \( i' = \text{GET-KILL-INDEX}(op[i, j]) \)
   8. \( j' = j \)
7. **PRINT-OPERATIONS**(\( op, i', j' \))
8. **PRINT**(\( \text{op}, i, j \))

(b) Explain how to cast the problem of finding an optimal DNA alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

The DNA alignment problem can be reduced to the edit distance problem by taking the following operation costs:

- \( \text{COST(COPY)} = -1 \) to account for the \( x'[j] = y'[j] \) and neither is space case.
- \( \text{COST(REPLACE)} = +1 \) to account for the \( x'[j] \neq y'[j] \) and neither is space case.
- \( \text{COST(DELETE)} = +2 \) and \( \text{COST(INSERT)} = +2 \) to account for the \( x'[j] \) or \( y'[j] \) is a space case.
- \( \text{COST(TWIDDLE)} = \infty \) and \( \text{COST(KILL)} = \infty \). In other words, the operations are not permitted.

Then, the negative of the cost minimized by **COMPUTE-EDIT-DISTANCE** is the score to maximize in the DNA alignment problem.