Problem 15.1-2:
Show, by means of a counterexample, that the following “greedy” strategy does not always determine an optimal way to cut rods. Define the density of a rod of length $i$ to be $p_i$, that is, its value per inch. The greedy strategy for a rod of length $n$ cuts off a first piece of length $i$, where $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n-i$.

Solution:
Here is a counterexample to prove that greedy algorithm doesn’t provide an optimal solution everytime:

<table>
<thead>
<tr>
<th>length $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>price $p_i$</td>
<td>1</td>
<td>20</td>
<td>33</td>
<td>36</td>
</tr>
<tr>
<td>$p_i/i$</td>
<td>1</td>
<td>10</td>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

Let the length of the rod be 4 in. As per the Greedy algorithm, the first cut for the rod would be at a length 3 as the density for length 3 rod is maximum(11) taking the price to 33. Now we are just left with 1 in length rod of price 1. The total price for the rod, thus would be 34 if we choose Greedy strategy.

However, instead if we cut the rod in two halves, viz. 2 small rods of length 2 each, then the total price of the rod would be 40 which is optimal.

Hence, we say that greedy strategy does not always provide an optimal solution.
**Problem 15.1-3:**

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:**

Here cost $c$ is associated with every cut, then we will have to subtract this cost from the revenue wherever a cut is made. Also whenever no cuts are made, we should not deduct cost $c$ from the associated revenue value. Thus max revenue for length $n$ rod is given by,

$$
r_n = \max \{ p_n, p_i + r_{[n-i]} - c \}, \text{where } 1 \leq i \leq n
$$

**Pseudocode:**

$MODIFIED\text{-}CUT\text{-}ROD(p, n, c)$

let $r[0..n]$ be a new array

$r[0] = 0$

for $j = 1$ to $n$

\[ q = p[j] \]

for $i = 1$ to $j-1$

\[ q = \max(q, p[i] + r[j-i] - c) \]

$r[j] = q$

return $r[n]$ 

Here all the modifications in the original bottom-up dynamic programming algorithm are highlighted in bold.

We are initializing $q$ with $p[j]$ and then running the inner for loop over $i$ from 1 to $j-1$ (in place of $j$), as for $i=j$ (i.e. when no cuts are made), we don’t want to do any cost deduction $c$. Had we run the inner for loop from 1 to $j$, then even in the case of no cuts, we would have deducted cost $c$ from the revenue, which is incorrect. Assigning $q$ with $p[j]$ takes care of the case with no cuts.

**Time Complexity:**

The time complexity for this algorithm would be same as that of the original bottom-up dynamic programming approach, which is $O(n^2)$. This is because we are just running the inner for loop 1 less time than the original along with the cost subtraction changes, both of which have no effect on the time complexity of the algorithm.