CS 463
Section 01
Spring 2021
Christopher K. Schmitt
Assignment 5

Problem 1

In order to maximize profit, we can create a simple recursive function that makes a cut at some position, k. We are then left with a rod of length $n - k$. We recursively cut the rod at every possible position, returning the optimal solution at the end. This approach is clearly exponential time, but involves re-computing potions of the pipe at each step. We can eliminate this with a "bottom-up" approach. We can create a temporary array to store the solutions of sub-problems and avoid re-computing them. We start by solving for the smaller rods, as they will be used to create our solution.

Algorithm 1 Rod Cutting

```
function CUT(cost, n)
    temp ← [0, ..., n]
    for i = 1 to n do
        x ← −∞
        for j = 1 to i do
            x ← MAX(x, cost[j] + temp[i − j])
        end for
        temp[i] ← x
    end for
    return temp[n]
end function
```

This algorithm belongs to $\mathcal{O}\left(n^2\right)$ because the MAX function and the inner-most addition are both constant time. We can therefore solve for the running-time with the following sigma-expression.

$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \frac{n^2+n}{2} \in \mathcal{O}\left(n^2\right)$$

The top-down approach (memoization) has identical performance characteristics, but uses a recursive method instead of an iterative one.

Problem 2

To compute the number of unique paths, we can use a naïve recursive algorithm to solve this problem in exponential time. This would involve re-computing the number of unique paths in identical sub-grids several times. To eliminate this, we can store the number of unique paths to get from any point to the destination in an $m \times n$ matrix. We can then use a bottom-up approach to fill the matrix, starting from the destination and iterating up to the start.

Algorithm 2 Unique Paths

function PATHS(m, n)

  temp $\leftarrow \begin{pmatrix} 0_{0,0} & 0_{0,1} & \cdots & 0_{0,n-1} \\ 0_{1,0} & 0_{1,1} & \cdots & 0_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0_{m-1,0} & 0_{m-1,1} & \cdots & 0_{m-1,n-1} \end{pmatrix}$

  for $i = 0$ to m **do**

    temp$[i][0] \leftarrow 1$

  end for

  for $i = 0$ to n **do**

    temp$[0][i] \leftarrow 1$

  end for

  for $i = 1$ to m **do**

    for $j = 0$ to n **do**

      temp$[i][j] \leftarrow$ temp$[i-1][j]$ + temp$[i][j-1]$

    end for

  end for

  return temp$[m-1][n-1]$

end function

This algorithm belongs to $\mathcal{O}(n \cdot m)$, which we can see by solving the following sigma expression.

$$m + n + \sum_{i=1}^{m} \sum_{j=0}^{n} 1 = m(n+1) + m + n = nm + 2m + n \in \mathcal{O}(nm)$$

It should that this relation only holds as long as $n \geq 2$. Using combinatorics, we can find an even cleaner solution. We can use the formula $\binom{m+n-2}{n-1}$. Subbing in 5 and 6 for m and n gives us $\binom{9}{5} = 126$.

Problem 3

Suppose we create a function, $f(i, j)$, that return the maximum profit path from the bottom to row to (i, j). This function will use the value $p(i, j)$ plus the result from the previous row. In this manner, we can avoid re-computing previous rows by filling a matrix in a bottom-up manner. A simpler solution however, would be to memoize the recursive definition. We can construct a recursive definition like so.

$$f(i,j) = \begin{cases} -\infty & j < 0 \\ -\infty & j \geq n \\ p(i,j) & i = 0 \\ max(f(i-1,j-1), f(i-1,j), f(i,j-1)) + p(i,j) & otherwise \end{cases}$$

By using the definition, we can see that this is an exponential time computation. However, memoization will mean that we only check each ordered pair once. On an $n \times n$ board, this would put us at $\mathcal{O}\left(n^2\right)$ complexity. To construct the path, we can create an algorithm like this.

Algorithm 3 Checkerboard

```
function CONSTRUCT(i, j, memo, path)
    if i = 0 then
        return path
    end if
    a ← memo[i - 1][j - 1] or construct(i - 1, j - 1, memo, path)
    b ← memo[i - 1][j] or construct(i - 1, j, memo, path)
    c ← memo[i][j - 1] or construct(i, j - 1, memo, path)
    memo[i][j] ← max(a, b, c) + p(i, j)
    path.push(j)
    return construct(i - 1, j, memo, path)
end function
```

Problem 4

We can either include or exclude each element in the knapsack. We could use a simple recursive solution to explore every possible combination of includes. Without memoization, this would involve re-computing many include-exclude combinations. We can instead build a table in a bottom-up manner to avoid this needless work.

Our algorithm will begin by populating a table with the base case. Since we cannot add items when we have none to choose from, the solution for the first row is always zero. At each row, i, we will compute the maximum value we could get by including the the i'th element. This will be equal to $temp[i-1][j] + value[i]$.

Algorithm 4 knapsack

function KNAPSACK$(n, w, weight, value)$

  temp $\leftarrow \begin{pmatrix} 0_{0,0} & 0_{0,1} & \cdots & 0_{0,1} \\ 0_{1,0} & 0_{1,1} & \cdots & 0_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0_{w,0} & 0_{w,1} & \cdots & 0_{w,n} \end{pmatrix}$

  for $i = 1$ to n **do**

    for $j = 1$ to w **do**

      max exclude $\leftarrow$ temp$[i-1][j]$

      max include $\leftarrow -\infty$

      if weight$[i] < j$ **then** ▷ If the knapsack can fit the item

        max include $\leftarrow$ value$[i]$ + temp[i][j - weight$[i]$]

      end if

      temp[i][j] $\leftarrow$ max(max include, max exclude)

    end for

  end for

  return temp$[n][w]$

end function

This algorithm fills every entry in a $|value| \times |weight|$ table. If the size of these collections are n and w respectively, then the complexity of this algorithm is $\mathcal{O}(nw)$. With the given conditions, the maximum knapsack value is 75.