Problem Set 4
Due: Wednesday, October 30th, 2019 (within the first 5 minutes of class)

Problem 1. [DYNAMIC PROGRAMMING]

McDowell’s is considering opening a series of restaurants along Route 7. The \( n \) possible locations are along a straight line, and the distances of these locations from the start of Route 7 are, in miles and in increasing order, \( m_1, m_2, \ldots, m_n \). The constraints are as follows:

- At each location, McDowell’s may open at most one restaurant. The expected profit from opening a restaurant at location \( i \) is \( p_i \), where \( p_i > 0 \) and \( i = 1, 2, \ldots, n \).
- Any two restaurants should be at least \( k \) miles apart, where \( k \) is a positive integer.

McDowell’s would like to find the set of locations that will yield the maximum total expected profit. For example, if the \( n \) locations are at miles: 0, 2, 5, 7, 10 and the \( n \) corresponding expected profits are: 2, 5, 4, 3, 1, and \( k = 4 \), then the optimal solution is the 2nd and 4th locations (i.e. at miles 2 and 7) with expected profit = 5 + 3 = 8.

(a) [21 points] Give an efficient algorithm to compute the maximum expected total profit subject to the given constraints. As we’ve done in class, your solution should describe the following:

- [2] The size of the array or matrix you will use to compute your solution.
- [3] What each entry of the array or matrix holds (in words).
- [7] The dynamic programming formulation (how to fill each entry, i.e. mathematically), including the base cases.
- [3] How to fill the entire array.
- [2] Where the optimal value is located once the array is filled.
- [4] A brief justification of the running time in terms of \( n \).
- [7] A brief explanation of how to find the locations. Hint: it will be helpful to maintain an additional array while you are computing the maximum profit.

(b) [15 points] Now that you’ve had some practice with solving problems with dynamic programming, you will combine this knowledge with your programming skills. The purpose of this exercise is to help you learn how to translate your pseudocode to actual code so you can understand some of the subtleties of dynamic programming: dealing with base cases, data structures, indexing, looping, and how to find the actual solution.

Write a program (either Java or python) to verify that your algorithm is correct. The program should take as command line \( n, k \), the \( n \) mile markers and the \( n \) corresponding expected profits and output the maximum total expected profit and the optimal locations.

For the example above, the command line arguments should be:
```
5 4 0 2 5 7 10 2 5 4 3 1
```
and the output should be:
```
8 at 4 2
```
Since the max profit is 8 and the optimal locations are at mile markers 4 and 2.

Here are some other test examples:
command line arguments: 5 10 10 20 25 30 40 100 100 100
ans: 400 at 5 4 2 1
command line arguments: 8 5 0 5 10 15 19 25 28 29 0 10 4 61 21 13 19 15
ans: 94 at 7 4 3 2
Submit your program via the link on canvas.

Problem 2. Subset Sum

(a) [10 pts] The following is input to the 01-Knapsack problem:

\( W \) (knapsack capacity) = 5

\[
\begin{array}{c|cc}
\text{item} & \text{weight} & \text{value} \\
1 & 2 & 3 \\
2 & 3 & 4 \\
3 & 4 & 5 \\
4 & 5 & 6 \\
\end{array}
\]

Apply the dynamic programming algorithm we discussed to fill in the following table and find the optimal value we can achieve (do not worry about the actual items). In the table, the columns represent the weight limits and the rows represent the item numbers.

\[
\begin{array}{ccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
0 & & & & & \\
1 & & & & & \\
2 & & & & & \\
3 & & & & & \\
4 & & & & & \\
\end{array}
\]

(b) [21 points] Now consider a new problem: Given a set of positive integers \( L = \{x_1, x_2, \ldots, x_n\} \) and a target \( b \), the Subset Sum problem asks you to find \( S \subseteq L \) such that the sum of the elements in \( S \) is no greater than \( b \) but is also as close to \( b \) as possible.

Give an efficient dynamic programming solution to find the best value (i.e. the sum closest to \( b \), not the actual set of integers). As usual, your solution should include:

- [2] The size of the array or matrix you will use to compute your solution.
- [3] What each entry of the array or matrix holds (in words).
- [7] The dynamic programming formulation (how to fill each entry, i.e. mathematically), including the base cases.
- [3] How to fill the entire array.
- [2] Where the optimal value is located once the array is filled.

(b) [6 points] Describe how to modify your solution if the \( x_i \)'s were not integers. Also describe how to modify your solution if \( b \) was not an integer (assuming the \( x_i \)'s are still integers).

(c) [7 points] Now describe how to find the actual set of items.

Problem 3.

Coin-Changing

(a) [12 points] Recall the recursive non-memoized version of Coin-Changing that we discussed on the first day of class. Although this algorithm always produces the correct answer (i.e. the minimum number of coins needed to make change for the given amount), for any set of coin denominations, it has an exponential running time. Formally prove that...
the running time of this algorithm is $O(m^n)$, where $m$ is the number of coin denominations and $n$ is the amount for which to make change. Hint: Consider the recurrence tree that is generated by this algorithm.

(b) [13 points] Prove that the greedy approach of always choosing the coin with the largest denomination yields the optimal solution if the coins are in U.S. denominations. Use the standard proof technique we discussed in class – i.e. prove (1) the greedy choice property and (2) the optimal substructure property. For simplicity, you may assume that $n \geq 25$. 