CSC373 – Problem Set 2

To avoid suspicions of plagiarism: at the beginning of your submission, **clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, that you consulted**.

Problem Set: due October 18, 2021 22:00

Answer each question completely, always justifying your claims and reasoning. Your solution will be graded not only on correctness, but also on clarity.

Answers that are technically correct that are hard to understand will not receive full marks. Mark values for each question are contained in the [square brackets].

You may work in groups of up to TWO to complete these questions.

1 Robot Knapsack

A robot is in an n by n room and is allowed to start at any column in the top row (row 1). The robot has a knapsack with an integer weight capacity W, and, if programmed correctly, will maximize the value of the items that it puts in the knapsack.

Each cell (i, j) in the room has zero or one item stored there. If an item is there, then the positive number $v_{i,j}$ gives its value and the positive integer $w_{i,j}$ gives its weight. If there is no item there, then both $v_{i,j} = 0$ and $w_{i,j} = 0$.

The robot makes a sequence of moves. The robot picks up any item at its starting cell, and then picks up any item on its cell after it completes a move. The allowable moves are to move down one row, diagonally down one row and to the left one column, or diagonally down one row and to the right one column. It's not necessary for the robot to end up in the bottom row (row n).

Your goal is to write an algorithm to maximize the value of the items that get stored in the knapsack. Step 4 should give the optimal value, and step 5 should give the actual path that the robot should take to realize that value.

[5] Follow the five steps to design a dynamic-programming algorithm to solve this problem. Include and clearly label each step in your submission.

Programming Question

The best way to learn a data structure or an algorithm is to code it up. In some problem sets, we will have a programming exercise for which you will be asked to write some code and submit it. You may also be asked to include a write-up about your code in the PDF/TEXfile that you submit. Make sure to **maintain your academic integrity** carefully, and protect your own work. The code you submit will be checked for plagiarism. It is much better to take the hit on a lower mark than risking much worse consequences by committing an academic offence.

2 How Many Ways?

The input to this problem is a text t, a pattern p, and a nonnegative integer n.

Consider moving from left to right in t, choosing n nonempty, nonoverlapping substrings; call these substrings $s_1, s_2, \ldots, s_n$. If $s_1 s_2 \ldots s_n = p$ (that is, concatenating the substrings together forms the same string as p), then we have a *match*.

The problem is to determine the number of ways that n substrings can be chosen from t so that we have a match with p.

Here's an example. Suppose t is `aababc`, p is `aabc`, and $n = 2$. Then the answer is 4. Each line below shows one way to get p from t by choosing 2 appropriate substrings. The (1) and (2) indicate the two chosen substrings.

```
(1)aab  ab  (2)c
(1)aa  ba  (2)bc
(1)a  ab  (2)abc
a  (1)a  b  (2)abc
```

1. [**4**] Follow the **first four** steps to design a dynamic-programming algorithm to solve this problem. Include and clearly label each step in your submission writeup. (Any guesses as to why I'm not asking for step 5?)

2. [**3**] Now implement the function `num_ways` in the starter code by writing a bottom-up dynamic programming solution.

   Requirements:

   - Your code must be written in Python 3, and the filename must be `ways.py`.
   - We will grade only the `num_ways` function; please do not change its signature in the starter code. You may include as many helper functions as you wish.
   - For each test-case that your code is tested on, your code must run within 5x the time taken by our solution. Otherwise, your code will be considered to have timed out.

   Please include comments in your code to explain what your algorithm is doing.

3 Visiting all Squares

You've decided to play a board game instead of studying for courses like CSC369 and CSC347.

The board game consists of $n \geq 1$ squares numbered $0, 1, \ldots, n-1$. There is a cost $d(i, j)$ to jump directly from square i to square j.

You can't arbitrarily jump around the board, though, because there's one important rule that you must follow: if you visit square numbered v, then you must have already visited *all* squares numbered less than v, or you must not yet have visited *any* square numbered less than v.

Your goal is to minimize your total cost to visit each square exactly once, subject to the above rule. You can start on whatever square you want and end on whatever square you want.

1. [**4**] Follow the **first four** steps to design a dynamic-programming algorithm to solve this problem. Include and clearly label each step in your submission writeup.

2. [**3**] Now implement the function `cheapest_cost` in the starter code by writing a bottom-up dynamic programming solution.

   Requirements:

   - Your code must be written in Python 3, and the filename must be `all_squares.py`.
   - We will grade only the `cheapest_cost` function; please do not change its signature in the starter code. You may include as many helper functions as you wish.
   - For each test-case that your code is tested on, your code must run within 5x the time taken by our solution. Otherwise, your code will be considered to have timed out.

   Please include comments in your code to explain what your algorithm is doing.