Algorithms - Assignment 7

Due May 2 by Midnight

(1) Let A be an array of integers. We wish to find a subarray A[i:j] (i.e. a subarray of consecutive elements from A, sometimes referred to as a *contiguous subsequence*) such that when we sum those entries, we get the largest possible sum. For example, if $A = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$, then a contiguous subarray with the maximum sum is $[4, -1, 2, 1]$, whose elements sum to 6.

   (a) Describe and implement a brute-force solution to the problem. What is the runtime?

   (b) Describe a dynamic programming solution which operates in linear time. Your description needs to clearly define the subproblems, express a recurrence relation between the problem and those subproblems, and clearly state what is being done in order to construct the solution (i.e. what we are maximizing, minimizing, etc). You should also justify the runtime.

   (c) Implement the solution in Python. Implement both the top down version *and* the bottom up version. For now, your code only needs to output the maximum sum, and not the contiguous subsequence witnessing it. Compare them with the brute-force solution to make sure that they work, and then run some tests and include a plot comparing them. The bottom up should be slightly faster.

   (d) Modify the top-down solution so that it outputs the actual subsequence along with the maximum sum. Would your approach need to be any different for the bottom up?

(2) You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.

   You'd ideally like to travel 200 miles a day, but this may not be possible depending on the spacing of the hotels. We would like to find the sequence of hotels to stop at which leaves us travelling *as close to* 200 miles as possible. Towards figuring out what this sequence of stops is, we define the *penalty* for a day's driving as $(200 - x)^2$, where x is the distance travelled that day. So if we travel exactly 200 miles in a day, then the penalty is zero, but if we fall short of or exceed that number, then the penalty ramps

up. Our goal is to plan our trip so as to minimize the total penalty accrued over the course of the trip, i.e. minimize the sum of the penalties for each day that we drive.

(a) Describe a dynamic programming algorithm for computing the minimum penalty possible over the course of the trip. Your description should satisfy the same criteria as part b of the previous problem. What is the runtime?

(b) Convert that description into pseudocode.

(c) Explain how you would modify that pseudocode in order to recover the actual sequence of hotels which minimizes the total penalty.

(3) You are given a string of n characters $s[1, \ldots, n]$ which you believe to be a corrupted text document in which all punctuation has vanished (so it looks something like 'thetraditionofalldeadgenerationsweighslikeanightmareonthemindsoftheliving'). You wish to reconstruct the document using a dictionary. To get a dictionary (mostly) working in python, we'll use the natural language toolkit package. Do a pip install nltk in your terminal, and then copy-paste the following code to wherever you plan on solving this problem:

```
import nltk
nltk.download('words')
from nltk.corpus import words

english_words = set(words.words())

def d(word):
    if len(word) == 1:
            if word.lower() != 'a' and word.lower() != 'i': return False
    return word.lower() in english_words
```

This gives you a function d(s), which (mostly) returns True if s is a word in the English dictionary and False if it's not. I say mostly because it's pretty finicky. It counts every one letter string as a valid word (why I had to add the if statement in the function), and also seems to disqualify certain plural words as not words (for example, in the above example string, it thinks 'weighs' isn't a word, yet has no problem with the word 'generations'). Whatever. It's good enough for our testing purposes. We just want a way to check if our code (mostly) works.

(a) Describe a dynamic programming algorithm that determines whether the string s can be reconstituted as a sequence of valid words. Your explanation should clearly define the subproblems, express a recurrence relation relating a problem to it's subproblems, and explain how this relationship would be used in a solution. The running time should be at most $O(n^3)$ on the assumption that calls to the d function above are constant time. (Hint: Use the edit distance example as a model for how to approach this. This is a 2D dynamic programming problem, and it will help to draw a grid for a sample string and think about the dependencies of it like we did for that problem.)

(b) Write the dynamic program in python. I actually found it easier to go with a bottom-up approach for this one, but it does not matter. We will test your code on a few strings that are two or three words long each, such as 'dogstorm' and 'catviper'.

(c) Modify your code to recover the original document (i.e. return the actual words separated by spaces when the answer is yes.)

(4) We described in class a dynamic programming algorithm which could find shortest paths of particular lengths. In particular, we noted that for a weighted graph $G = (V, E, w)$ and a given starting node s, if $SP(v, i)$ is the shortest path from s to v which traverses exactly i many edges, then

$$SP(v, i) = \min_{(u,v)\in E}(SP(u, i-1) + w(u, v))$$

This implies a dynamic program for finding shortest paths. We also noted in class that this program had a suspiciously familiar runtime: $O(|V||E|)$ - the same as Bellman Ford. In this exercise you will investigate that relationship and see that these are really just the top down and bottom up versions of the same dynamic programming algorithm.

(a) Implement this dynamic program for shortest paths from the top down in python. Your program should be able to return all shortest paths from some starting node to every other node which traverses exactly i many edges *for all* $i \in 0, 1, \ldots, |V|$ (many of these will be ∞). The keys of your memo dictionary should be 2-tuples (u, i), where u is a node of the graph and i is a number of edges traversed (ranging between 0 and $|V|$). Make sure to initialize your memo so that $(s, 0) = 0$, and $(u, 0) = \text{inf}$ for all other nodes $u \neq s$. Your program can either return the entire subproblem dictionary, but for the purposes of comparing it to your Bellman Ford algorithm from the previous homework, you should also add an option which takes the minimum distance from s to every other node out of all possible i's in order to return an output that is hopefully the same as your Bellman-Ford function. Use both algorithms on the example graph from the previous assignment (the one linked on the canvas page under the assignment) to confirm that your implementation works.

(b) Run some tests of your top-down dynamic program against Bellman-Ford and show that they have comparable runtimes, and include a matplotlib plot in your write-up. You can use the same random graph function from the previous assignment. Don't go too crazy with the sample size; just range n from 20 to 130 with a step size of 2 or something like that. Your top-down should be a decent amount slower, although both are still slow as molasses compared to Dijkstra.

(c) We never mentioned when discussing Bellman-Ford that it was capable of returning shortest paths of a specific number of edge 'hops' (or if I did mention it, I didn't give any details). To show yourself how the original Bellman-Ford algorithm is really a bottom up DP, explain how you could modify it to return

shortest paths that traverse exactly i edges for a given i. (Hint: Our original Bellman-Ford performs it's outer loop $|V|$ many times. Can you identify a loop invariant?)