Dynamic Programming Problems - Hints Sheet

These are the sample problems which are intended to provide a hands-on introduction to solving problems using Dynamic Programming.

1. Coin Change Problem
2. Optimum Path Problem
3. Billboard Location Problem
4. Additional Problem - DNA Sequence Alignment

Problem 1 - Coin Change Problem

Objective: Given a amount A and n coins, $c_1 < c_2 < c_3 < \ldots < c_n$, write a program to calculate the minimum number of coins required to make the exact change for the amount 'A'.

Dynamic Solution: Start at zero and iterate up to the target amount 'A'. Using the available coins, calculate the minimum number of coins to go from a previous value that can be reached with 1 of the available coins.

For each value, v, in the range from 1 to A then you pick the
$f(0) = 0$
$f(v) = 1 + \min(f(v - c_i)_{for\ i=1\ to\ n})\ where\ c_n <= v$

Example:
This is a walk-through of a sequence of steps to illustrate how the final answer can be determined.
Given a set of coins with values of 1p, 2p, 4p, make the total of 5p.
$f(0) = 0$
$f(1) = 1 + \min(f(0)) = 1 + \min(0) = 1$ Only consider 1p coin.
$f(2) = 1 + \min(f(2-1), f(2-2)) = 1 + \min(f(1), f(0)) = 1$ Consider 1p & 2p coins.
$f(3) = 1 + \min(f(3-1), f(3-2)) = 1 + \min(f(2), f(1)) = 2$ Consider 1p & 2p coins.
$f(4) = 1 + \min(f(4-1), f(4-2), f(4-4)) = 1 + \min(f(3), f(2), f(0)) = 1$ Consider 1p, 2p & 4p coins
$f(5) = 1 + \min(f(5-1), f(5-2), f(5-4)) = 1 + \min(f(4), f(3), f(1)) = 2$

Consider 1p, 2p & 4p coins.

> Note: Don't forget to store the results of each value of $f(v)$ as v increases from 1 to A to mimimise the number of calculations and avoid recursion.

Problem 2 - Optimum Path Through Matrix

Objective: Given a 2-D matrix of positive integers, find the path from the top-left corner to the bottom-right of the matrix where the elements on the path have the smallest total sum. You are only allowed to move either right or down (i.e. can't move up, to the left. or diagonally)

This is a sample grid with the optimum path highlighted in blue.

8	7	4	2	3	5
3	2	1	2	5	2
2	4	5	3	3	6
6	4	5	4	3	2
1	3	4	5	6	3
7	3	2	1	3	5

$$Path_{opt} = 8 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow 3 \rightarrow 2 \rightarrow 3 \rightarrow 5 = 35$$

Hint: There are only 2 ways to get to the final square. Which option should be selected?

Dynamic Solution:
In order to reach the final cell, we have to come from either the cell immediately above it or from the one to it's immediate left. In order to minimise the total sore we would choose which of these 2 options had the lowest score to reach it. We can use this logic to pick the best route to the preceeding cell, and so on back to the top-left corner.

We treat the origin matrix as a grid (called a) of width w and height h with $(1, 1)$ at the top-left and (x, y) at the bottom-right. We can then build a parallel grid (called b) with the same dimensions that holds the minimum value to reach each location.

We take the smallest value from b immediately to the left or above the target cell and add it to the cell's value in the original grid, a.

So the general formula to calculate the best value for a cell in b looks like:

$$b(x,y) = a(x,y) + \min(b(x-1,y), b(x,y-1))$$

The only variations on this are in the first row where there is no row above and the first column where there is no column to the left. In these two cases you only have 1 possible preceding cell.

Worked Example:
Given the following $input$ matrix, calculate optimum path from top-left corner to bottom right:

1	**2**	**6**
3	**4**	**5**
9	**7**	**3**

Start by calculating the top row in the $result$ matrix as the accumulative sum of the cells

$$result(1,n) = \sum_{x=1}^{n} input(1,x)$$

So top row of $result$ = $[1,3,9]$

You can then use this first row to calculate the 2nd row in the $result$ matrix.
First element on the row r is given by

$$result(r,1) = input(r,1) + result(r-1,1)$$

Subsequent elements on the row:

$$result(r,n) = input(r,n) + \min(result(r-1,n), result(r,n-1))$$

So 2nd row in $result$ = $[4,7,12]$
and 3rd row in $result$ = $[13,14,15]$

$Path_{opt} = 1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3$

Problem 3 - Billboard Locations

Objective: Suppose you're managing construction of billboards on the M62 between Leeds & Manchester, a distance of M miles. The possible sites for advertising hoardings are given by numbers $x_1 < x_2 < \cdots < x_n$, each in the interval $[0, M]$, specifying their position in miles measured from Leeds. If you place a billboard at position x_i , you receive a revenue of $r_i > 0$.

Regulations imposed by the Department of Transport require that no two billboards be within n miles or less of each other (i.e. the distance between successive billboards must be at least $n + 1$ miles). You'd like to place billboards at a set of the sites so as to maximize your total revenue, subject to this restriction.

Calculate the maximum achievable revenue and the location of the billboards which generates this figure.

Dynamic Solution: For each location which has the potential of placing billboard there is a choice to either place or not place a billboard.
If we build a list of the maximum revenues at point d called $income_d$ starting with $income_0 = 0$
The decision will be determined using the following calculation:

$$income_d = \max(income_{d-1}, income_{d-(n+1)} + r_d))$$

NB If $d < n + 1$ then the 2nd term in the $\max()$ just becomes r_d

We can then build the list of $income$ out to the required distance.

Additional Problem - DNA Sequence Alignment

We want to find the best way to align 2 sequences of DNA to minimise the number of differences between corresponding characters. For each, character in the sequences you have the option of either using the given character or inserting a gap "$-$" which will allow the subsequent characters to align better.

For example, if we aligned the following 2 sequences **ACGGTC** and **ATCGCC** we could produce an optimal alignment as:

> **A-CGGTC**
> **ATCG-CC**

This would have an 'Edit Distance' of 3 since there are 3 positions (2, 5 & 6) where the characters in the 2 generated sequences are different. This distance is called the

Levenshtein Distance and forms the basis of the Needleman-Wunsch Algorithm which determines the distance between 2 DNA or Protein sequences in bioinformatics.

This problem should build upon your solution to the Optimum Path problem with the following difference that allows an additional option to move diagonally Left & Down in a single step. It uses an insertion cost of 1 for each gap inserted and a substitution cost of one if a diagonal move is made when the 2 characters are different.

Useful Accumulate Function

When using Dynamic Programming I have found that in many cases it is useful to have a function which fills a void between the typical $fold()$ and $reduce()$ functions.

```
import operator
def accumulate(iterable,
               firstFn=lambda x:x,
               subsequentFn=operator.add):
    it = iter(iterable)
    try:
        previous = firstFn(next(it))
    except StopIteration:
        return
    yield previous
    for item in it:
        previous = subsequentFn(item, previous)
        yield previous```
```

This function is supplied out-of-the-box in Python 3 and the code above is my implementation in Python 2.
The $accumulate()$ function takes 3 parameters:

1. An iterator containing the individual key items
2. A function to be applied to derive the initial value for 1st item in the iterator - Defaults to the identity function.
3. A function to be applied for each subsequent item in the iterator which takes the item value and the previous summary value to calculate the next value in the results list. This defaults to the addition function.

If the $accumate()$ function is invoked with a standard range then it will cumulatively create a running total
i.e. $accumulate(xrange(1,5)) = [1,3,6,10]$ (i.e. 1, 1+2, 1 + 2 +3, 1 + 2 + 3 + 4)

Supplying multiplier operation instead of default addition produces:
$accumulate(xrange(1,5), subsequentFn = operator.mult) = [1,2,6,24]$ (i.e. 1, 1*2, 1 * 2 * 3, 1 * 2 * 3 * 4)