University of Toronto at Scarborough
CSCC73 - Algorithm Design and Analysis

Assignment #4: Dynamic Programming 1

Due: Feb 11, 2023 at 11.59pm This exercise is worth 5% of your final grade.

Warning: Your electronic submission on MarkUs affirms that this exercise is your own work and no one else's, and is in accordance with the University of Toronto Code of Behaviour on Academic Matters, the Code of Student Conduct, and the guidelines for avoiding plagiarism in CSCC73. Late assignments will not be accepted. If you are working with a partner your partners' name must be listed on your assignment and you must sign up as a "group" on Gradescope. Recall you must not consult **any outside sources except your partner, textbook, TAs and instructor**.

1. With all your school work you're running out of time to complete simple tasks like laundry. Since you are a poor student, you realize that if you're going to outsource your laundry there are only two ways you can afford to do this. Either you ask your mom - who immediately tells you to "grow up" or you get together with friends to organize a group laundry service. You poll all your friends to determine who wants in on which weeks and how many loads they each have. From this you have a weekly schedule for n weeks where each value is a number of loads ℓ_i for week i. You research the two closest laundromats that offer pick-up and delivery. Laundromat L_1 and laundromat L_2 have different methods for charging for their services.

   - L_1 charges a fixed rate r per load (so it costs $r \cdot \ell_i$ to pick-up, wash and deliver a week's load ℓ_i.)
   - L_2 makes contracts for a fixed amount w per week, independent of the number of loads. However, contracts with company L_2, must be made in blocks of three consecutive weeks at a time.

   A *schedule*, for you and your friends laundry, is a choice of laundromat (L_1 or L_2) for each of the n weeks, with the restriction that laundromat L_2, whenever it is chosen, must be chosen for blocks of three contiguous weeks at a time. The *cost* of the schedule is the total amount paid to laundromat L_1 and L_2, according to the description above.

   Give a polynomial-time algorithm that (makes you the laundry hero and) takes a sequence of weekly load values $\ell_1, \ell_2, \ldots \ell_n$ and returns a *schedule* of minimum cost. Explain your algorithm and justify your recurrence relation and complexity. A formal proof is not necessary.

   Example Suppose $r = \$10$, $w = \$80$, and the sequence of values is

   $$5, 8, 10, 11, 9, 6, 7.$$

   Then the optimal schedule would be to choose laundromat L_1 for the first two weeks, then laundromat L_2 for a block of three consecutive weeks and then laundromat L_1 for the final three weeks.

   Solution.

   In order to find the optimal schedule over i weeks we will consider the smaller problems of optimal schedules over a shorter number of weeks.

   Notice that when considering the i^{th} week we can consider using the weekly service L_1 or the 3 week min service L_2. If the optimal solution uses the weekly service during week i then the optimal solution to the subproblem of the previous $i-1$ weeks plus the cost of week i gives us an optimal solution. If

using L_2 then we know that it must have been used for 3 weeks, so consider the optimal subproblem on the previous $i-3$ weeks and add the cost $3w$.

More formally, Let $OPT(i)$ denote the minimum cost of a solution for weeks 1 through i. In an optimal solution, we use either laundromat L_1 or L_2 for the i^{th} week. If we use laundromat L_1 we pay $r \cdot \ell_i$ and can consider the optimal schedule up until week $i-1$. If we use laundromat L_2 for week i then we pay $3w$ for this contract and laundry is take care of all the way up to and including week $i-2$; thus we can use the optimal schedule through week $i-3$ and then invoke this contract. This gives:

$$OPT(i) = \begin{cases} 0 & i = 0 \\ r\ell_i + OPT(i-1) & 1 \leq i \leq 2 \\ min(r\ell_i + OPT(i-1), 3w + OPT(i-3)) & i \geq 3 \end{cases}$$

We can build up these OPT values in order of increasing i, spending constant time per iteration with the initialization $OPT(i) = 0$ for $i \leq 0$. The desired value is $OPT(n)$ and we can get the schedule be tracing back through the array of OPT values. Complexity is therefore $\mathcal{O}(n)$.

2. Suppose that you are given a sequence S of price quotes for a particular stock. You wish to locate trends in the stock price over time. In particular, you wish to find the longest trend in S that is strictly increasing. Design an algorithm to find the *longest increasing trend* of S. For example, if the sequence S is

$$S = 0.45, 0.43, 0.42, 0.46, 0.47, 0.52, 0.49, 0.50, 0.51, 0.53, 0.48, 0.44$$

a longest increasing trend is 0.42, 0.46, 0.47, 0.49, 0.50, 0.51, 0.53 where as 0.45, 0.46, 0.47, 0.52, 0.53 is not.

Solution.

The subproblems we will use to solve our problem on n values $S = \{x_1, x_2, \ldots, x_n\}$ will be the optimal solution on $n-1$ values with a restriction on the largest value of the maximum element in the trend.

We define a dynamic programming solution to this problem by constructing a recurrence for the length of the longest increasing trend. To motivate this recurrence, consider the last number x_n in S. Either x_n is in the longest trend or it is not. If it is, then the optimal solution is the longest increasing trend in $x_1, \ldots, x_{n-1}$ that is strictly less than x_n. If x_n is not in the longest increasing trend, then the optimal solution is the longest increasing trend in $x_1, \ldots, x_{n-1}$.

We formalize this as follows:

- Let $Opt(i,k)$ be the optimal solution on $x_1, \ldots, x_i$ such that the trend is strictly less than k.
- If $i = 1$ then we have a single item and if its less than k we include it for a length of 1 or if at least k then we cannot include it.
- If x_i is larger than k we cannot include it so optimal solution must be $Opt(x_{i-1}, k)$.
- Otherwise, $x_i < k$ and so we have the option to either include it giving $Opt(x_{i-1}, x_i) + 1$ or not include it so we carry over the current upper limit of k for $Opt(x_{i-1}, k)$. Take the max of the two.

Then, we can define $Opt(i,k)$ as:

$$Opt(i,k) = \begin{cases} 0 & \text{if } i = 1, x_1 \geq k \\ 1 & \text{if } i = 1, x_1 < k \\ Opt(i-1,k) & \text{if } x_i \geq k \\ \max(Opt(i-1,k), 1 + Opt(i-1,x_i)) & \text{if } x_i < k \end{cases}$$

In the following algorithm, let $M[][]$ represent $Opt(i, k)$

```
Algorithm Longest_Inc_Trend(n, sequence X)
   for (j = 1; j ≤ n; j++)
      if (x_1 ≥ x_j)
         M[1][j] = 0;
      else
         M[1][j] = 1;
   for (i = 1; i ≤ n; i++)
      for (j = 1; j ≤ n; j++)
         if (x_i ≥ x_j )
            M[i][j] = M[i − 1][x_j];
         else
            M[i][j] = max(M[i − 1][x_j], 1 + M[i − 1][x_i])
   return (M);
```

The value of the longest trend is then one of the values of $M[n]$, i.e., in the bottom row. We can find it by walking the bottom row. The actual trend can be returned by tracing back through M. The complexity is simply the complexity of filling in the matrix M, or $\mathcal{O}(n^2)$.