CSC373— Algorithm Design, Analysis, and Complexity — Spring 2016

Solutions for Tutorial Exercise 3: Dynamic Programming, Part II

---

1. **Road Trip.** Suppose you are going on a long road trip, which starts at kilometer 0, and along the way there are n hotels. The k^{th} hotel is at d_k kilometers from the start, and the hotels have been sorted so that $0 < d_1 < d_2 < \ldots < d_n$. Let $p_k > 0$ denote the price (in dollars) of a room at the k^{th} hotel.

   Each night of the trip you must stop at one of the hotels. On the j^{th} night we denote the choice of the hotel at distance $d_{m(j)}$ by the index $m(j)$. This hotel must further down the road than the previous night, so $m(j) > m(j-1)$. Also, you must stop at the n^{th} hotel at the end of your trip.

   A travel plan $\{m(j)\}_{j=0}^{J}$ then specifies the hotels to stay at during each night of the trip. We include $m(0) = 0$ for convenience, but this does not denote a hotel stay. The constraints on $m(j)$ are

   $$m(0) = 0,\ m(j) < m(j+1) \text{ for } 0 \leq j < J, \text{ and } m(J) = n. \tag{1}$$

   You can assume $0 < J \leq n$.

   You would prefer to drive roughly 600 km per day, but you are somewhat flexible on this. However, due to the placement (or prices) of the hotels, driving exactly this distance isn't always appropriate. To model the trade-off between your preference on the daily driving distance and your preference towards cheaper hotels consider the penalty function $C(x) = (max[x-600, 0])^2 Q$ for driving a distance $x = d_{m(j)} - d_{m(j-1)}$ on day j. Here $Q > 0$ is a constant (with the units of dollars per km^2). Think of $C(x)$ as the term that allows a direct comparison between your discomfort in having to drive x kilometers with a difference in the price of hotels.

   You wish to plan your trip in such a way as to minimize the total cost

   $$\mathcal{O}(J) = \sum_{j=1}^{J} \left[C(d_{m(j)} - d_{m(j-1)}) + p_{m(j)}\right], \tag{2}$$

   where m must satisfy the constraints stated in equation (1).

   (a) Consider the following greedy algorithm. Suppose $j > 0$ and the greedy algorithm has chosen the first $(j-1)$ hotels, $m(0), \ldots, m(j-1)$. And suppose $m(j-1) < n$ (i.e., it hasn't yet planned the last hotel). Then the algorithm chooses the j^{th} hotel to be the $m(j)$ which minimizes $C(d_{m(j)} - d_{m(j-1)}) + p_{m(j)}$, subject to $m(j-1) < m(j) \leq n$. That is, this greedy algorithm chooses the next hotel to be the next cheapest single move further along the road from the current hotel $m(j-1)$. The next hotel index, $m(j)$, is selected without looking ahead at how the choice effects travel on subsequent days.

   Prove that this greedy algorithm does not always give the minimum cost travel plan.

   (b) Give an efficient algorithm that determines the minimum cost plan. That is, subject to the constraints in (1) with $1 \leq J \leq n$, compute a sequence $m(1), m(2), \ldots, m(J)$ with the minimum possible total cost $\mathcal{O}(J)$, as defined in (2).

   (c) **Optional** Implement your algorithm in part (b) (say in Python). Consider $Q = 5 \times 10^{-3}$ dollars/km^2. On a given example, describe what happens to the optimal solution when you increase or decrease the value of Q.

Solution for Road Trip Question

1a Suppose there are $n = 3$ hotels. Let $Q = 0.005$, $p_k = 100$ for each $k = 1, 2, 3$, and $d_1 = 400$, $d_2 = 600$, $d_3 = 700$. Then the greedy algorithm chooses $m(1) = 2$, and is then forced to choose $m(2) = 3$. The total cost is $0 + 100 + (max[100 - 600, 0])^2 Q + 100$, which is \$200. Doing the trip in one day costs $(700 - 600)^2 Q + 100$, which is \$150. Therefore the greedy algorithm is not optimal.

1b Define $\mathcal{P}(k, j)$ to be the optimal cost for travel from the beginning of the trip through the j^{th} day, given that you stay at the k^{th} hotel on the j^{th} night. Here $\mathcal{P}(k, j)$ includes the price of all the hotels so far,

including the k^{th} hotel, and it includes the modelled prices for driving possibly further than your preferred distance each day. For simplicity we include the beginning of the trip as $k = 0$, $j = 0$, and $\mathcal{P}(0, 0) = 0$.

Since there are only n hotels and you must move forward each day, then on day j we must be at least at the j^{th} hotel, so we can assume $n \geq k \geq j$. Therefore we consider a $(n+1) \times (n+1)$ table of costs $\mathcal{P}(k, j)$, with

$$\mathcal{P}(k, j) = \infty \text{ for } 0 \leq k < j. \tag{3}$$

The condition that we start at distance 0 on the first day is represented by

$$\mathcal{P}(0, 0) = 0, \text{ and } \mathcal{P}(k, 0) = \infty \text{ for } 1 \leq k \leq n. \tag{4}$$

Consider day $j > 0$ with $j \leq n$, and suppose we stay at hotel k that night. The minimal cost to arrive at this state is defined to be $\mathcal{P}(k, j)$. Suppose we know the optimal costs $\mathcal{P}(k', j')$ for $(k', j') < (k, j)$ (using a lexical order). We wish to find an expression for $\mathcal{P}(k, j)$ in terms of these simpler sub-problems $\mathcal{P}(k', j')$

On the previous night, the $(j-1)^{st}$, we must have stayed at some hotel i with $j - 1 \leq i < k$. From above, the optimal cost of the first $j - 1$ days travel is then $\mathcal{P}(i, j - 1)$, and the cost of the j^{th} day's travel and accommodation, is $C(d_k - d_i) + p_k$. Therefore the minimimum cost of having this $(j-1)^{st}$ night at hotel i is $\mathcal{P}(i, j - 1) + C(d_k - d_i) + p_k$. In order to minimize $\mathcal{P}(k, j)$ we should choose a hotel i which minimizes this cost, and therefore

$$\mathcal{P}(k, j) \;=\; \min_{j-1 \leq i < k} \left[\mathcal{P}(i, j - 1) + C(d_k - d_i) + p_k\right]. \tag{5}$$

We can solve for $\mathcal{P}(k, j)$ by using the equatons (3), (4), and successively evaluating (5) by iterating through an outer loop over hotel locations $k = 1, 2, \ldots, n$ and an inner loop over days $j = k, k + 1, \ldots, n$.

It follows that the optimum number of days is therefore

$$J = \arg \min_{1 \leq J \leq n} \mathcal{P}(n, J), \tag{6}$$

and the minimum cost of the trip is $\mathcal{O}(J) = \mathcal{P}(n, J)$ for this J.

In order to recover the sequence of hotels, we begin by setting $m(J) = n$. For $j = J, J - 1, \ldots, 1$ (in decreasing order), we set $m(j - 1)$ to be the smallest i such that the minimum is achieved in equation (5) with $k = m(j)$. That is, $m(j - 1)$ equals the smallest index i such that

$$\mathcal{P}(m(j), j) \;=\; \mathcal{P}(i, j - 1) + C(d_{m(j)} - d_i) + p_{m(j)}. \tag{7}$$

Such an i must exist by construction.

1c We skip the implementation. Increasing Q causes the optimum solution to have the mean daily driving distance closer to 600 km, where that is possible by the hotel locations. The variation in the cost of the hotels becomes increasingly irrelevant. When $Q > 0$ is decreased the optimum solution has more flexibility to choose the cheapest hotels and/or to reduce the number of days of travel. Eventually, when Q is nearly zero, the optimum solution consists of driving the whole way to the n^{th} hotel in one "day" (i.e., $J = 1$ and $m(1) = n$).

2. **Substring Counting.** Given a pattern string $X = (x(1), x(2), \ldots, x(n))$, with individual characters $x(i)$, we wish to find how many times this pattern appears in a second string $Y = (y(1), y(2), \ldots, y(m))$. In order to be a match, all the characters in the pattern X must appear in the same left to right order in Y, but they need not be successive characters in Y. We wish to count the number of distinct matches of this type. (In the notation of string regular expressions, we're talking about matching the pattern `[x(1).*x(2) ... .* x(n)]`.)

   More precisely, each particular match is specified by an ordered set of indices $I = (i(1), i(2), \ldots, i(n))$ where $1 \leq i(1) < i_2 < \ldots < i_n \leq m$ and $x(k) = y(i(k))$ for each $k \in \{1, 2, \ldots, n\}$. We consider two matches different if the corresponding n-tuples $I_1 = (i_1(1), \ldots, i_1(n))$ and $I_2 = (i_2(1), \ldots, i_2(n))$ are different, that is, $i_1(k) \neq i_2(k)$ for at least one index k.

   For example, given $X =$ '$algor$' and $Y =$ '$aaalllgggooorrraaa$', the number of matches is 3^5 since each of the five characters in X can match any one of the first three appearances of the same character in Y. Note we need complete matches of the pattern string so, for example, the last three characters in the Y for this example cannot be part of any match.

   Alternatively, for $X =$ '$algor$' and $Y =$ '$jaalggororot$', one possible match is $I_1 = (2, 4, 5, 7, 8)$ and another is $I_2 = (2, 4, 6, 7, 8)$. The total number of possible matches in this case turns out to be 12 (although you probably want to use your algorithm to check this).

   Describe a dynamic programming approach which computes the maximum number of different matches for any two input strings X (the pattern) and Y. (Note you do not need to find each of these matches, just count them.)

2. **Solution for Substring Counting Question**

   Consider the sub-problem, indexed by (j, k), which consists counting the number of ways the prefix string $X_j = (x(1), \ldots, x(j))$ matches with Y and where the last character in this pattern matches with the k^{th} character in Y (that is, $x(j) = y(k)$ and $i(j) = k$). We will use $C(j, k)$ to denote the number of these matches for each of these sub-problems, for $j = 0, 1, \ldots, n$ and $k = 1, 2, \ldots, m$. We define that $C(0, k) = 0$ for each k.

   We imagine solving for $C(j, k)$ recursively. Consider any match which satisfies the conditions of the (j, k) sub-problem. Since $x(j)$ must equal $y(k)$ and $i(j) = k$ then we can append this individual character match to each of the shorter pattern matches that match the prefix $(x(1), \ldots, x(j-1))$ to the prefix $(y(1), \ldots, y(k-1))$. (Note we enforce such a prefix match to actually end before the k^{th} character in Y.) Suppose it ends at the m^{th} character, so $x(j-1) = y(m)$, $i(j-1) = m$ and $m < k$. The number of different ways to find a submatch of exactly this form is then $C(j-1, m)$. Since this is true for each $m < k$ at which there is a match, and all of these must be different matches, it follows that

$$C(j,k) = \begin{cases} \sum_{m<k \ | \ y(m)=x(j-1)} C(j-1, m), & \text{if } x(j) = y(k), \\ 0, & \text{if } x(j) \neq y(k). \end{cases} \tag{8}$$

   The sum above is equivalent to simply $\sum_{m<k} C(j-1, m)$, where we have used the fact that $C(j-1, m) = 0$ whenever $x(j-1) \neq y(m)$.

   Recall that $C(n, k)$ is the number of matches for the sub-problem in which the string $X_n = (x(1), \ldots, x(n)) = X$ matches with Y **and** where the last character in this pattern matches with the k^{th} character in Y (that is, $x(j) = y(k)$ and $i(j) = k$). To find the total number of matches we need to sum over all possible end positions k. The total number of matches is therefore $N = \sum_{k=1}^{m} C(n, k)$.

3. **Word Segmentation.** This is problem 5, p.316 of the Kleinberg and Tardos text.

Suppose you are given character strings formed from English words, but which have had all the spaces and punctuation removed. For example, you might be given 'meetateight', which you probably interpret as the three words 'meet at eight'. Consider the problem of segmenting these individual words from such a string without spaces or punctuation. (Some written languages don't use spaces.)

More formally, given a long string $S = (y(1), \ldots, y(n))$ the problem is to find a set of locations (or break-points) i_k at which to break this string S into words. That is, you need to find i_k, for $k = 0, 1, \ldots, K$, such that $0 = i_0 < i_1 < i_2 < \ldots < i_K = n$, such that these locations break the original string S into the K individual words

$$S_k = (y(i_{k-1}), y(i_{k-1}+1), \ldots, y(i_k)), \text{ for } k = 1, 2, \ldots, K. \tag{9}$$

Note we wish to find both the number of words K and the break-points $\{i_k\}_{k=0}^K$. For example, if $S =$ 'meetateight' then the corresponding breakpoints would be $i_1 = 4$, $i_2 = 6$ and $i_3 = 11$.

In order to begin you will need to have access to something like an English dictionary. Suppose you are given a function `quality`$(\ X)$ which accepts any input string $X = (x(1), x(2), \ldots, x(m))$ and returns a number that indicates how plausible the string X is in terms of a single English word. That is, if X corresponds to exactly an English word then `quality`$(\ S)$ will be large, and if X is far from any English word then `quality`$(\ S)$ will be small (and possibly negative). (Assume the empty string of length zero has a negative quality, i.e., `quality`$(\prime\prime) < 0$.)

Your problem is then as follows. Given any string S, find a sorted set of breakpoints $\{i_k\}_{k=0}^K$, with $i_0 = 0$ and $i_K = |S|$, which maximizes the sum of the qualities of the segmented words, that is, maximizes

$$Q(\{i_k\}_{k=0}^K) \equiv \sum_{k=1}^{K} \texttt{quality}(S_k). \tag{10}$$

(a) Derive a dynamical programming solution to determine the maximum possible quality of any segmentation. That is, find the value $\bar{Q}$ where

$$\bar{Q} = \max\{Q(\{i_k\}_{k=0}^K) \mid 0 = i_0 < i_1 \ldots < i_K = |S|\}. \tag{11}$$

Note in this part you only need to compute the maximum quality $\bar{Q}$, not corresponding break-points $\{i_k\}_{k=0}^K$.

(b) What is the run-time of your algorithm for (a), assuming the calls to `quality`$(\ S)$ run in $O(1)$ time?

(c) Modify your algorithm in part (a) (if necessary) so that a maximum quality segmentation $\{i_k\}_{k=0}^K$ can be computing without the need for any additional calls to the funciton `quality`$(\ S)$ (beyond what is already needed to compute $\bar{Q}$ in part (a)).

3. Solution for Word Segmentation Question

3a Given the string $(y(1), \ldots, y(n))$ and the function quality(S), let $OPT(j)$ denote the optimum quality of possible segmentations for the prefix string $S_j = (y(1), \ldots, y(j))$.

For $j > 1$ consider a segmentation of S_j with the optimum quality $OPT(j)$. Consider the last word used in this segmentation, consisting of the sub-string $S_{i,j} = (y(i), \ldots, y(j))$.

Case i, for $1 < i \leq j$:
Since $S_{i,j}$ is the last word in the segmentation, then

$$OPT(j) \geq T(S_{i-1}) + \text{quality}(S_{i,j}), \tag{12}$$

where $T(S_{i-1})$ is the quality of some segmentation of the prefix string S_{i-1}. We see from equation (12) that in order for $OPT(j)$ to be the maximum quality, $T(S_{i-1})$ must actually be the optimal quality of any segmentation of the prefix string S_{i-1}. That is, we can replace $T(S_{i-1})$ by $OPT(i-1)$. Therefore

$$OPT(j) \geq OPT(i-1) + \text{quality}(S_{i,j}) \tag{13}$$

for each i with $1 < i \leq j$.

Case $i = 1$ and $j \geq 1$:
When $i = 1$ the last segment $S_{i,j}$ is actually the whole prefix string S_j. Therefore

$$OPT(j) = \text{quality}(S_{1,j}). \tag{14}$$

Note this case also applies for prefix strings of length 1, i.e., $j = 1$.

Since these are all the possible cases, $OPT(j)$ must be given by the maximum over them. We therefore have

$$OPT(j) = \max_{1 \leq i \leq n} \left[OPT(i-1) + \text{quality}(S_{i,j}) \right], \tag{15}$$

where we define $OPT(0) = 0$.

```
[qMax, p] = optSegQual(y, n)
    // Input y = (y(1), ... y(n)) the string
              n – the length of the string
    // Output: qMax the maximum quality of any segmentation of y
    //           p a n-array of indicies to the beginning of the last word.

    Allocate q, an array of length n. // q(j) will store OPT(j).
    Allocate p, an array of length n. // p(j) will store the starting index of a word that
                                      // ends at character j, i.e., the word Y(p(j),...,j).

    // Prefix string of length 1 has a unique segmentation
    q(1) = quality(y(1))
    p(1) = 1

    // Loop over increasing lengths of the prefix string
    for j = 2..n
        q(j) = quality(y(1..j)) // Include the one word segmentation.
        for i = 2..j            // Find the best quality of multi-word segmentations
            q(j) = max( q(j), q(i-1) + quality(y(i..j)))
            If q(j) was increased in the line above, set p(j) = i
        end
    end

    return q(n), p(·).
```

3b This algorithm runs in $O(n^2)$ time, where $n = |y|$, with $O(n^2)$ calls to `quality(S)`.

3c Return some bread crumbs $p(\cdot)$ along with the optimization table $qMax$ from the function `optSegQual(y, n)` above. The index $p(j)$ stores the index at the beginning of the last word in the best segmentation of the prefix $S_j = (y(1), \ldots, y(j))$. Specifically, $p(n)$ is the beginning of the last word in a best-quality segmentation. If $p(n) > 1$ then the second last word ends at $j = p(n) - 1$ and it begins at $p(j)$. By continuing this until $p(j) = 1$, each of the words in an optimal segmentation can be recovered along with the number of words.