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

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

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

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

   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 $m(j-1) < m(j) \le n$. 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 $1 \le J \le n$, 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.

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 $1 \le i(1) < i_2 < \ldots < i_n \le m$ 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.)

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

   $$S_k = (y(i_{k-1}), y(i_{k-1} + 1), \ldots, y(i_k)), \text{ for } k = 1, 2, \ldots, K. \tag{3}$$

   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`$('') < 0$.)

   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

   $$Q(\{i_k\}_{k=0}^K) \equiv \sum_{k=1}^{K} \texttt{quality}(S_k). \tag{4}$$

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

   $$\bar{Q} = \max\{Q(\{i_k\}_{k=0}^K) \mid 0 = i_0 < i_1 \ldots < i_K = |S|\}. \tag{5}$$

   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)).