COMS21103: Problem set 4

2015/2016

Remark: Most problem sets for the next few weeks will contain at least one starred problem, which is more challenging. If any of the problems seem unclear, please post a question on the Blackboard discussion board.

1. Let A be an array containing n integers. A subarray of A is a contiguous subset of elements of A. The maximum subarray sum MSS is defined to be the maximum, over all subarrays of A, of the sum of the elements in the subarray. For example, if $A = (1, -2, 3, -1, 2)$, $MSS = 4$, achieved by the subarray $(3, -1, 2)$.

   (a) Let $MSS(j)$ be the maximum subarray sum that can be achieved by any subarray of A that finishes at position j. Give a recurrence expressing $MSS(j)$ in terms of $MSS(j-1)$.

   (b) Use your recurrence to give a memoized recursive dynamic programming algorithm for computing MSS. Your algorithm should run in time $O(n)$.

   (c) Give an iterative algorithm which computes MSS without any recursive calls.

   Answer (sketch): For $j \geq 1$, $MSS(j) = \max\{MSS(j-1)) + A[j], A[j]\}$. For $j = 0$, $MSS(0) = 0$. The memorized algorithm follows by writing down the recurrence (and performing the standard memoization). The iterative algorithm should compute $MSS(j)$ in increasing order of j. The final step in both algorithms is to compute MSS as the maximum of all $MSS(j)$

2. Suppose that you are in charge of placing posters for the next CSS social on the n lamp posts along Woodland Road. The lampposts are conveniently numbered 1 to n. Each lamppost has a distance x_i (measured in meters from the bottom of Woodland Road) and a value v_i (based on the number of student who will see that poster). Your goal is to plan the placement of the posters to maximise the total value, denoted Val. This is the sum of the values of the lampposts that you put posters on. While there is no explicit limit on the number of posters you can place, according to Council regulations, you cannot place two posters within 10 meters of each other.

   (a) Let $Val(i)$ be the maximum total value that can be achieved by only considering the first i lampposts. Give a recurrence expressing $Val(i)$ in terms of smaller subproblems.

   (b) Use your recurrence to give a memoized recursive dynamic programming algorithm for computing Val. Your algorithm should run in time $O(n)$.

   (c) Give a iterative algorithm which computes Val without any recursive calls.

   Answer (sketch): The 'cheat' way to solve this problem is to see it as a special case of the weighted interval scheduling problem. Set up an input for the weighted interval scheduling problem as follows: You have one interval for each lamppost with start time at $x_i - 5$ and

it's finish time at $x_i + 5$. The weight of the interval is v_i. This gives a solution which runs in $O(n \log n)$ time (by the algorithm given in lectures).

This does not answer the question however, which asks for a recurrence to be given explicitly. However, considering the approach in lectures for the weighted interval scheduling problem can help us here. To compute $Val(i)$ we consider two cases: either we include a poster on lamppost i or we don't. In the latter case, $Val(i) = Val(i-1)$. In the former case, we cannot include any lamppost $j < i$ with $x_j > x_i - 10$. Let $p(i)$ be the largest such j. In the former case, $Val(i) = v_i + Val(p(i))$. The recurrence is then $Val(i) = \max(Val(i-1), Val(p(i)) + v_i)$.

However, we still need to compute all the $p(i)$ values. We could use the binary search approach from the lecture on weighted interval scheduling problem. However, this will take $O(n \log n)$ time. This is improved to $O(n)$ time by observing that the $p(i)$ values are non-decreasing i.e. $p(i+1) \geq p(i)$. We can use this to compute them by scanning the x_i values (in increasing i order). More specifically $p(1) = 0$. We then compute $p(i)$ by considering lampposts $j = p(i-1), (p(i-1)+1), (p(i-1)+2) \ldots$ until we find the first lampost j such that $x_{j+1} > x_i - 10$. We then have by definition the $p(i) = j$. The time taken to determine $p(i)$ is $O(p(i) - p(i-1) + 1)$. The time to compute all $p(i)$ is therefore $\sum_i O(p(i) - p(i-1) + 1) = O(p(n) + n) = O(n)$.

The memorized algorithm follows by writing down the recurrence (and performing the standard memoization). The iterative algorithm should compute $Val(i)$ in increasing order of i.

3. A sequence is a palindrome if it reads the same forwards as backwards. For example the sequence (A, B, C, C, B, A) is a palindrome. The following sequences are also palindromes (A, B, A), (D, E, E, D), and (A). The sequence (A, B, C) is not a palindrome.

   Given a sequence S (of length n), the palindromic subsequence problem is to find the length of the longest subsequence of S which is a palindrome. For example in the sequence $S = (D, A, B, C, A)$ the longest palindromic subsequence is (A, B, A) so the output should be 3.

   Notation: A subsequence of S is any sequence which can be obtained from S by deleting elements (but not rearranging them). The sequences (A, A) and (D, C, A) are both subsequences of (D, A, B, C, A). The sequence (C, B) is not a subsequence of (D, A, B, C, A).

   (a) Formulate a recurrence for the palindromic subsequence problem.

   (b) Use your recurrence to give a memoized recursive dynamic programming algorithm for the palindromic subsequence problem.

   (c) What is the time complexity of your memoized algorithm?

   (d) Give a iterative algorithm which solves the palindromic subsequence problem without any recursive calls.

   (e) What is the time complexity of your iterative algorithm?

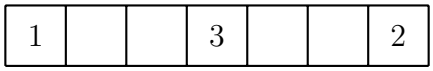
   Answer (sketch): Let $P(i, j)$ denote the length of the longest palindromic subsequence of sequence $S[i, j]$. Here $S[i, j]$ is the sequence formed from the i-th element of S upto and including the j-th element of S. Therefore $S[1, n] = S$ and $P(1, n)$ is the output to the problem. First observe that for any i, $P(i, i) = 1$. A suitable recurrence is given by,

$$P(i,j) = \begin{cases} P(i+1, j-1) + 2 & \text{if } S[i] = S[j] \\ \max\{P(i, j-1), P(i+1, j)\} & \text{otherwise} \end{cases}$$

The memorized algorithm follows by writing down the recurrence (and performing the standard memoization). The answer is then $P(1, n)$. The memoized algorithm runs in $O(n^2)$ time. The iterative algorithm should compute $P(i, j)$ in "length order". That is for each $\ell \in [1, n]$ (in increasing order) it computes $P(i, j)$ for all $i - j = \ell$. Observe that this preserves dependencies. The iterative algorithm also runs in $O(n^2)$ time.

4. ($\star$) The Solitary Drinking Problem is defined as follows. There are n seats in a line at a bar, and a sequence of drinkers comes in to use them. The first drinker sits in seat 1. All subsequent drinkers sit in a seat which is as far as possible from anyone else (if there is more than one, picking the lowest-numbered one). To avoid having to make small talk, drinkers do not want to sit next to anyone else. When this can no longer be achieved, any new drinkers give up and leave.

   For example, with 7 seats and numbering drinkers in order of appearance, the situation at the end of this process, when no more drinkers can sit down, is as follows:



   For a bar with n seats, let $B(n)$ be the number of filled seats at the end of this process. The first values of $B(n)$, starting from $n = 1$, are $1, 1, 2, 2, 3, 3, 3, \ldots$

   Give a dynamic programming algorithm which efficiently computes $B(n)$. Code up your algorithm and use it to compute $B(1000000)$.

   Answer (sketch): One way to look at this is as follows. Let $B'(n)$ be the number of filled seats in a block of n initially empty seats, in the modified setting where we assume there are drinkers already sitting either side of the block, and the next drinker wants to sit as close to the middle of the block as possible. Then $B'(1) = B'(2) = 0$, and $B'(n) = B'(\lceil (n-1)/2 \rceil) + B'(\lfloor (n-1)/2 \rfloor) + 1$ for $n \geq 3$. Finally, $B(n) = B'(n-2) + 2$. It turns out that $B(1000000) = 475712$.

   As a bonus question, you might wonder how long it takes to compute $B(n)$ by this approach? In particular, which how many (different) values of $B'(i)$ are computed?

For more practice, there are many more dynamic programming exercises by Jeff Erickson online at `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/05-dynprog.pdf` (I would start with question 3a).