CSCI-GA.1170-001/002 Fundamental Algorithms November 10, 2015

Solutions to Problem 1 of Homework 7 (14 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, November 10*

Suppose you are given an array $A[1, \ldots, n]$ of numbers, which may be positive, negative, or zero.

(a) (4 points) Let $S_{i,j}$ denote $A[i] + A[i+1] + \cdots + A[j]$. Use dynamic programming to give an $O(n^2)$ algorithm to compute $S_{i,j}$ for all $1 \leq i \leq j \leq n$, and hence compute $\max_{i,j} S_{i,j}$.

Solution:

$$S[i,j] = \begin{cases} A[i] & \text{if } i = j \\ S[i,j-1] + A[j] & \text{if } i < j \end{cases}$$

Algorithm: $\textsc{MaxSumSubArray}(A)$
$S \leftarrow \textsc{NewArray}(n \times n)$
$max \leftarrow -\infty$
for $i \leftarrow$ *1 to* n **do**
  $S[i,i] = A[i]$
  if $S[i,i] > max$ **then**
    $max \leftarrow S[i,i]$
    $max_i \leftarrow i$, $max_j \leftarrow i$
end
for $i \leftarrow$ *1 to* n **do**
  for $j \leftarrow i{+}1$ *to* n **do**
    $S[i,j] = S[i,j-1] + A[j]$
    if $S[i,i] > max$ **then**
      $max \leftarrow S[i,j]$
      $max_i \leftarrow i$, $max_j \leftarrow j$
  end
end
Return (max, max_i, max_j)

Algorithm 1: Dynamic Programming Algorithm to calculate maximum sum sub-array in $O(n^2)$ time

From the above two *for* loops it is clear that the algorithm takes $O(n+n-1+\ldots+1) = O(n^2)$ time □

(b) (6 points) Let $L[j]$ denotes $\max_{i \leq j} S_{i,j}$. Give a recurrence relation for $L[j]$ in terms of $L[1, \ldots, j-1]$. Use your recurrence relation to give an $O(n)$ time dynamic programming algorithm to compute $L[1 \ldots n]$, and hence compute $\max_{i,j} S_{i,j}$.

Solution:

$$L[j] = \begin{cases} A[1] & \text{if } j = 1 \\ A[j] & \text{if } L[j-1] \leq 0 \\ L[j-1] + A[j] & \text{if } L[j-1] > 0 \end{cases}$$

Algorithm: MaxSumSubArray(A)
$L \leftarrow$ NewArray(n)
$L[1] \leftarrow A[1]$
$max \leftarrow L[1]$
for $i \leftarrow 2$ *to* n **do**
    if $L[i\text{-}1] \leq 0$ **then**
        $L[i] = A[i]$
    else
        $L[i] = L[i-1] + A[i]$
    if $L[i] > max$ **then**
        $max \leftarrow L[i]$
end
Return max

Algorithm 2: Dynamic Programming Algorithm to calculate maximum sum sub-array in $O(n)$ time

In the above algorithm, there is only one *for* loop that runs from 2 to n. Hence the running time is $O(n)$ □

(c) (2 points) Assume you use recursion (without memorization) to compute the answers to part(a) and part(b). Will both running times stay at $O(n^2)$ and $O(n)$, respectively, only one of them (which one?), or none?

Solution:

Part(a)

If we don't use the memory, then

Calculating the elements of the first row of the matrix takes $1 + 2 + \ldots + n = O(n^2)$ time

Calculating the elements of the second row of the matrix takes $1+2+\ldots+n-1 = O((n-1)^2)$ time

$\vdots$

Calculating the elements of the last row of the matrix takes $1 + 2 + \ldots + n - 1 = O(1^2)$ time

Therefore, total time taken to calculate the matrix elements without memorization will be $O(1^2 + 2^2 + \ldots + (n-1)^2 + n^2) = O(n^3)$. The run time increases by a factor of n

Part(b)

The running time will still be $O(n)$ without memorization using recursion. The maximum-sum-subarray of $A[1 \ldots j]$ makes a recursive call to maximum-sum-subarray of $A[1 \ldots j-1]$ which returns two fields, $L[j-1]$ which will be used to evaluate $L[j]$ and also the maximum sum sub array of $A[1 \ldots j-1]$. This maximum is compared to $L[j]$ and will be updated to $L[j]$ if it's lesser and return this maximum at the end. Thus the running time is sill $O(n)$ without using memorization $\square$

(d) (4 points) Suggest appropriate modifications to your algorithm in part (b) to give an $O(n)$ algorithm to compute $\max_{i,j} P_{i,j}$, where $P_{i,j} = A[i] \cdot A[i+1] \cdots A[j]$. Assume that multiplication of any two numbers takes $O(1)$ time.

Solution:

Let $L[j]$ denotes $\max_{i \le j} P_{i,j}$, then

$$L[j] = \begin{cases} A[1] & \text{if } j = 1 \\ S[j-1]A[j] & \text{if } (L[j-1] \le 0 \text{ and } A[j] \le 0) \text{ or } (L[j-1] \ge 0 \text{ and } A[j] \ge 0) \\ A[j] & \text{if } (L[j-1] > 0 \text{ and } A[j] < 0) \text{ or } (L[j-1] < 0 \text{ and } A[j] > 0) \end{cases}$$

Algorithm: MaxProductSubArray(A)
$L \leftarrow$ NewArray(n)
$L[1] \leftarrow A[1]$
$max \leftarrow L[1]$
for $j \leftarrow 2$ *to* n **do**
  if $(L[j-1] \le 0$ *and* $A[j] \le 0)$ *or* $(L[j-1] \ge 0$ *and* $A[j] \ge 0)$ **then**
    $L[j] = L[j-1]A[j]$
  else if $(L[j-1] > 0$ *and* $A[j] < 0)$ *or* $(L[j-1] < 0$ *and* $A[j] > 0)$ **then**
    $L[j] = A[j]$
  if $L[j] > max$ **then**
    $max \leftarrow L[j]$
end
Return max

Algorithm 3: Dynamic Programming Algorithm to calculate maximum product sub-array in $O(n)$ time

Assuming that the multiplication of any two numbers takes $O(1)$ time, the running time of the above algorithm is clearly $O(n)$ □