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:

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

```
1  Algorithm: MaxSumSubArray(A)
2  S ← NewArray(n × n)
3  max ← −∞
4  for i ← 1 to n do
5  |  S[i,i] = A[i]
6  |  if S[i,i] > max then
7  |  |  max ← S[i,i]
8  |  |  max_i ← i, max_j ← i
9  end
10 for i ← 1 to n do
11 |  for j ← i+1 to n do
12 |  |  S[i,j] = S[i,j−1] + A[j]
13 |  |  if S[i,i] > max then
14 |  |  |  max ← S[i,j]
15 |  |  |  max_i ← i, max_j ← j
16 |  end
17 end
18 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:

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

```
1  Algorithm: MaxSumSubArray(A)
2  L ← NewArray(n)
3  L[1] ← A[1]
4  max ← L[1]
5  for i ← 2 to n do
6  |  if L[i-1] ≤ 0 then
7  |  |  L[i] = A[i]
8  |  else
9  |  |  L[i] = L[i − 1] + A[i]
10 |  if L[i] > max then
11 |  |  max ← L[i]
12 end
13 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 $A[1 \dots j]$ makes a recursive call to maximum-sum-subarray of $A[1 \dots j-1]$ which returns two fields, $L[j-1]$ which will be used to evaluate $L[j]$ and also the maximum sum sub array of $A[1 \dots j-1]$. 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 $\max_{i \leq j} P_{i,j}$, then

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

```
1  Algorithm: MaxProductSubArray(A)
2  L ← NewArray(n)
3  L[1] ← A[1]
4  max ← L[1]
5  for j ← 2 to n do
6  |  if (L[j − 1] ≤ 0 and A[j] ≤ 0) or (L[j − 1] ≥ 0 and A[j] ≥ 0) then
7  |  |  L[j] = L[j − 1]A[j]
8  |  else if (L[j − 1] > 0 and A[j] < 0) or (L[j − 1] < 0 and A[j] > 0) then
9  |  |  L[j] = A[j]
10 |  if L[j] > max then
11 |  |  max ← L[j]
12 end
13 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 $O(n)$ □

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

Solutions to Problem 2 of Homework 7 (8 points)

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

Assume that you have an n by n checkerboard. You must move a checker from the bottom left corner (position $(1,1)$) square the board to the top right corner (position (n,n)) square. In each step you may either

- move the checker up one square, or
- move the checker diagonally one square up and to the right, or
- move the checker right one square.

If you move a checker from square $x = (i,j)$ to square $y = (i',j')$ you get $p(x,y)$ dollars. You are told all of the $p(x,y)$ a priori. The $p(x,y)$ may be negative, zero or positive. You want to get as much money as possible.

(a) (4 pts) Let $M[i,j]$ be the highest profit you can collect from position $(1,1)$ to (i,j). Write a dynamic programming recurrence relation for $M[i,j]$ (do not forget the initial condition). Based on this recurrence relation, analyze the running time of the dynamic programming algorithm to compute $M[n,n]$.

Solution:

The best way to get to any square (i,j) is the best way to get from the square to it's left or the from the square to it's bottom or the from the diagonal square. Therefore,

$$\begin{aligned} M[i,j] = max\{&M[i,j-1] + p((i,j-1),(i,j)),\\ &M[i-1,j-1] + p((i-1,j-1),(i,j)),\\ &M[i-1,j] + p((i-1,j),(i,j))\} \end{aligned}$$

Note that the base case is $M[1,1] = 0$ and in the above equation, there will be no diagonal and bottom square for all the squares in the first row and there will be no diagonal and left squares for all the squares in the first column.

To fill an entry in the matrix M we look the adjacent three entries and appropriately fill the corresponding entry. Therefore, the total time taken to build this matrix is $\leq 3n^2 = O(n^2)$

□

(a) (4 pts) Give the "bottom-up" pseudocode for an efficient procedure CheckerBoard(n) for computing $M[n, n]$.

Solution:

```
1  Algorithm: CheckerBoard(n)
2  M ← NewArray(n × n)
3  M[1, 1] ← 0
4  for i ← 2 to n do
5  |  M[i, 1] ← M[i − 1, 1] + p((i − 1, 1), (i, 1))
6  end
7  for j ← 2 to n do
8  |  M[1, j] ← M[1, j − 1] + p((1, j − 1), (1, j))
9  end
10 for i ← 2 to n do
11 |  for j ← 2 to n do
12 |  |  max ← M[i, j − 1] + p((i, j − 1), (i, j))
13 |  |  if M[i − 1, j − 1] + p((i − 1, j − 1), (i, j)) > max then
14 |  |  |  max ← M[i − 1, j − 1] + p((i − 1, j − 1), (i, j))
15 |  |  else if M[i − 1, j] + p((i − 1, j), (i, j)) > max then
16 |  |  |  max ← M[i − 1, j] + p((i − 1, j), (i, j))
17 |  |  max ← M[i, j]
18 |  end
19 end
20 Return M[n, n]
```

Algorithm 4: Dynamic Programming Algorithm to calculate maximum profit in checkerboard in $O(n^2)$ time

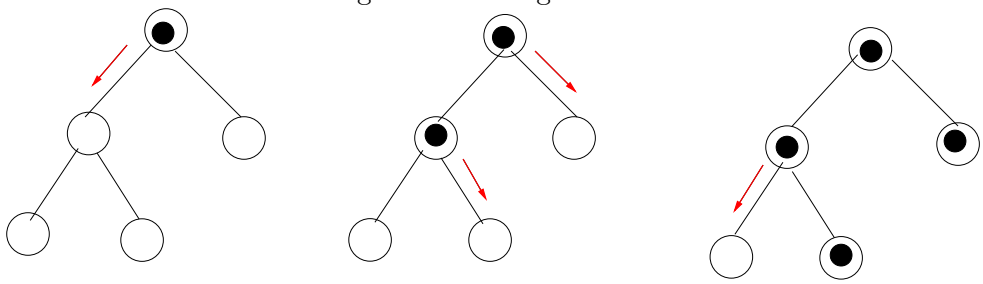
From the above psuedocode, it is clear that the dynamic programming algorithm runs in $O(n^2)$ time □

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

Solutions to Problem 3 of Homework 7 (8 (+6) points)

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

Suppose we need to distribute a message to all the nodes in a rooted (not necessarily binary) tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. For example, the minimum number of rounds it takes to distribute the message in the tree given below is 3.



Note that the order in which the messages are distributed matters. For example, in the above tree, if the root node sends the message to the right child in the first round, then the number of rounds will be 4.

Assume that a tree T is given with nodes labeled $\{0, 1, 2, \ldots, n-1\}$ and the node 0 is the root of the tree. Further there is a two-dimensional $n \times n$ array $Child[][]$, where $k = Child[i][0]$ is the number of children of the node labeled i and $Child[i][1], Child[i][2], \ldots, Child[i][k]$ denote the labels corresponding to the children of node i. The remaining entries in the array are -1.

(a) (4 points) Let ROUNDS(i) be a function computing the minimum number of rounds it takes to distribute the message from node i to all nodes in the subtree rooted at i. Give a recursive formula to compute ROUNDS(i) as a function of ROUNDS(j_1), ..., ROUNDS(j_k), where $j_1, \ldots, j_k$ denote the children of node i.

HINT: What is the order in which each of the children get the message?

Solution:

For any leaf node i, we have

$$Rounds(i) = 0$$

For any non-leaf node i, let $j_{\pi_1}, \ldots, j_{\pi_k}$ be a permutation of $j_1, \ldots, j_k$ such that

$$Rounds(j_{\pi_1}) \geq Rounds(j_{\pi_2}) \geq \ldots \geq Rounds(j_{\pi_k})$$
$$\implies Rounds(i) = Max\{1 + Rounds(j_{\pi_1}), 2 + Rounds(j_{\pi_2}), \ldots, k + Rounds(j_{\pi_k})\}$$

□

(b) (4 points) Write the pseudocode for the recursion with memorization dynamic programming procedure for computing ROUNDS(0).

Solution:

In the following psuedocode *Rounds* is an array of size n which holds the value of Rounds at every node in the tree and the base call is EVALROUNDS(0)

```
1  Algorithm: EvalRounds(i)
2  if Rounds(i) is not NULL then
3  |  Return Rounds[i]
4  end
5  if Child[i][0] is 0 then
6  |  Rounds[i] ← 0
7  |  Return Rounds[i]
8  end
9  ChildrenRounds ← Vector(int)
10 for j ← 1 to n do
11 |  if Child[i][j] is not -1 then
12 |  |  Rounds[Child[i][j]] ← EvalRounds(Child[i][j])
13 |  |  ChildrenRounds.Add(Rounds(Child[i][j]))
14 |  end
15 end
16 DescendingSort(ChildrenRounds)
17 Rounds[i] ← −∞
18 for j ← 1 to |ChildrenRounds| do
19 |  Rounds[i] ← Maximum(Rounds[i], j + ChildrenRounds[j])
20 end
21 Return Rounds[i]
```

Algorithm 5: Dynamic Programming Algorithm to calculate *Rounds*

□

(b) (4 points (**Extra credit**)) Analyze the running time of your algorithm.

Solution:

To evaluate *Rounds*(0), we start the base recursive call from the root node 0 which recursively calls each of it's child and when the recursive call is returned back from it's child, store the returned value into the appropriate entry of *Rounds* (used for memorization) corresponding to the respective child.

Each recursive call visits every node exactly once. Therefore, this takes $O(n)$ time in total. Then we sort the *Rounds* of the node's children and then find the maximum of $j + ChildrenRounds[j]$. If every node i has k_i children, then the total time this step takes will be $\sum_{i=0}^{n-1} O(k_i \log k_i) + O(k_i)$ where $\sum_{i=0}^{n-1} k_i = n$

Therefore, the total time taken to evaluate *Rounds*(0) will be $O(n) + \sum_{i=0}^{n-1} O(k_i \log k_i)$ where $\sum_{i=0}^{n-1} k_i = n$ □

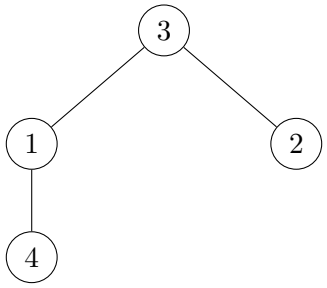
CSCI-GA.1170-001/002 Fundamental Algorithms November 10, 2015

Solutions to Problem 4 of Homework 7 (10 points)

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

Given an undirected rooted tree T with n nodes, with possibly negative weights $w(v)$ assigned to its vertices v, the weight of the tree is the sum of the weights of all the nodes in the tree. (The weight of the empty tree is 0.)

A subtree is any connected subgraph of a tree, including an empty tree as a pathological special case. For example, in the tree below the subtrees are $\emptyset$, $\{3\}$, $\{1\}$, $\{2\}$, $\{4\}$, $\{3,1\}$, $\{3,2\}$, $\{1,4\}$, $\{3,1,4\}$, and $\{3,1,2\}$, but not $\{3,4\}$.



The goal of the problem is to design a polynomial time algorithm to find a subtree (possibly empty) with maximum weight, and analyze its running time.

(a) (4 pts) Given a node v, let $T(v)$ be the complete subtree of T rooted at v (so $T = T(T.root)$), and WithRoot(v) to be the maximum weight of a all subtrees of $T(v)$ which must include the node v. For example, if the tree above, WithRoot(1) is the maximum weight of sub-trees $\{1\}$ and $\{1,4\}$ (but not $\{4\}$ or the empty tree). Give the dynamic programming recurrence relation for WithRoot(v) (don't forget the base when v is a leaf in T), and use it to analyze the running time of a dynamic-programming procedure to compute the values WithRoot(v) for all nodes $v \in T$ (including with $T.root$).

Solution:

This problem is similar to the problem of Maximum-Sum-SubArray. In this case, the array is replaced by a tree. Let T be a tree with a root v and let $v_1, \ldots, v_n$ be the children of v. By analogy it is easy to observe that, *WithRoot(v)* is simply the union of all *WithRoot(v_i)* which have weight greater than 0 together with v

$$WithRoot(v) = \begin{cases} v & \text{v is a leaf} \\ \bigcup_{\geq 0} \{WithRoot(v_i)\} \cup v & \text{otherwise} \end{cases}$$

Start from the root node v and make recursive calls to each of it's child and when the recursive call is returned back, store the returned value into the appropriate entry of the array (used for memorization) corresponding to the respective node.

A recursive call is made on every node of the tree exactly once. Therefore the time taken to calculate *WithRoot(v)* for every $v \in T$ is $O(|v|)$, where $|v|$ denotes the number of nodes in the tree T

□

(b) (4 pts) Given a node v, let $T(v)$ be the complete subtree of T rooted at v (so $T = T(T.root)$), and TOTAL(v) to be the maximum weight of a all subtrees of $T(v)$ which must may or may not include the node v. Assume you already solved part (a) and computed all values WITHROOT(v).

Give the dynamic programming recurrence relation for TOTAL(v) (don't forget the base when v is a leaf in T), and use it to analyze the running time of a dynamic-programming procedure to compute the values TOTAL(v) for all nodes $v \in T$ (including with $T.root$, which gives the answer to the original problem). What is the running time of this algorithm?

Solution:

Given that TOTAL(v) may or may not include the the node v. We have already analyzed the case when v is included. Therefore, now analyze the case when v is excluded and then take the maximum of both these cases. When v is excluded it will just be the maximum weight of one of it's children $v_1, \ldots, v_n$

$$WithoutRoot(v) = \begin{cases} \emptyset = 0 & \text{v is a leaf} \\ max\{Total(v_1), \ldots, Total(v_n)\} & \text{otherwise} \end{cases}$$

$$Total(v) = max\{WithRoot(v), WithoutRoot(v)\}$$

The running time of $WithoutRoot(v)$ is similar to that of $WithRoot(v)$. A recursive call is made on every node of the tree exactly once. Hence the running time will be $O(|v|)$.

Calculating $Total(v)$ involves taking maximum of two quantities which takes $O(1)$ time. Therefore, the total running time of this algorithm is $O(|v|)$ □

(c) (2 pts) What will the running-time of the procedures in parts (a) and (b) be if we use standard recursion, and not dynamic programming?

Solution:

Note that the running time of part(a) is similar to that of part(b). Hence the running time of both these procedures will be effected in a similar way if we use standard recursion and not dynamic programming.

Let the total number of nodes in T be n and v be any node in the tree then the time taken to calculate $WithRoot(v)$ and $Total(v)$ is $O(|v|)$. Let $v_1, v_2, \ldots v_n$ be the the nodes in the tree, then the total time taken without memorization is

$$T(n) = \sum_{i=1}^{n} O(|v_i|)$$

In the worst case, the tree takes the structure of a linked list. Then the above sum will evaluate to $O(n^2)$. Therefore if we use recursion without memorization, the running time increases by a factor of n.

However, note that in part(a) and part(b), we used memory to store the values returned by the recursive calls. Therefore, if we are allowed to use memorization in standard recursion, the running time will $O(n)$. So the running time depends on if we're allowed to use memorization or not in the standard recursion

□