CS515 - Algorithms & Data Structures

Practice Assignment 1

Vy Bui - 934370552

Instructor: Professor Glencora Borradaile

The School of Electrical Engineering and Computer Science
Oregon State University

Problem 1

Suppose you are given an array A[1..n] of integers, which may be positive, negative, or zero. Describe a linear-time (i.e. O(n)-time) algorithm that finds the largest sum of elements in a contiguous, nonempty subarray A[i..j]. For example, given the array [-6, 12,-7, 0, 14,-7, 5] as input, your algorithm should return the integer 19 (the sum of [12,-7, 0, 14]).

For the sake of analysis, assume that comparing, adding, or multiplying any pair of numbers takes O(1) time.

Recursive Formulation

First, let $MS(k)$ denote the function that find the nonempty contiguous subarray with the largest sum in $A[k : n]$. Second, let $MP(k)$ finds the contiguous subarray that starts from k and has the largest sum. Note that MPs can be empty. For example, let $A = [-3, -1, 2, 1]$, $MS(2) = 3, MP(2) = 2$, with the two last elements as MS and the last three elements as MP. However, $MS(1) = 3, MP(1) = 0$ with the two last elements as MS and the empty MP.

MP and MS can be defined recursively as follows:

$$MP(k) = \begin{cases} 0 & k > n \lor k \leq 0 \\ max(0, A[k]) & k = n \\ max(0, A[k] + MP(k+1)) & otherwise \end{cases}$$

$$MS(k) = \begin{cases} 0 & k > n \lor k \leq 0 \\ A[k] & k = n \\ max(A[k], A[k] + MP(k+1), MS(k+1)) & otherwise \end{cases}$$

Proof/Explanation

Proof 1a. For MP's recursive fomular, the first line prevents the algorithm to get out of the boundary of the subject array. In the base case on the second line, there is only one element in $A[k : k]$, so the result must be either empty or $A[k]$. For the inductive step, assume that $MP(k+1)$ returns the max prefix (contiguous subarray that starts from $k+1$ and has the largest sum) of $A[k]$, then $MP[k]$ must be $max(0, A[k] + MP(k+1))$. Because if there exists a larger prefix $A[k : j]$ so that $sum(A[k : j]) = A[k] + sum(A[k+1 : j]) > A[k] + MP(k+1)$, then $sum(A[k+1 : j] > MP(k+1))$, which contradicts the assumption that $MP(k+1)$ is the max prefix of $A[k+1 : n]$. Proof completes.

Proof 1b. For MS's recursive formula, the first line prevents the algorithm to get out of the boundary of the subject array. The second line is the base case in which we consider one element at the right end of the array. $A[k]$ is the only non-empty subarray in $A[k : k]$ so it must also be MS of $A[k : k]$. For the inductive step, assume that $MS(k+1)$ returns the max subarray of $A[k+1 : n]$ and $MS(k+1)$ returns that max prefix, $MS(k)$ is guaranteed to be the maximum of the three case mentioned in the third line of the $MS(k)$ formula. Because if there exists a better solution, that is a contiguous subarray whose sum is greater than all of these cases, it will either contain $A[k]$ or not. In the first case when the subarray $S1$ does not contain $A[k]$, it must be within $A[k+1:n]$. $sum(S1) > MS(k+1)$ contradicts the assumption that $MS(k+1)$ is the max subarray of $A[k+1:n]$, hence there not exists such an subarray. In the second case where the subarray $S2$ contains $A[k]$, it can be decomposed to $sum(S2) = A[k] + sum(A[k+1:j])$. Again, $sum(S2) > A[k] + MP(k+1) \iff sum(A[k+i:j]) > MP(k+1)$ contradicts the assumption that $MP(k+1)$ is the max prefix, thus there not exists such an subarray. Proof completes!

Pseudocode Because $MS(k)$ only depends on $MS(k+1)$ and $MP(k)$ only depends on $MP(k+1)$, we can implement them iteratively in the same loop.

Algorithm 1 $(A[1:n])$

```
$ms \leftarrow A[k]$
$mp \leftarrow max(0, A[k])$
$k \leftarrow n - 1$
while $k \geq 1$ do
    $mp \leftarrow mp + A[k]$
    $max\_sum \leftarrow max(A[k], ms, mp)$
    if $mp == max\_sum$ then
        $ms \leftarrow mp$
    else if $A[k] == max\_sum$ then
        $ms \leftarrow A[k]$
        $mp \leftarrow A[k]$
    else
        $mp \leftarrow max(0, mp)$
    end if
    $k \leftarrow k - 1$
end while
      return $ms$
```

Runing Time and Space Analysis There are n iterations, each of which has constant number of operations, hence the algorithm has $O(n)$ time complexity. Furthermore, the algorithm uses $O(1)$ space.

Problem 2
String A is a supersequence of string B if string B can be obtained from string A by removing letters. For example, the strings BARNYARDSNACK, YUMMYBANANAS, and BWAN-WANWA are supersequences of the string BANANA. Give a dynamic program for finding the length of the shortest string that is a supersequence of two input strings A and B.

We can create the shortest supersequence of A and B by modifying one of them to make it become a supersequence of the other with the least modifications. To keep the resulted sequence a supersequence of the starting string, only insertions are allowed. The proof will be given in the subsequent section.

The problem now becomes finding the least number of insertions to turn one string to another. This resembles the Edit Distance problem mentioned in [Eri19] but simpler because it only allows insertions. Intuitively, starting with a longer string leads to less insertions, but both ways will result an optimal solution.

Recursive Formulation

Let $MI(i, j)$ denote the program that can return the minimum of insertions to transform $A[i : n]$ to a supersequence of $B[j : m]$.

$$MI(i,j) = \begin{cases} j & i = 0 \\ 0 & j = 0 \\ min(MI(i, j+1) + 1, MI(i+1, j+1)) & otherwise \end{cases}$$

Proof/Explanation

Assume that we know the minimum number of insertions k needed to transform A to a supersequence S of B. S will be also a supersequence of A because we can always remove the inserted letters to transform it back to A. To prove that S is the shortest supersequence of A and B, proof by contradiction is used. Assume that there exists a shorter supersequence R of A and B. We can always transform A to R by only inserting some m letters into A. Because R is shorter than S, m is shorter than k, which contradicts with the assumption that k is the minimum number of insertions needed to transform A to a supersequence of B. Therefore, R does not exist and S must be the shortest supersequence of A and B.

Pseudocode

```
Algorithm 2 ($A[1:n], B[1:m]$)
  if $i == 0$ then return j
  end if
  if $j == 0$ then return 0
  end if
  Make sure that n is the length of the longer string and m is the length of the shorter string
  $j \leftarrow m$
  while $j > 0$ do
      $i \leftarrow j$
      while $i > 0$ do
          $MI[i,j] \leftarrow min(MI[i,j+1], MI(i+1,j+1))$
      end while
  end while
        return $MI[1,1]$
```

Runing Time Analysis

The algorithm takes constant time to fill in less than half the table of size mxn, thus takes $O(nm)$ time in total.

Problem 3
Find the length (number of edges) of the longest path in a binary tree.

Observation 1: A binary tree T includes two subtrees L and R, and the root node. The longest path of T, LP(T) can be a path within L, a path within R (2), or a combination of the longest path from the root node to some node in L and the longest path from the root node to some node in R (3). The longest path from some node in L to the root node is the depth of L + 1, and the same applies to R.

Observation 2: The depth of tree T, D(T), is the largest of the depth of the left subtree and the depth of the right subtree.

Recursive Formulation

$$D(T) = \begin{cases} -1 & T = \emptyset \\ 0 & T.L = \emptyset \wedge T.R = \emptyset \\ 1 + max(D(L), D(R)) & otherwise \end{cases}$$

$$LP(T) = \begin{cases} -1 & T = \emptyset \\ 0 & T.L = \emptyset \wedge T.R = \emptyset \\ max(LP(L), LP(R), D(L) + D(R) + 2) & otherwise \end{cases}$$

Proof/Explanation

We will prove that there not exist a path in T that is longer than LP(L), LP(R), and D(L) + D(R) + 2, using proof by contradiction. Assume that there exists a longer path (x, y) with $x, y \in L \cup R \cup r$ where r is the root node. If $x, y \in L$, then it contradicts with the assumption that LP(L) is the longest path in L, which shows that such path does not exist. The same argument can be applied to path within the right tree. Lastly, if $x \in L$ and $y \in R$ then it has to go through r. We can break the path into two pieces, (x, r) and (r, y) so that $|(x, y)| = |(x, r)| + |(r, y)|$, which can not be greater than D(L) + D(R). Thus, such a path does not exist.

Pseudocode

For this problem, a recursive implementation seems to be more efficient, readable, and easier to implement compared to an iterative implementation.

Algorithm 3 $D(T)$

```
if $T == \emptyset$ then return -1
end if
if $T.L == \emptyset \wedge T.R == \emptyset$ then return 0
end if
    return $1 + max(D(T.L), D(T.R))$
```

Algorithm 4 $LP(T)$

```
if $T == \emptyset$ then return -1
end if
if $T.L == \emptyset \wedge T.R == \emptyset$ then return 0
end if
    return $max(LP(T.L), LP(T.R), D(T.L) + D(T.R) + 2)$
```

The iterative implementation can be done by first searching for the leaves of the tree using DF-S/BFS. Then filling LP from the bottom layer up to the top layer. Because D and LP of trees rooted at nodes at height h only depends on their children at height $h+1$, each node will be visited exactly once.

Runing Time Analysis

The iterative implementation needs to compute LP from the bottom up, that is from the leaves up to the root. The problem is we do not know the leaves in advance so it needs to traverse the trees one more time to find the leaves. For memory, it needs O(n) of memory to keep the results in the worst case that the tree is a full binary tree, which has (n + 1) / 2 leaves. On the other hand, a recursive algorithm only needs to traverse through the tree once because a node depends on exactly its two children nodes. Therefore, it takes O(n) time and O(n) space of stack to store the recursive calls.

Problem 4
Suppose you are given an n x n bitmap, represented by a 2-dimensional array M[1..n, 1..n] of 0s and 1s. A solid block in M is a subarray of the form M[i..i', j..j'] containing only 1s. Describe an algorithm to find the area of the maximum solid block in M in $O(n^3)$ time. If you can do that, try to design a faster algorithm that runs in $O(n^2)$ time.

Observation 1: Let us call the maximum solid block of $M[1:n, 1:n]$ as $MSB[i:i', j:j']$. Observe that $MSB[i:i', j:j']$ is also the maximum solid block of the "sub-right-bottom-matrix" $M[i:n, j:n]$. A maximum solid block in M is guaranteed to be the maximum solid block of the some of the sub-right-bottom-matrices. The maximum of the MSB of these sub-matrices is the final solution to the problem.

Recursive Formulation Let us call the first dimension of the matrix as height, and the second dimension as width. Let $R(i,j)$ denotes the rectangle $M[i:n, j:n]$. Let $W[i,j]$ be the width of the widest block in $R(i,j)$ that contains $M[i,j]$ and has height of one, as shown by the green block in Figure 1. Let $H[i,j]$ be the height of the tallest solid block in $R(i,j)$ that contains $M[i,j]$ and has width of one, as shown by the blue block in Figure 1. Let $L[i,j] = (h,w)$ be the height and width of the largest solid block in $R(i,j)$ that contains $M[i,j]$, as shown by the red block in Figure 1.

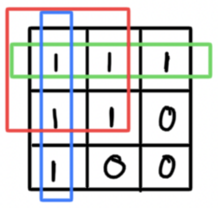

Figure 1: Suboptimal structure. The solution to the black rectangle depends on the solutions to the green, blue, and red rectangles

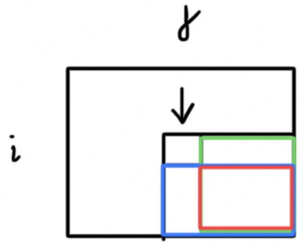

Figure 2: Suboptimal structure. The solution to the black rectangle depends on the solutions to the green, blue, and red rectangles

The recursive formulations of those functions/tables can be defined as follows

$$W[i,j] = \begin{cases} 0 & M[i,j] = 0 \\ \begin{cases} 0 & j = n \\ 1 + W[i,j+1] & otherwise \end{cases} & otherwise \end{cases}$$

$$H[i,j] = \begin{cases} 0 & M[i,j] = 0 \\ \begin{cases} 0 & i = n \\ 1 + H[i+1,j] & otherwise \end{cases} & otherwise \end{cases}$$

$$L[i,j] = \begin{cases} (0,0) & i,j < 1 \lor i,j > n \\ (0,0) & H[i,j] = 0 \\ \begin{cases} (H[i,j],1) \\ (1,W[i,j]) \\ (min(H[i,j], L[i+1,j+1][0]+1), min(W(i,j), L[i+1,j+1][1]+1)) \end{cases} & otherwise \end{cases}$$

In the otherwise case (line 3,4,5) of the formula $L[i,j]$, the program find the largest solid block between $H[i,j]$, $W[i,j]$, and $min(H[i,j], L[i+1,j+1][0]+1) \times min(W(i,j), L[i+1,j+1][1]+1)$.

Proof/Explanation

Proof 4a.

Base case: for rectangles whose top-left corner is on the right edge of M, the widest solid blocks obviously have width of one, if they exists.

Inductive Hypothesis: assume that $W[i,j+1]$ is the width of the widest block in $R(i,j)$ that contains $M[i,j]$ and has height of one.

Inductive Step: assume that there exists some k that $W[i,j] = |M[i,j:k]| > W[i,j+1]+1$. Removing $M[i,j]$ from $M[i,j:k]$ results in $M[i,j+1:k]$, which contains $M[i,j]$ and longer than $W[i,j+1]$. This contradicts the assumption that $W[i,j+1]$ is the width of the widest block in $R(i,j)$ that contains $M[i,j]$ and has height of one, which shows that $W[i,j] = W[i,j+1]+1$ is true.

The same reasoning can be used for $H[i,j]$ recursive formulation.

Proof 4b.

Base case: for rectangles whose top-left corner is on the right edge of M, the largest solid blocks obviously have size equal its H. Similarly, for rectangles whose top-left corner is on the bottom edge of M, the largest solid blocks obviously have size equal its W.

Inductive Hypothesis: assume that $L[i+1,j+1]$ are the height and width of the largest solid block in $R(i,j)$ that contains $M[i,j]$.

Inductive Step: by looking at Figure 2, it is obvious that $L[i,j]$ must be one of the three mentioned in the formula. First, the block has to be within the bound of $W[i,j]$ and $H[i,j]$, the otherwise contradicts the assumption of $W[i,j]$ or $H[i,j]$. Second, the right-bottom corner of the block has to be within $L[i+1,j+1]$, otherwise, $L[i+1,j+1]$ is not the largest solid block of $R(i,j)$. Proof completes!

Pseudocode

Algorithm 5 $M[1:n, 1:n]$

```
if $M == \emptyset$ then return $0$
end if
$maxArea \leftarrow 0$, $W \leftarrow []$, $H \leftarrow []$, $L \leftarrow []$, $i \leftarrow n$, $j \leftarrow n$
while $i \geq 1$ do
    while $j \geq 1$ do
        if $M[i,j] == 0$ then
            $W[i,j] \leftarrow 0$, $H[i,j] \leftarrow 0$, $L[i,j] \leftarrow (0,0)$
        else if $j == n \wedge i == n$ then
            $W[i,j] \leftarrow 1$, $H[i,j] \leftarrow 1$, $L[i,j] \leftarrow (1,1)$
        else if $j == n$ then
            $W[i,j] \leftarrow 1$, $H[i,j] \leftarrow H[i+1,j] + 1$, $L[i,j] \leftarrow H[i,j]$
        else if $i == n$ then
            $W[i,j] \leftarrow W[i,j+1]$, $H[i,j] \leftarrow 1$, $L[i,j] \leftarrow W[i,j]$
        else
            $W[i,j] \leftarrow W[i,j+1] + 1$
            $H[i,j] \leftarrow H[i+1,j] + 1$
            $lh \leftarrow min(H[i,j], L[i+1,j+1][0] + 1)$
            $lw \leftarrow min(W(i,j), L[i+1,j+1][1] + 1)$
            $lArea \leftarrow lh * lw$
            $largestBlock \leftarrow max(H[i,j], W[i,j], crossArea)$
            if $lArea == largestBlock$ then
                $L[i,j] \leftarrow (lh, lw)$
            else if $W[i,j] == largestBlock$ then
                $L[i,j] \leftarrow (1, W[i,j])$
            else
                $L[i,j] \leftarrow (H[i,j], 1)$
            end if
            $maxArea \leftarrow max(maxArea, largestBlock)$
        end if
        $j \leftarrow j - 1$
    end while
    $i \leftarrow i - 1$
end while
    return $maxArea$
```

Runing Time Analysis

For time complexity, because the code in the inner loop takes constant time, the algorithm takes $O(n^2)$ in total. For space complexity, Algorithm 5 consumes $O(n^2)$ space to store the W, H, L tables for the entire M. However, notice that $L(i, j)$ only depends on the row below and the right cell. Therefore, we only need to keep track of W, H, L tables of $M[i+1, 1:n]$ and $M[i, j+1]$, thus reduce the space complexity to $O(n)$.

References

[Eri19] Jeff Erickson. *Algorithms.* 2019.