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 Let $LSA(k)$ denote the function that can find the subarray of $A[k : n]$ with the largest sum of its elements. LSA returns a pair of the sum of the largest subarray and its prefix sum in the aforementioned order. For example, let $A = [-1, 2, 1]$, $LSA(1) = (3, -1)$, with the two last elements as the largest subarray and the first element as the prefix.

LSA can be defined recursively as follows:

$$LSA(k) = \begin{cases} (0,0) & k > n \wedge k < 1 \\ (A[k], 0) & A[k] > LSA(k+1)[0] \wedge LSA(k+1)[0] < -LSA(k+1)[1] \\ (A[k] + LSA(k+1)[0] + LSA(k+1)[1], 0) & A[k] \geq -LSA(k+1)[1] \wedge LSA(k+1)[0] \geq -LSA(k+1)[1] \\ (LSA(k+1)[0], A[k] + LSA(k+1)[1]) & A[k] < -LSA(k+1)[1] \wedge A[k] < LSA(k+1)[0] \end{cases}$$

Proof/Explanation

For simplicity, in this section, let $LSA(k)$ denote the subarray of interest only, but not its prefix. This problem's optimal substructure property can be described as follows. The optimal solution to the problem with $A[k : n]$ can be found based on the optimal solution to the problem with $A[k+1 : n]$. In particular, assume that we know the optimal solution to the problem of $A[k+1 : n]$, adding $A[k]$ to the problem results in two new candidate largest subarrays. First, $A[k]$ can be combined with $LSA(k+1)$ to create a new solution. Note that this comes with the cost of the sum of the prefix of $LSA(k+1)$ because the new solution has to be contiguous. The second candidate is $A[k]$ itself. $LSA(k)$ has the largest sum among these two new candidates and $LSA(k+1)$. Note that the combination solution is prioritized when a tie happens in order to open up opportunities for further combination.

We can use proof by contradiction to prove this algorithm. Assume that there exists a better solution $LSA'(k)$ that is not one of the three candidates, $A[k]$, the combination, and $LSA(k+1)$. First, because $LSA'(k)$ is not $A[k]$, it must be a subarray of $A[k+1 : n]$. This makes $LSA'(1)$ the subarray of the largest sum of $A[k+1 : n]$ because $sum(LSA'(k)) > sum(LSA(k+1))$ based on our assumption. However, this contradicts another assumption that $LSA(2)$ is the optimal solution to the problem with $A[2 : n]$. The proof completes!

Pseudocode

Observe that $LSA(k)$ only depends on $LSA(k+1)$, we can implement this algorithm iteratively from $LSA(n)$ to $LSA(0)$. The solution is $LSA(0)$.

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

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

Runing Time and Space Analysis There are n iteration, 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 BWANWANWA 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 is the shortest supersequence of A and B.

Pseudocode

Runing Time Analysis

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 2 $D(T)$

```
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 3 $LP(T)$

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

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.

Recursive Formulation

Proof/Explanation

Pseudocode

Runing Time Analysis

References

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