Design and Analysis of Algorithms: Homework #2

Due in class on February 15, 2018

Professor Kasturi Varadarajan

Alic Szecsei

Problem 1

Suppose we are given an array $A[1..n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are six local minima in the following array:

9	**7**	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9

Table 1: Example array

We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. *[Hint: With the given boundary conditions, the array **must** have at least one local minimum. Why?]*

Solution

Algorithm 1 Algorithm for finding a local minimum

```
function FindLocalMinimum(A[1..n])
    median ← n/2
    if A[median] > A[median − 1] then
        return FindLocalMinimum(A[1..n/2])
    else if A[median] > A[median + 1] then
        return FindLocalMinimum(A[n/2..n])
    else
        return median
    end if
end function
```

Given the boundary conditions, the array **must** have at least one local minimum. To prove this, we can assume that the array has no local minimum. This means that the array must be either always increase or always decrease throughout the array; formally, $\forall x \in \{1, \ldots, n\}$, $A[x] > A[x-1]$ or $\forall x \in \{1, \ldots, n\}$, $A[x] > A[x+1]$. However since $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$, this cannot be true; specifically it must *start* by decreasing such that $A[x-1] \geq A[x]$ and then, at some point, stop decreasing, such that $A[x] \leq A[x+1]$. This point is the local minimum, since $A[x-1] \geq A[x] \leq A[x+1]$.

Given some median point, m, we can detect whether our array is increasing or decreasing at that point by examining the neighboring points. If the array is increasing, $A[m] \geq A[m-1]$. We know that the array starts by decreasing, so it must have started increasing since then; thus we can narrow our search for the local minimum to the first half of the array.

If the array is decreasing, $A[m] \geq A[m+1]$. Since the array ends with an increase, we know it must start increasing at some point after the median; thus we can narrow our search to the last half of the array.

If neither of these options is true, we must be at either a local maximum or minimum. If we are at a local maximum, then $A[m] \geq A[m-1]$, so we simply search the first half of the array; if we are at a local minimum, $A[m] \not> A[m+1]$ and $A[m] \not> A[m-1]$, so we reach our final branch and simply return m.

Problem 2

Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in the union of A and B in $\Theta(\log n)$ time. For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, your algorithm should return the median of $A \cup B$. You can assume that the arrays contain no duplicate elements. *[Hint: First solve the special case $k = n$.]*

Solution

Algorithm 2 Algorithm for finding the kth smallest element in a union

```
function FindKSmallest(A[1..m], B[1..n], k)
    if k = 1 then
        if m = 0 then
            return B[1]
        else if n = 0 then
            return A[1]
        else
            return min(A[1], B[1])
        end if
    else
        mid ← ⌊k/2⌋
        if mid ≥ m + 1 then
            FindKSmallest(A, B[mid + 1..n], k − mid)
        else if mid ≥ n + 1 then
            FindKSmallest(A[mid + 1..m], B, k − mid)
        else if A[mid] < B[mid] then
            FindKSmallest(A[mid + 1..m], B, k − mid)
        else
            FindKSmallest(A, B[mid + 1..n], k − mid)
        end if
    end if
end function
```

As an example, we will run through the algorithm with $A = \{1, 3, 5, 7\}$ and $B = \{2, 4, 6, 8\}$, with $k = 4$.

Since $k \neq 1$, we determine a middle element, $mid = \left\lfloor \frac{k}{2} \right\rfloor = 2$. Since $mid < m + 1$ and $mid < n + 1$, we compare the second element of each array. A has a smaller element, since $A[2] = 3 < B[2] = 4$, so we recurse with a new $A = \{5, 7\}$, B remaining the same, and $k = k - mid = 4 - 2 = 2$.

k is still not 1, so we determine our middle element, $mid = \left\lfloor \frac{k}{2} \right\rfloor = 1$. Since $mid < m + 1$ and $mid < n + 1$, we compare the first element of each array. B has a smaller element, since $B[1] = 2$, so we recurse with a new $B = \{4, 6, 8\}$, A remaining the same, and $k = k - mid = 2 - 1 = 1$.

Now, $k = 1$. Since neither A nor B are empty, we simply find the minimum of their first elements; $B[1] = 4$ and $A[1] = 5$, so we return 4, as expected.

In general, we can divide the two arrays into four sections: A_1, A_2, B_1, and B_2, such that A_1 and B_1 have length $\frac{k}{2}$. Since we are looking for the kth smallest element, there are $k - 1$ smaller elements we wish to discard. Let us define A_{1_M} to be the largest element in A_1 and B_{1_M} to be the largest element in B_1.

Problem 2 continued on next page…

Let us first suppose that the kth smallest element is in either A_1 or B_1; without loss of generality, suppose it is in A_1. Then, since $|A_1 \cup B_1| = k$, we know that up to $\frac{k}{2} - 1$ of the $k-1$ smaller elements are also in A_1. Therefore, there must be at least $\frac{k}{2}$ of the $k-1$ smaller elements in B_1. By definition, the kth smallest element will be larger than any of the $k-1$ smaller elements, so $A_{1_M} > B_{1_M}$, and our algorithm can safely remove B_1, as we know it contains only elements that are smaller than the kth smallest element.

Alternatively, we can suppose that the kth smallest element is in either A_2 or B_2; without loss of generality, suppose it is in A_2. Then, A_1 must contain $\frac{k}{2}$ of the $k-1$ smaller elements, since A_1 and B_1 contain $\frac{k}{2}$ elements. The opposite array (B_1) must therefore contain fewer than $\frac{k}{2}$ of the $k-1$ smaller elements, and will therefore have its largest element B_{1_M} be greater than A_{1_M}; our algorithm then safely removes A_1, as we know it contains only elements that are smaller than the kth smallest element.

Since we have discarded elements smaller than the kth smallest, we update our value of k, removing the number of elements we discarded - $\frac{k}{2}$. We continue in this manner, discarding elements when possible, until we have reached our trivial base case, where $k = 1$ and we can simply find the minimal element of $A \cup B$.

Problem 3

For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return the root and the depth of this subtree.

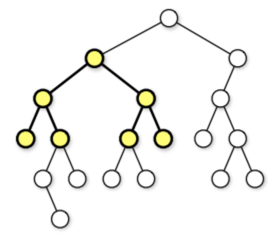

Figure 1: The largest complete subtree of this binary tree has depth 2.

Solution

Algorithm 3 Algorithm for finding the largest complete subtree

function LargestCompleteSubtreeAt($node$)
    if $node.left$ exists and $node.right$ exists **then**
        $childDepth \leftarrow$ the minimum of LargestCompleteSubtreeAt($node.left$) and LargestCompleteSubtreeAt($node.right$)
        return $1 + childDepth$
    else
        return 0
    end if
end function
function LargestCompleteSubtreeAt($root$)
    $rootDepth \leftarrow$ LargestCompleteSubtreeAt($root$)
    if $node.left$ exists **then**
        $leftDepth, leftNode \leftarrow$ LargestCompleteSubtree($node.left$)
    end if
    if $node.right$ exists **then**
        $rightDepth, rightNode \leftarrow$ LargestCompleteSubtree($node.left$)
    end if
    $maxDepth \leftarrow$ the maximum of $rootDepth$, $leftDepth$, and $rightDepth$
    if $maxDepth = rootDepth$ **then**
        return $maxDepth, root$
    else if $maxDepth = leftDepth$ **then**
        return $maxDepth, leftNode$
    else
        return $maxDepth, rightNode$
    end if
end function

Our first function exists to determine the largest complete subtree beginning at a particular node. This way, we can separate our concerns to aid with recursion. Our second function repeatedly calls the first function to determine the largest complete subtree at any position in the tree; we do this by recursing through it and finding the largest complete subtree starting from every node in the tree.

The run-time of LargestCompleteSubtreeAt has a recurrence of $T_1(n) = 2T_1(\frac{n}{2}) + 1$. Using the Master Theorem, we can solve this recurrence:

$$\begin{aligned}
a &= 2 \\
f(\frac{n}{b}) &= 1 \\
f(n) &= 1 \\
af(\frac{n}{b}) &= Kf(n) \\
2 \times 1 &= K \times 1 \\
K &= 2
\end{aligned} \tag{1}$$

Since $K > 1$, the Master Theorem states that $T_1(n) = \Theta(n^{\log_b a})$; with $b = 2$ and $a = 2$, we get $T_1(n) = \Theta(n^{\log_2 2})$, which trivially reduces to $T_1(n) = \Theta(n^1) = \Theta(n)$.

The unoptimized run-time of LargestCompleteSubtree has a recurrence of $T_2(n) = 2T_2(fracn2) + T_1(n)$. Using the Master Theorem, we get:

$$\begin{aligned}
a &= 2 \\
f(\frac{n}{b}) &= \frac{n}{2} \\
f(n) &= n \\
af(\frac{n}{b}) &= Kf(n) \\
2\frac{n}{2} &= K \times n \\
K &= 1
\end{aligned} \tag{2}$$

Since $K = 1$, the Master Theorem states that $T_2(n) = \Theta(f(n) \log_b n)$, so $T_2(n) = \Theta(n \log_2 n)$.

This could easily be optimized with memoization; we can cache the results of LargestCompleteSubtreeAt. In this case, no repeated calculations of LargestCompleteSubtreeAt occur, so rather than running in $\Theta(n \log_2 n)$, it would run in $\Theta(n)$.

Problem 4

(a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common subsequence* of A and B is both a subsequence of A and a subsequence of B. Give a simple recursive definition for the function $lcs(A, B)$, which gives the length of the *longest* common subsequence of A and B.

(b) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Give a simple recursive definition for the function $los(A)$, which gives the length of the longest oscillating subsequence of an arbitrary array A of integers.

(c) Call a sequence $X[1..n]$ *accelerating* if $2 \times X[i] < X[i-1] + X[i+1]$ for all i. Give a simple recursive definition for the function $lxs(A)$, which gives the length of the longest accelerating subsequence of an arbitrary array A of integers.

Solution

Part A

Algorithm 4 Algorithm for finding the length of the longest common subsequence

```
function LCS(A[1..m], B[1..n])
    if m = 0 or n = 0 then
        return 0
    else if A[1] ≠ B[1] then
        a ← LCS(A, B[2..n])
    else
        a ← 1
        while a < m and a < n and A[a] = B[a] do
            a ← a + 1
        end while
        a ← the maximum of a and LCS(A, B[a..n])
    end if
    b ← LCS(A[2..m], B)
    return the maximum of a and b
end function
```

Problem 4 continued on next page…

Part B

Algorithm 5 Algorithm for finding the length of the longest oscillating subsequence

```
function LOS(X[1..n])
    if n ≤ 1 then
        return n
    else
        i ← 1
        while i + 1 < n do
            if i is even then
                if X[i] ≥ X[i + 1] then
                    break
                end if
            else
                if X[i] ≤ X[i + 1] then
                    break
                end if
            end if
            i ← i + 1
        end while
        return the max of i and LOS(X[2..n])
    end if
end function
```

Part C

Algorithm 6 Algorithm for finding the length of the longest accelerating subsequence

```
function LXS(X[1..n])
    if n ≤ 1 then
        return n
    else
        i ← 2
        while i + 1 < n do
            if 2 × X[i] ≥ X[i − 1] + X[i + 1] then
                break
            end if
            i ← i + 1
        end while
        return the max of 1 + i and LXS(X[2..n])
    end if
end function
```