Largest Sum Subsequence - Proof of Correctness

Here's the Largest Sum Subsequence algorithm we developed in class:

```
1  LSS(A, lo, hi) // A is a length-n array of numbers
2     // Base case: only one element in the array
3      if hi == lo, return A[lo]
4      // Divide: split the array in half
5      mid = (hi + lo)/2
6      // Conquer: solve the LSS problem on each half
7      left = LSS(A, lo, mid)
8      right = LSS(A, mid+1, hi)
9      // Combine: find the largest crossing sum and compare
10     cross = LargestCrossingSum(A, lo, mid, hi)
11     return max{left, right, cross}

12 LargestCrossingSum(A, lo, mid, hi)
13     leftSum = A[mid] // running total
14     largestL = leftSum // largest sum found so far
15     for i = mid-1 down to lo
16         leftSum += A[i]
17         if leftSum > largestL
18             largestL = leftSum
19     // repeat the same idea to find largestR
20     return largestL + largestR
```

Theorem 1. *For any n that is a power of 2, `LSS` correctly finds the largest sum subsequence in a length-n array.*

Proof. We will prove this by induction on n. (Note that the proof will assume that the LSS is unique—there are not two different subsequences with the same largest value—but only minor adaptations are required to handle the more general case where the LSS may not be unique.)

Base case ($n = 1$): When $n = 1$, we are computing the largest sum subsequence of an array containing a single element. The answer should be that element. Sure enough, if our array contains only one element we fall into the base case of our `LSS` algorithm, which returns the only element in our array. Hence `LSS` is correct when $n = 1$.

Inductive hypothesis: Assume that `LSS` gives the correct answer for any input array of length n for some $n \geq 1$.

Inductive step: We will show that `LSS` gives the correct answer for any input array of length $2n$.

Our `LSS` algorithm has three steps:

- The call to `LSS(A, lo, mid)`. The purpose of this call is to find the LSS in the left half of the array. The left half of the array is an array of size n, so by the inductive hypothesis we

know that this call correctly finds and returns the value of the largest sum subsequence that lies entirely in the left half of the array. Call this value v_{left}.

- The call to `LSS(A, mid+1, hi)`. The purpose of this call is to find the LSS in the right half of the array. The right half of the array is an array of size n, so by the inductive hypothesis we know that this call correctly finds and returns the value of the largest sum subsequence that lies entirely in the right half of the array. Call this value v_{right}.
- The call to `LargestCrossingSum(A, lo, mid hi)`. The purpose of this call is to find the LSS that crosses over the middle. By Lemma 1 below (a lemma is a "mini theorem" that we use along the way to proving a larger theorem) `LargestCrossingSum` correctly finds and returns the value of the largest sum subsequence that crosses the middle. Call this value v_{cross}.

Call the true value of the largest sum subsequence v_{max}. We observe that there are three possibilities for where v_{max} could fall within our length-$2n$ array:

1. v_{max} lies entirely in the first n elements of the array (i.e., entirely in the left half).
2. v_{max} lies entirely in the last n elements of the array (i.e., entirely in the right half).
3. v_{max} crosses over the middle (i.e., it contains at least one element in the left half and at least one element in the right half).

We will show that the `LSS` algorithm correctly finds v_{max} in each of the three cases.

Case 1 (v_{max} lies entirely in the left half of the array): Then it must be the case that $v_{left} = v_{max}$: if $v_{left} > v_{max}$ then we contradict the fact that v_{max} is the LSS of the entire array, and if $v_{left} < v_{max}$ then the call to `LSS(A, lo, hi)` did not return the correct answer, since v_{max} lies entirely in the left half and thus should have been considered by the left-hand recursive call.

It must also be the case that $v_{right} < v_{max}$: if $v_{right} > v_{max}$, then we have found a larger LSS for the entire array, contradicting the fact that v_{max} is the LSS of the entire array. Similarly, we must have that $v_{cross} < v_{max}$.

We have $v_{max} = v_{left} > v_{right}, v_{cross}$, so the return statement in line 11 correctly returns v_{max}. Hence in Case 1 `LSS` correctly finds the largest sum subsequence of the entire array.

Case 2 (v_{max} lies entirely in the right half of the array): This case is entirely symmetric to Case 1, and so is omitted.

Case 3 (v_{max} crosses over the middle): Then it must be the case that $v_{cross} = v_{max}$: if $v_{cross} > v_{max}$ then we have found a larger subsequence for the entire array, contradicting the fact that v_{max} is the LSS for the entire array, and if $v_{cross} < v_{max}$ then the call to `LargestCrossingSum` did not return the correct answer, since v_{max} crosses the middle and should have been considered by `LargestCrossingSum`.

It must also be the case that $v_{left} < v_{max}$: if $v_{left} > v_{max}$ then we have found a larger LSS for the entire array, contradicting the fact that v_{max} is the LSS of the entire array. Similarly, we must have that $v_{right} < v_{max}$.

We have $v_{max} = v_{cross} > v_{left}, v_{right}$, so the return statement in line 11 correctly returns v_{max}. Hence in Case 3 `LSS` correctly finds the largest sum subsequence of the entire array.

In all three cases, `LSS` finds the correct answer for an array of size $2n$. Hence by induction, `LSS` is correct for all values of n that are powers of 2. □

Lemma 1. *`LargestCrossingSum` correctly finds the largest sum of any contiguous subsequence containing the elements at both `mid` and `mid+1`.*

Proof. Initially, `leftSum` contains just the element at `A[mid]`, as does `largestL`. So before we start the loop in lines 15-18, `largestL` contains the largest subsequence in the range from `mid` to `mid`. After one iteration of the loop, we have added the element at `A[mid-1]` to our running total in `leftSum` and replaced the largest sum found so far in `largestL` if our new running total is greater than the value previously stored in `largestL`. At this point `largestL` contains the largest subsequence starting in the range from `mid-1` to `mid` and ending at `mid`. We repeat this process: after iteration `i` of our loop, `largestL` contains the largest subsequence starting in the range from `i` to `mid` and ending at `mid`. By the end of the loop, `largestL` contains the largest subsequence starting in the range from `lo` to `mid` and ending at `mid`.

By similar reasoning, at the end of the step in line 19 `largestR` contains the largest subsequence starting at `mid+1` and ending in the range from `mid+1` to `hi`.

Finally, we'll show that `largestL + largestR` is the largest sum subsequence that crosses the middle. Suppose it's not. Then there is some other contiguous subsequence that crosses the middle and has larger value; call that sequence's value `x`. We can split up `x` at `mid` so that `xLeft` is the sum to the left of `mid` (including `mid`) and `xRight` is the sum to the right of `mid` (not including `mid`). We know that `largestL > xLeft` and `largestR > xRight`. So we could make `x` larger by replacing `xLeft` and `xRight` with `largestL` and `largestR`. This contradicts the assumption that `x` has larger value than `largestL + largestR`. So `largestL + largestR` must be the largest sum subsequence that crosses the middle. □