CMPE 365 Lab 2

Riley Becker

September 27, 2018

Part 1

The program appears to work. I tested as many of the fringe cases as I could think of, including the ones given in the problem statement. The cases I tested were:

- The given example
- A list whose length is a power of 2
- A list whose length is not a power of 2
- A list whose sublist with the largest sum is the whole list
- A list whose sublist with the largest sum contains a negative number
- A list with all negative values
- A list whose sublist with the largest sum is repeated twice
- A list with two different sublists whose sum is the maximum (one longer than the other)
- A list with zeroes on either end of the sublist with the largest sum
- A list whose entries are all equal and positive except a block in the middle
- A list whose entries are all equal and negative except a block in the middle

These tests seem to cover all possible fringe cases.

The computational complexity of the algorithm is $O(n)$ because the recursive equation giving the complexity is

$$T(n) = 2T(n/2) + O(1)$$

with a base case of $T(1) = O(1)$. The memory complexity probably isn't great because I'm returning so many things each time the function is called.

Part 2

The intuitive idea would be to simply replace all the elements in the maximal sublist with 0 so that they don't contribute to the sums of the lists on either side. However, this idea doesn't work if all the remaining values in the list are negative because it second largest sum as 0 and the sublist with this sum as [0], which obviously is not possible if all the remaining numbers are negative.

Another idea we could implement to remedy this is to simply replace all the elements of the maximal sublist by the minimum value in the list. This, of course, only works if the minimum value in the list is negative, because otherwise the replacement numbers would increase the sum of the sublists on either side of the replaced portion of the list and could result in an incorrect solution.

I implemented a combination of these two ideas by replacing the elements in the maximal sublist with 0 if the minimum value in the list is positive and the minimum value in the list if not. This solves the issue with the first method because if the remaining numbers are all negative the maximal sublist will be replaced with negative numbers. It also solves the issue with the second method because the new numbers will never contribute to a sum, and in fact will 'block' the sum from crossing over the replaced sublist. Note that the only case in which the replacement number will be 0 is if all the numbers in the list are non-negative, in which case the entire list after the replacement will be zeroes.

Part 3

The second-largest array will be either the second-largest non-overlapping or the second-largest overlapping. We'll find the second-largest non-overlapping and compare it to the sublists of the maximal array, and they should give us the second-largest array overall. We will also need to check the arrays connected to the maximal sublist to see what the maximal sums are and see what they add to the maximal sublist (check the max left sum of the remaining list on the right and the max right sum of the remaining list on the left). The maximal sum of all these subarrays can

Note

I did two iterations of the code. The first was done by simply copying the new sublist every time I divided the problem, and I didn't pass sums up when I made my way up the recursion tree. The second iteration is exactly the same code, except that I worked with indexes instead of copying the array every time and I passed sums up the recursion tree. The first iteration is $O(n \log n)$ because the equation describing the complexity is $T(n) = T(n/2) + O(n)$.