Largest Sum Sub-Array Solution

by Nathan Feaver; July 25th, 2011

Problem Statement:

You're given an array containing both positive and negative integers, that array may be quite large, and you are asked to find the sub-array of any size with the largest sum. Small examples:

[0,-1,3,4,-10] = [3,4] = Sum of 7
[1,3,4,-1,6] = [1,3,4,-1,6] = Sum of 13
[1,3,4,-12,10] = [10] = Sum of 10

Solution Method:

I first approached this problem by finding the solution manually for several different arrays of larger size. It is impressive how quickly the eye can spot the highest numbers and deduce the correct sub-array. For example, let's look at the following array of 20 integers:

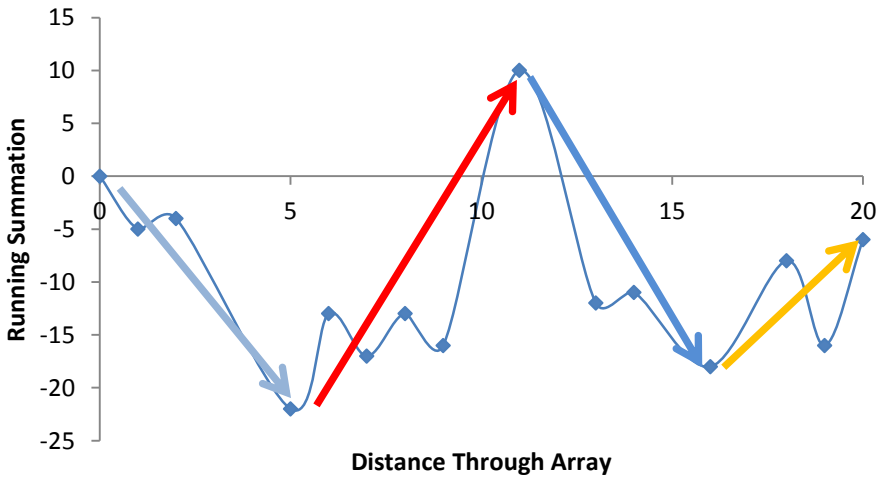
[-5 1 -7 -1 -10 9 -4 4 -3 11 15 -8 -14 1 -3 -4 6 4 -8 10]

After looking at it for a while, regions with high numbers and regions with low numbers begin to stand out, including the max sum sub-array ('++ : High' region):

[-5 1 -7 -1 -10 9 -4 4 -3 11 15 -8 -14 1 -3 -4 6 4 -8 10]

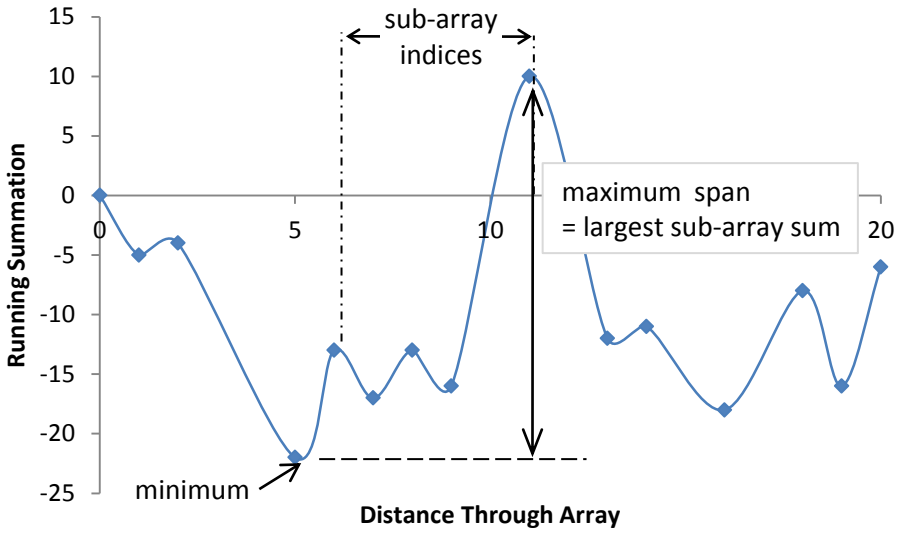
Quickly, we can narrow in on the sub-array that we are looking for: It is the '++ : High' region, bounded by low regions and a second high region that is not quite high enough to warrant inclusion in the sub-array.

To keep track of this computationally, I kept a running sum so that negative regions are characterized by negative slopes and positive regions are characterized by positive slopes. Additionally, in order to cut down on processing time later on, neighboring elements with the same sign are combined (although this uses more memory). The running summation for this array is seen below with the expected slopes in the 'low' and 'high' regions:



This step is completed in the first **for** loop: lines 86 – 111

Once the summation is completed, the program must find the largest span from a local minimum to a local maximum to locate the largest sub-array sum. This is done quickly by stepping through the new summation array one more time, looking for the minimum and checking the difference from it at each point.



This step is completed in the second **for** loop: lines 122 – 133

Important Variable Explanations:

array – original array of 1000 uniformly distributed random numbers ranging from -15 to 15
newArray – combines neighboring values of same sign from 'array' and adds them to previous element in 'newArray': the running summation
pos – Keeps track of neighboring positive/negative regions
count – keeps track of location in newArray
maxSpan – summation of largest sum sub-array
leftInd – left index of largest sum sub-array
rightInd – right index of largest sum sub-array

Performance Comparison:

Comparison of the CPU time was carried out against Kadane's algorithm (most-efficient linear time solution) as well as a brute force method. Kadane's algorithm is found on lines 137 – 157 and the brute force method is found on lines 162 – 178.

CPU time was averaged over 10 computations in variables:
t(10) – times for method described here
tk(10) – times for Kadane's algorithm
tb(10) – times for brute force algorithm

Results for 1000 array elements:
Kadane's algorithm is around 150 times faster than my algorithm [both O(n)]
My algorithm is around 2.5 times faster than the brute force algorithm [$O(n^2)$]