Max-Subarray Sum Problem and Solution Algorithms

Erdal Sidal Dogan, Alp Gokcek
MEF University
November 17, 2019

***Abstract*—Maximum Subarray Sum is a well-known problem in the field of computer science. There are multiple number of solution algorithms with different complexities. In this paper, we demonstrated and compared 3 of these algorithms with quadratic, linear, and logarithmic complexities.**

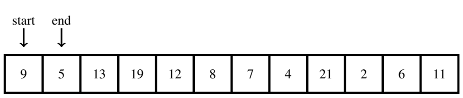
I. The Problem

The Maximum Subarray Sum problem is the task of finding the contiguous subarray with largest sum in a given array of integers. Each number in the array could be positive, negative, or zero. For example: Given the array $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ the solution would be $[4, -1, 2, 1]$ with a sum of 6.

II. Solutions

A. Brute-Force Approach

This is the most intuitive solution to anyone. You basically traverse over the array and compare every possible combination of start and end index for the soultion array.



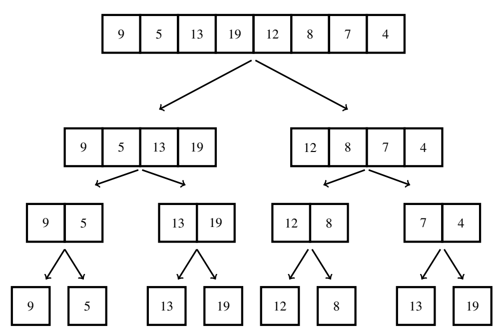
The *start* index will be incremented by 1 everytime the *end* index reaches the end of the array. Then the *end* index will start from the element right next to the start element. At every iteration the sum between *start* and *end* indexes will be calculated. Hence, the maximum sum will be determined by computing every sum for the every possible sub-array. The complexity of this algorithm is $\mathcal{O}(n^3)$. With a little improvement we can convert this algorithm to be $\mathcal{O}(n^2)$. Instead of calculating the sum between two array indicies at every iteration from scratch, we know that the current sum will be the (current element + previous sum). Consequently, eliminating the loop which is used for calculating the sum from the algorithm will reduce the time complexity.

Algorithm 1 Brute-Force

$n \leftarrow len(array)$
$max\_sum \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to n **do**
    $sum += array[j]$
    if $sum > max\_sum$ **then**
      $max\_sum \leftarrow sum$
    end if
  end for
end for

B. Divide & Conquer Approach

Another solution is to divide the array into half recursively and computing the max subarray sum for each half and the sub array for crossing both halfs. After calculating the summation for these 3 cases, we choose the largest one, thus we determine the maximum sub array.



Algorithm 2 Divide & Conquer

```
function MAX_CROSSING_SUB_ARRAY(array, l, m, h)
    left_max_sum ← −100000
    sum_l ← 0
    for i ← m downto l − 1 do
        sum_l ← sum_l + array[i]
        if sum_l > left_max_sum then
            left_max_sum ← sum_l
        end if
    end for
    sum_r ← 0
    right_max_sum ← −100000
    for j ← m + 1 to h + 1 do
        sum_r ← sum_r + array[j]
        if sum_r > right_max_sum then
            right_max_sum ← sum_r
        end if
    end for
return left_max_sum + right_max_sum)
end function

function MAX_SUB_ARRAY(array, l, h)
    m ← ((h + l)/2)
    if l = h then
        return array[l]
    end if
     return max(max_subarray(array, l, m),
max_subarray(array, m + 1, h),
max_crossing_subarray(array, l, m, h)
)
end function
```

C. Linear Time

Algorithm 3 Linear Time

```
max_so_far ← −100000
max_ending_here ← −100000
n ← length(array)
for i ← 1 to n do
    max_ending_here ← max_ending_here+array[i]
    if max_ending_here < array[i] then
        max_ending_here ← array[i]
    end if
    if max_so_far < max_ending_here then
        max_so_farmax_ending_here
    end if
end for
```