Max-Subarray Sum Problem and Solution Algorithms

Erdal Sidal Dogan, Alp Gokcek
doganer@mef.edu.tr, gokcekal@mef.edu.tr
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.**

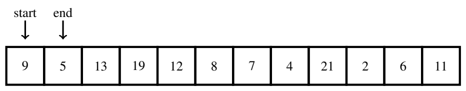
I. The Problem

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.



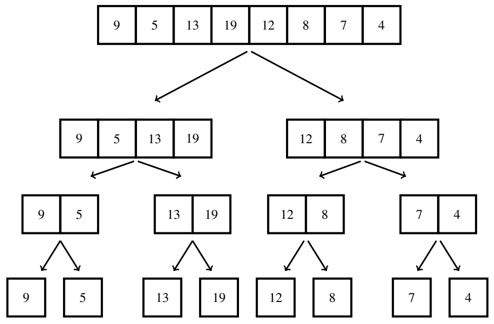
The *start* index will be incremented by 1 everytime the *end* index reaches the end of the array. Then the *end* index will start from the element right next to the start element. At every iteration the sum between *start* and *end* indexes will be calculated. Hence, the maximum sum will be determined 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

```
1: $n \leftarrow len(array)$                          ▷ $1$
2: $max\_sum \leftarrow 0$                           ▷ $1$
3: for $i \leftarrow 1$ to $n$ do                     ▷ $n+1$
4:     $sum \leftarrow 0$                            ▷ $n$
5:     for $j \leftarrow i$ to $n$ do                 ▷ $\sum_{i=0}^{n}(n-i+2)$
6:         $sum+ = array[j]$                        ▷ $\sum_{i=0}^{n}(n-i+1)$
7:         if $sum > max\_sum$ then                 ▷ $\sum_{i=0}^{n}(n-i+1)$
8:             $max\_sum \leftarrow sum$            ▷ $\sum_{i=0}^{n}(n-i+1)$
9:         end if
10:    end for
11: 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.



The recurrence relation of the *Divide & Conquer algorithm* is as follows; $T(n) = 2T(n/2) + n$

By using *Master's Method*, we conclude that this algorithm's complexity is $\mathcal{O}(n \log n)$

Algorithm 2 Divide & Conquer

```
1: function MAX_CROSSING_SUB_ARRAY($array, l, m, h$)
2:     $left\_max\_sum \leftarrow -100000$
3:     $sum\_l \leftarrow 0$
4:     for $i \leftarrow m$ downto $l - 1$ do
5:         $sum\_l \leftarrow sum\_l + array[i]$
6:         if $sum\_l > left\_max\_sum$ then
7:             $left\_max\_sum \leftarrow sum\_l$
8:         end if
9:     end for
10:    $sum\_r \leftarrow 0$
11:    $right\_max\_sum \leftarrow -100000$
12:    for $j \leftarrow m + 1$ to $h + 1$ do
13:        $sum\_r \leftarrow sum\_r + array[j]$
14:        if $sum\_r > right\_max\_sum$ then
15:            $right\_max\_sum \leftarrow sum\_r$
16:        end if
17:    end for
18: return $left\_max\_sum + right\_max\_sum)$
19: end function
20:
21: function MAX_SUB_ARRAY(array, l, h)
22:     $m \leftarrow ((h + l)/2)$
23:     if $l = h$ then
24:         return $array[l]$
25:     end if
          return max(max_subarray(array, l, m),
26: max_subarray(array, m + 1, h),
27: max_crossing_subarray(array, l, m, h)
28: )
29: end function
```

C. Linear Time

Another solution is by just traversing the array once and comparing the current element with maximum sub array ending there. Since the problem is finding the maximum contiguous subarray, we pick the larger one, a.k.a Kadane's Algorithm. The complexity is calculated as the summation of the costs of all the lines; $C_1 + C_2 + C_3 + C_4 + n(C_4 + C_5 + C_6 + C_7 + C_8 + C_9) = \mathcal{O}(n)$ Where C_n denotes the cost of the line n.

Algorithm 3 Linear Time

```
1: $max\_so\_far \leftarrow -100000$                          ▷ 1
2: $max\_ending\_here \leftarrow -100000$                     ▷ 1
3: $n \leftarrow length(array)$                               ▷ 1
4: for $i \leftarrow 1$ to $n$ do                             ▷ $n + 1$
5:     $max\_ending\_here \leftarrow max\_ending\_here + array[i]$
   ▷ $n$
6:     if $max\_ending\_here < array[i]$ then                 ▷ $n$
7:         $max\_ending\_here \leftarrow array[i]$            ▷ $n$
8:     end if
9:     if $max\_so\_far < max\_ending\_here$ then             ▷ $n$
10:        $max\_so\_farmax\_ending\_here$                    ▷ $n$
11:    end if
12: end for
```

III. Comparisons

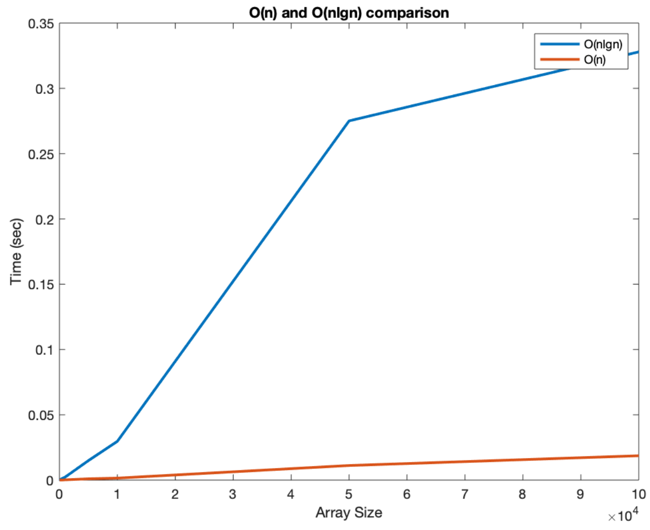


Fig. 1: Divide & Conquer Algorithm ($\mathcal{O}(n)$) vs Linear Time Algorithm ($\mathcal{O}(n \log n)$)

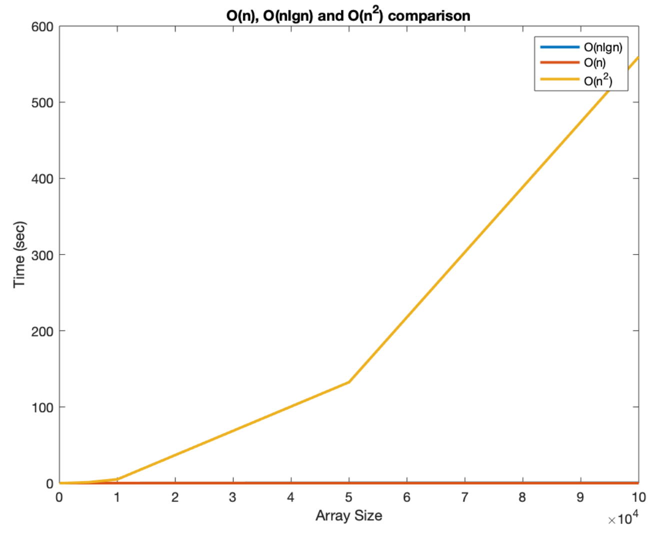


Fig. 2: Brute-Force ($mathcalO(n^2)$)vs .Divide & Conquer ($\mathcal{O}(n)$) vs Linear Time ($\mathcal{O}(n \log n)$)

As the Fig. 1 and Fig. 2 indicates, Brute-Force algorithm is way slower comparing to D&C and Linear Time algorithms, espicially in larger data sets.