Melis Akarçay - 041401015

Yasin Yağız Gülten – 041501037

# COMP 303 – Analysis of Algorithms

## Project : Maximum Subarray Sum

1. Pseudocode of the brute-force algorithm to find the subarray with a running time complexity of O(n2) :

```
                                                        cost        time
declare array                                           c0          1
size(array) <- n                                        c1          1
max <- a[0]                                             c2          1
for i <- 0 to n                                         c3          n + 1
        currentSum <- 0                                 c4          n
        for j <- i to n                                 c5          n*(n+1)
                currentSum <- currentSum + array[j]     c6          n*n
                do if currentSum > max                  c7          n*n
                        then max <- currentSum          c8          n*n
                        left <- i                       c9          n*n
                        right <-j                       c10         n*n
                        maxsubarray <- array[left to right]  c11    n*n
print max value                                         c12         1
```

Total Cost :
(c0*1)+(c1*1)+(c2*1)+(c3*(n+1))+(c4*n)+(c5*(n(n+1)))+(c6*n*n)+(c7*n*n)+(c8*n*n)+(c9*n*n)+(c10*n*n)+(c11*n*n)(c12*1)

= O(n2)

Graph of the code's execution time :

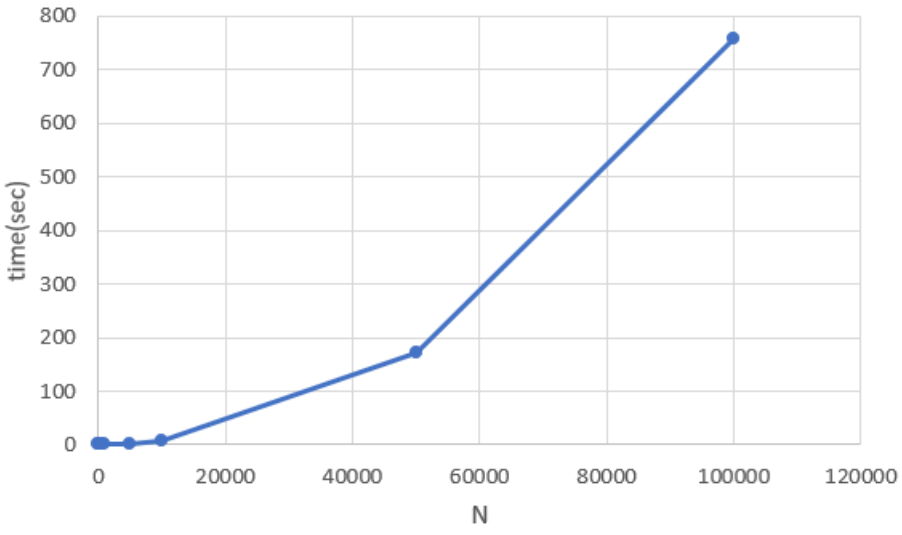


figure 1. O(n2) execution time graph

O(N2)'s runtime grows faster than our input size. As shown in the figure2, execution time graph is similiar to O(n2). When finding the execution time of the algorithm we've used the time() function and calculate the difference of starting and ending time.

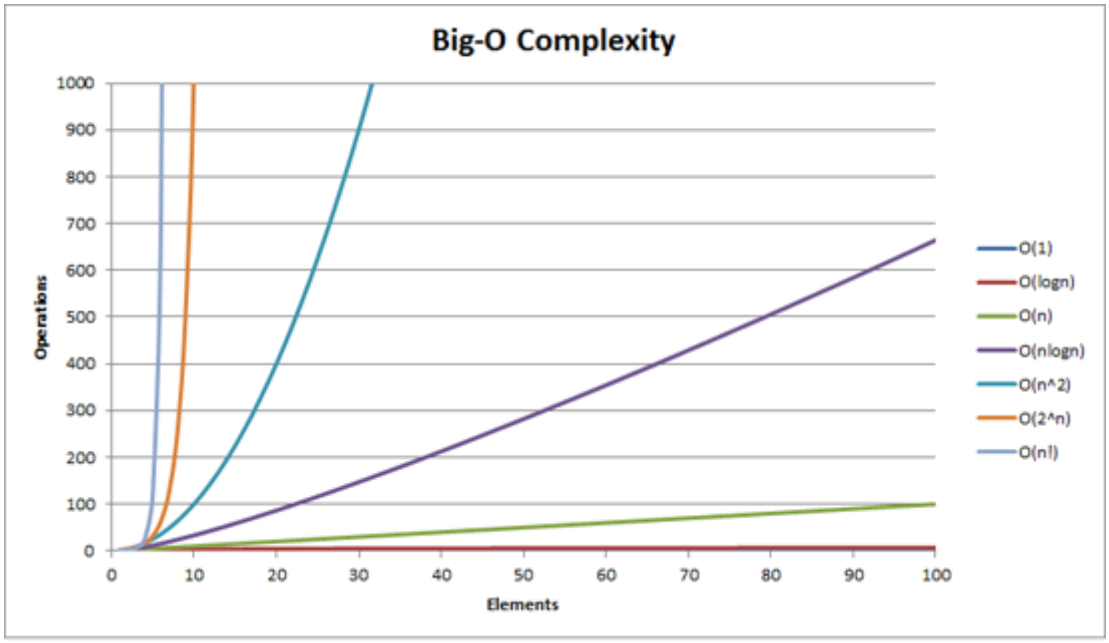


figure 2. big o notation

2. Pseudocode of the brute-force algorithm to find the subarray with a running time complexity of O(nlgn) :

```
declare array
function maxCrossSum(array, l, m, h)
        for i<-m to l-1                                          size : n/2
                i <- i – 1
                sum <- sum + array[i]
                do if sum > left_sum
                        then left_sum <- sum
        sum <- 0
        for i<-m+1 to h+1                                        size : n/2
                sum <- sum + array[i]
                do if sum > right_sum
                        then right_sum <- sum
        return left_sum + right_sum
function maxsubarray(array, l, h)                                size : n
        do if l = h
                then return array[l]
        m <- (1+h)/2
        return max(maxsubarray(array, l, m),
maxsubarray(array,m+1,h),maxCrossSum(Array, l, m, h))
```

Total Cost : T(n) = 2T(n/2) + O(n) = O(nlogn)

Finding maxsubarray's size is n and that means its run time is O(n) in worst case. We are making two recursive calls with using input n/2. With using the iteration method we reached the O(nlogn).

Graph of the code's execution time :

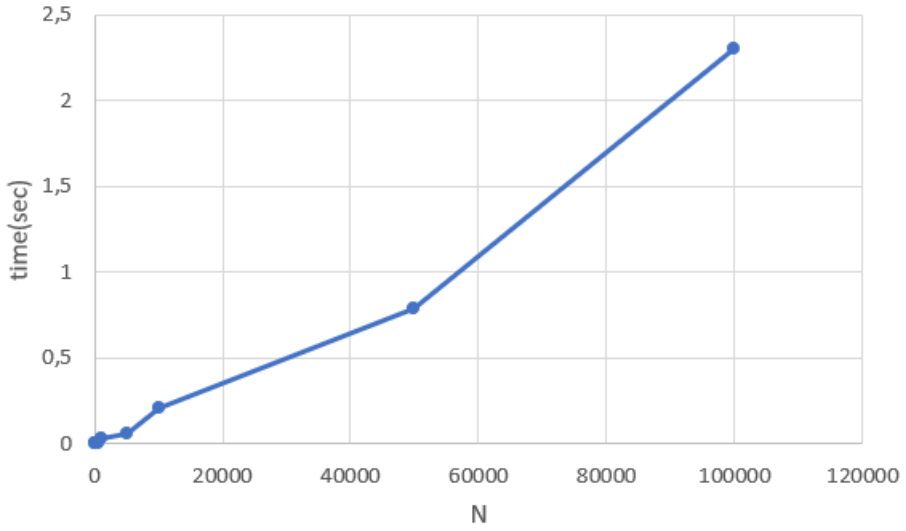


figure 3. O(nlogn) execution time graph

3. Pseudocode of the brute-force algorithm to find the subarray with a running time complexity of O(n) :

| | cost | time |
|---|---|---|
| declare array | c1 | 1 |
| define size as N | c2 | 1 |
| max_value = a[0] | c3 | 1 |
| temp_max = a[0] | c4 | 1 |
| for i<-1 to N | c5 | N |
| temp_max <- MAX(a[i] , temp_max + a[i]) | c6 | N - 1 |
| do if temp_max > max_value | c7 | N - 1 |
| then max_value <- temp_value | c8 | N - 1 |

Total Cost : (c1*1)+(c2*1)+(c3*1)+(c4*1)+(c5*N)+(c6*(N-1))+(c7*(N-1))+(c8*(N-1)) =O(n)

Graph of the code’s execution time :

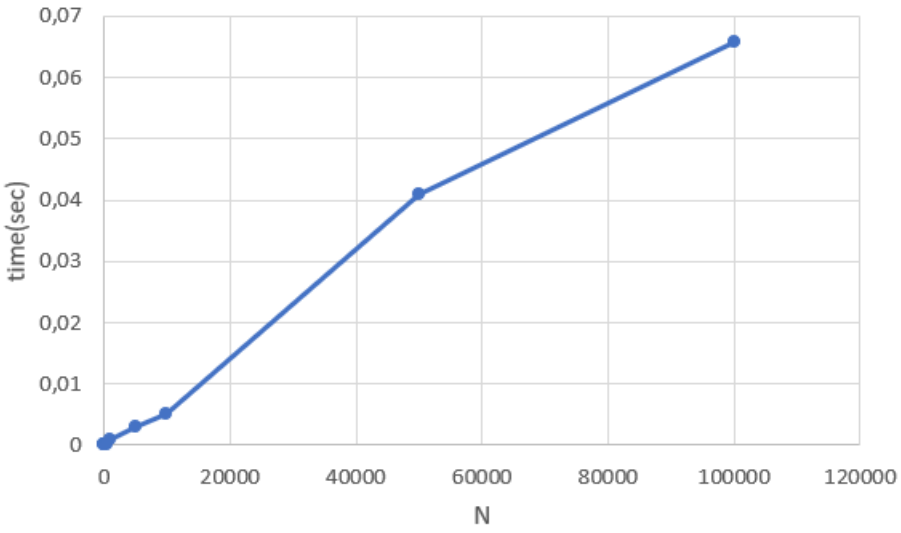


figure4. O(n) execution time graph

4. According to the values we’ve obtained, it moves from fast to slow in the order of O (n), O(nlogn), O (n2). When we compare the results with the graphs in figure 2, we saw that we have reached the right results. The problem size n0 which gives the crossover point at which algorithm with the running time O(nlgn) beats the algorithm with the running time O(n2) is 50000. We have learned how to analyze the algorithms we use with this project. When choosing the algorithm, we have to make the choice according to the available memory and time constraints for finding the efficient one. We've used the number of data in execution time and loops to analyze algorithms. So we've found the most efficient algorithm.