National Polytechnic Institute
Superior School of Computer Sciences

Algorithm Analysis.

# Practice 6 - Maximum Subarray Problem.

*Hernandez Martinez Carlos David.*
*Burciaga Ornelas Rodrigo Andres.*

*davestring@outlook.com.*
*andii_burciaga@live.com.*

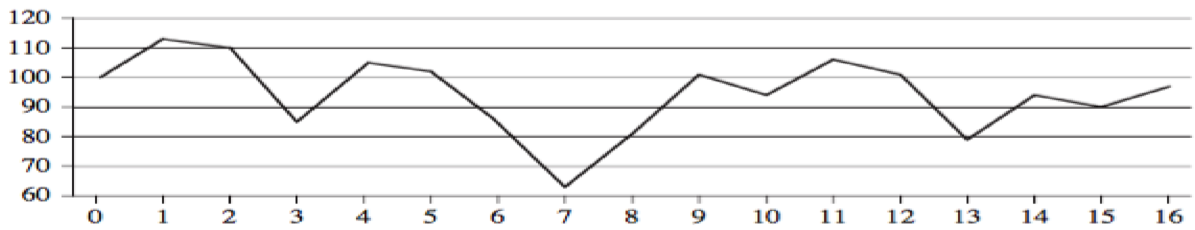
*Group: 2cv3.*

October 19, 2017

# Contents

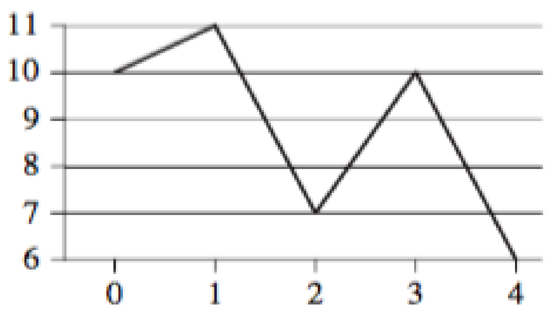
# 1 Introduction

Suppose that you been offered the opportunity to invest in the Volatile Chemical Corporation. Like the chemicals the company produces, the stock price of the Volatile Chemical Corporation is rather volatile. You are allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day. To compensate for this restriction, you are allowed to learn what the price of the stock will be in the future. Your goal is to maximize your profit. Figure 1 shows the price of the stock over a 17-day period. You may buy the stock at any one time, starting after day 0, when the price is 100 dollars per share. Of course, you would want to "buy low, sell high" buy at the lowest possible price and later on sell at the highest possible price to maximize your profit. Unfortunately, you might not be able to buy at the lowest price and then sell at the highest price within a given period. In Figure 1, the lowest price occurs after day 7, which occurs after the highest price, after day 1. You might think that you can always maximize profit by either buying at the lowest price or selling at the highest price. For example, in Figure 1, we would maximize profit by buying at the lowest price, after day 7. If this strategy always worked, then it would be easy to determine how to maximize profit: find the highest and lowest prices, and then work left from the highest price to find the lowest prior price, work right from the lowest price to find the highest later price, and take the pair with the greater difference. Figure 2 shows a simple counterexample demonstrating that the maximum profit sometimes comes neither by buying at the lowest price nor by selling at the highest price.




| Day | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Price | 100 | 113 | 110 | 85 | 105 | 102 | 86 | 63 | 81 | 101 | 94 | 106 | 101 | 79 | 94 | 90 | 97 |
| Change | | 13 | −3 | −25 | 20 | −3 | −16 | −23 | 18 | 20 | −7 | 12 | −5 | −22 | 15 | −4 | 7 |

Figure 1: Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.




| Day | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Price | 10 | 11 | 7 | 10 | 6 |
| Change | | 1 | −4 | 3 | −4 |

Figure 2: An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of 3 dollars per share would be earned by buying after day 2 and selling after day 3. The price of 7 dollars after day 2 is not the lowest price overall, and the price of 10 dollars after day 3 is not the highest price overall.

# 2 Basic Concepts:

The ***Strassen's*** algorithm, implements the divide-and-conquer paradigm.

## 2.1 Divide-and-Conquer Paradigm:

The divide-and-conquer paradigm involves three steps at each level of the recursion:

- ***Divide:*** Divide the problem into a number of sub-problems that are smaller instances of the same problem.
- ***Conquer:*** Conquer the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub-problems in a straightforward manner.
- ***Combine:*** Combine the solutions to the sub-problems into the solution for the original problem.

# 3 Development:

In this section we will implement the **Maximum Subarray** algorithms. We divided the program in 4 python modules to have a better control of the code.

- ***main.py:*** Control the sequence of execution.
- ***maximum_subarray.py:*** This module implements the 3 different algorithms to find a solution to the *Maximum Subarray* problem.
- ***plot.py:*** Plot the computational time of the algorithms.
- ***global_variables.py:*** As the name of the module, stores the global variables of the program.

***Observation:*** *The code that we will show bellow doesn't include the counter in each line of the algorithm, this because to make notice only the essential parts. For that, see subsection 3.4.*

### 3.1 Main.py

This module has the *__main__* implementation, as we can see in the code bellow, in line 2 calls the method **create ( ... )** and stores the returning value in a variable **A** which is nothing more than an array. The variables **high** and **mid** stores the length of **A** and the length divided by 2 respectively, then, in lines 7, 11, 15 we call the 3 algorithms to find the *Maximum Subarray* ( See section 3.2 ), and for each algorithm its result will be shown on the screen thanks to the **printer ( ... )** method. Finally, to plot the temporal complexity for the 3 algorithms we call the method **plot ( ... )** ( See section 3.3 and 3.4 ).

```
1  if ( __name__ == "__main__" ):
2      A = create ( )
3      high = int ( len ( A ) )
4      mid = int ( high / 2 )
5
6      # Find the maximum subarray using a Brute-Force Algorithm.
7      max_left , max_right , result = brute_force ( A )
8      printer ( A, max_left , max_right , result , 1 )
9
10     # Find the maximum subarray usign a Crossing Algorithm.
11     max_left , max_right , result = crossing ( A, 0, mid , high )
12     printer ( A, max_left , max_right , result , 2 )
13
14     # Find the maximum subarray usign a Recurrence Algorithm.
15     max_left , max_right , result = recurrence ( A, 0, high )
16     printer ( A, max_left , max_right , result , 3 )
17
18     plot ( )
```

Above there were mentioned 2 methods: **create ( ... )** and **printer ( ... )**. The first one as it name says, creates and returns an array of size $2^{10}$ of random positive and negative integers.

```
1  def create ( ):
2      n = 2 ** 10
3      A = [ random.randint ( -n, n ) for i in range ( n ) ]
4      return A
```

The second method only displays on screen the input array, the maximum subarray found and the sum of this last. As we can see in line 1, the function receive as parameter a variable **flag** which stores a integer. If it's equals to 1, then, the program will know that the other parameters received are from the **Brute-Force** solution, if it's equals to 2, then, are from the **Crossing** solution, and for any other case, are from the **Recursive** implementation.

```
1  def printer ( A, max_left , max_right , result , flag ):
2      if ( flag == 1 ):
3          print ( __FORMAT_1 )
4          print ( "\n\tArray: {}".format ( A ) )
5      elif ( flag == 2 ):
6          print ( __FORMAT_2 )
7      else:
8          print ( __FORMAT_3 )
9      print ( "\n\tMaximum Subarray: {}".format ( A [ max_left : max_right + 1 ] ) )
10     print ( "\n\tSum: {}\n\n".format ( result ) )
```

In the latest code, we can see in lines 3, 6, 8 a call to the method *print* that has as parameter 3 variables:

- **__FORMAT_1**.
- **__FORMAT_2**.
- **__FORMAT_3**.

This variables only store a *String* to give format to our output.

### 3.2 Maximum_subarray.py

Has we have mentioned this module implements 3 different algorithms to find a solution to the *Maximum Subarray Problem.*

#### 3.2.1 A Brute-Force Solution:

This algorithm has a running time of $\theta$ ( $n^2$ ). As we can see in the code bellow, this algorithm only tries every possible combination thanks to the *for* loops declared in lines 10 and 11.

```
1  def brute_force ( A ):
2      sums = [ 0 ]
3      for i in A:
4          sums.append ( sums [ -1 ] + i )
5  
6      max_sum = int ( -1e100 )
7      left_index = -1
8      right_index = -1
9      # Search for the maximum subarray indexes.
10     for i in range ( len ( A ) ):
11         for j in range ( i , len ( A ) ):
12             if ( sums [ j + 1 ] - sums [ i ] > max_sum ):
13                 max_sum = sums [ j + 1 ] - sums [ i ]
14                 left_index = i
15                 right_index = j
16     # Return statement.
17     return left_index , right_index , max_sum
```

#### 3.2.2 A Crossing Solution:

The following code in line 1 receive as parameters the input array **A** and the lowest, middle and highest positions of A and it returns a tuple containing the indices demarcating a maximum subarray that crosses the midpoint, along with the sum of the values in a maximum subarray.

```
1  def crossing ( A, low , mid , high ):
2      left_sum = int ( -1e100 )
3      _sum = 0
4      for i in range ( mid - 1, low - 1, -1 ):
5          _sum = _sum + A [ i ]
6          if ( _sum > left_sum ):
7              left_sum = _sum
8              max_left = i
9  
10     right_sum = int ( -1e100 )
11     _sum = 0
12     for i in range ( mid , high ):
13         _sum = _sum + A [ i ]
14         if ( _sum > right_sum ):
15             right_sum = _sum
16             max_right = i
17     # Return statement.
18     return max_left , max_right , right_sum + left_sum
```

This procedure works as follows. Lines 2 - 8 find a maximum subarray of the left half, A [ low ... mid ]. Since this subarray must contain A [ mid ], the **for** loop of lines 4 - 8 starts the index *i* at mid and works down to low, so that every subarray it considers is of the form A [ i ... mid ]. Lines 2 - 3 initialize the variables *left_sum*, which holds the greatest sum found so far, and sum, holding the sum of the entries in A [ i ... mid ]. Whenever we find, in line 6, a subarray A [ i ... mid ], with a sum of values greater than *left_sum*, we update *left_sum* to this subarrays sum in line 7, and in line 8 we update the variable *max_left* to record this index *i*. Lines 10 - 16 work analogously for the right half, A [ mid + 1 ... high ]. Here, the **for** loop of lines 12 - 16 starts the index *j* at *mid + 1* and works up to *high*, so that every subarray it considers is of the form A [ mid + 1 ... j ]. Finally, line 18 returns the indices *max_left* and *max_right* that demarcate a maximum subarray crossing the midpoint, along with the sum *left_sum + right_sum* of the values of the subarray A [ max_left ... max_right ].

### 3.2.3 A Divide-and-Conquer Solution:

The initial call to RECURRENCE ( A, 1, A.length ) will find a maximum subarray of A [ 1 ... n ]. Similar to CROSSING, the recursive procedure RECURRENCE returns a tuple containing the indices that demarcate a maximum subarray, along with the sum of the values in a maximum subarray.

```
1  def recurrence ( A, low , high ):
2      if ( high == low + 1 ):
3          return low , high , A [ low ]
4      else:
5          mid = int ( ( low + high ) / 2 )
6          left_low , left_high , left_sum = recurrence( A, low , mid )
7          right_low , right_high , right_sum = recurrence ( A, mid , high )
8          cross_low , cross_high , cross_sum = crossing ( A, low , mid , high )
9          if ( left_sum >= right_sum and left_sum >= cross_sum ):
10             return left_low , left_high , left_sum
11         elif ( right_sum >= left_sum and right_sum >= cross_sum ):
12             return right_low , right_high , right_sum
13         else:
14             return cross_low , cross_high , cross_sum
```

Line 2 tests for the base case, where the subarray has just one element. A subarray with just one element has only one subarray - itself - and so line 3 returns a tuple with the starting and ending indices of just the one element, along with its value. Lines 5 - 14 handle the recursive case. Line 5 does the divide part, computing the index mid of the midpoint. Lets refer to the subarray A [ low ... mid ] as the ***left-subarray*** and to A [ mid + 1 ... high ] as the ***right-subarray***. Because we know that the subarray A [ low ... high ] contains at least two elements, each of the left and right subarrays must have at least one element. Lines 6 and 7 conquer by recursively finding maximum subarrays within the left and right subarrays, respectively. Lines 9 - 14 form the combine part. Line 8 finds a maximum subarray that crosses the midpoint. ( Recall that because line 8 solves a subproblem that is not a smaller instance of the original problem, we consider it to be in the combine part.) Line 9 tests whether the left subarray contains a subarray with the maximum sum, and line 10 returns that maximum subarray. Otherwise, line 11 tests whether the right subarray contains a subarray with the maximum sum, and line 12 returns that maximum subarray. If neither the left nor right subarrays contain a subarray achieving the maximum sum, then a maximum subarray must cross the midpoint, and line 14 returns it.

## 3.3 Plot.py

This module plot the temporal complexity of the algorithms, which for the *Brute-Force* it's $\theta\ (\ n^2\ )$, *Crossing* is $\theta\ (\ n\ )$ and *Recurrence* has $\theta\ (\ n \cdot log_2\ (\ n\ )\ )$. The procedure works as follows, in line 18 makes a call to the function *initialize ( )*, this function returns the lists where $s\_1$, $s\_2$, $s\_3$ are the sizes of the array, $t\_1$, $t\_2$, $t\_3$ are the computational time and $p\_1$, $p\_2$, $p\_3$ are the proposed functions for brute-force, crossing and recurrence respectively. This parameters are extracted from the lists of tuples *parameters_1*, *parameters_2* and *parameters_3* as we can see in lines 3 - 13. Lines 23, 34 and 45 divide the plot into subplots which the first will be for *Brute-Force*, the second for *Crossing* and finally the third for *Recurrence*. Once the parameters are set, lines 28 - 29, 39 - 40 and 49 - 50 plot the temporal complexity for each algorithm.

```
1  def initialize ( ):
2      # Parameters S ( n ) -size-.
3      s_1 = list ( map ( lambda x:x [ 0 ], gb.parameters_1 ) )
4      s_2 = list ( map ( lambda x:x [ 0 ], gb.parameters_2 ) )
5      s_3 = list ( map ( lambda x:x [ 0 ], gb.parameters_3 ) )
6      # Parameters T ( t ) -time-.
7      t_1 = list ( map ( lambda x:x [ 1 ], gb.parameters_1 ) )
8      t_2 = list ( map ( lambda x:x [ 1 ], gb.parameters_2 ) )
9      t_3 = list ( map ( lambda x:x [ 1 ], gb.parameters_3 ) )
10     # Proposed functions.
11     p_1 = list ( map ( lambda x: ( 10/8 ) * x, t_1 ) )
12     p_2 = list ( map ( lambda x: ( 10/8 ) * x, t_2 ) )
13     p_3 = list ( map ( lambda x: ( 10/8 ) * x, t_3 ) )
14     return s_1, s_2, s_3, t_1, t_2, t_3, p_1, p_2, p_3
15
16 def plot ( ):
17     # Initialize the plot points.
18     s_1, s_2, s_3, t_1, t_2, t_3, p_1, p_2, p_3 = initialize ( )
19     plt.figure ( "Maximum Subarray Problem", figsize = ( 14, 7 ) )
20
21     # BRUTE-FORCE MAXIMUM SUBARRAY ALGORITHM PLOT.
22
23     plt.subplot ( 1, 3, 1 )
24     # Figure title.
25     plt.title ( "Brute-Force Max Subarray ( {}, {} )".format ( gb.parameters_1 [ -1 ] [ 0 ],
26         gb.parameters_1 [ -1 ] [ 1 ] ), size = "small" )
27     plt.ylabel ( "Time ( t )", color = ( 0.3, 0.4, 0.6 ), family = "cursive" )
28     plt.plot ( s_1, p_1, "#000000", linestyle = "--", label = "3/2 n^2" )
29     plt.plot ( s_1, t_1, "#0B3B0B", linewidth = 3, label = "n^2" )
30     plt.legend ( loc = "upper left" )
31
32     # MAXIMUM CROSSING SUBARRAY ALGORITHM PLOT.
33
34     plt.subplot ( 1, 3, 2 )
35     # Figure title.
36     plt.title ( "Max Crossing Subarray ( {}, {} )".format ( gb.parameters_2 [ -1 ] [ 0 ],
37         gb.parameters_2 [ -1 ] [ 1 ] ), size = "small" )
38     plt.xlabel ( "Size ( n )", color = ( 0.3, 0.4, 0.6 ), family = "cursive" )
39     plt.plot ( s_2, p_2, "#000000", linestyle = "--", label = "3/2 n" )
40     plt.plot ( s_2, t_2, "#610B0B", linewidth = 3, label = "n" )
41     plt.legend ( loc = "upper left" )
42
43     # MAXIMUM SUBARRAY RECURENCE ALGORITHM PLOT.
44
45     plt.subplot ( 1, 3, 3 )
46     # Figure title.
47     plt.title ( "Max Subarray ( {}, {} )".format ( gb.parameters_3 [ -1 ] [ 0 ],
48         gb.parameters_3 [ -1 ] [ 1 ] ), size = "small" )
49     plt.plot ( s_3, p_3, "#000000", linestyle = "--", label = "3/2 n log ( n )" )
50     plt.plot ( s_3, t_3, "#4C0B5F", linewidth = 3, label = "n log ( n )" )
51     plt.legend ( loc = "upper left" )
52
53     plt.show ( )
```

### 3.4 Calculating The Temporal Complexity:

To calculate the temporal complexity of the 3 algorithms it's necessary to put a counter in each line of the codes and store the results in a list, where each element it's a tuple that in its first element stores the size of the arrays and in the second element stores the counter. In ***global_variables.py*** are declared the variables that we will use for this purpose:

(i) **parameters_1**: List that stores the parameters of the points to plot for the Brute-Force Maximum Subarray Algorithm. Each element it's a tuple that stores the size of the Array and the temporal complexity in its first and second element respectively.

(ii) **parameters_2**: List that stores the parameters of the points to plot for the Maximum Crossing Subarray Algorithm. Each element it's a tuple that stores the size of the Array and the temporal complexity in its first and second element respectively.

(iii) **parameters_3**: List that stores the parameters of the points to plot for the Maximum Subarray Algorithm using Recursion. Each element it's a tuple that stores the size of the Array and the temporal complexity in its first and second element respectively.

(iv) **time**: Variable that stores the temporal complexity of each algorithm.

As we can see, the following codes are exactly the same showed in section 3.2 but with the counter **time** placed in each line:

```python
def brute_force ( A ):
    gb.time += 1
    sums = [ 0 ]
    gb.time += 1
    for i in A:
        gb.time += 1
        sums.append ( sums [ -1 ] + i )
        gb.time += 1
    gb.time += 1
    max_sum = int ( -1e100 )
    gb.time += 1
    left_index = -1
    gb.time += 1
    right_index = -1
    gb.time += 1
    for i in range ( len ( A ) ):
        gb.time += 1
        for j in range ( i , len ( A ) ):
            gb.time += 1
            if ( sums [ j + 1 ] - sums [ i ] > max_sum ):
                gb.time += 1
                max_sum = sums [ j + 1 ] - sums [ i ]
                gb.time += 1
                left_index = i
                gb.time += 1
                right_index = j
                gb.time += 1
            gb.time += 1
        gb.time += 1
    # Return statement.
    gb.time += 1
    return left_index , right_index , max_sum
```

```python
def crossing ( A, low, mid, high ):
    gb.time += 1
    max_left, max_right = 0, 0
    gb.time += 1
    left_sum = int ( -1e100 )
    gb.time += 1
    _sum = 0
    gb.time += 1
    for i in range ( mid - 1, low - 1, -1 ):
        gb.time += 1
        _sum = _sum + A [ i ]
        gb.time += 1
        if ( _sum > left_sum ):
            gb.time += 1
            left_sum = _sum
            gb.time += 1
            max_left = i
        gb.time += 1

    gb.time += 1
    right_sum = int ( -1e100 )
    gb.time += 1
    _sum = 0
    gb.time += 1
    for i in range ( mid, high ):
        gb.time += 1
        _sum = _sum + A [ i ]
        gb.time += 1
        if ( _sum > right_sum ):
            gb.time += 1
            right_sum = _sum
            gb.time += 1
            max_right = i
        gb.time += 1
    # Return statement.
    gb.time += 1
    return max_left, max_right, right_sum + left_sum
```

```python
def recurrence ( A, low, high ):
    gb.time += 1
    if ( high == low + 1 ):
        gb.time += 1
        return low, high, A [ low ]
    else:
        gb.time += 1
        mid = int ( ( low + high ) / 2 )
        gb.time += 1
        left_low, left_high, left_sum = recurrence ( A, low, mid )
        gb.time += 1
        right_low, right_high, right_sum = recurrence ( A, mid, high )
        gb.time += 1
        cross_low, cross_high, cross_sum = crossing ( A, low, mid, high )
        gb.time += 1
        if ( left_sum >= right_sum and left_sum >= cross_sum ):
            gb.time += 1
            return left_low, left_high, left_sum
        elif ( right_sum >= left_sum and right_sum >= cross_sum ):
            gb.time += 1
            return right_low, right_high, right_sum
        else:
            gb.time += 1
            return cross_low, cross_high, cross_sum
```

With the counter now set, the main method needs to be modified too. In line 2 prevails the call to the method *create ( )* which return an array **A** of random positive and negative integers of size A [ low ... high ]. In lines 4, 10 and 18 there are 3 **for** loops that will run from 0 to *high*. In lines 5, 13 and 20 the program call the algorithms *brute_force ( ... )*, *crossing ( ... )* and *recurrence ( ... )* respectively, but, if we see in the parameters of each function we send **A** but as a subarray A [ 0 ... i ] where *i* it's the **for** variable, this will allow us to find the plot points for each size of the array until reaching the original highest position. In lines 6, 14 and 21 the sizes of this subarrays and the temporal time calculated will be stored in the lists ***parameters_*** $\alpha$ as a tuple, where $\alpha$ can be 1, 2 or 3 with respect to which algorithm its evaluating. Then in lines 7, 15, and 22 the counter **time** is reset.

```
1  if ( __name__ == "__main__" ):
2      A = create ( )
3      # Find the maximum subarray using a Brute-Force Algorithm.
4      for i in range ( len ( A ) ):
5          max_left , max_right , result = brute_force ( A [ :i ] )
6          gb.parameters_1.append ( ( len ( A [ :i ] ), gb.time ) )
7          gb.time = 0
8      printer ( A, max_left , max_right , result , 1 )
9      # Find the maximum subarray usign a Crossing Algorithm.
10     for i in range ( len ( A ) ):
11         high = int ( len ( A [ :i ] ) )
12         mid = int ( high / 2 )
13         max_left , max_right , result = crossing ( A [ :i ], 0, mid , high )
14         gb.parameters_2.append ( ( len ( A [ :i ] ), gb.time ) )
15         gb.time = 0
16     printer ( A, max_left , max_right , result , 2 )
17     # Find the maximum subarray usign a Recurrence Algorithm.
18     for i in range ( 1, len ( A ) ):
19         high = int ( len ( A [ :i ] ) )
20         max_left , max_right , result = recurrence ( A [ :i ], 0, high )
21         gb.parameters_3.append ( ( len ( A [ :i ] ), gb.time ) )
22         gb.time = 0
23     printer ( A, max_left , max_right , result , 3 )
24     plot ( )
```

# 4 Results:

All the code shown above doesn't have significance if its operation is not shown. This section will show you the ***console*** output and the graphic of the temporal complexity of the algorithms previously mentioned. Also, I attach a table with the plot points for each test.

***Observation:*** *From this point in each plot figure that we were analyzing the left plot will correspond to the **Brute-Force** algorithm, the plot of the center to the **Crossing Subarray** algorithm and the right one to the **Recursive Maximum Subarray** algorithm.*

***Observation:*** *From this point in each plot figure that we were analyzing the black pointed functions will be the asymptotic proposed ones, where for the left plot it's* $\boldsymbol{g}\ (\ \boldsymbol{n}\ ) = \frac{10}{8} \cdot n^2$, *for the center will be* $\boldsymbol{h}\ (\ \boldsymbol{n}\ ) = \frac{10}{8}\ \boldsymbol{n}$ *and for the right plot will be* $\boldsymbol{s}\ (\ \boldsymbol{n}\ ) = \frac{10}{8} \cdot n \cdot log_2(n)$.

## 4.1 Positive Ordered Integers Array:

The first test will consist in evaluate the running time of the algorithms analyzing an array of ordered integers of size $2^{10}$.

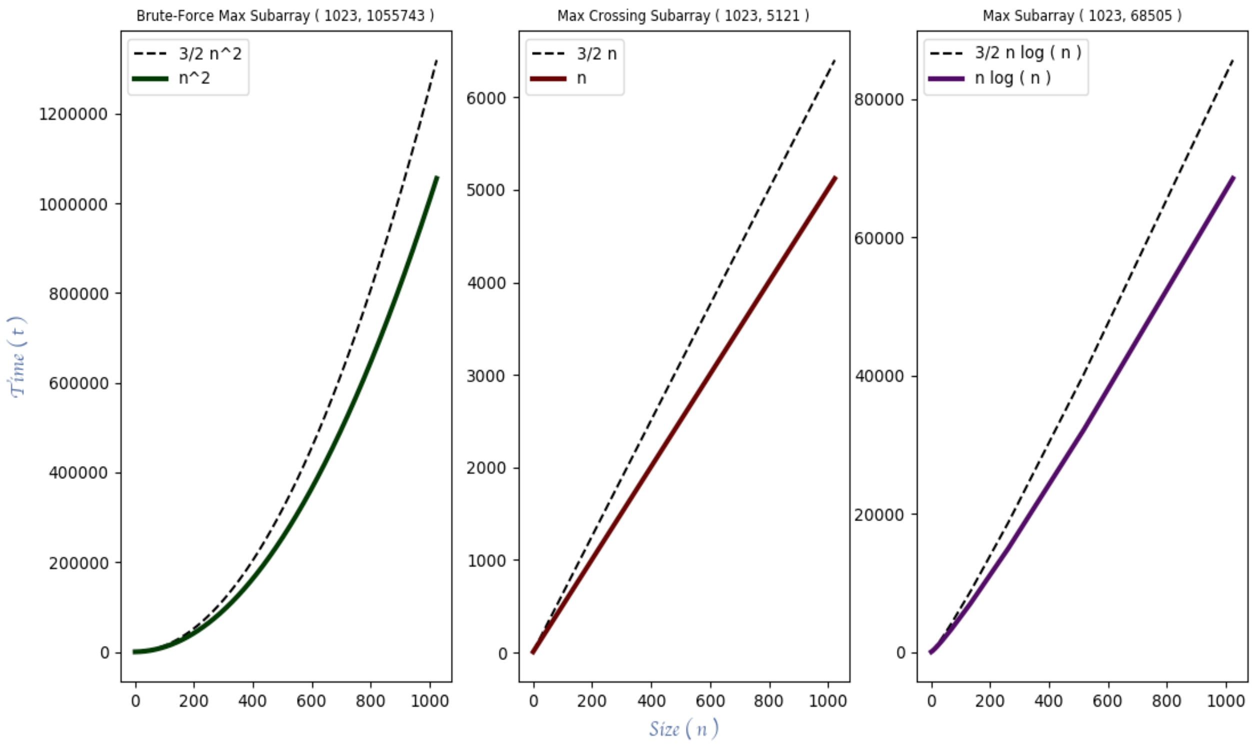



Figure 4.1.0: Algorithms results for an array of size $2^{10}$.

***Observation:*** *Since the size it's* $2^{10}$ *the parameters of the points of each algorithm are to many, so we decide to not attach only for this case the table of mapping values and the console output only the plot.*

The second test will consist in evaluate the running time of the algorithms analyzing an array of ordered integers of size $2^5$.


Figure 4.1.1: Console output of the program.

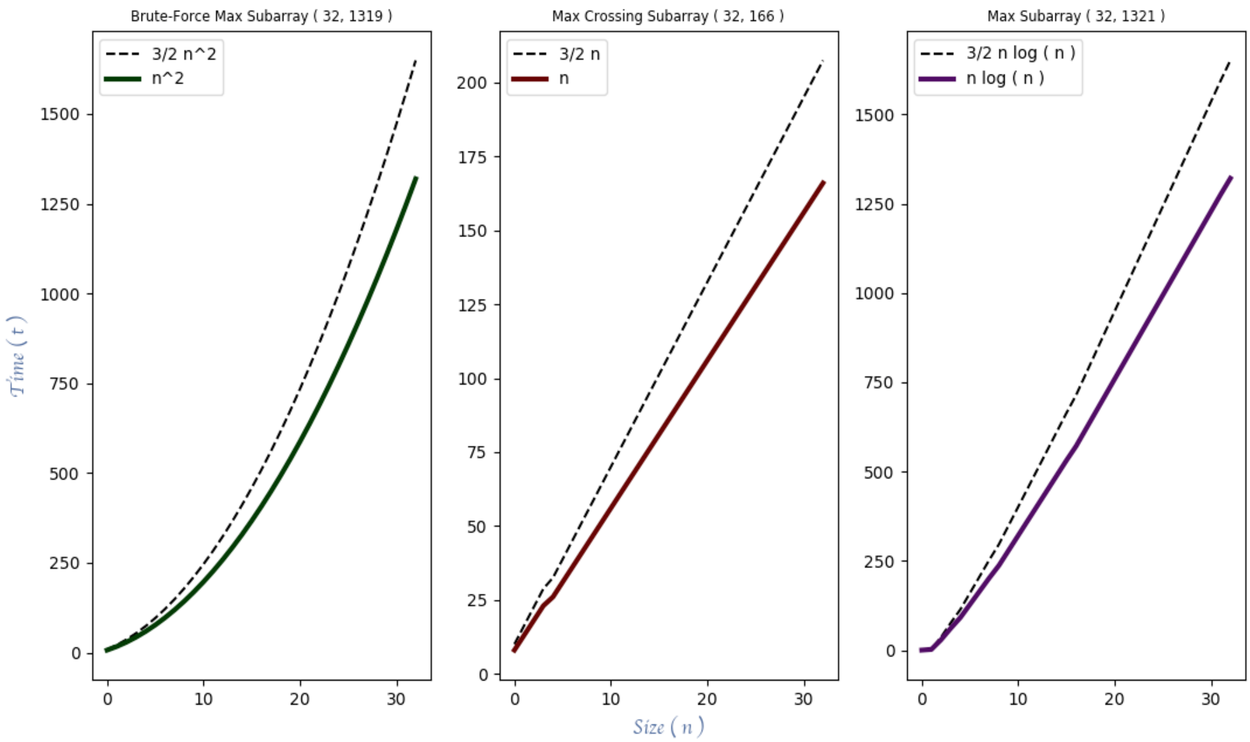



Figure 4.1.2: Algorithms running time for an array of size $2^5$.

The following table shows the points of the plots where the first column it's the *Size* of the array, the second, third and fourth columns describes the computational time of *Brute-Force*, *Crossing* and *Recurrence* Maximum Subarray Algorithms respectively.

| Size ( n ) | Brute-Force Time ( t ) | Crossing Time ( t ) | Recurrence Time ( t ) |
|---|---|---|---|
| 0 | 7 | 8 | 0 |
| 1 | 17 | 13 | 2 |
| 2 | 29 | 18 | 29 |
| 3 | 43 | 23 | 61 |
| 4 | 59 | 26 | 91 |
| 5 | 77 | 31 | 128 |
| 6 | 97 | 36 | 165 |
| 7 | 119 | 41 | 202 |
| 8 | 143 | 46 | 237 |
| 9 | 169 | 51 | 279 |
| 10 | 197 | 56 | 321 |
| 11 | 227 | 61 | 363 |
| 12 | 259 | 66 | 405 |
| 13 | 293 | 71 | 447 |
| 14 | 329 | 76 | 489 |
| 15 | 367 | 81 | 531 |
| 16 | 407 | 86 | 571 |
| 17 | 449 | 91 | 618 |
| 18 | 493 | 96 | 665 |
| 19 | 539 | 101 | 712 |
| 20 | 587 | 106 | 759 |
| 21 | 637 | 111 | 806 |
| 22 | 689 | 116 | 853 |
| 23 | 743 | 121 | 900 |
| 24 | 799 | 126 | 947 |
| 25 | 857 | 131 | 994 |
| 26 | 917 | 136 | 1041 |
| 27 | 979 | 141 | 1088 |
| 28 | 1043 | 146 | 1135 |
| 29 | 1109 | 151 | 1182 |
| 30 | 1177 | 156 | 1229 |
| 31 | 1247 | 161 | 1276 |
| 32 | 1319 | 166 | 1321 |

Table 1: Plot points for Figure 4.1.2.

## 4.2 Random Integers:

The third test will consist in evaluate the running time of the algorithms analyzing an array of disordered random positive and negative integers of size $2^5$.

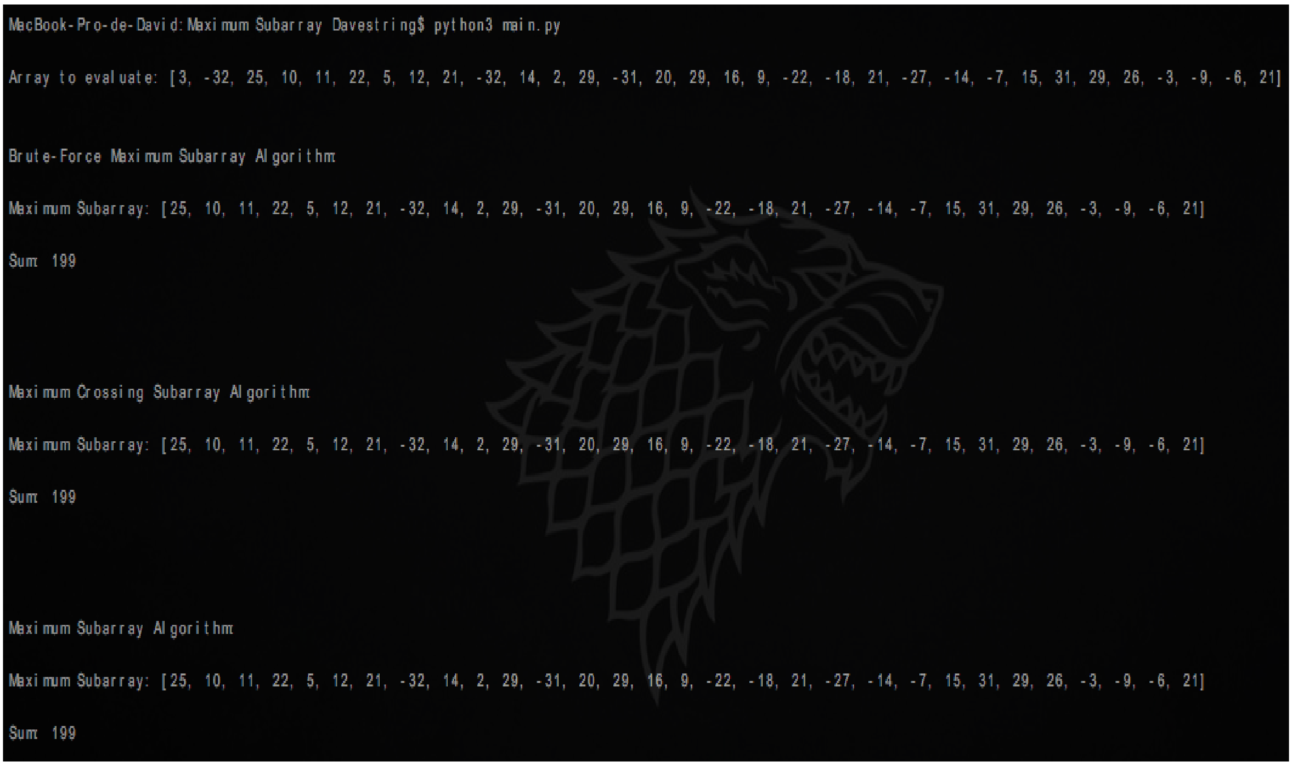



Figure 4.2.0: Console output of the program.

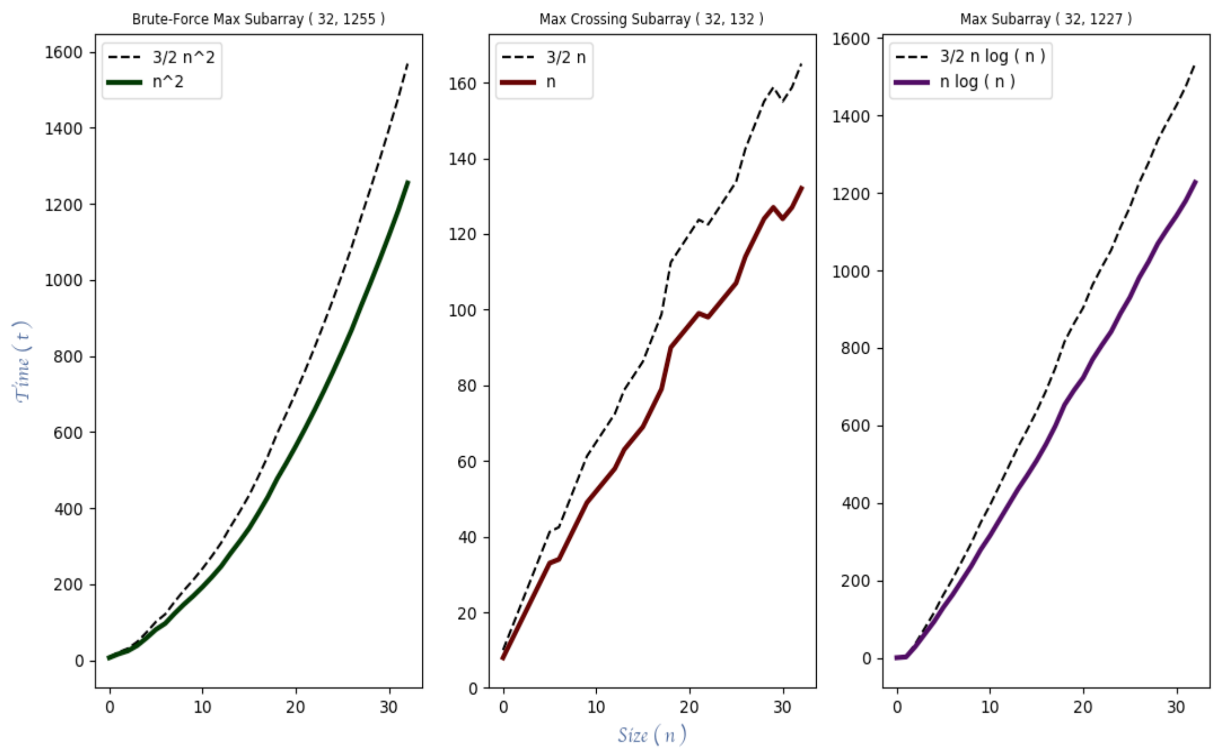



Figure 4.2.1: Algorithms running time for an array of size $2^5$.

The following table shows the points of the plots where the first column it's the *Size* of the array, the second, third and fourth columns describes the computational time of *Brute-Force*, *Crossing* and *Recurrence* Maximum Subarray Algorithms respectively.

| Size ( n ) | Brute-Force Time ( t ) | Crossing Time ( t ) | Recurrence Time ( t ) |
|---|---|---|---|
| 0 | 7 | 8 | 0 |
| 1 | 17 | 13 | 2 |
| 2 | 25 | 18 | 29 |
| 3 | 39 | 23 | 61 |
| 4 | 59 | 28 | 93 |
| 5 | 81 | 33 | 130 |
| 6 | 97 | 34 | 163 |
| 7 | 123 | 39 | 200 |
| 8 | 147 | 44 | 237 |
| 9 | 169 | 49 | 279 |
| 10 | 193 | 52 | 315 |
| 11 | 219 | 55 | 355 |
| 12 | 247 | 58 | 395 |
| 13 | 281 | 63 | 435 |
| 14 | 313 | 66 | 471 |
| 15 | 347 | 69 | 509 |
| 16 | 387 | 74 | 551 |
| 17 | 429 | 79 | 598 |
| 18 | 477 | 90 | 653 |
| 19 | 519 | 93 | 690 |
| 20 | 563 | 96 | 723 |
| 21 | 609 | 99 | 770 |
| 22 | 657 | 98 | 807 |
| 23 | 707 | 101 | 842 |
| 24 | 759 | 104 | 889 |
| 25 | 813 | 107 | 930 |
| 26 | 869 | 114 | 981 |
| 27 | 931 | 119 | 1022 |
| 28 | 991 | 124 | 1069 |
| 29 | 1053 | 127 | 1106 |
| 30 | 1117 | 124 | 1141 |
| 31 | 1183 | 127 | 1180 |
| 32 | 1255 | 132 | 1227 |

Table 2: Plot points of Figure 4.2.1.

The fourth test will consist in evaluate the running time of the algorithms analyzing an array of disordered random positive and negative integers of size $2^4$.

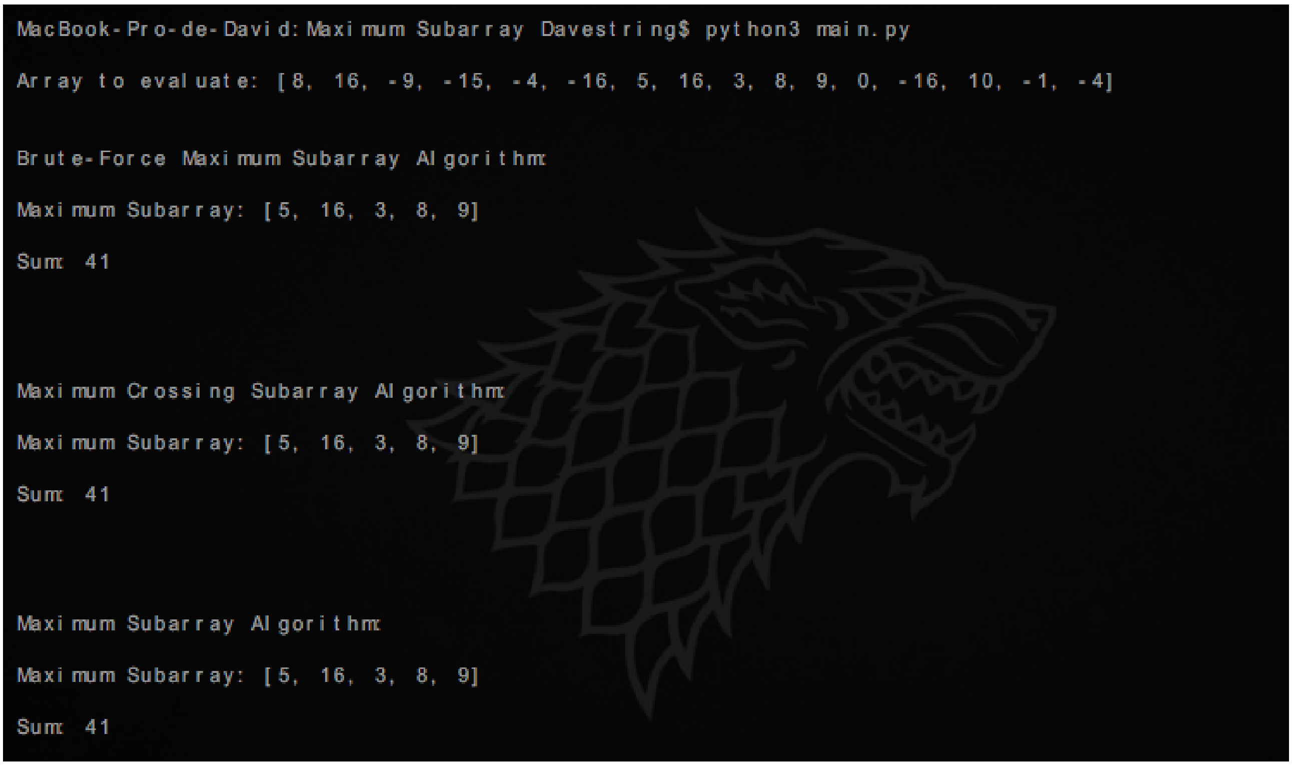



Figure 4.2.2: Console output of the program.

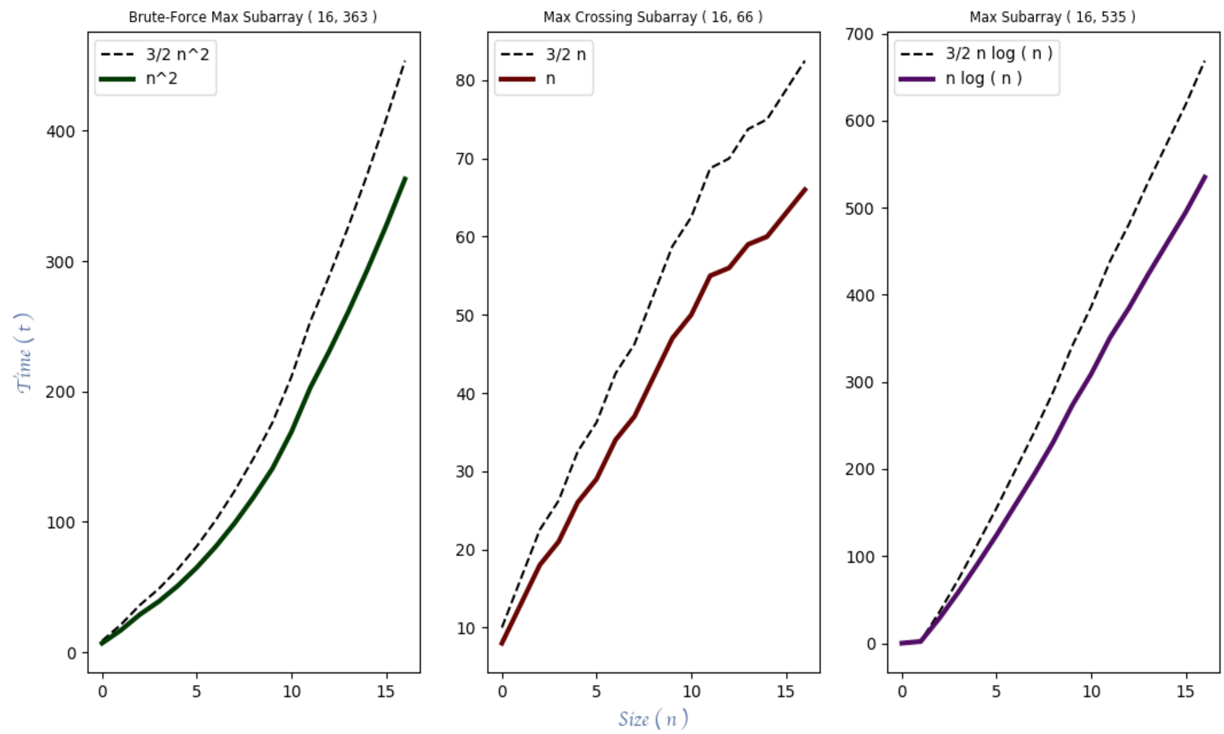



Figure 4.2.3: Algorithms running time for an array of size $2^4$.

The following table shows the points of the plots where the first column it's the *Size* of the array, the second, third and fourth columns describes the computational time of *Brute-Force*, *Crossing* and *Recurrence* Maximum Subarray Algorithms respectively.

| Size ( n ) | Brute-Force Time ( t ) | Crossing Time ( t ) | Recurrence Time ( t ) |
|---|---|---|---|
| 0 | 7 | 8 | 0 |
| 1 | 17 | 13 | 2 |
| 2 | 29 | 18 | 29 |
| 3 | 39 | 21 | 59 |
| 4 | 51 | 26 | 91 |
| 5 | 65 | 29 | 124 |
| 6 | 81 | 34 | 159 |
| 7 | 99 | 37 | 194 |
| 8 | 119 | 42 | 231 |
| 9 | 141 | 47 | 273 |
| 10 | 169 | 50 | 309 |
| 11 | 203 | 55 | 351 |
| 12 | 231 | 56 | 385 |
| 13 | 261 | 59 | 423 |
| 14 | 293 | 60 | 459 |
| 15 | 327 | 63 | 495 |
| 16 | 363 | 66 | 535 |

Table 3: Plot points of Figure 4.2.3.

## 4.3 Negative Integers:

The fifth test will consist in evaluate the running time of the algorithms analyzing an array of disordered random negative integers of size $2^5$.

```
Array to evaluate: [-4, -7, -29, -1, -17, -11, -23, -30, -14, -22, -6, -29, -3, -4, -24, -29, -15, -26, -2, -29, -26, -27, -25, -6, -26, -3, -15, -24, -13, -28, -6, -32]

Brute-Force Maximum Subarray Algorithm:

Maximum Subarray: [-1]

Sum: -1



Maximum Crossing Subarray Algorithm:

Maximum Subarray: [-29, -15]

Sum: -44



Maximum Subarray Algorithm:

Maximum Subarray: [-1, -17]

Sum: -1
```

Figure 4.3.0: Console output of the program.

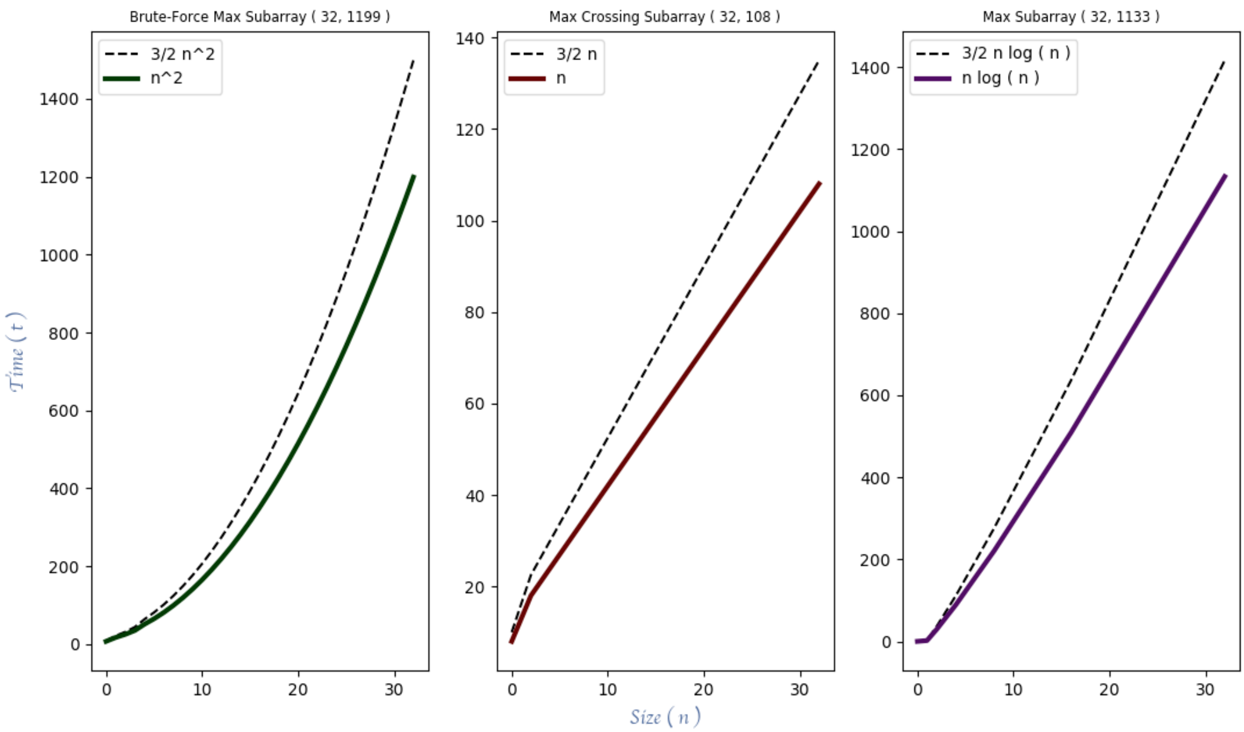



Figure 4.3.1: Algorithms running time for an array of size $2^5$.

The following table shows the points of the plots where the first column it's the *Size* of the array, the second, third and fourth columns describes the computational time of *Brute-Force*, *Crossing* and *Recurrence* Maximum Subarray Algorithms respectively.

| Size ( n ) | Brute-Force Time ( t ) | Crossing Time ( t ) | Recurrence Time ( t ) |
|---|---|---|---|
| 0 | 7 | 8 | 0 |
| 1 | 17 | 13 | 2 |
| 2 | 25 | 18 | 29 |
| 3 | 35 | 21 | 59 |
| 4 | 51 | 24 | 89 |
| 5 | 65 | 27 | 122 |
| 6 | 81 | 30 | 155 |
| 7 | 99 | 33 | 188 |
| 8 | 119 | 36 | 221 |
| 9 | 141 | 39 | 257 |
| 10 | 165 | 42 | 293 |
| 11 | 191 | 45 | 329 |
| 12 | 219 | 48 | 365 |
| 13 | 249 | 51 | 401 |
| 14 | 281 | 54 | 437 |
| 15 | 315 | 57 | 473 |
| 16 | 351 | 60 | 509 |
| 17 | 389 | 63 | 548 |
| 18 | 429 | 66 | 587 |
| 19 | 471 | 69 | 626 |
| 20 | 515 | 72 | 665 |
| 21 | 561 | 75 | 704 |
| 22 | 609 | 78 | 743 |
| 23 | 659 | 81 | 782 |
| 24 | 711 | 84 | 821 |
| 25 | 765 | 87 | 860 |
| 26 | 821 | 90 | 899 |
| 27 | 879 | 93 | 938 |
| 28 | 939 | 96 | 977 |
| 29 | 1001 | 99 | 1016 |
| 30 | 1065 | 102 | 1055 |
| 31 | 1131 | 105 | 1094 |
| 32 | 1199 | 108 | 1133 |

Table 4: Plot points of Figure 4.3.1.

The sixth test will consist in evaluate the running time of the algorithms analyzing an array of disordered random negative integers of size $2^4$.

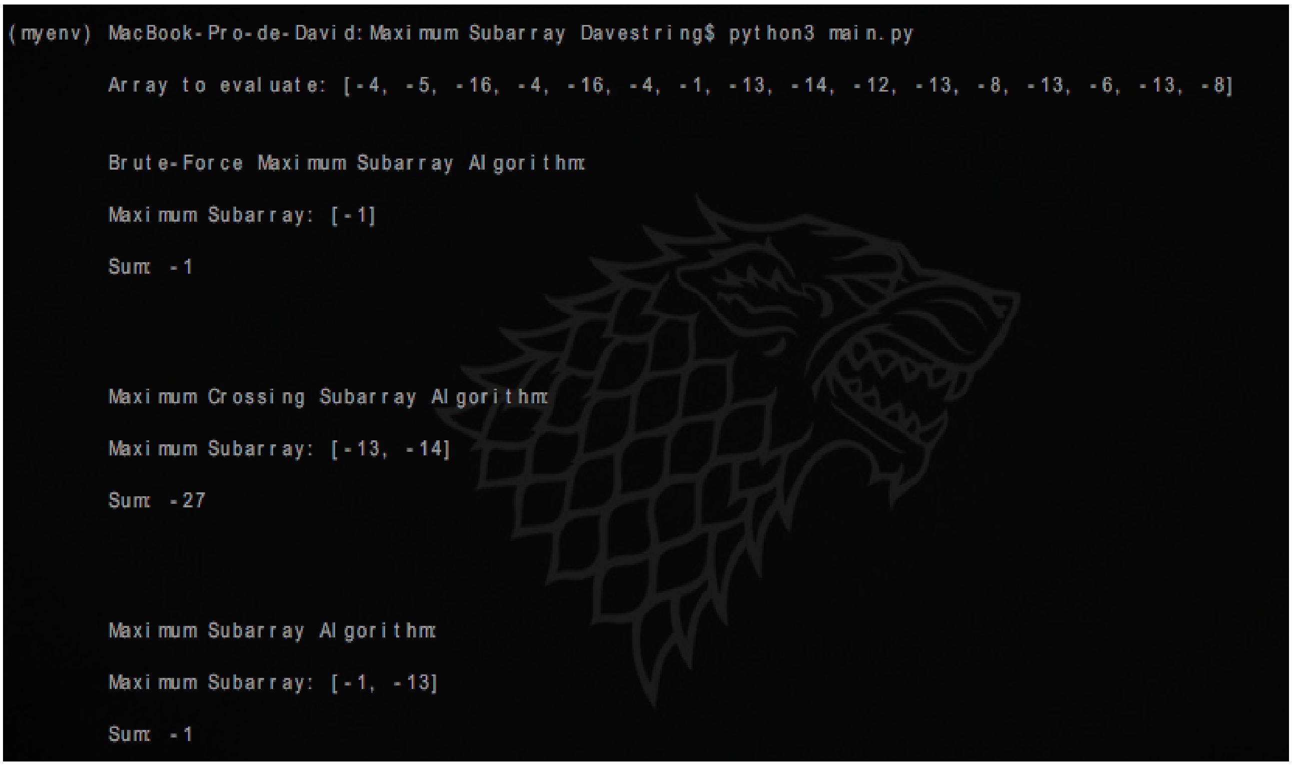



Figure 4.3.2: Console output of the program.

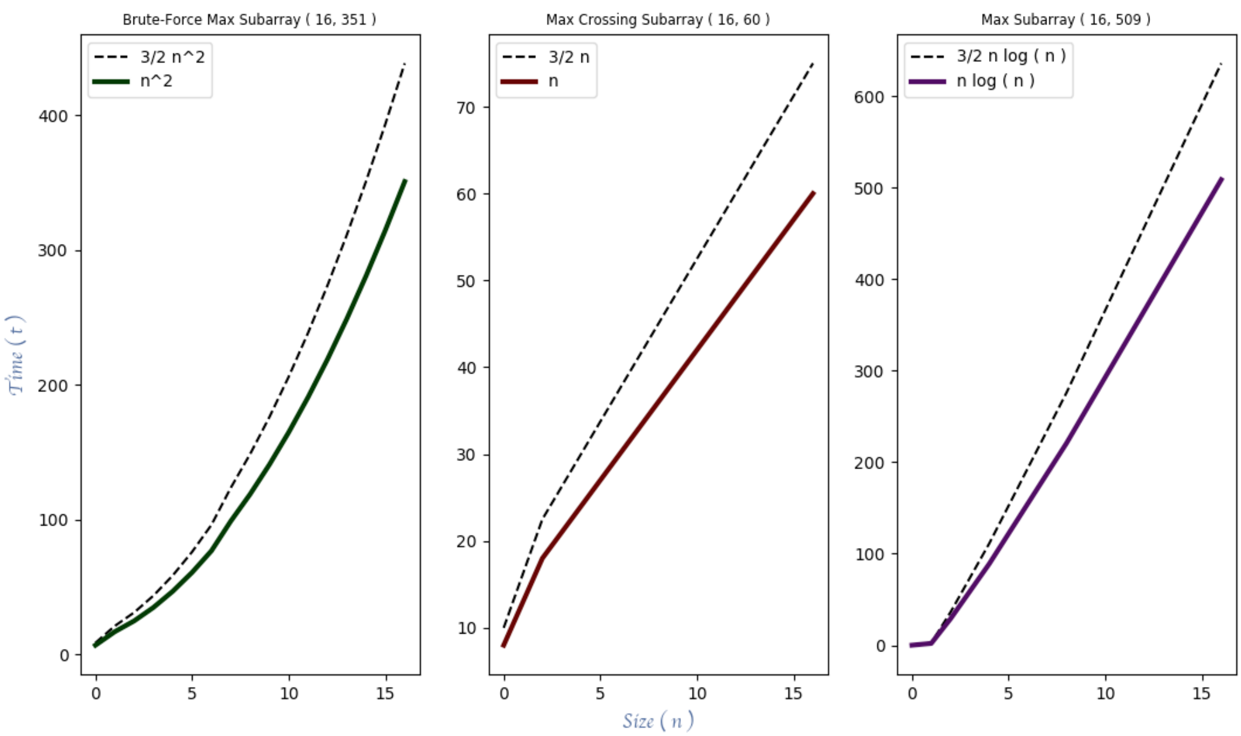



Figure 4.3.3: Algorithms running time for an array of size $2^4$.

The following table shows the points of the plots where the first column it's the *Size* of the array, the second, third and fourth columns describes the computational time of *Brute-Force*, *Crossing* and *Recurrence* Maximum Subarray Algorithms respectively.

| Size ( n ) | Brute-Force Time ( t ) | Crossing Time ( t ) | Recurrence Time ( t ) |
|---|---|---|---|
| 0 | 7 | 8 | 0 |
| 1 | 17 | 13 | 2 |
| 2 | 25 | 18 | 29 |
| 3 | 35 | 21 | 59 |
| 4 | 47 | 24 | 89 |
| 5 | 61 | 27 | 122 |
| 6 | 77 | 30 | 155 |
| 7 | 99 | 33 | 188 |
| 8 | 119 | 36 | 221 |
| 9 | 141 | 39 | 257 |
| 10 | 165 | 42 | 293 |
| 11 | 191 | 45 | 329 |
| 12 | 219 | 48 | 365 |
| 13 | 249 | 51 | 401 |
| 14 | 281 | 54 | 437 |
| 15 | 315 | 57 | 473 |
| 16 | 351 | 60 | 509 |

Table 5: Plot points of Figure 4.3.3.

# 5 Annexes:

In the following section we will formally demonstrate the complexity of the algorithms previously mentioned.

## 5.1 Maximum Subarray-Brute Force

Proposed algorithm for maximum subarray using Brute-Force.

**function** BRUTE-FORCE-MAXIMUM-SUBARRAY (A)

```
1 max = -∞
2 for i = 0 to A.lenght
3     sum = 0
4     for j = i to A.lenght
5         sum += A[j]
6         if ( sum > max )
7             max = sum
8 return max
```

- ***Demonstration:***
- ***Analyzing the complexity of each line:***
  1. Line 1 = $\theta$ ( 1 ).
  2. Line 2 = $\theta$ ( $n$ ).
     (i) line 3 = $\theta$ ( 1 ).
     (ii) line 4 = $\theta$ ( $n$ ).
     (iii) line 5, 6, 7 = $\theta$ ( 1 ).
  3. Line 8 = $\theta$ ( 1 ).

- ***Then, from all lines we can conclude:***

$$T\ (\ n\ )\ =\ \theta\ (\ n\ (\ 1\ +\ n\ +\ 1\ )\ )\ =\ \theta\ (\ n^2\ ) \tag{1}$$

- ***Finally:***

$$Brute - Force\ Maximum\ Subarray\ \in\ O\ (\ n^2\ ) \tag{2}$$

## 5.2 Maximum Crossing Subarray:

Demonstration that maximum crossing subarray has linear complexity: $\boldsymbol{T\ (\ n\ )\ =\ (\ n\ )}$ order.

**function** FIND-MAX-CROSSING-SUBARRAY (A, low, mid, high)

```
1  left_sum = −∞
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
5      if sum > left_sum
6          left_sum = sum
7          max_left = i
8  rigth_sum = −∞
9  sum = 0
10 for j = mid + 1 to high
11     sum = sum + A[j]
12     if sum > right_sum
13         right_sum = sum
14         max_right = j
15 return ( max_left , max_right , left_sum + right_sum )
```

- ***Demonstration:***
- ***Analyzing the complexity of each line:***
  1. Line 1, 2 = $\theta$ ( 1 ).
  2. Line 3, 4, 5, 6, 7 = $\theta$ ( $\frac{n}{2}$ ).
  3. Line 8, 9 = $\theta$ ( 1 ).
  4. Line 10, 11, 12, 13, 14 = $\theta$ ( $\frac{n}{2}$ ).
  5. Line 15 = $\theta$ ( 1 ).

- ***Then, from all lines:***

$$T\ (\ n\ )\ =\ \theta\ (\ \frac{n}{2}\ +\ \frac{n}{2}\ )\ =\ \theta\ (\ n\ ) \tag{3}$$

- ***Finally:***

$$Maximum\ Crossing\ Subarray\ \in\ \theta\ (\ n\ ) \tag{4}$$

### 5.3 Maximum Subarray:

Demonstration that maximum subarray has complexity $\boldsymbol{T\ (\ n\ )\ =\ (\ n\ log\ n\ )}$ order:

**function** FIND-MAXIMUM-SUBARRAY (A, low, high)

```
if high == low
    return (low, high, A[low])
else mid = [(low+high) / 2]
    (left_low, left_high, left_sum) = FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (rigth_low, rigth_high, rigth_sum) = FIND-MAXIMUM-SUBARRAY(A, mid+1, high)
    (cross_low, cross_high, cross_sum) = FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left_sum ≤ right_sum and left_sum ≤ cross_sum
        return (left_low, left_high, left_sum)
    else if right_sum ≤ left_sum and right_sum ≤ cross_sum
        return (rigth_low, rigth_high, rigth_sum)
    else return (cross_low, cross_high, cross_sum)
```

- ***Demonstration:***
- ***First, our recurrence equations are:***

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(\frac{n}{2}) + cn & \text{if } n > 1 \end{cases} \tag{5}$$

The next figure show how we can solve recurrence. For convenience, we assume that $n$ is an exact power of 2. The total number of levels of the recursion tree in the figure is $lgn + 1$, where $n$ is the number of leaves, corresponding to the input size. An informal inductive argument justifies this claim. The base case occurs when $n = 1$, in which case the tree has only one level. Since $lg1 = 0$, we have that $lgn + 1$ gives the correct number of levels. Now assume as an inductive hypothesis that the number of levels of a recursion tree with $2^i$ leaves is $lg2^i + 1 = i + 1$ (since for any value of $i$, we have that $lg2^i = i$).
To compute the total cost represented by the recurrence equation, we simply add up the costs of all the levels. The recursion tree has $lgn + 1$ levels, each costing $cn$,for a total cost of $cn(lgn + 1) = cnlgn + cn$. Ignoring the low-order term and the constant $c$ gives the desired result of $\theta(nlgn)$.

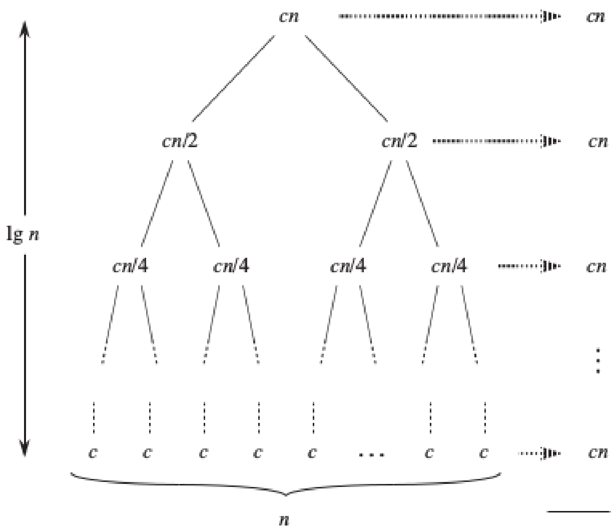



Figure : Rcurrence Solve for Maximum Subarray.

- ***Finally:***

$$Maximum\ Subarray\ \in\ \theta\ (\ n\log\ n\ ) \tag{6}$$

# 6 Conclusion:

I never realize the importance of the ***divide-and-conquer*** paradigm, I realize that maybe I applied it many times, but never take conscience what I was really doing, an example it's the ***Binary-Search***, this algorithm was maybe the first that I programmed, but up to this point, I didn't know that uses this paradigm. Now, I'm quite intrigued what other applications will have ***divide-and-conquer***. It's important to remark that in this practice the implementation of divide-and-conquer paradigm solve the maximum subarray problem in linear o $n \cdot log_2\ (n)$ time, which it's better than the iterative in an square time.

- Hernandez Martinez Carlos David.

This time we could use the algorithms properties to demonstrate the complexity of some algorithms, we increase in complexity about the programming task, for instance, the programming level was a bit interesting. The new issue was the recursion of the algorithm, I think this time the algorithm was very useful than other many programs that we were analyzing.

- Burciaga Ornelas Rodrigo Andres.

# 7 Bibliography References:


[ 1 ] Baase and Van Gelder. ”Computer Algorithms: Introduction to Design and Analysis”. Addison-Wesley.

[ 2 ] Thomas H. Cormen. ”Introduction to Algorithms”. The MIT press.