National Polytechnic Institute
Superior School of Computer Sciences

Algorithm Analysis.

Practice 4 - Divide and Conquer: QuickSort Algorithm.

Hernandez Martinez Carlos David.
Burciaga Ornelas Rodrigo Andres.

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

Group: 2cv3.

September 14, 2017

Contents

1 Introduction

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several sub-problems that are similar to the original problem but smaller in size, solve the sub-problems recursively, and then combine these solutions to create a solution to the original problem.

1.1 Analyzing Divide-and-Conquer Algorithms:

When an algorithm contains a recursive call to itself, we can often describe its running time by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. As before, we let $\boldsymbol{T(n)}$ be the running time on a problem of size $\boldsymbol{n}$. If the problem size is small enough, say $\boldsymbol{n \leq c}$ for some constant $\boldsymbol{c}$, the straightforward solution takes constant time, which we write as $\theta(1)$. Suppose that our division of the problem yields a subproblems, each of which is $\frac{1}{b}$ the size of the original. It takes the time $\boldsymbol{T(\frac{n}{b})}$ to solve one subproblem of size $\frac{n}{b}$, and so it takes time $\boldsymbol{aT(\frac{n}{b})}$ to solve $\boldsymbol{a}$ of them. If we take $\boldsymbol{D(n)}$ time to divide the problem into subproblems and $\boldsymbol{C(n)}$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence:

$$T(n) = \begin{cases} \theta(1) & \text{if } n \leq c \\ aT(\frac{n}{b}) + D(n) + C(n) & otherwise \end{cases} \tag{1}$$

2 Basic Concepts:

The ***Quicksort*** algorithm has a worst-case running time of θ (n^2) on an input array of n numbers. Despite this slow worst-case running time, ***Quicksort*** is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is θ ($n\ log$ (n)), and the constant factors hidden in the θ ($n\ log$ (n)) notation are quite small. It also has the advantage of sorting in place and it works well even in virtual-memory environments.

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.

2.2 Quicksort Algorithm:

- ***Divide:*** Partition (rearrange) the array ***A [p...r]*** into two (possibly empty) subarrays ***A [p...q-1]*** and ***A [q+1...r]*** such that each element of ***A [p...q-1]*** is less than or equal to ***A [q]*** , which is, in turn, less than or equal to each element of ***A [q+1...r]***. Compute the index q as part of this partitioning procedure.
- ***Conquer:*** Sort the two subarrays ***A [p...q-1]*** and ***A [q+1...r]*** by recursive calls to ***Quicksort***.
- ***Combine:*** Because the subarrays are already sorted, no work is needed to combine them: the entire array ***A [p ... r]*** is now sorted.

3 Development:

Based on the divide-and-conquer procedure explained in section 2.2, in the following sections we will formally demonstrate the temporal complexity of the algorithms ***Quicksort*** and ***Partition***, as well, we will explain our implemented code.

I divided the program in 6 python modules, to have a better control of the code.

- ***main.py:*** Control the sequence of execution.
- ***menu.py:*** Creates the list **A** to sort of length 'n', entered by the user in console.
- ***quicksort.py:*** Principal algorithm, split the list **A** several times. This algorithm implements the divide-and-conquer paradigm.
- ***partition.py:*** Secondary algorithm, sort the list **A**.
- ***graph.py:*** Plot the computational time of 'quicksort.py' and 'partition.py'
- ***gb.py:*** Store the global variables to use.

3.1 Main.py:

This module controls the structural sequence of execution. When the call to ***menu ( )*** ends, the program will have the list to sort ***A*** created (***A*** it's stored in 'gb.py' as a global variable), then the program will send it to ***quicksort (...)*** to begin the sorting process. But the program will not send the complete 'list' at the first time, as we can see in the code showed bellow, in line 3 there is a *for* loop, and will iterate from ***0 to 'n'*** where ***'n'*** it's the size of ***A***, in each *iteration* the loop will "split" ***A*** in a sublist ***B*** of size ***i***, where ***i*** will be the index of the loop. In other words ***B = A [0, ..., i]*** and ***B*** will be the 'list' that ***quicksort (...)*** will receive as parameters. This process has a reason to be, as we can see in line 6, the list ***parameters*** stores a *tuple*, where the first element it's the ***length*** of the list ***B*** and the second element it's the ***computation time*** that the algorithm takes to sort that 'list' ***([Length (n), Time (t)])***. This *parameters* will help later to plot ***length*** against ***time***.

```
1  def main ( ):
2      menu ( )
3      for i in range ( len ( gb.n ) + 1 ):
4          m, gb.time = gb.n [ :i ], 0
5          quicksort ( m, 0, len ( m ) - 1 )
6          gb.parameters.append ( ( len ( m ), gb.time ) )
7      print ( "\n\tSorted list: ", m, "\n" )
8      print ( "\n\tQuicksort Parameters: ", gb.parameters )
9      print ( "\n\tPartition Parameters: ", gb._parameters )
10     graph ( )
11 main ( )
```

Observation: *In our program, the list* ***A*** *and* ***B*** *has the name* ***n*** *and* ***m*** *respectively.*

3.2 Menu.py

This module display on screen two options:

- Create a list ***A*** of random elements.
- Create a list ***A*** with all the element sorted in decreasing order.

Observation: *The second option it's the worst case for the algorithm* ***quicksort****.*

Observation: *Before create* ***A****, the program will display another option asking for its length.*

```
1  def menu ( ):
2      ans, length = -1, -1
3      # Menu option 1: User must select if create a list of random numbers or
4      # a list in the worst case for the algorithm, sorted in decreasing order.
5      while ( ans != 1 and ans != 2 ):
6          print ( "\n\n\t\t\tQUICK-SORT\n\n\t Select one of the following options." )
7          print ( "\n\t1.- Random case." )
8          print ( "\t2.- Worst case." )
9          ans = int ( input ( "\tAnswer: " ) )
10     # Menu option 2: User must choose the length of the list.
11     while ( length < 0 ):
12         print ( "\n\tLength of the list: " )
13         length = int ( input ( "\tAnswer: " ) )
14     # Create a list of random numbers. Otherwise a list in decreasing
15     # order from 'length' to 0, on intervals of 1.
16     if ( ans == 1 ):
17         gb.n = list ( np.random.randint ( 0, 100, size = length ) )
18     else:
19         gb.n = list ( range ( length, 0, -1 ) )
20         gb.flag = True
21     print ( "\n\tList to sort: ", gb.n )
```

3.3 Gb.py

This module only stores the global variables that we are going to use:

(i) ***_parameters:*** List that store the parameters of the points to plot for Partition. Each element it's a tuple that stores the length of the sorted list, and the time that the algorithm took to execute its process the first time it was called.

(ii) ***parameters:*** List that store the parameters of the points to plot for Quicksort. Each element it's a tuple that stores the length of the sorted list, and the time that the algorithm takes to sort it.

(iii) ***_time:*** Counter that stores the computational time that the algorithm Partition takes to finish his execution process.

(iv) ***time:*** Counter that stores the computational time that the algorithm Quicksort takes to sort the list 'n'.

(v) ***flag:*** Flag used in graph.py, help to create the labels of the proposed function, depending if the user select worst or random case.

(vi) ***n:*** List to sort.

3.4 Graph.py

When this point is reached, on screen the user will visualize two graphs, the left one will be the temporal complexity for ***Quicksort*** and the right one for ***Partition***. Both are comparing ***time*** of execution vs ***length***.

In line 22 starts the code for ***Quicksort*** graph. As we can see in line 27, we use the 'list' ***parameters***, as we said, this is a 'list' of tuples, and in every tuple we will have 2 elements a *length* and a *time*, well, this tuples will be the points to plot, so, the only thing rest to do is divide *parameters* into two lists, one for the computational time, and another one for all sizes, this process it's performed in the lines 27 and 29 respectably.

```
1  def graph ( ):
2      labels ( )
3      # Window title.
4      plt.figure ( "Quicksort Algorithm", figsize = ( 14, 7 ) )
5
6      # Right graph: Temporal complexity of Partition.
7      plt.subplot ( 1, 2, 2 )
8      # Figure title.
9      plt.title ( "Partition ( " + str ( gb._parameters [ -1 ] [ 0 ] + 1 ) + ", " + str ( gb._param
10     # Parameter Time ( t ).
11     _t = list ( map ( lambda x: x [ 1 ], gb._parameters ) )
12     # Parameter Size ( n ).
13     _s = list ( map ( lambda x: x [ 0 ] + 1, gb._parameters ) )
14     # Axes names.
15     plt.xlabel ( "Size ( n )", color = ( 0.3, 0.4, 0.6 ), family = "cursive", size = "large" )
16     plt.ylabel ( "Partition Time ( t )", color = ( 0.3, 0.4, 0.6 ), family = "cursive", size = "la
17     # Plot.
18     plt.plot ( _s, _t, "#778899", linewidth = 3, label = function1 )
19     plt.plot ( _s, g1, "#800000", linestyle = "--", label = proposed1 )
20     plt.legend ( loc = "upper left" )
21
22     # Left graph: Temporal complexity of Quicksort.
23     plt.subplot ( 1, 2, 1 )
24     # Figure title.
25     plt.title ( "Quicksort ( " + str ( gb.parameters [ -1 ] [ 0 ] ) + ", " + str ( gb.parameters
26     # Parameter Time ( t ).
27     t = list ( map ( lambda x: x [ 1 ], gb.parameters ) )
28     # Parameter Size ( n ).
29     s = list ( map ( lambda x: x [ 0 ], gb.parameters ) )
30     # Axes names.
31     plt.xlabel ( "Size ( n )", color = ( 0.3, 0.4, 0.6 ), family = "cursive", size = "large" )
32     plt.ylabel ( "Quicksort Time ( t )", color = ( 0.3, 0.4, 0.6 ), family = "cursive", size = "la
33     # Plot.
34     plt.plot ( s, t, "#778899", linewidth = 3, label = function2 )
35     plt.plot ( s, g2, "#800000", linestyle = "--", label = proposed2 )
36     plt.legend ( loc = "upper left" )
37     plt.show ( )
```

Similar for the graph of ***Partition***, in line 6 starts the code that will describe the plot. As we can see in line 11, we use the 'list' ***_parameters*** (this one stores the computational time only for Partition). For plotting, we will repeat the process of separating ***_parameters*** into two lists. This process it's performed in the lines 11 and 13.

3.5 Partition.py

This module implements the algorithm ***partition***, as we mention in section 2.2. Quicksort rearrange the array ***A [p...r]*** into two (possibly empty) subarrays ***A [p...q-1]*** and ***A [q+1...r]***. Partition receive one of this subarrays, such that each element of ***A [p...q-1]*** is less than or equal to ***A [q]***, which is, in turn, less than or equal to each element of ***A [q+1...r]***. In this case the algorithm will exchange elements, thus, everything at the left of ***A [q]*** will be less to this element, and all at the right of ***A [q]*** will be greater to this element, also compute the index ***q*** as part of this partitioning procedure.

```
1  def partition ( n, p, r ):
2      x = n [ r ] #   Pivot
3      i = p # Border
4      for j in range ( p, r ):
5          if ( n [ j ] < x ):
6              aux = n [ j ]
7              n [ j ] = n [ i ]
8              n [ i ] = aux
9              i += 1
10     aux = n [ i ]
11     n [ i ] = n [ r ]
12     n [ r ] = aux
13     return i
```

For calculate the temporal complexity of this algorithm, it's necessary to add the counter ***_time*** in each line of our code, as we can see, in line 26, we will sum ***_time*** to the counter ***time***, because, as we mention in section 3.3, ***time*** stores the temporal complexity for ***Quicksort*** and ***Partition*** it's part of this algorithm. In line 27 the program make a comparison, if the last element added it's fewer than ***r*** then append another element to ***_parameters*** i.e The list only stores the computational time of the first time that ***Quicksort*** calls ***Partition***. In line 29, ***_time*** it's restarted.

```
1  def partition ( n, p, r ):
2      x = n [ r ] # pivot
3      gb._time += 1
4      i = p # border
5      gb._time += 1
6      for j in range ( p, r ):
7          gb._time += 1
8          if ( n [ j ] < x ):
9              aux = n [ j ]
10             gb._time += 1
11             n [ j ] = n [ i ]
12             gb._time += 1
13             n [ i ] = aux
14             gb._time += 1
15             i += 1
16             gb._time += 1
17     gb._time += 1
18     aux = n [ i ]
19     gb._time += 1
20     n [ i ] = n [ r ]
21     gb._time += 1
22     n [ r ] = aux
23     gb._time += 1
24     # Sum the temporal complexity of Partition '_time' to the temporal
25     # complexity of Quicksort 'time'.
26     gb.time += gb._time
27     if ( r > gb._parameters [ len ( gb._parameters ) - 1 ][ 0 ] ):
28         gb._parameters.append ( ( r, gb._time ) )
29     gb._time = 0
30     return i
```

3.6 Quicksort.py

Principal algorithm, based on the index returned by ***Partition***, rearrange the array ***A [p...r]*** into two subarrays ***A [p...q-1]*** and ***A [q+1...r]*** as we can see in lines 4 y 5. If the ***if*** sentence it's true, the algorithm will repeat the split process.

```
1 def quicksort ( n, p, r ):
2     if ( p < r ):
3         q = partition ( n, p, r )
4         quicksort ( n, p, q - 1 )
5         quicksort ( n, q + 1, r )
```

For calculate the temporal complexity of this algorithm, similar to section 3.5, it's necessary to put the counter ***time*** in each line of the code.

```
1 def quicksort ( n, p, r ):
2     gb.time += 1
3     if ( p < r ):
4         q = partition ( n, p, r )
5         gb.time += 1
6         quicksort ( n, p, q - 1 )
7         gb.time += 1
8         quicksort ( n, q + 1, r )
9         gb.time += 1
```

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.

4.1 Quicksort Results:

4.1.1 Worst case:

First test for ***Quicksort***. The program will plot the ***time*** that the algorithm takes to sort a list of length $n = 6$.

Observation: *We are going to analyze the worst case when the list it's sorted in decreasing order.*

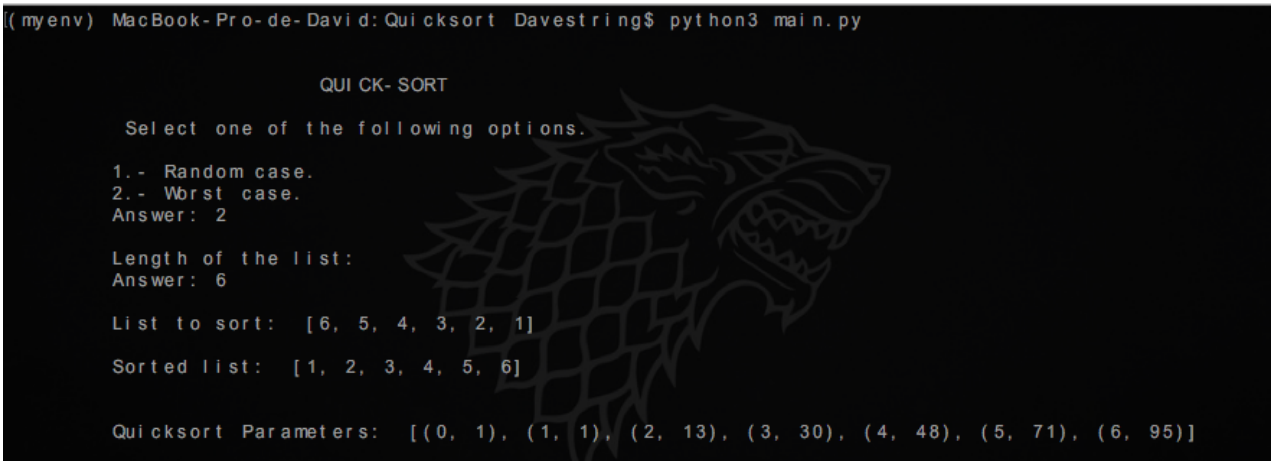

Figure 4.1.1.0: Testing Quicksort with a list of length n = 6.

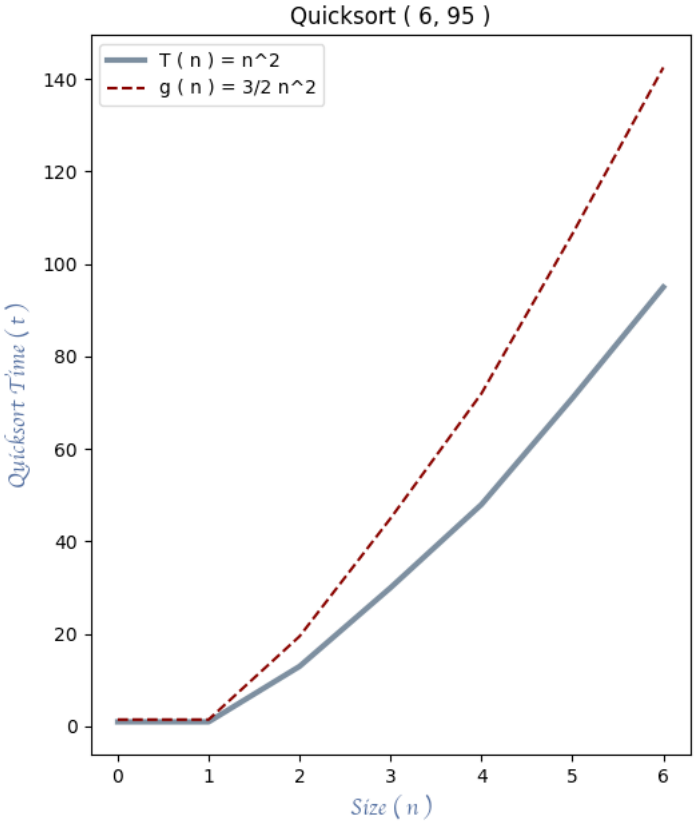

Figure 4.1.1.1: Graph for Figure 4.1.1.0.

Length (n)	Time (t)
0	1
1	2
2	13
3	30
4	48
5	71
6	95

Table 1: Plot point for Figure 4.1.1.1.

Second test for ***Quicksort***. The program will plot the ***time*** that the algorithm takes to sort a list of length ***n = 20***.

Observation: *We are going to analyze another worst case when the list it's sorted in decreasing order.*

```
Select one of the following options.

1.- Random case.
2.- Worst case.
Answer: 2

Length of the list:
Answer: 20

List to sort:  [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Sorted list:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Figure 4.1.1.2: Testing Quicksort with a list of length n = 20.

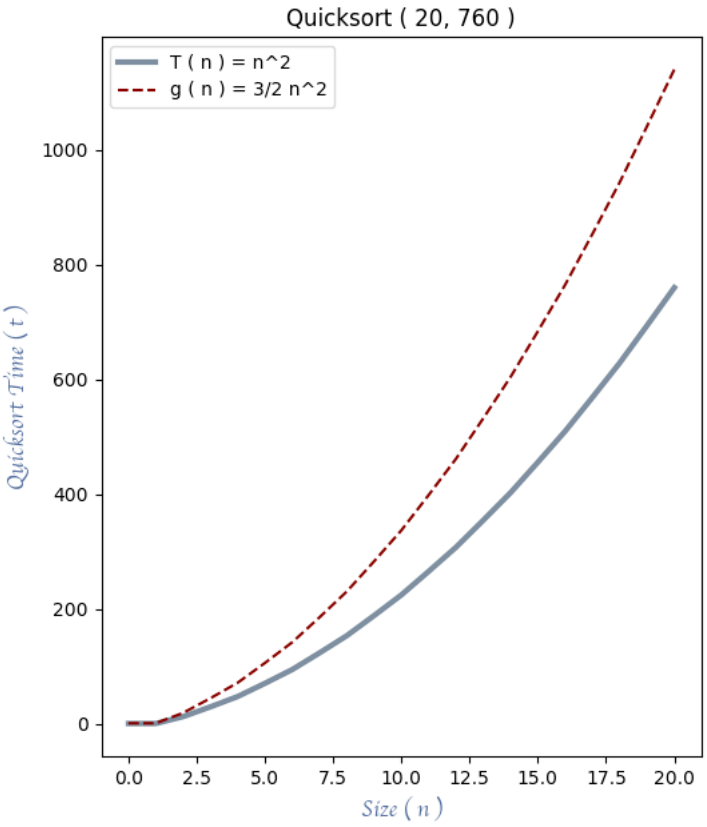

Figure 4.1.1.3: Graph for Figure 4.1.1.2.

Length (n)	Time (t)
0	1
1	2
2	13
3	30
4	48
5	71
6	95
7	124
8	154
9	189
10	225
11	266
12	308
13	355
14	403
15	456
16	510
17	569
18	629
19	694
20	760

Table 2: Plot point for Figure 4.1.1.3.

Observation: *The* ***red*** *graph represents our proposed function for the worst case of* ***Quicksort*** *and the* ***blue*** *one it's the temporal complexity.*

Observation: *The function proposed is:* ***g(n) = (3/2)*** n^2.

4.1.2 Random case:

Third test for ***Quicksort***. The program will plot the ***time*** that the algorithm takes to sort a list of length ***n = 10***.

Figure 4.1.2.0: Testing Quicksort with a list of length n = 10.

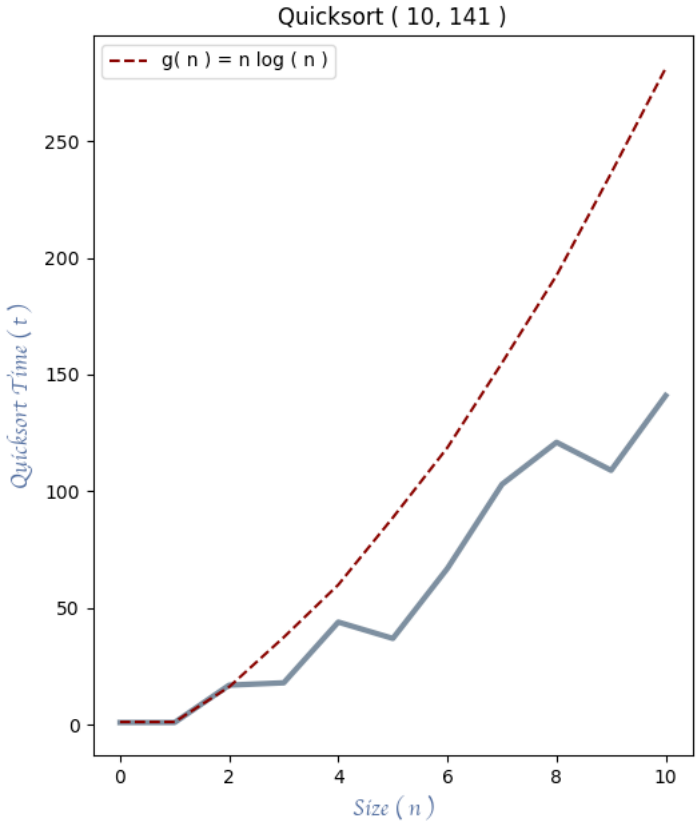

Figure 4.1.2.1: Graph for Figure 4.1.2.0.

Length (n)	Time (t)
0	1
1	1
2	17
3	18
4	44
5	37
6	67
7	103
8	121
9	109
10	141

Table 3: Plot point for Figure 4.1.2.1.

Fourth test for ***Quicksort***. The program will plot the ***time*** that the algorithm takes to sort a list of length ***n = 15***.

```
(myenv) MacBook-Pro-de-David:Quicksort Davestring$ python3 main.py

                    QUICK-SORT

 Select one of the following options.

1.- Random case.
2.- Worst case.
Answer: 1

Length of the list:
Answer: 15

List to sort:  [41, 19, 31, 32, 59, 31, 19, 58, 41, 82, 11, 19, 28, 26, 75]

Sorted list:  [11, 19, 19, 19, 26, 28, 31, 31, 32, 41, 41, 58, 59, 75, 82]
```

Figure 4.1.2.2: Testing Quicksort with a list of length n = 15.

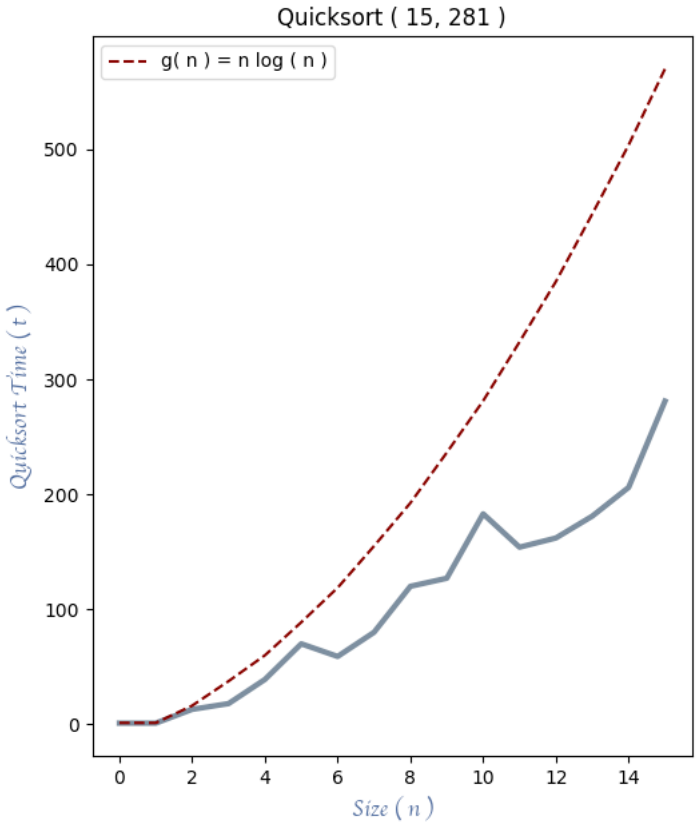

Figure 4.1.2.3: Graph for Figure 4.1.2.2.

Length (n)	Time (t)
0	1
1	1
2	13
3	18
4	39
5	70
6	59
7	80
8	120
9	127
10	183
11	154
12	162
13	181
14	206
15	281

Table 4: Plot point for Figure 4.1.2.3.

Observation: *The* ***red*** *graph represents our proposed function for any other case of* ***Quicksort*** *and the* ***blue*** *one it's the temporal complexity.*

Observation: *The function proposed is:* ***g(n) = n log (n).***

4.2 Partition Results:

4.2.1 Worst case:

First test for ***Partition***. The program will plot the ***time*** that the algorithm takes to finish his process with a list sorted in decreasing order of length $n = 6$.

Observation: *We are going to analyze the worst case when the list it's sorted in decreasing order.*

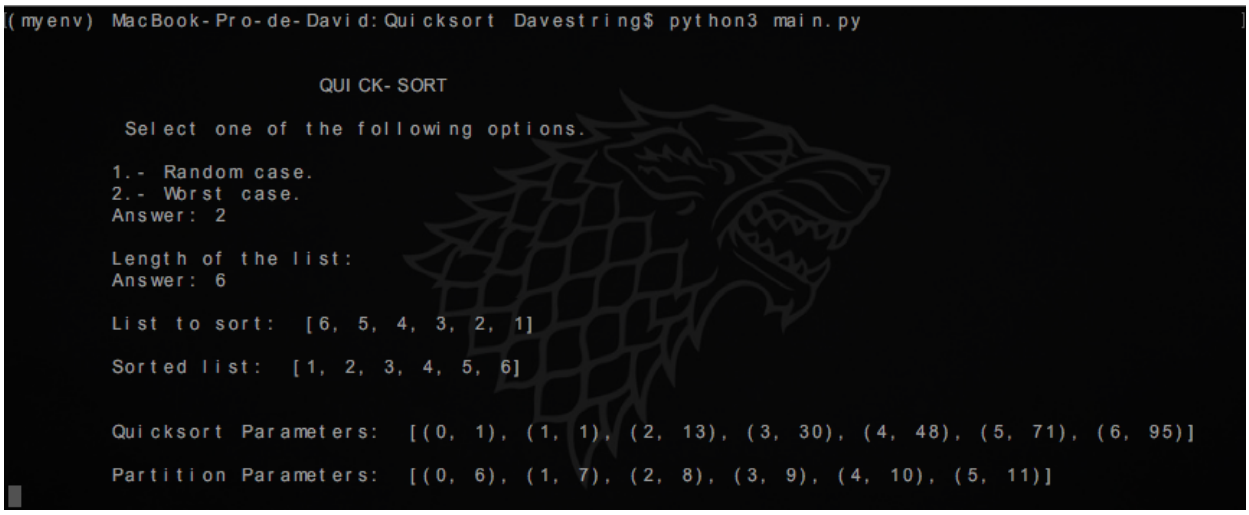

Figure 4.2.1.0: Testing Partition with a list of length n = 6.

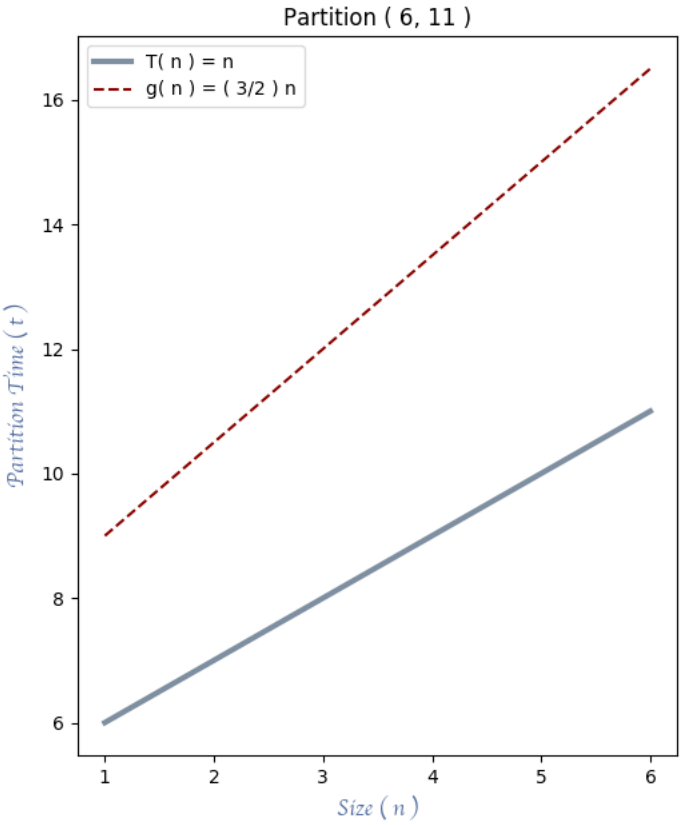

Figure 4.2.1.1: Graph for Figure 4.2.1.0.

Length (n)	Time (t)
0	6
1	7
2	8
3	9
4	10
5	11
6	12

Table 5: Plot point for Figure 4.2.1.1.

Second test for ***Partition***. The program will plot the ***time*** that the algorithm takes to finish his process with a list sorted in decreasing order of length $n = 20$.

```
                    QUI CK- SORT

 Select one of the following options.

1.- Random case.
2.- Worst case.
Answer: 2

Length of the list:
Answer: 20

List to sort:  [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Sorted list:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Figure 4.2.1.2: Testing Partition with a list of length n = 20.

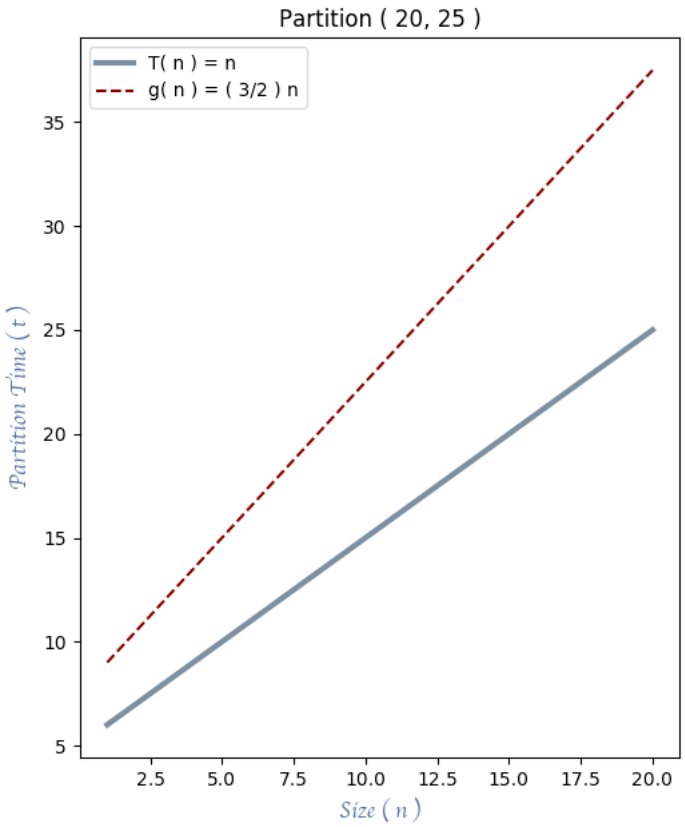

Figure 4.2.1.3: Graph for Figure 4.2.1.2.

Length (n)	Time (t)
0	6
1	7
2	8
3	9
4	10
5	11
6	12
7	13
8	14
9	15
10	16
11	17
12	18
13	19
14	20
15	21
16	22
17	23
18	24
19	25
20	26

Table 6: Plot point for Figure 4.2.1.3.

Observation: *The* ***red*** *graph represents our proposed function for the worst case of* ***Partition*** *and the* ***blue*** *one it's the temporal complexity.*

Observation: *The function proposed is:* $g(n) = (3/2)n$.

4.2.2 Random case:

Third test for ***Partition***. The program will plot the ***time*** that the algorithm takes to finish his process with a list of random elements of length ***n = 10***.

Figure 4.2.2.0: Testing Partition with a list of length n = 10.

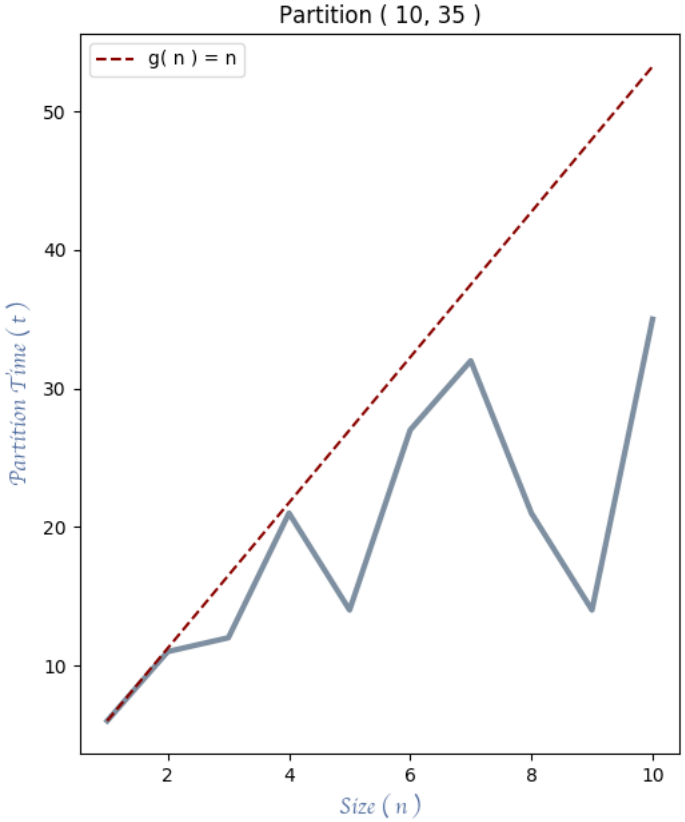

Figure 4.2.2.1: Graph for Figure 4.2.2.0.

Length (n)	Time (t)
0	6
1	11
2	12
3	21
4	14
5	27
6	32
7	21
8	14
9	35

Table 7: Plot point for Figure 4.2.2.1.

Fourth test for ***Partition***. The program will plot the ***time*** that the algorithm takes to finish his process with a list of random elements of length $n = 15$.

```
(myenv) MacBook-Pro-de-David:Quicksort Davestring$ python3 main.py

                    QUICK-SORT

 Select one of the following options.

1.- Random case.
2.- Worst case.
Answer: 1

Length of the list:
Answer: 15

List to sort:  [41, 19, 31, 32, 59, 31, 19, 58, 41, 82, 11, 19, 28, 26, 75]

Sorted list:  [11, 19, 19, 19, 26, 28, 31, 31, 32, 41, 41, 58, 59, 75, 82]
```

Figure 4.2.2.2: Testing Partition with a list of length n = 15.

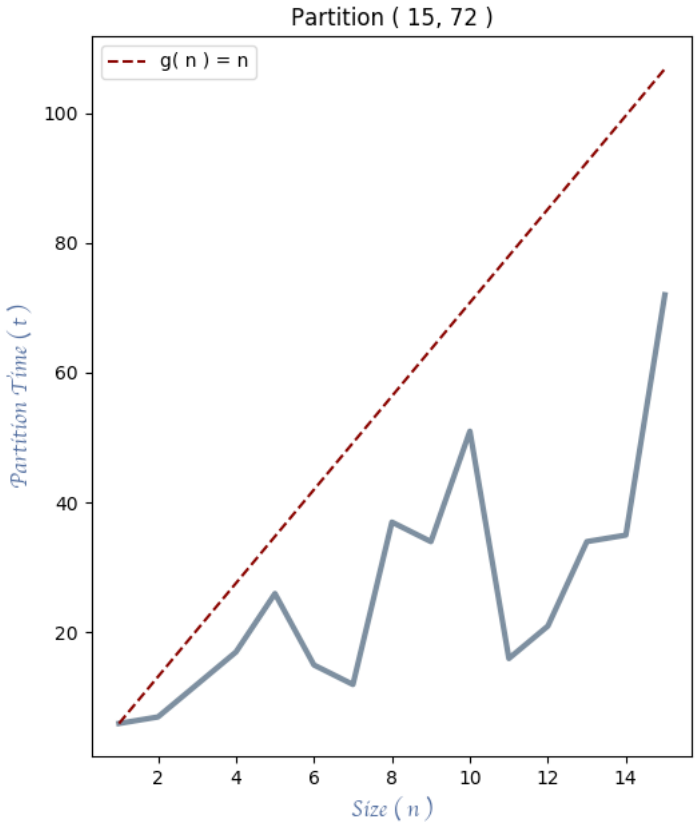

Figure 4.2.2.3: Graph for Figure 4.2.2.2.

Length (n)	Time (t)
0	6
1	7
2	12
3	17
4	26
5	15
6	12
7	37
8	34
9	51
10	16
11	21
12	34
13	35
14	72

Table 8: Plot point for Figure 4.2.2.3.

Observation: *The* ***red*** *graph represents our proposed function for any other case of* ***Partition*** *and the* ***blue*** *one it's the temporal complexity.*

Observation: *The function proposed is:* $g(n) = (3/2)\,n$.

5 Annexes

In the following lines we will formally demonstrate ***Partition*** and ***Quicksort***.

5.1 Quicksort Demonstration:

Demonstrate that Quicksort algorithm has ***T (n) = (n) (log (n))*** order:

function Quicksort (A, p, r):

```
1 if ( p < r )
2     q = partition ( a, p, r )
3     Quicksort ( A, p, q )
4     Quicksort ( A, q + 1, r )
```

- ***Demonstration:***

Although the pseudocode for Quicksort works correctly, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2, the ***pivot*** always will split the array in the middle and ***Partition*** has order θ (n) (see section 5.2 for the demonstration), we can use the following equation:

- ***Quicksort recurrence equation:***

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

- ***Let*** $k = log_2(n)$***, then*** $n = 2^k$***, substituting:***

$$\begin{aligned} T(\ n\ ) &= 2T(\ \frac{n}{2}\ ) + cn && (3)\\ &= 2T(2^{k-1}) + c(\ 2^k\ ) && (4)\\ &= 2^2T(\ 2^{k-2}\ ) + 2c(\ 2^k\ ) && (5)\\ &= 2^3T(\ 2^{k-3}\ ) + 3c(\ 2^k\ ) && (6)\\ & && (7) \end{aligned}$$

- ***Then:***

$$T(\ n\ ) = 2^iT(\ 2^{k-i}\ ) + ic(\ 2^k\ ) \tag{8}$$
$$(9)$$

- ***Let k - i = 0, then k = i:***

$$
\begin{aligned}
T(\ n\ ) &= 2^k T(\ 1\ ) + (\ kc\ )\ (\ 2^k\ ) && (10)\\
&= (\ c\ )\ (\ 2^k\ )\ +\ (\ kc\ )\ (\ 2^k\ ) && (11)\\
&= (\ c\ )\ (\ 2^k\ )\ [\ 1\ +\ k\ ] && (12)\\
&= (\ cn\ )\ [\ log_2(\ n\ )\ +\ 1\ ] && (13)\\
&= cn\ log_2(\ n\ )\ +\ cn && (14)\\
&= c\ [\ n\ log_2(\ n\ )\ +\ n\ ] && (15)
\end{aligned}
$$

- ***Finally:***

$$Quicksort \in \theta\ (\ n\ log_2(\ n\ )\ )\ ). \tag{16}$$

5.2 Partition Demonstration:

Formally demonstrate that ***Partition*** algorithm has linear complexity.

Partition (A, p, r):

```
1    x = A[r]
2    i = P - 1
3    for  j = p to j <= r - 1 do
4        if ( A[j] <= x )
5            i ++
6            exchange ( A[i], A[j] )
7    exchange ( A[i + 1] , A[r] )
8    return i + 1;
```

- ***Demonstration:***
- ***Analyzing the complexity of each line:***
  1. Line 1, 2 = θ (1).
  2. Line 3 = θ (n).
     (i) Line 4, 5, 6 = θ (1).
  3. Line 7, 8 = θ (1).
- ***Then, from all lines:***

$$T\ (\ n\ )\ =\ \theta\ (\ 1\ +\ (\ n\ *\ 1\ )\ +\ 1\ )\ =\ \theta\ (\ n\ ) \tag{17}$$

- ***Finally:***

$$Partition \in\ \theta\ (\ n\ ) \tag{18}$$

5.3 Questionnaire:

1. ***What value of*** q ***returns Partition when all the elements in array*** $A[p, ..., r]$ ***have the same value ?***

   When we take a look to the algorithm we can see that if the element has the same value or the value is lower than the pivot, the algorithm exchange the positions in the array, first the element in position $A[j]$ is changed by the element which is in the position $A[q]$ then, q is moving up at the same time of j which is the index of the elements bigger than pivot, so when it is finished the first time the q has the same value as r (value of position or pivot), in that first case r is equal to n, where n is the size of the array is ***n - 1***, and the quicksort is called once again, for this time the q equal ***n - 2***, the next time q equal ***n - 3*** and so on, this is one of the worst cases in this algorithm, because is always doing the merge n times in place of $T(\ \frac{n}{2}\ )$. This procedure it's represented in Figure 5.1.0.

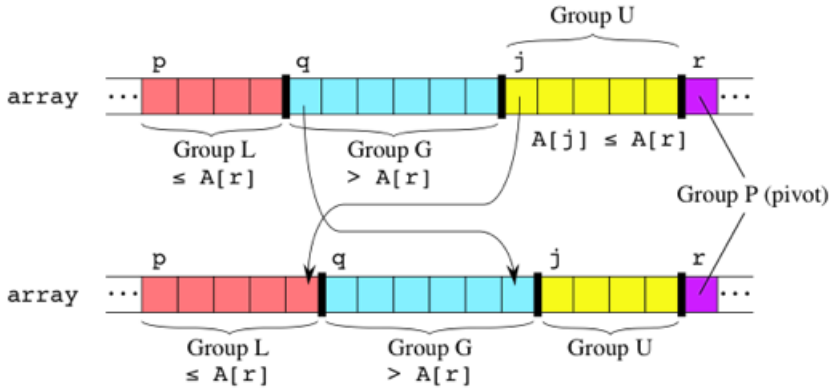

Figure 5.1.0: Array partition.

In the image above is illustrated the previous procedure, in this case the group G is always 0 because there are not number greater than the pivot, when the index j reaches the value index of pivot, the q takes the same value, so if the n equal 10 where n is the number of elements, the first time q is 9 because the position of the element $A[10]$ is 9, then 8, then 7 and so on.

Observation: *This is the worst case in the algorithm, the analytic demonstration is covered in section 5.2.*

2. ***What is the QuickSort runtime when all array elements have the same value?***

When Quicksort always has the most unbalanced partitions possible (or all positions have the same value), then the original call takes a time ***cn*** for some constant, the recursive call for ***n - 1*** elements takes time of ***c(n - 2)***, the recursive call for ***n - 2*** takes a constant time of ***c(n - 3)*** and so on.

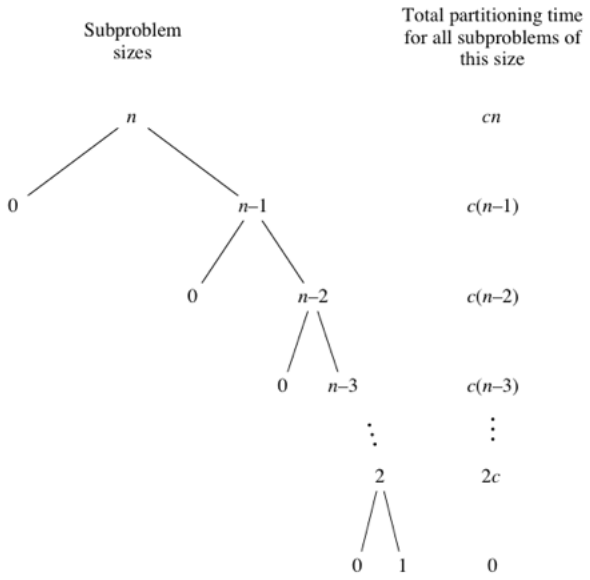

Figure:

When we add the times to do partitions for each level, we get as we have seen in other practices:

$cn + c(n-1) + c(n-2) + ... + 2c = \sum_{j=1}^{i=-2} = \frac{(n-2)(n-1)}{2}$

We can see that we have the arithmetic series, in notation big θ, the complexity order is $\theta \in (n^2)$

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 ***Bubble-Sort***, 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***.

- 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 *log* complexity about the algorithm, I think this time the algorithm was very useful than other many programs that we were analyzing, because a lot of programs needs to sort their data or collections.

- 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.