

NATIONAL POLYTECHNIC INSTITUTE
SUPERIOR SCHOOL OF COMPUTER SCIENCES

ALGORITHM ANALYSIS.

Practice 3 - Divide and Conquer: MergeSort Algorithm.

Hernandez Martinez Carlos David.
Burciaga Ornelas Rodrigo Andres.

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

Group: 2cv3.

September 8, 2017

Contents

1	Introduction:	2
1.1	Analyzing Divide-and-Conquer Algorithms:	2
2	Basic Concepts:	3
2.1	Divide-and-Conquer Paradigm:	3
2.2	Merge-Sort Algorithm:	3
3	Algorithms:	4
3.1	Merge Procedure:	4
3.2	Merge-Sort Procedure:	5
4	Development:	6
4.1	Merge-Sort:	6
4.1.1	Main.py	6
4.1.2	Graph.py	7
4.1.3	Merge.py	7
4.1.4	Mergesort.py	8
4.2	Merge:	9
4.2.1	Main.py:	9
4.2.2	Graph.py	10
4.2.3	Mergesort.py	11
4.2.4	Merge.py	12
5	Results:	13
5.1	Merge-Sort Results:	13
5.2	Merge Results:	14
6	Annexes	15
6.1	Merge Demonstration:	15
6.2	Merge-Sort Demonstration:	16
6.3	Function A:	18
6.4	Function B:	20
7	Conclusion:	22
8	Bibliography References:	23

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.

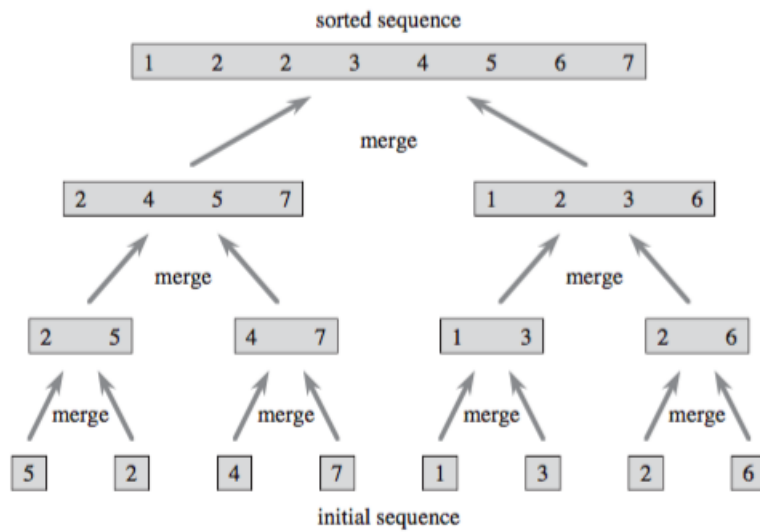


Figure 1.1.0: The operation of merge sort on the array $A = 5, 2, 4, 7, 1, 3, 2, 6$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

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 $T(n)$ be the running time on a problem of size n . If the problem size is small enough, say $n \leq c$ for some constant 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 $\frac{n}{b}$ the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which $a \neq b$). It takes the time $T(\frac{n}{b})$ to solve one subproblem of size $\frac{n}{b}$, and so it takes time $a T(\frac{n}{b})$ to solve a of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence:

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

2 Basic Concepts:

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 Merge-Sort Algorithm:

The *merge sort algorithm* closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- (i) **Divide:** Divide the n -element sequence to be sorted into two sub-sequences of $n/2$ elements each.
- (ii) **Conquer:** Sort the two sub-sequences recursively using merge sort.
- (iii) **Combine:** Merge the two sorted sub-sequences to produce the sorted answer.

The recursion bottoms out when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order. The key operation of the merge sort algorithm is the merging of two sorted sequences in the combine step.

3 Algorithms:

The following section will explain the procedure of *Merge* and *Merge-Sort* algorithms.

3.1 Merge Procedure:

Our *MERGE* procedure takes time $\theta(n)$, where $n = r - p + 1$ is the total number of elements being merged, and it works as follows. Returning to our card playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are comparing just the two top cards. Since we perform at most n basic steps, merging takes $\theta(n)$ time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a *sentinel* card, which contains a special value that we use to simplify our code. Here, we use `inf` as the sentinel value, so that whenever a card with `inf` is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

function MERGE (A, p, q, r):

```
1   n1 = q - p + 1
2   n2 = r - q
3   let L [ 1, ..., n1 + 1 ] and R [ 1, ..., n2 + 1 ] be new arrays
4   for i = 1 to n1 do
5       L [ i ] = A [ p + i - 1 ]
6   for j = 1 to n2 do
7       R [ j ] = A [ q + j ]
8   L [ n1 + 1 ] = inf
9   R [ n2 + 1 ] = inf
10  i = 1
11  j = 1
12  for k = p to r do
13      if L [ i ] <= R [ j ]
14          A [ k ] = L [ i ]
15          i++
16      else A [ k ] = R [ j ]
17          j++
```

In detail, the MERGE procedure works as follows. Line 1 computes the length n_1 of the subarray $A [p \dots q]$ and line 2 computes the length n_2 of the subarray $A [q + 1 \dots r]$. We create arrays L and R (left and right) of length $n_1 + 1$ and $n_2 + 1$, respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4 - 5 copies the subarray $A [p \dots q]$ into $L [1 \dots n_1]$, and the **for** loop of lines 6 - 7 copies the subarray $A [q + 1 \dots r]$ into $R [1 \dots n_2]$. Lines 8 - 9 put the sentinels at the ends of the arrays L and R . Lines 10 - 17, illustrated in Figure 3.1.0, performs $r - p + 1$ basic steps by maintaining the following loop invariant:

At the start of each iteration of the **for** loop of lines 12 - 17, the sub-array $A [p \dots k - 1]$, contains the $k - p$ smallest element of $L [1 \dots n_1 + 1]$ and $R [1 \dots n_2 + 1]$ in sorted order. Moreover, $L [i]$ and $R [j]$ are the smallest elements of their arrays that have not been copied back into A .

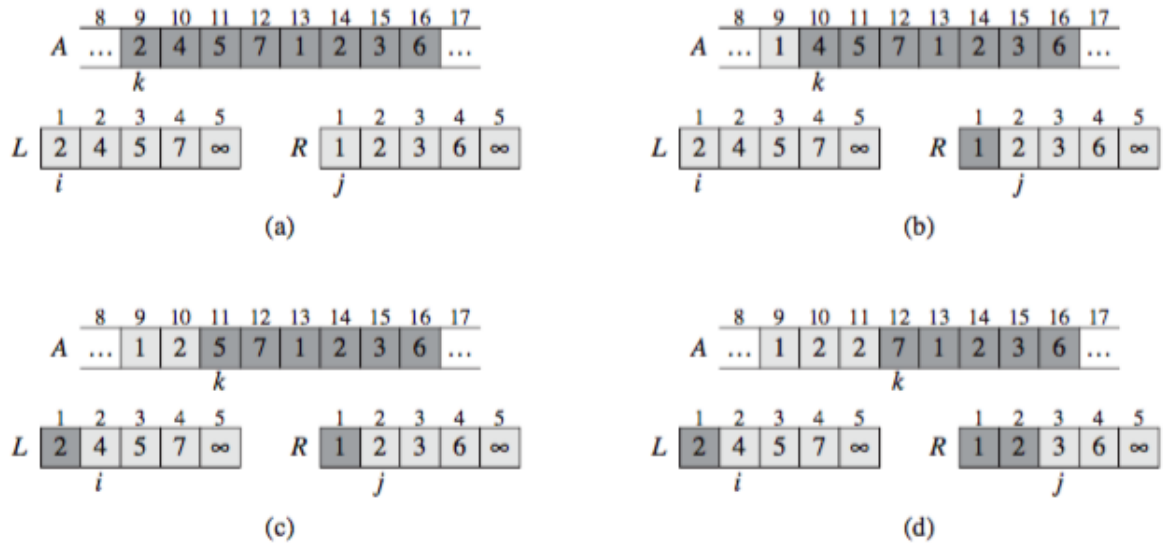


Figure 3.1.0: Merge algorithm procedure.

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12 - 17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

3.2 Merge-Sort Procedure:

The procedure MERSE-SORT (A, p, r), sorts the elements in the subarray A [p ... r] if $p \geq r$ the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index **q** that partitions A [p ... r] into two subarrays: A [p ... q], contains $\lfloor \frac{n}{2} \rfloor$ elements, and A [q + 1 ... r], containing $\lfloor \frac{n}{2} \rfloor$ elements.

function MERSE-SORT (A, p, r):

```

1  if ( p < r )
2      q = ( p + r ) / 2
3      MERSE-SORT ( A, p, q )
4      MERSE-SORT ( A, q + 1, r )
5      MERSE ( A, p, q, r )

```

To sort the entire sequence $A = (A[1], A[2], \dots, A[n])$, we make the initial call MERGE-SORT (A, 1, A.length), where $A.length = n$. Figure 1.1.0 illustrate the operation of the procedure bottom-up when n is a power of 2. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $\frac{n}{2}$ are merged to form the final sorted sequence of length n .

4 Development:

Based on the procedure explained above for both algorithms (*Merge-Sort* and *Merge*). In the following sections we will formally demonstrate the temporal complexity of each algorithm and the implemented code.

4.1 Merge-Sort:

The program it's divided in four python modules to have a better control of the code:

- *main.py*: Control the execution sequence, as well generate the list to sort and calls the *Merge-Sort* algorithm.
- *mergesort.py*: Contains the *Merge-Sort* algorithm.
- *merge.py*: Contains the *Merge* algorithm.
- *graph.py*: Plot the *size* against the *time* that the algorithms takes to sort the list generated in *main.py*.

4.1.1 Main.py

In *main.py* the program will ask the user to enter a number '*n*', this parameter will be the size of a 'list' of random numbers between $[-100, 100]$. The method that does this process it's *menu* ():

```
1 def menu ( ):  
2     ans = 0  
3     print ( "\n\n\tMerge-Sort Algorithm:\n" )  
4     ans = int ( input ( "\tAdd the size of the list: " ) )  
5     return list ( np.random.randint ( -100, 100, size = ans ) )
```

When the call to *menu* () ends, the method will return the 'list' of size '*n*'-Let's call this list *A*-, then the program will send it to *mergesort* (...) 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 4 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 *mergesort* will receive as parameters. This process has a reason to be, as we can see in the line 5, *mergesort* returns 2 values, *m* and *param*, the first one it's the 'list' sorted, and the second one is a *tuple*, where the first element it's the *size* of the list *B* and the second element it's the *computation time* that the algorithm takes to sort that 'list' ([*Size* (*n*), *Time* (*t*)]). In line 6, the program will append the *tuple* to a new 'list' named *parameters*.

```
1 def main ( ):  
2     parameters, n = [ ], menu ( )  
3     print ( "\n\tList to sort: ", n )  
4     for i in range ( len ( n ) + 1 ):  
5         m, param = mergesort ( n [ :i ] )  
6         parameters.append ( param )  
7     print ( "\n\tSorted list: ", m, "\n" )  
8     print ( "\tMergeSort Parameters: ", parameters, "\n" )  
9     graph ( parameters )
```

So then, the loop finish when $B = A$. As result, we will have the list *A* sorted, and all the *parameters* from *B_i* to *A*. This parameters will help to plot the *computation time* against the *size* of *A*.

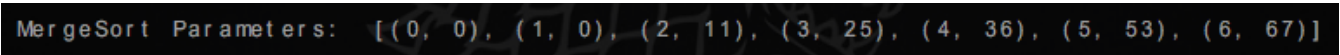


Figure 4.1.1.0: *Parameters* list, where the first element it's the *size* of *B_i* and the second element it's the *computation time* of *mergesort*. The evaluated list it's of size $n = 6$ as we can see in the last *tuple*.

4.1.2 Graph.py

Graph (...) plot *Size* (*n*) where '*n*' it's the length of *A*, against *Time* (*t*) where *t* it's the computational time that *mergesort* takes to sort this list. As we can see, the parameters received it's the 'list' *parameters*, as we said, this is a 'list' of tuples, and in every tuple we will have 2 elements a *size* 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 all the sizes, and another one for all the computational times, this process performed in the lines 8 and 10.

```
1 def graph ( parameters ):  
2     # Window title.  
3     plt.figure ( "Merge-Sort Algorithm" )  
4     # Graph title.  
5     plt.title ( "Merge-Sort ( size , time ) = ( " + str ( parameters [ len ( parameters ) - 1 ]  
6         [ 0 ] ) + " , " + str ( parameters [ len ( parameters ) - 1 ] [ 1 ] ) + " )." )  
7     # Parameter Time ( t ).  
8     t = list ( map ( lambda x: x [ 1 ], parameters ) )  
9     # Parameter Size ( n ).  
10    n = list ( map ( lambda x: x [ 0 ], parameters ) )  
11    # Proposed function  $g(n) = (3/2)(n)(\log(n))$ .  
12    _t = list ( map ( lambda x: ( 3/2 ) * x, t ) )  
13    # Axes names.  
14    plt.xlabel ( "Size ( n )", color = ( 0.3, 0.4, 0.6 ), family = "cursive",  
15        size = "large" )  
16    plt.ylabel ( "Time ( t )", color = ( 0.3, 0.4, 0.6 ), family = "cursive",  
17        size = "large" )  
18    # Plot.  
19    plt.plot ( n, t, "#778899", linewidth = 3, label = "T ( n ) = ( n ) ( log ( n ) )" )  
20    plt.plot ( n, _t, "#800000", linestyle = "—", label = "g(n)=(3/2)(n)(log ( n ))" )  
21    plt.legend ( loc = "upper left" )  
22    plt.show ( )
```

Observation: We propose a function $g(n) = (3/2)(n)(\log(n))$. In line 12, we take the elements of *t* and multiply each one for (3/2), for later plot $t_{g(n)}$ against *size* (*n*).

4.1.3 Merge.py

The function *Merge* (...) it's very simply, takes two 'lists' and star to sort and combine those 'lists'.

Observation: For now we will center on this code, in section 3.2 we will calculate the temporal complexity for this algorithm. And we will explain it more thoroughly

```
1 def merge ( left , right ):  
2     m, i, j, time = [ ], 0, 0, 1  
3     # Convine and sort the lists 'left' and 'right'.  
4     while ( i < len ( left ) and j < len ( right ) ):  
5         time += 1  
6         if ( left [ i ] <= right [ j ] ):  
7             m.append ( left [ i ] )  
8             time += 1  
9             i += 1  
10            time += 1  
11        else:  
12            m.append ( right [ j ] )  
13            time += 1  
14            j += 1  
15            time += 1  
16    time += 1  
17    # The loop may break before all the rest of the elements in the lists  
18    # 'left' and 'right' are appended, hence, append the remaining elements.  
19    m.extend ( left [ i: ] )  
20    time += 2  
21    m.extend ( right [ j: ] )  
22    return m, time
```

4.1.4 Mergesort.py

This it's the principal algorithm for this section, *mergesort* will receive as parameter a 'list' -Let's name this 'list' *A*-, of integer elements, and, if the length of *A* it bigger than 1, then the algorithm will split *A* in two halves, and will recursively call itself sending as parameter the first and the second half as we can see in lines 7 and 8, and will repeat this process until reaching the algorithm *bottom out*.

```
1 def mergesort ( n ):  
2     # If the list has at most 1 element, return that list.  
3     if ( len ( n ) <= 1 ):  
4         return n  
5     # middle: Stores the integer length of the list splitted in 2.  
6     middle = int ( len ( n ) / 2 )  
7     left = mergesort ( n [ :middle ] )  
8     right = mergesort ( n [ middle: ] )  
9     # 'merge' convine and sort the left and right parts of the original list.  
10    return merge ( left , right )
```

As we can see in the code shown above, in line 10 we call *merge (...)* so, the return statement will be a combined and sorted 'list'. To calculate the temporal complexity of this algorithm, we need to add a counter in each line (*time*):

```
1 def mergesort ( n ):  
2     time = 0  
3     # If the list has at most 1 element, return that list.  
4     if ( len ( n ) <= 1 ):  
5         return n, ( len ( n ), time )  
6     # middle: Stores the integer length of the list splitted in 2.  
7     middle = int ( len ( n ) / 2 )  
8     time += 1  
9     left , parameters = mergesort ( n [ :middle ] )  
10    time += 1 + parameters [ 1 ]  
11    right , parameters = mergesort ( n [ middle: ] )  
12    time += 1 + parameters [ 1 ]  
13    # 'merge' convine and sort the left and right parts of the original list.  
14    m, t = merge ( left , right )  
15    time += 1 + t  
16    return m, ( len ( n ), time )
```

Now, apart of return only the sorted and combined 'list', will return a tuple named *parameters* (The tuple *parameters* it's different than the 'list' with the same name that we talked about earlier). As we can see in line 16, *parameters* inside has a counter named *time*, this counter will not be the computational time of the algorithm unless we sum the other *times* of the recursively calls as we can see in lines 10 and 12. Thus, when the process it's finished, the algorithm will return *A* sorted and a tuple with the *length of A* and the *computational time* that the algorithm takes to sort *A*.

Observation: The tuple *parameters* returned in this algorithm in *main.py* will become an element of the 'list' *parameters*.

Observation: As we can see in line 14, *Merge* also return his computation time (*t*), this value it's also summed to the counter of the temporal complexity of *Merge-Sort*.

4.2 Merge:

Basically it's the very same program of *Merge-Sort* but with some modifications. Still, we will explain the functionality of it. The program it's divided in four python modules to have a better control of the code:

- *main.py*: Control the execution sequence, as well generate the list to sort.
- *mergesort.py*: Contains the *Merge-Sort* algorithm.
- *merge.py*: Contains the *Merge* algorithm.
- *graph.py*: Plot *length of the 'list'* against the *time* that the algorithm *Merge* takes to sort it.

4.2.1 Main.py:

As well as Section 3.1.1, this program has a *menu ()* method that does exactly the same. The difference between this *main.py* and the one seen previously it's what *mergesort (...)* returns.

```
1 def main ( ):  
2     n = menu ( )  
3     print ( "\n\tList to sort: ", n )  
4     n, parameters = mergesort ( n )  
5     print ( "\n\tSorted list: ", n )  
6     print ( "\n\tMerge Parameters: ( ", parameters [ 0 ], " , " ,  
7         parameters [ 1 ] - 1, " )\n" )  
8     graph ( parameters )  
9 main ( )
```

As we can see, *mergesort (...)* also receive a 'list' *A* and returns two variables, in this case, we won't need a *for loop* because this time we are calculating the time that *Merge* takes to sort *one* simply 'list' and not the time that *MergeSort* takes to sort a 'list' doing recursive calls. The first returned variable it's *A* sorted, and the second one it's only a tuple with two variables: The *Length* of *A* and the *computational time* that *Merge* takes to sort *A*. So then, let *Parameters (Size (n), Time (t))*, where '*n*' = *A* and *t* it's the computational time.

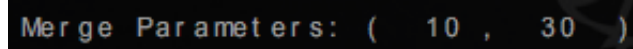


Figure 4.2.1.0: *Parameters* tuple, where the first element it's the *size* of *A* and the second element it's the *computation time* that *Merge* takes to sort and combine this list. For this example *A* it's of size *n = 10* as we can see.

4.2.2 Graph.py

Graph (...) plot **Size** (n) where ' n ' it's the length of the 'list' **A**, against **Time** (t) where t it's the computational time that **Merge** takes to sort this list. As we can see, the function receive the tuple **parameters**, as we said, stores 2 elements the **size** and the **time**, well, this tuple will gave us indirectly the points to plot, so, the only thing rest to do is divide **parameters** into two lists, one for the size, and another one for the computational time, this process is performed in the lines 8 and 10.

```
1 def graph ( parameters ):  
2     # Window title .  
3     plt.figure ( "Merge Algorithm" )  
4     # Plot title .  
5     plt.title ( "Merge ( size , time ): ( " + str ( parameters [ 0 ] ) + " , "  
6         + str ( parameters [ 1 ] - 1 ) + " )" )  
7     # Parameter Time ( t ) .  
8     t = np.arange ( 0 , parameters [ 1 ] + 1 , int ( ( parameters [ 1 ] ) / ( parameters [ 0 ] ) ) )  
9     # Parameter Size ( n ) .  
10    n = np.arange ( 0 , parameters [ 0 ] + 1 )  
11    # Proposed function  $g ( n ) = ( 3/2 ) n$  .  
12    _t = list ( map ( lambda x: ( 3/2 ) * x , t ) )  
13    # Names of the axes .  
14    plt.ylabel ( "Time ( t )" , color = ( 0.3 , 0.4 , 0.6 ) , family = "cursive" , size = "large" )  
15    plt.xlabel ( "Size ( n )" , color = ( 0.3 , 0.4 , 0.6 ) , family = "cursive" , size = "large" )  
16    # Plot .  
17    plt.plot ( n , t , "#778899" , linewidth = 3 , label = "T( n ) = n " )  
18    plt.plot ( n , _t , "#800000" , linestyle = "—" , label = "g( n ) = ( 3/2 )( n )" )  
19    plt.legend ( loc = "upper left" )  
20    plt.show ( )
```

Observation: In the tuple **parameters** always the **time** value will be bigger than the **size** of **A** so, to have the correctly points to plot we will need that the lists **t** and **n** in lines 8 and 10 respectively, be of the same size, for this the list **t** will be from '**0**' to '**k**' in intervals of (k / l) where **k** will be the computational time and **l** the length of **A**.

Observation: We propose a function $g(n) = (3/2) (n)$. In line 12, we take the elements of **t** and multiply each one for $(3/2)$, for later plot $t_{g(n)}$ against **size** (n).

4.2.3 Mergesort.py

This algorithm it's the same showed in section 3.1.4, but with some modifications in the return statement. The algorithm does the same process explained above, but with the difference that in the lines 7 and 8, the variable parameters doesn't store the **computational time** of *mergesort*, but rather the **computational time** of *merge* and the size of **A** where **A** it's the list to sort.

```
1 def mergesort ( n ):  
2     # If the list has at most 1 element, return that list.  
3     if ( len ( n ) <= 1 ):  
4         return n, ( len ( n ), 0 )  
5     # middle: Stores the integer part of the list length divided over 2.  
6     middle = int ( len ( n ) / 2 )  
7     left, parameters = mergesort ( n [ :middle ] )  
8     right, parameters = mergesort ( n [ middle: ] )  
9     # 'merge' convine and sort the left and right parts of the original list.  
10    return merge ( left , right )
```

4.2.4 Merge.py

This it's the principal algorithm for this section, *merge* will receive as parameter two 'list' -Lets name this 'lists' *B* and *C*-, which are the *left* and *right* halves of a 'list' *A*.

```
1 def merge ( left , right ):  
2     m, i, j = [ ], 0, 0  
3     # Convine and sort the lists 'left' and 'right'.  
4     while ( i < len ( left ) and j < len ( right ) ):  
5         if ( left [ i ] <= right [ j ] ):  
6             m.append ( left [ i ] )  
7             i += 1  
8         else:  
9             m.append ( right [ j ] )  
10            j += 1  
11    # The loop may break before all the rest of the elements in the lists  
12    # 'left' and 'right' are appended, hence, append the remaining elements.  
13    m.extend ( left [ i: ] )  
14    m.extend ( right [ j: ] )  
15    return m
```

The principal function of this algorithm it's to sort *A* using the *divide-and-conquer* paradigm, of course, this algorithm works perfectly with *mergesort*. To calculate the computation time that this algorithm takes to sort a list of length '*n*' it's necessary to put a counter in each line -Lets call this counter *time*-, after finishing the algorithm process the program will store *time* in a tuple along with the length of *A*, together will conform the tuple that we have already been talking about.

```
1 def merge ( left , right ):  
2     m, i, j, time = [ ], 0, 0, 1  
3     # Convine and sort the lists 'left' and 'right'.  
4     while ( i < len ( left ) and j < len ( right ) ):  
5         time += 1  
6         if ( left [ i ] <= right [ j ] ):  
7             m.append ( left [ i ] )  
8             time += 1  
9             i += 1  
10            time += 1  
11        else:  
12            m.append ( right [ j ] )  
13            time += 1  
14            j += 1  
15            time += 1  
16    time += 1  
17    # The loop may break before all the rest of the elements in the lists  
18    # 'left' and 'right' are appended, hence, append the remaining elements.  
19    m.extend ( left [ i: ] )  
20    time += 1  
21    m.extend ( right [ j: ] )  
22    time += 1  
23    #print ( "\nArray: ", m, "\tCounter: ", count )  
24    return m, ( len ( m ), time )
```

5 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.

5.1 Merge-Sort Results:

Test of Merge-Sort Algorithm. The program will plot the **time** that the algorithm takes to sort a 'list' of size $n = 6$.

```
(myenv) MacBook-Pro-de-David: Merge-Sort Computational Time Davestring$ python3 main.py

Merge-Sort Algorithm
Add the size of the list: 6
List to sort: [6, 5, 4, 3, 2, 1]
Sorted list: [1, 2, 3, 4, 5, 6]
MergeSort Parameters: [(0, 0), (1, 0), (2, 11), (3, 25), (4, 36), (5, 53), (6, 67)]
```

Figure 5.1.0: Console output of Merge-Sort algorithm.

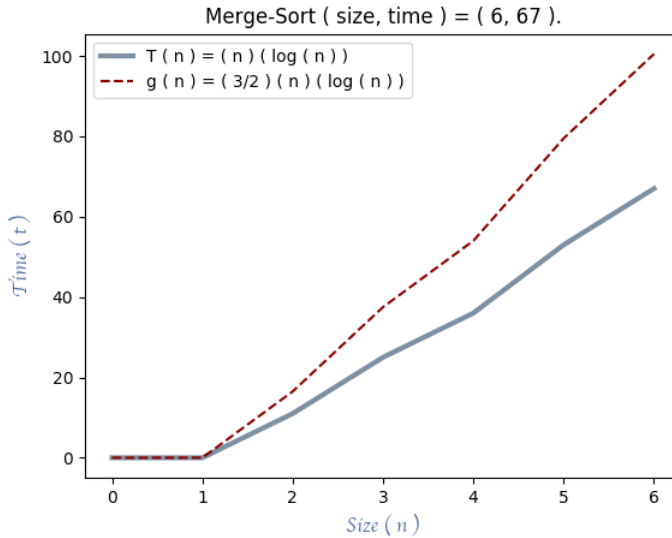


Figure 5.1.1: Plot of Figure 4.1.0.

Size (n)	Time (t)
0	0
1	0
2	11
3	25
4	36
5	53
6	67

Table 1: Plot points of Figure 4.1.1.

Observation: Remember that in **main.py** the program split the 'list' into sublists B_i and calculates the computational time that takes to sort each one until $B_i = A$. The column **Size** in the table it's the length of this B_i lists.

Observation: The plot has two curves, the blue one its the computation time of our algorithm: $T(n) = (n)(\log(n))$, and the red one it's a proposed function $g(n) = (3/2)(n)(\log(n))$ where $T(n) \in \theta(g(n))$.

Observation: For this example we are analyzing a worst case.

5.2 Merge Results:

Test of Merge Algorithm. The program will plot the *time* that the algorithm takes to sort a 'list' of size $n = 10$.

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

Merge-Sort Algorithm:

Add the size of the list: 10

List to sort: [83, -29, -7, 43, -40, 45, -42, -25, -15, 80]
Sorted list: [-42, -40, -29, -25, -15, -7, 43, 45, 80, 83]
Merge Parameters: ( 10 , 30 )
```

Figure 5.2.0: Console output of Merge algorithm.

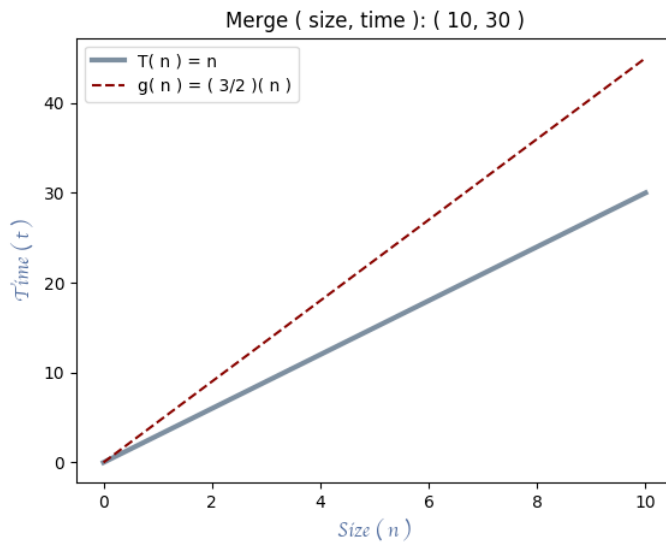


Figure 5.2.1: Plot of Figure 4.2.0.

Size (n)	Time (t)
0	0
1	3
2	6
3	9
4	12
5	15
6	18
7	21
8	24
9	27
10	30

Table 2: Plot points of Figure 4.2.1.

Observation: The plot has two curves, the blue one its the computation time of our algorithm: $T(n) = (n)$, and the red one it's a proposed function $g(n) = (3/2)(n)$ where $T(n) \in \theta(g(n))$.

6 Annexes

In the following lines we will formally demonstrate 4 algorithms, including *Merge* and *Merge-Sort*.

6.1 Merge Demonstration:

Demonstrate that Merge algorithm has *linear* order:

function MERGE (A, p, q, r):

```

1   n1 = q - p + 1
2   n2 = r - q
3   Let L [ 1, ..., n1 + 1 ] and R [ 1, ..., n2 + 1 ] be new arrays.
4   for i = 1 to n1 do
5       L [ i ] = A [ p + i - 1 ]
6   for j = 1 to n2 do
7       R [ j ] = A [ q + j ]
8   i = 0
9   j = 0
10  for k = p to r do
11      if L [ i ] <= R [ j ]
12          A [ k ] = L [ i ]
13          i++
14      else A [ k ] = R [ j ]
15          j++

```

- *Demonstration:*

- *Analyzing the complexity of each line:*

1. Line 1, 2, 3 = $\theta (1)$.

2. Line 4 = $\theta (n_1)$.

(i) Line 5 = $\theta (1)$.

3. Line 6 = $\theta (n_2)$.

(i) Line 7 = $\theta (1)$.

4. Line 8, 9 = $\theta (1)$.

\implies

5. Line 10 = $\theta (r - p + 1) = \theta (n_3)$.

(i) Line 11 = $\theta (1)$.

(ii) Line 12 = $\theta (1)$.

(iii) Line 13 = $\theta (1)$.

(iv) Line 14 = $\theta (1)$.

(v) Line 15 = $\theta (1)$.

1. Line 1, 2, 3 = $\theta (1)$.

2. Line 4 - 5 = $\theta (n_1 * 1) = \theta (n_1)$.

3. Line 6 - 7 = $\theta (n_2 * 1) = \theta (n_2)$.

4. Line 8, 9 = $\theta (1)$.

5. Line 10 - 11 - 12 - 13 - 14 - 15 = $\theta (n * 1 * 1 * 1 * 1 * 1) = \theta (n_3)$.

- *Then, from all lines:*

$$T (n) = \theta (n_1 + n_2 + n_3 + 1 + 1) = \theta (n) \quad (2)$$

- *Finally:*

$$Merge \in \theta (n) \quad (3)$$

6.2 Merge-Sort Demonstration:

Demonstrate that MergeSort algorithm has $T(n) = (n)(\log(n))$ order:

function MergeSort (A, p, r):

```

1 if ( p < r )
2     q = ( p + r ) / 2
3     MergeSort ( A, p, r )
4     MergeSort ( A, q + 1, r )
5     Merge ( A, p, q, r )

```

- **Demonstration:**

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly $\frac{n}{2}$. We reason as follows to set up the recurrence for $T(n)$ the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

- **From equation 1:**

- Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \theta(n)$.
- Conquer:** We recursively solve two subproblems, each of size $\frac{n}{2}$ which contributes $2T(\frac{n}{2})$ to the running time.
- Combine:** We have already noted that the MERGE procedure on an n -element subarray takes time $\theta(n)$ and so $C(n) = \theta(n)$

- **Then, our recurrence equations is:**

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(\frac{n}{2}) + cn & \text{if } n > 1 \end{cases} \quad (4)$$

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

$$T(n) = 2T(\frac{n}{2}) + cn \quad (5)$$

$$= 2T(2^{k-1}) + c(2^k) \quad (6)$$

$$= 2^2T(2^{k-2}) + 2c(2^k) \quad (7)$$

$$= 2^3T(2^{k-3}) + 3c(2^k) \quad (8)$$

$$(9)$$

- **Then:**

$$T(n) = 2^i T(2^{k-i}) + ic(2^k) \quad (10)$$

$$(11)$$

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

$$T(n) = 2^k T(1) + (kc) (2^k) \quad (12)$$

$$= (c) (2^k) + (kc) (2^k) \quad (13)$$

$$= (c) (2^k) [1 + k] \quad (14)$$

$$= (cn) [\log_2(n) + 1] \quad (15)$$

$$= (cn) \log_2(n) + (cn) \quad (16)$$

$$(17)$$

- *Finally:*

$$\text{MergeSort} \in \theta((n)(\log_2(n))). \quad (18)$$

6.3 Function A:

Calculate the complexity order of the following algorithm in the best (Ω) and the worst (O) case (it's not necessary to do the analysis line-per-line, in this case, you can apply properties of the algorithms):

Function A (n even):

```
1   i = 0
2   while i < n do
3       to j = 1 until j = 10 do
4           Action(i)
5           j++;
6   i+=2;
7   Suppose that action  $\in \theta(1)$ 
```

- **Demonstration:**

Observation: Best case: Let $n = 0$.

- **Analyzing the complexity of each line:**

1. Line 1 = $\theta(1)$.
2. Line 2 = $\theta(1)$.

Observation: When $n = 0$, the code only evaluate the condition of the while loop, then, because $n = i$, the condition isn't fulfilled, so the code inside the while it's not executed, thus.

$$T(n) = \theta(1 + 1) = \theta(1) \quad (19)$$

- **Finally:**

$$\text{Function A} \in \Omega(1) \quad (20)$$

- *Demonstration:*

Observation: Analyzing the worst case:

- *Analyzing the complexity of each line:*

1. Line 1 = $\theta (1)$.
2. Line 2 = $\theta (n)$.
 - (i) Line 3 = $\theta (1)$.
 - (ii) Line 4 = $\theta (1)$.
 - (iii) Line 5 = $\theta (1)$.
 - (iv) Line 6 = $\theta (1)$.

- *Then, from Line 1 and 2:*

$$T (n) = \theta (n + 1) = \theta (n) \quad (21)$$

- *Finally:*

$$\text{Function } A \in O (n) \quad (22)$$

6.4 Function B:

Calculate the complexity order of the following algorithm in the best (Ω) and the worst (O) case (it's not necessary to do the analysis line-per-line, in this case, you can apply properties of the algorithms):

Function B ($A[0, \dots, n-1]$, x integer):

```
1  for i = 0 i < n do
2      if (A[i] < x)
3          A[i] = min(A[0], ..., n-1])
4      else if (A[i] > x)
5          A[i] = max(A[0], ..., n-1])
6      else
7          exit
```

- **Demonstration:**

Observation: Best case: Let 'x' be in the first position of A.

- **Analyzing the complexity of each line:**

1. Line 1 = $\theta(1)$.
 - (i) Line 2 = $\theta(1)$.
 - (ii) Line 4 = $\theta(1)$.
 - (iii) Line 6 = $\theta(1)$.
 - (iv) Line 7 = $\theta(1)$.

Observation: The lines 3 and 5 doesn't count because 'x' doesn't fulfill the condition.

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

$$T(n) = \theta(1) \quad (23)$$

- **Finally:**

$$\text{Function B} \in \Omega(1) \quad (24)$$

- *Demonstration:*

Observation: Worst case: Let 'x' be in the position $\lfloor n - 1 \rfloor$ or isn't be in A.

- *Analyzing the complexity of each line:*

1. Line 1 = $\theta (n)$.
 - (i) Line 2, 3 = $\theta (n_1)$.
 - (ii) Line 4, 5 = $\theta (n_2)$.
 - (iii) Line 6, 7 = $\theta (1)$.

Observation: Line 6 and 7 are $\theta(1)$ if 'x' it's in the position $\lfloor n - 1 \rfloor$.

- *Then, from all lines:*

$$T (n) = \theta (n (n_1 + n_2 + 1)) = \theta (n^2) \quad (25)$$

- *Finally:*

$$\text{Function } B \in O (n^2) \quad (26)$$

7 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.

8 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.