Homework III - Analysis of Algorithms

Edwing Alexis Casillas Valencia
October 9, 2025

Abstract

In this homework, we analyze the Tim Sort algorithm on Python and compare it with the three linear sorts (Counting, Radix and Bucket sorts).

PROBLEM 1

Implement the three linear sortings using C

(a) Counting Sort
(b) Radix Sort
  i. Queue version
  ii. Give its new complexity in big O and prove it
(c) Bucketsort

Answer

Attached to this document are the three linear sortings in C.
The complexity for Radix sort using queues is

$$T(n,d) = O(1) + d * [O(n) + O(n+10)] \tag{1}$$

Simplifying we have

$$T(n,d) = O(1) + d * [O(2n+10)] \tag{2}$$

$$T(n,d) = O(1) + O(2dn + 10d) \tag{3}$$

$$T(n) = O(d * n) \tag{4}$$

where d is the number of digits on the array. Also, there is an implementation on the code to confirm this, and it give us the following result.

```
Comprobando que T(n,d) es proporcional a d * n:

n        d        Tiempo(s)    d * n          T/(d*n)
-------------------------------------------------------------
1000     2        0.001000     2000           0.0000005000
1000     4        0.001000     4000           0.0000002500
1000     6        0.001000     6000           0.0000001667
5000     2        0.002000     10000          0.0000002000
5000     4        0.004000     20000          0.0000002000
5000     6        0.008000     30000          0.0000002667
10000    2        0.004000     20000          0.0000002000
10000    4        0.013000     40000          0.0000003250
10000    6        0.013000     60000          0.0000002167
```

Fig. 1: Probe of O(d*n)

Here we can see that when d increases from 2 to 6 (having n=1000), the execution time remains with the same value, and when n increases, the time value also increases d times. So, what does this means?, it means that when we have a constant value of d, the behavior is O(n), and when we have a non constant value of d, we have O(d). The last column show us that proportion of time against the behavior O(d*n)

PROBLEM 2

Compare them against Tim Sort the one in Python

(a) Please provide with plots
(b) Explain why the differences
  i. Use the papers
    A. Auger, Nicolas; Nicaud, Cyril; Pivoteau, Carine (2015). "Merge Strategies: from Merge Sort to TimSort"
    B. Auger, Jugé, Nicaud, Pivoteau (2018). "On the Worst-Case Complexity of TimSort"

Answer

(a) Here are the plots of execution time, taking an average time of each sorting algorithm after running 5 times each one.

```
===COUNTING SORT===
Your array is: 9  1  5  2  4  7  3
Ordered array
Your array is: 1  2  3  4  5  7  9
execution time: 4217 microseconds
```

Fig. 2: Counting sort

```
===RADIX SORT===
Your array is: 170  45  75  90  2  802  24  66
Ordered array
Your array is: 2  24  45  66  75  90  170  802
execution time: 1328 microseconds
```

Fig. 3: Radix sort using queue

```
Original: Array: 0.780 0.170 0.390 0.260 0.720 0.940 0.210 0.120 0.230 0.680
Ordenado: Array: 0.120 0.170 0.210 0.230 0.260 0.390 0.680 0.720 0.780 0.940
execution time: 1080 microseconds
```

Fig. 4: Bucket sort

```
Given Array is
[-2, 7, 15, -14, 0, 15, 0, 7, -7, -4, -13, 5, 8, -14, 12]
After Sorting Array is
[-14, -14, -13, -7, -4, -2, 0, 0, 5, 7, 7, 8, 12, 15, 15]

Exec time is: 0.00016780001169536263
```

Fig. 5: Tim sort (167 microseconds)

Also here is their asymptotic behavior

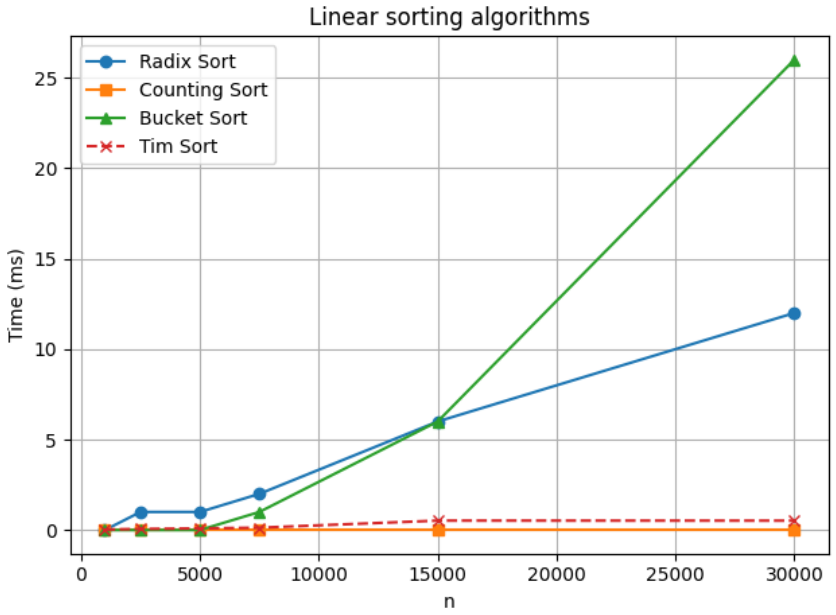

Fig. 6: Asymptotic behavior

(b) We can see that the execution time of Tim sort is less than the linear ones. This is due Tim sort is implemented as a hybrid of Merge sort and Insertion Sort, using a fusion of its recent runs on the system cache. This hybrid has a complexity of O(n log n) on the worst case and O(n) on the best case. Unfortunately, to reach the best case, we need to apply a pre-sorting on

the input array in order to help the Tim sort algorithm; in fact, Tim sort uses stacks to drive the runs ensuring that the length of the runs grow at least as fast as Fibonacci numbers, keeping the stack height within a logarithmic limit.

Couting Sort and radix sort work more efficiently for integers with a smaller range. Bucket Sort assumes that the data is already uniformly distributed.

One point to keep in mind is that Tim Sort is a more general approach; it works really good with partially sorted data. This means that there will be some cases that even if it's pretty good, another algorithms could be a better approach for specific data types as you can see in the following image.

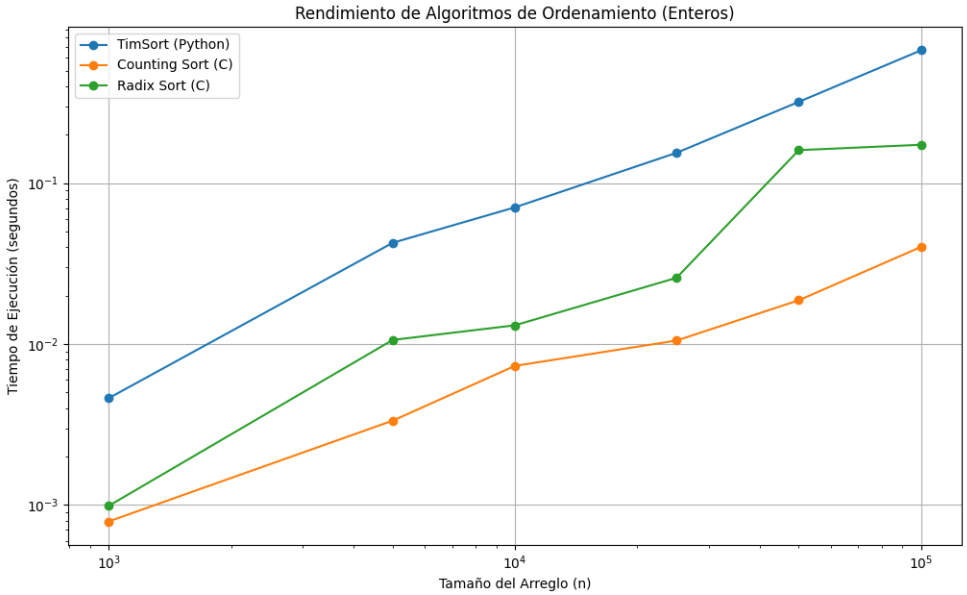

Fig. 7: specific cases

Complexity between algorithms are on the following table.

Algorithm	Worst Case	Average Case
Counting Sort	$O(n+k)$	$O(n)$
Radix Sort	$O(d*k)$	$O(n)$
Bucket Sort	$O(n^2)$	$O(n)$
Tim Sort	$O(nlogn)$	$O(n)$

TABLE I: Algorithms Complexity