Wasteland Sort

Complexity and Analysis of an Integer Sorting Algorithm

Markian Hromiak

October 10, 2020

Abstract

Sorting is one of the basic lifebloods of computation, and as such, any improvement in complexity costs will show drastically over with an algorithm's repeated and frequent use. Some sorting algorithms cater to a subset of data types. For example, quicksort and mergesort handle many floats, longs, strings and integers while radix sort is specialized to integers, and so on. With this highlighted emphasis on efficiency and the constraint of data types in mind, the goal was set to develop a universal sorting algorithm that could run in O(n) time. In this paper, we present an algorithm which brings us one step closer to this ideal sort. With a time complexity of O(n + m) and a space complexity of O(max(m, n)), the integer-favored sort entitled 'Wasteland sort' is illustrated, analyzed and compared empirically with established sorting algorithms.

Contents

1 Introduction

Languages and libraries for non-parallel computation host many standards when it comes to sorting algorithms. For example, Python's list.sort() method is backed by the timsort algorithm, while Java uses a double-pivot quicksort. Both of these algorithms are exceptional but leave space for improvement in complexity. Ideally, one would look at a list and have the list be sorted in O(1) time. This is unrealistic by computing standards as unsorted values must be moved, giving us a lower bound on performance of O(n). One way of sorting n values with complexity very close to O(n) is to sort by using the range of values. While the idea isn't new, a literature search prior to this project yielded no practical implementations of the idea — indeed, the idea was left as a practice problem in an algorithms textbook! Now we may say that there exists at least one practical implementation of the idea.

2 Description and pseudocode of the algorithm

To describe the algorithm with prose, imagine that you are standing in the middle of a giant desert — a wasteland, if you would. Because wastelands are generally hazardous environments, your survival instinct kicks in and you scan the entire horizon around you to search for landmarks such as cities or an oasis for shelter. You notice a number of landmarks, and want to determine which landmark is closest to you by sorting them. You also want to keep track of the distance of each other landmark as well in case the closest landmark turns out to be inhospitible.

Here the prose stops being of any use and data structures take over. The algorithm begins with a list of "landmarks" being passed in — imagine the numbers as being the number of kilometers away each landmark is to you.

We first find the miximum and minimum values of the landmarks and initialize an array, from now on known as the wasteland, to be of size range(landmarks) + 1. Using these values, we calibrate the 0th position of the wasteland, the length of which is the range of the landmarks, to represent the minimum value. For example, with landmarks [4,7,10,3,20] we have a range of 17 and the 0th position representing the integer '3', the 16th position representing '20'. Next, we iterate through our landmarks list, keeping count of how many times an integer appears in its representative slot in the wasteland. Finally, going through the wasteland once, for every index which is nonzero, we override the landmarks array with the index's representative number x times, where x is the counted value stored in the array.

With the general outline of the algorithm's function, let's turn to a line-by-line complexity analysis.

Listing 1: Wasteland Sort

```
def wasteland_sort(landmarks: List[int]) -> list:
      if len(landmarks) < 1:
       return landmarks
      least, most = find_extremes(landmarks)
      wasteland : list = [0] * (most - least + 1)
      index : int = 0
      for value in landmarks:
              index = value - least
              wasteland[index] += 1

      index = 0
      for x in range(len(wasteland)):
              item: int = wasteland[x]
              while item > 0:
                item -= 1
                if reverse:
                  landmarks[(len(landmarks) - 1) - index]
                   = least + x
                else:
                  landmarks[index] = least + x
                index += 1
```

Listing 2: Find Extremes

```
def find_extremes(landmarks : list):
   if landmarks == []:
       raise ValueError("Could not find maxima of the list")
   most, least = None, None

   for val in landmarks:
       if val > most:
           most = val
       if val < least:
           least = val
   return least, most
```

3 Complexity analysis

Both Wasteland sort and its helper function are listed above. Here we ask what the spacial and time complexities of Wasteland sort are.

3.1 Time complexity

As wasteland_sort make a subprocess call to find_extremes, it makes sense to begin time analysis with the child function.

3.1.1 Find Extremes

Let n be the length of the function parameter 'landmarks'. Checking if landmarks is empty is an O(1) operation.

Declaring and initializing most and least sum in line 4 to a total of O(1) time.

Line 6 begins a loop over all n values of landmarks, giving a base complexity of O(n), ignoring all contents; however, looking at lines 7-10, it is clear that assigning values to most and least gives a full complexity of O(1) + O(1), which is just O(1) worst-case. Multiplying these two complexities as they lie within a nested loop, we find that find_extremes runs with a total complexity of O(n).

$$\begin{aligned} O(find\_extremes) &= O(1) + 2 * O(1) + O(n) * [O(1) + O(1)] \\ &= 3 * O(1) + 2 * O(n) \\ &= O(n) \end{aligned} \tag{1}$$

3.1.2 Wasteland Sort

Given that find_extremes has a complexity of O(n), we can begin analyzing our sorting algorithm.

In line 2 we perform one O(1) check for the size of landmarks. If the check fails, then it is not worth sorting the list.

In line 4 we assign most and least to be the result of a call to find_extremes(), giving this a O(n) complexity.

In line 5 we create a list of zeros of size (most - least + 1). Let us call the length of this wasteland m. This operation therefore runs in O(m) time.

Line 7 runs at a base complexity of O(n), where n is the length of the landmarks array. All operations inside of this loop run in O(1) time, giving a total complexity of O(n).

Line 11 is O(1) for declaration and initialization of an integer.

Line 12 runs in O(m) base time for the length of the wasteland. The inner while loop is where things become interesting. While running through m indices, we will loop through n values to repopulate the landmarks array. In any case, we will while through each of m elements a portion of n times. Because we run through m once and n once, the overall loop complexity is O(m+n).

Therefore, our final complexity for Wasteland sort is:

$$
\begin{aligned}
O(Wasteland\_sort) &= O(1) + O(n) + O(m) * [O(1) + O(1)] + O(1) + O(m+n) \\
&= 2 * O(1) + O(n) + 2 * O(m) + O(m+n) \\
&= O(m+n)
\end{aligned}
\tag{2}
$$

If we really want to be picky, we can say that the complexity is max(2 * O(m), O(m + n)), where there can be a case where n < m; however, if we disregard the coefficient of 2, our complexity stands.

3.2 Space complexity

The time complexity of Wasteland sort runs just short of O(n) worst case. We will see that memory complexity runs well, but falls short of the ideal O(1). For all space complexity analysis, we will disregard the size of the input as taken memory.

3.2.1 Find Extremes

The space complexity for variables most and least in line 4 are both O(1). No other memory-using structures are created, giving find_extremes a memory complexity of O(1).

3.2.2 Wasteland Sort

- The size of the wasteland on line 5 is O(m), where m is the range of landmarks.

- Least and most on line 4 both take O(1) space, as does the function call. 'index' on line 6 takes O(1) space as well, with the remaining calls to index made strictly for reassignment.

- In the for loop through the wasteland starting at line 12, we go simply iterate through existing memory while reassigning value. No new memory is allocated aside from the declaration of item, which points to existing memory.

In the end, our worst-case memory value is O(m). Thus, the memory complexity for Wasteland sort is:

$$
\begin{aligned}
O(Wasteland\_Sort) &= O(m) + O(1) + O(1) + O(1) \\
&= O(m) + 3 * O(1) \\
&= O(m)
\end{aligned}
\tag{3}
$$

4 Sorting Algorithm Comparison

Now that we have a theoretical analysis of Wasteland sort's complexity costs, we turns to gather empirical data. We chose to compare Wasteland sort to Heapsort, Mergesort, Quicksort, Radix sort and Timsort. In particular, we chose Radix sort as both Radix and Wasteland specialize in sorting integers. We chose Timsort due to it's high efficiency and use as Python's standard sorting implementation. Finally, Heapsort Mergesort and Quicksort are introductory sorting algorithms that are widely used.

In order to sort we needed first to generate an appropriate data set. Using numpy, ten sets of data were generated, following this format: (small, medium, large, close, far). Small data included 1000 elements within the range 100, medium included 10000 within range 1000, large 40000 withing 4000, close 20000 within 2 and far 20000 within 100000. By varying both the amount and density of our data, we were able to generate a data set that covered most edge cases for sorting; the empty set was considered arbitrary for sorting purposes.

Each algorithm ran ten times over the entire data set, sorting each category individually. These times were then averaged and plotted as the graphs you can see in appendix A. Note that Radix sort was exempted from these graphs. The reason for this is that Radix sort ran so slowly that its bar rendered the remaining data points unreadable. The remaining data shows that Wasteland sort outperforms all other sorts excepting Timsort. Given the theoretical bounds calculated, with optimization it may be possible to outperform Timsort, but that is work for the future.

5 Conclusion and Future Work

Results have shown that Wasteland sort is a practical integer sorting algorithm that outperforms a wide selection of commonly-used sorting algorithms for data either large or small, clustered or spread. As such, its use in place of the other algorithms is recommended. Further optimization should be done in order to challenge Timsort and outperform it. Additionally, as objects can be represented as integers, objects' representations can be sorted; however, Wasteland sort is unable to sort floating point type primitives. Further research must be done into fitting floats into the wasteland. Finally, as each array's element is replaced, Wasteland sort is in-place but not stable, meaning that true object sorting cannot yet be done. Regardless of it's shortcomings, Wasteland sort provides a solid first step in our approach to developing an O(n) general-purpose sorting algorithm.

A Runtime Bar Graphs

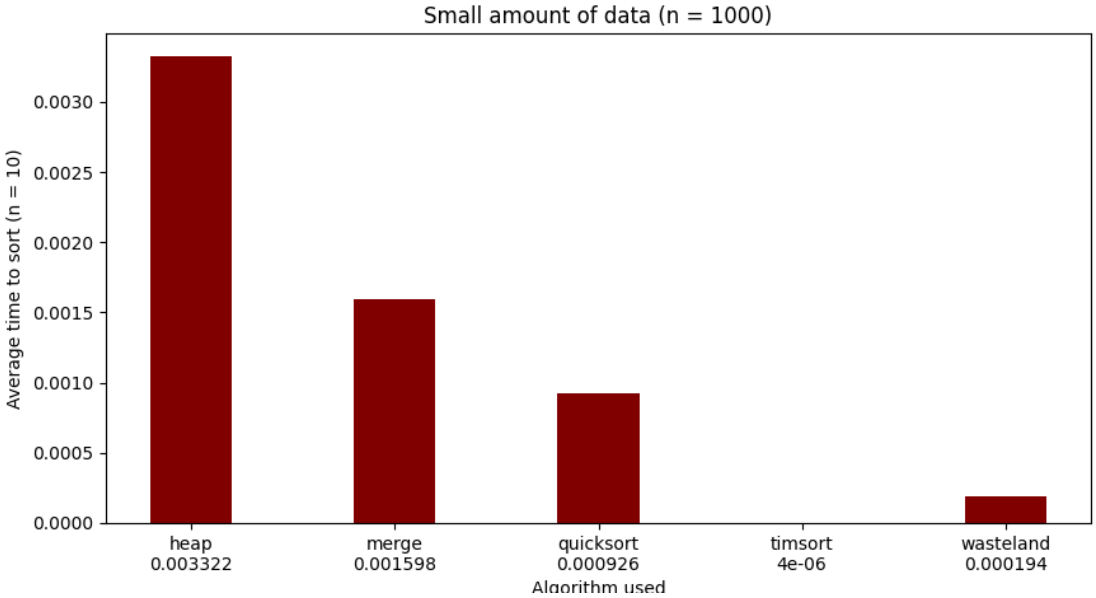

Figure 1: Comparison of performance for n = 1000

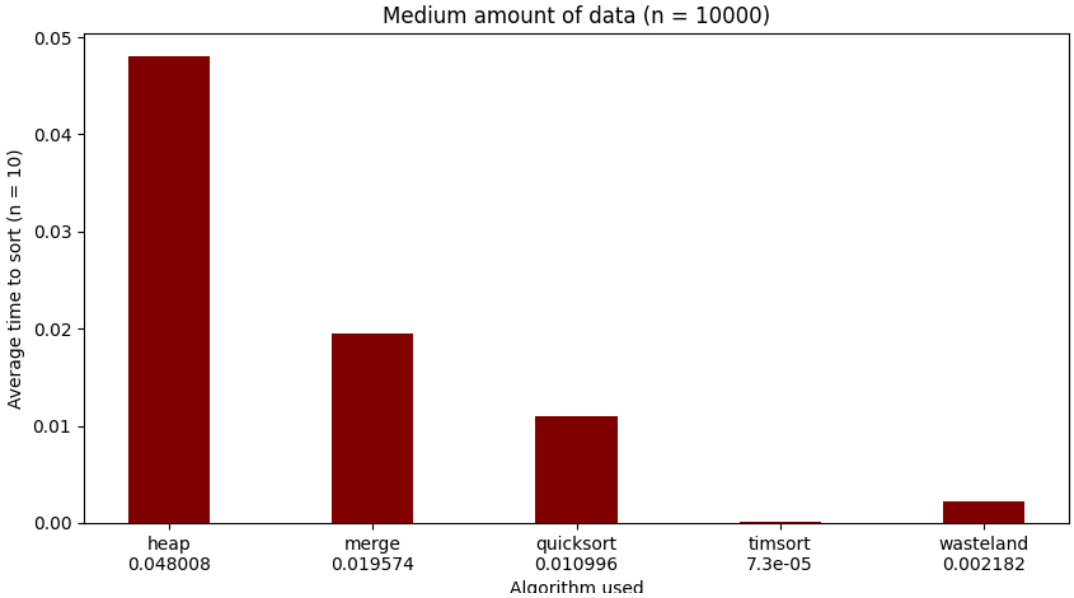

Figure 2: Comparison of performance for n = 10000

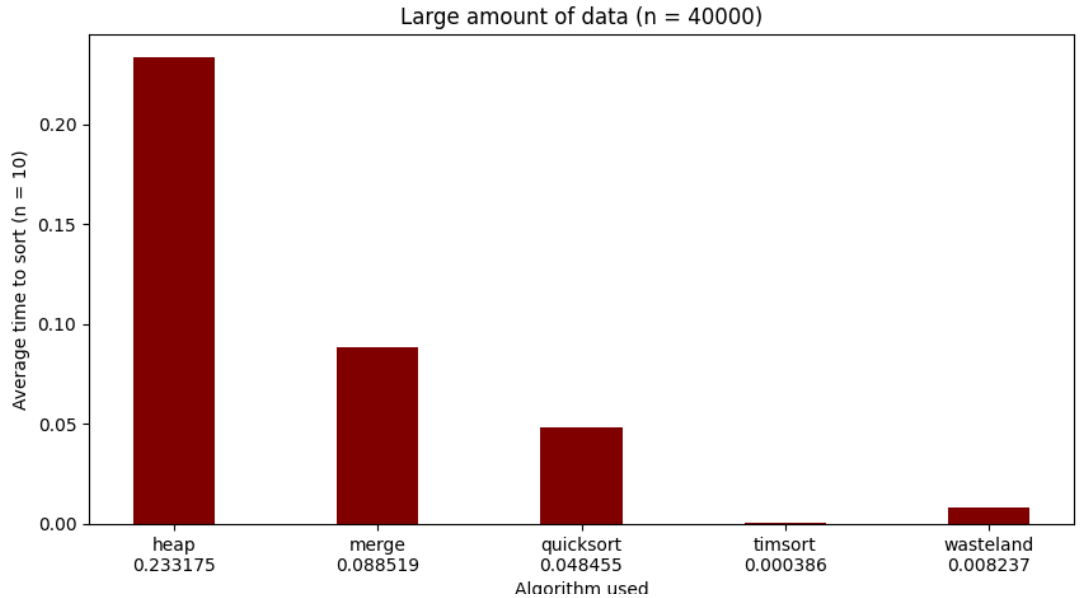

Figure 3: Comparison of performance for n = 40000

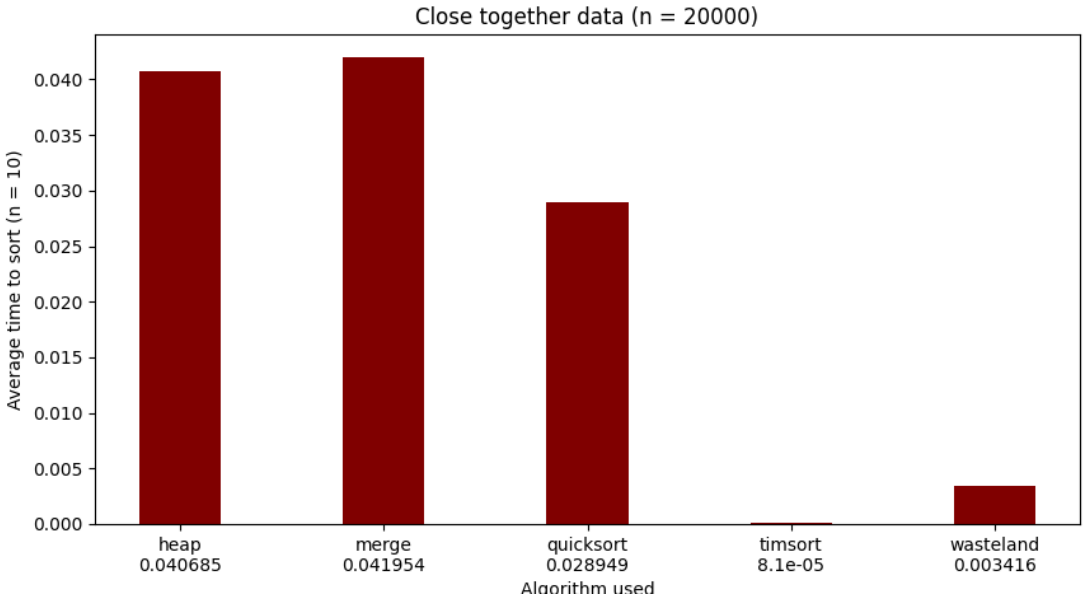

Figure 4: Comparison of performance for n = 20000, range(2)

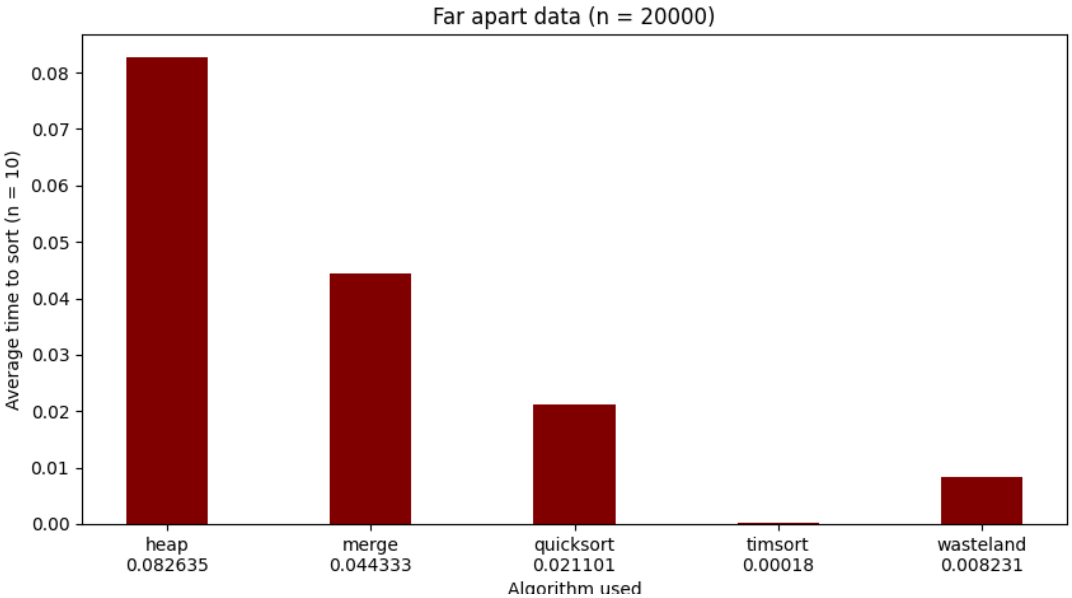

Figure 5: Comparison of performance for n = 20000, range(100000)

B Runtime Data

Below is an example of the readout produced in the **stats** module

```
('Small', 'Medium', 'Large', 'Close', 'Far', 'Sort name')

([0.00359344482421875, 0.04397063255310059, 0.23652143478393556,
 0.0480381965637207, 0.10899782180786133], 'heap')

([0.001960420608520508, 0.024118661880493164, 0.10649950504302978,
 0.0502837181091308 6, 0.055884814262390135], 'merge')

([0.0011498451232910157, 0.013671588897705079, 0.06537356376647949,
 0.0358097076416015 6, 0.024433994293212892], 'quicksort')

([4.434585571289063e-06, 7.069110870361328e-05, 0.0003722190856933594,
 9.300708770751954e-05, 0.00017440319061279297], 'timsort')

([0.00019304752349853517, 0.0021480560302734376, 0.009821963310241 7,
 0.0040182828903198246, 0.0095915079116821 28], 'wasteland')

([0.44091339111328126, 3.34621524810791, 12.998563098907471,
 4.832594871520996, 6.543259334564209], 'radix')
```