Module 8: Portfolio Project

Analysis of Algorithms and Data Structures

Nolan Byrnes

CSC506 – Design and Analysis of Algorithms

Colorado State University – Global Campus

Professor Isaac Gang

November 4th, 2022

**Portfolio Project**

Valdarrama notes that the “Timsort is near and dear to the Python community because it was created by Tim Peters in 2002 to be used as the standard sorting algorithm of the Python language” (Valdarrama, 2019). I am curious as to why Tim Peters decided to use this algorithm over the many other sorting algorithms out there. My assumption is that the Timsort is the best sorting algorithm out there, because for the algorithm to be included in Python’s standard library, you would imagine Tim Peters would only want to use the best sorting algorithm as the default sort method for the Python language.

**Testing Sorting Algorithms**

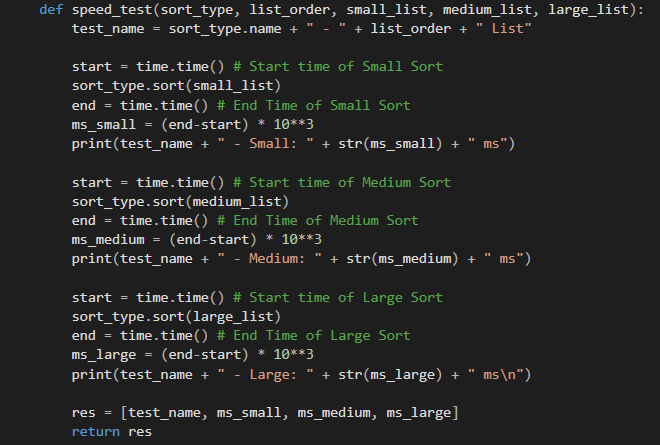
For this portfolio project, I recreated the TimSort, Quicksort, Merge sort, and Insertion sort in Python so that the performance of each sort can be compared. I saved the start time and end time of each sorting execution, and afterwards, I had the start time subtracted from the end time and converted the value to represent milliseconds. I had each of the sorting methods perform a sort on a list of integers that were sorted in increasing order, decreasing order, and random order. The range of values for each sort started from 1 to the length of the list. I decided to test the sorting methods on a list of 1000, 10000, and 100,000 integers.

**Implementation**

To test the sorting algorithms, I created a function that took in the sorting object, the name of the test being performed, and a copy of a small, medium, and large list. Observe the speed test function in figure 1 below.

Figure 1

Speed Test function



Note. The speed\_test() function was used to gather the amount of milliseconds it took to sort a small, medium, and large sized list.

I had the function calculate the start time, perform the sort, capture the end time, and calculate the amount of milliseconds that occurred for each of the lists that were passed in, and had the results returned in a list. This function was called a total of three times for each of the four sorting algorithms (Timsort, Quicksort, Insertion Sort, Merge Sort). By having the object passed in as an input parameter of this function, I was able to reduce the amount of code that would have been required to use to test the four sorting algorithms.

I also developed an accuracy test function that will take in the sorting object and would perform a sort on a hard coded list of integers. After the sorting has been completed, I had the sorted list compared to what the sorted list should be, and if the two lists were a match, the algorithm passed the accuracy test. I included this function so that I can ensure that the sorting algorithms that I implemented were sorting correctly and returning what we expected them to. I also chose to have the length of the list to have 51 elements because the timsort had a minimum run of 32. If the list was below 32 elements, then the timsort would have only been using the Insertion Sort part of the algorithm during the accuracy test. Observe the accuracy\_test() function in figure 2 below.

Figure 2

Accuracy Test Function

A screenshot of a computer

Description automatically generated with medium confidence

Note. This function was created to ensure that the sorting algorithms were working correctly.

To create the lists of increasing, decreasing, and random lists, I had functions that would generate them through a for-loop. For the random list, I used the Python module Random to generate a random integer in between zero and the length of the list. Observe the functions for creating the lists in figure 3.

Figure 3

Creating Lists functions

Text

Description automatically generated

Note. These functions were used to create an increasing, decreasing, and random list.

In the main() function, I had the four sorting objects created, and after performing an accuracy test on all four of the sorting methods, it would then test the sorting algorithms to increasing lists, decreasing lists, and random lists. For each of the speed tests, I made sure to have each of the four sorting algorithms sort a copy of the generated lists so that there would not be an unfair advantage if one random list would be more sorted than the other. After running the speed tests on the four sorting algorithms, I had the results saved into a .csv file. Observe the implemented main() function and the execution results in figures 4 and 5 below.

Figure 4

Main Function

Text

Description automatically generated

Note. The main() function tests the algorithms for accuracy, then proceeds to test the sorting algorithms, and save the results to a .csv file.

Figure 5

Execution Results

Text

Description automatically generated

Note. This represents the execution results of the speed and accuracy testing of the four sorting algorithms. The result of executing this program also creates a .csv file with all of the acquired data.

**Timsort**

**Implementation**

The Timsort algorithm is a hybrid sort that combines the Merge Sort, and the Insertion Sort (GeeksForGeeks, 2019). The Timsort is also a stable algorithm, which “maintains the relative order of the items with equal sort keys” (FreeCodeCamp, 2019, para. 5). This means that in cases where there are duplicate values in an array, the repeated value will not end up with a lower index than the first.

The timsort will divide the list of integers into runs, sort those runs by using the insertion sort, and merge them together with the combine function used in the merge sort. Depending on the size of the array being sorted, each run will contain between 32 and 64 elements. GeeksForGeeks mentions that the “merge function performs well when size subarrays are powers of 2” (GeeksForGeeks, 2022b, para. 2). If the length of the array is less than 32, the insertion sort is performed on the list, and no merging is performed. The implemented Timsort class can be observed in Figure 6 below.

Figure 6

Timsort Class

Text

Description automatically generated

Note. The Timsort class combines the merging step of the Merge algorithm with the Insertion Sort to be more efficient.

**Analysis**

The Timsort has a time complexity of O(n) in the best-case scenario, and O(N log N) for both the worst and average scenarios (GeeksForGeeks, 2019). Looking at the results in Table 1 below, we can see that the amount of time it took to sort the lists that were in decreasing, and random order do not have much of a time difference, although sorting in random order performed slightly better than the decreasing list. This is perhaps since with a random order, some of the elements within the collection may have been sorted in increasing order, whereas with a decreasing list, none of the elements are in the increasing order that the sort is working to achieve.

**Table 1**

Timsort Speed Test Results

|  |  |  |  |
| --- | --- | --- | --- |
| List Name | Small List (1000) | Medium List (10000) | Large List (100000) |
| Increasing List | 1.001358032 | 8.006811142 | 100.091218948364 |
| Decreasing List | 2.00176239013671 | 18.0165767669677 | 236.351490020751 |
| Random List | 2.00223922729492 | 20.0178623199462 | 263.239622116088 |

Note. This table represents the amount of milliseconds it took the timsort to sort an increasing, decreasing, and random lists with lists of sizes 1,000, 10,000, and 100,000 elements.

**Quicksort**

**Implementation**

The Quicksort is a divide and conquer algorithm that will pick an element as a pivot and partitions the array around the pivot point (GeeksForGeeks, 2022a). When the pivot point is declared, the algorithm places all values smaller than the pivot point to the left of the pivot, and values greater than the pivot to the right. The Quicksort is also an unstable algorithm, which means that the sort “does not maintain the key-value pair’s initial order” (Patel, 2022, para. 6). The quicksort implementation code can be viewed in figure 7 below.

Figure 7

Quick\_sort Class

Text

Description automatically generated

Note. The partition of the quick sort is determined by selecting the middle index of the list.

Analysis

In the best-case scenario, the QuickSort has a time complexity of O(N log N) (Patel, 2022). For the best case scenario to occur with the QuickSort, the partitioning algorithm must always choose the middle element as the pivot. In my QuickSort, I had the partitioning algorithm always choose the middle index as the pivot point. As you can see in Table 2 below, the sort times for the increasing and decreasing lists of the quicksort are very similar. This is due to the middle element being the pivot point of the partitioning function being the middle index. Because the middle index was used as the pivot point, the increasing and decreasing lists were the best-case scenarios for this implementation of the quicksort.

**Table 2**

Quicksort Speed Test Results

|  |  |  |  |
| --- | --- | --- | --- |
| List Name | Small List (1000) | Medium List (10000) | Large List (100000) |
| Increasing List | 1.00111961364746 | 9.00769233703613 | 106.096506118774 |
| Decreasing List | 1.00064277648925 | 9.00840759277343 | 110.100030899047 |
| Random List | 1.00088119506835 | 14.0085220336914 | 167.156219482421 |

Note. This table represents the amount of milliseconds it took the quicksort to sort an increasing, decreasing, and random lists with lists of sizes 1,000, 10,000, and 100,000 elements.

Lysecky mentions that the “quicksort algorithm's runtime is typically O(N log N)” (Lysecky, 2019, para. 7). The random order lists would be classified as the Quicksort’s average case, since with the random numbers, there is no guarantee that the middle index will be the middle element of that list. Comparing the speeds of the random list to the increasing and decreasing list, you can tell that since the middle element was not always selected, the execution time was affected, taking 57 extra milliseconds to sort the random list of 100,000 than it did the decreasing list of 100,000.

The worst-case scenario for the quicksort is where the “partitioning algorithm always chooses the largest or smallest element as the pivot point” (Patel, 2019, para. 28). When I was developing the quick sort, when I had the last index of an array sorted in decreasing order selected as the pivot point, I would receive a stack overflow error as soon as the list had a length of 997. You can view the code and execution results of this implementation of the quicksort in figures 8 and 9 below.

Figure 8

Quicksort with Last Index Partition

Text

Description automatically generated

Note. This is the code used for a Quicksort sorting algorithm where the last index of the array is always selected as the pivot point.

Figure 9

Quicksort last index partition execution results

Text

Description automatically generated

Note. These are the results that occur when running the code displayed in Figure 8, where we are performing a quicksort on a list of 997 elements in decreasing order, with the last index always being selected as the partition.

By selecting the middle index as the pivot instead, I would not receive a stack overflow error for either the increasing or decreasing list, and both the increasing and decreasing lists became the best-case scenarios for the implemented quicksort, since the middle element of the list was always chosen.

**Merge Sort**

**Implementation**

The merge sort is a sorting algorithm that recursively splits the list in half, and “is based on the divide and conquer paradigm” (GeeksForGeeks, 2022, para. 1) . The base case for this recursive algorithm is where the elements of the array can no longer be divided. The class for the merge sort can be viewed in figure 10 below.

Figure 10

Merge\_sort Class

Text

Description automatically generated

Note. The Merge\_sort class recursively calls .sort() until the length of the list is less than 2.

Each time that the algorithm splits the array in half, it recursively calls the merge sort on the two split arrays. After the list has been split into the until each are a list of 1 integer, the merge sort will then begin merging them together by looping through the indexes of the left and right list. If the value of the index in the left list is greater than the one in the right list, the value of the left list will be appended to the resulting list, and vice-versa. When the left or right index has reached its length, it will append the remaining numbers of the list of the opposite to the resulting list.

**Analysis**

GeeksForGeeks mentions that “the time complexity of Merge Sort is O(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves” (GeeksForGeeks, 2022, para. 8). When sorting the lists in increasing and decreasing order, the merge sort was able to sort them in just about the same amount of time, as shown in Table 3 below. For the random list, you can see that it takes longer for the merge sort to complete, and this is because the merge sort is having to have more comparisons when combining the list back together. Although the time complexity is categorized as O(N log N) for the increasing, decreasing, and random lists, it took longer to sort the random lists due to the increased amount of comparisons that the merge sort had to perform when sorting the list.

**Table 3**

Merge Sort Speed Test Results

|  |  |  |  |
| --- | --- | --- | --- |
| List Name | Small List (1000) | Medium List (10000) | Large List (100000) |
| Increasing List | 2.00176239013671 | 14.0130519866943 | 172.15609550476 |
| Decreasing List | 2.00176239013671 | 14.0128135681152 | 168.153047561645 |
| Random List | 2.00200080871582 | 23.0209827423095 | 282.256603240966 |

Note. This table represents the amount of milliseconds it took the merge sort to sort an increasing, decreasing, and random lists with lists of sizes 1,000, 10,000, and 100,000 elements.

**Insertion Sort**

**Implementation**

The insertion sort algorithm works similarly to how one would sort their hand of cards when playing a card game. the algorithm will loop through the indexes of the array, starting at index one. In each iteration of the loop, it will perform a nested loop. In this nested loop, it will check to see if the value of the index before it is greater, and if it is, it will switch the two indexes. This nested loop runs until the value of the index before it is less than the current indexes value, or the zeroth index has been replaced. Observe the python implementation of the insertion sort in figure 11 below.

Figure 11

Insertion\_sort Class

Text

Description automatically generated

Note. The Insertion\_sort class had the sort() method implemented to match the other objects, and the insertion\_sort() method developed so that the Timsort could also use it.

With the Insertion\_sort() class shown in figure 11 above, I have the class methods .sort() and insertion\_sort(). The reasoning behind this is because the Timsort also uses the insertion sort by calling .insertion\_sort(). When the insertion sort is running on its own for the speed tests, .sort() is being used to start the sort. When the timsort is using the insertion sort, it is calling the insertion\_sort() method directly.

**Analysis**

The insertion sort has a time complexity of O(n) in the best-case scenario, and O(n^2) in the average and worst-case scenario (Alake, 2022, para 21-22). The sorting speeds of the insertion sort can be viewed in Table 4 below.

**Table 4**

Insertion Sort Speed Test Results

|  |  |  |  |
| --- | --- | --- | --- |
| List Name | Small List (1000) | Medium List (10000) | Large List (100000) |
| Increasing List | 0 | 1.00064277648925 | 8.00704956054687 |
| Decreasing List | 50.0452518463134 | 5839.36166763305 | 586480.398416519 |
| Random List | 26.0336399078369 | 2995.19300460815 | 296315.2115345 |

Note. This table represents the amount of milliseconds it took the Insertion sort to sort an increasing, decreasing, and random lists with lists of sizes 1,000, 10,000, and 100,000 elements.

When the insertion sort is sorting a list that is already in increasing order, it performed extremely well, taking only 8 milliseconds to sort 100,000 integers, running with a time complexity of O(n). The insertion sort’s worst-case scenario is when all the integers are in decreasing order, and the results in Table 4 above highlight that the decreased list is the worst-case scenario. The average case for the insertion sort would be where the numbers are random, and with the results of the insertion sort sorting the random list, it performed better than the decreasing list. Although the average and worst-case scenario are both categorized as having a time complexity of O(n^2), it took less time to sort the random list compared to the decreasing list. This is because with the random list, the sort did not have to move as many elements than it did with the decreasing list.

**Comparisons of Sorting Algorithms**

Comparing the timsort with the merge sort, I noticed that the timsort had a slight performance increase than the merge sort. When it came to sorting 100,000 random integers, the timsort was able to complete the sort 19.02 milliseconds faster than the merge sort. When it came to sorting a decreasing list, the merge sort performed better than the timsort. This is because a timsort uses the insertion sort on the runs, and since the insertion sorts worst case scenario is when the numbers are in decreasing order, it affected the timsort’s runtime. When the timsort and merge sort are compared with the increasing list, the timsort performed better than the merge sort. The factor that helped the timsort in this case is that when the insertion sort is sorting numbers in increasing order, the time complexity is O(n).

When we compare the timsort with the insertion sort, the insertion sort performed faster when the list was in increasing order. This could be since with the insertion sort, it was able to loop through all of the indexes in O(n) time. The timsort had to split the list into runs and perform the insertion sort on each of those runs before merging them back together. In the decreasing list, the timsort was able to perform the sorting faster than the insertion sort. The worst-case scenario for the insertion sort is when the numbers are in decreasing order and has a time complexity of O(n^2), whereas it is O(n log n) for the timsort.

Comparing the timsort to the quicksort, the timsort was able to outperform the quicksort when we were sorting an increasing list, however, when it came to a decreasing or random list, the quicksort was able to sort the list faster. The decreasing list is timsort’s worst-case scenario and has a time complexity of O(n log n). Although the worst-case scenario for the quicksort is O(n^2), a decreasing list is not the worst-case scenario for my implementation of the quick sort.

**Conclusion**

At first glance of the results that I received through my speed tests, the quicksort seemed to outperform the timsort, and would be the fastest sorting algorithm. The catch with the quicksort that its worst-case scenario is when the chosen pivot is always the smallest or largest element and has a time complexity of O(n^2). With my implementation of the quicksort, I chose to have the pivot point to be the middle index of the array, since when I attempted to have the quicksort implemented with the last index being selected as the partition, I would receive a stack overflow. With the increasing, decreasing, and random lists, the quicksort did not have to handle its worst-case scenario due to the pivot selection. The timsort performed better than the insertion sort and the merge sort because the timsort algorithm takes the best out of both algorithms to be more efficient. The timsort also has the benefit of being a stable algorithm, unlike the quicksort. If Python’s .sort() function was using the quicksort algorithm, it could be faster for most cases, but would not be reliable since the worst-case scenario of a quicksort could result in a stack overflow.

**References**

Alake, R. (2022, August 21). Insertion sort Explained–A data scientists algorithm guide. NVIDIA Technical Blog. https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide

FreeCodeCamp. (2018, December 17). Stability in sorting Algorithms — A treatment of equality. <https://www.freecodecamp.org/news/stability-in-sorting-algorithms-a-treatment-of-equality-fa3140a5a539/>

GeeksForGeeks. (2022, September 1). Merge sort algorithm. <https://www.geeksforgeeks.org/merge-sort/>

GeeksForGeeks. (2022a, July 14). QuickSort. <https://www.geeksforgeeks.org/quick-sort/>

GeeksForGeeks. (2021, June 26). TimSort. <https://www.geeksforgeeks.org/timsort/>

Lysecky, R., & Vahid, F. (2019). 3.8: Quicksort. In Design and Analysis of Algorithms. zyBook.

OpenGenus IQ. (2020, June 7). Tim sort. <https://iq.opengenus.org/tim-sort/>

Patel, H. (2022, March 14). An overview of QuickSort algorithm. Medium. <https://towardsdatascience.com/an-overview-of-quicksort-algorithm-b9144e314a72>

Valdarrama, R. (2020, April 15). Sorting algorithms in Python – Real Python. Python Tutorials – Real Python. <https://realpython.com/sorting-algorithms-python>