Project 2 Report

CSCI 3330 Algorithims Hu

Chandler Richmond

This project has been created to examine four different sorting algorithms including bubble sort, merge sort, quick-sort and insertion sort. The accompanying code is used in order to display the similarities and differences in speed between the different sorting mechanisms in regard to computational performance. The coding language used in this project will be Python using the Spyder programming tool in order to produce code that can accurately display the capabilities of each type of sort. The project composed of the python code and report has been created and submitted by Chandler Richmond as the sole contributor to the project. His responsibilities have included the arrangement of the sorting algorithms in the program, the research regarding the studied algorithms, as well as the trial and error of the program. In order to accurately compare the sorting times of each sorting algorithm, a timer function has been implemented to keep track of the time it takes to sort an array of 500 integers as a standard for the different types of sorts used. These 500 integers have been randomly generated and stored in a text file named generated.txt to be sorted by the four algorithms in the program.

The sorting algorithms included in this project will be bubble-sort, merge-sort, quick-sort, and insertion sort.

Bubble Sort

Bubble Sort is the first option displayed in the program menu. Overall, this sorting algorithm is simple to understand but inefficient to use because of the method that it uses to sort. In this algorithm two numbers are compared at a time. If the numbers are in the incorrect spot in relation to each other, then they will switch places until the entire array is in the correct order. This sorting algorithm has both an average and wort-case run time of O(n^2). Because of this, it has been identified as the slowest of the four algorithms studied. In our program, the bubble sort method was able to sort the array of 500 integers in 0.111964 seconds. The bubble sort code below was originally pulled from

JavaTPoint

https://www.javatpoint.com/bubble-sort-in-python

![Text

Description automatically generated]()

Merge-Sort

Merge Sort is the second menu option present in the program menu and sorts data based off of the divide and conquer idea which divides an array in halves in order to rearrange the integers. The average case regarding runtime of merge sort is O(n log n) which is faster than many other sorting algorithms. In our program, merge sort was able to sort the array of 500 integers in 0.004192 seconds on average and places the sort at a close 3rd place in the algorithm comparison. The function code which has been included in the project has been pulled from

Medium.com

https://medium.com/@george.seif94/a-tour-of-the-top-5-sorting-algorithms-with-python-code-43ea9aa02889

![Text

Description automatically generated]()

Quick-Sort –

Quick Sort is the third option in the user interface and uses a divide an conquer method which compares the ends of an array and switches the integers if they are in the wrong spots in relation to each other. Quick sort has a best and average case time complexity of O (n log n) and a worst time complexity of O(n^2). Regarding the sorting of the 500 integer list, quick sort was able to accomplish the task in 0.00409 seconds which places this algorithm in 2nd place in relation to the other options. The function code below which is present in the program has been pulled from the wiki at

Brilliant.org

https://brilliant.org/wiki/quick-sort/

![Text

Description automatically generated]()

Insertion Sort

Insertion Sort was chosen as the final fourth option of the user interactive menu and treats the array as a list that it is constantly rearranging. This sorting algorithm starts at the beginning of the array and steps through it while moving the newly found minimum to the beginning of the array. Insertion sort has a best and wort case time complexity of O(n^2). However, during the sorting of the text array of 500 integers, insertion sort was able to complete the task in 0.000938 seconds on average which put this sort in 1st place among the other algorithms. The insertion sort code used in the program is pictured below and was pulled from the wiki at

Brilliant.org

https://brilliant.org/wiki/insertion/

![Text

Description automatically generated]()

**DESIGN**

The design of the program comparing the sort algorithms will be straightforward in order to quickly display and understand the complexity of each of the four algorithms with regard to the same array data across the board. The user will be able to view a menu when first opening the program to choose which sorting algorithm they would like to use and will then be able to review the time it takes to sort the 500 items from the referenced text file through program output. Due to the structure of the program, the user is able to change or edit the text file which contains the array of integers in order to add diversity to the trials when comparing the algorithms. Regarding the collection of output data, 10 trials which each sorting algorithm were recorded to provide an accurate average sort time per algorithm. This data has been arranged in the conclusion section of the report with the results.

**Implementation and Testing**

|  |  |  |  |
| --- | --- | --- | --- |
| Bubble Sort (seconds) | Merge Sort (seconds) | Quick Sort (seconds) | Insertion Sort (seconds) |
| 1. 0.14249515533447266 | 1.  0.00524449348449707 | 1.  0.007178306579589844 | 1.  0.0009970664978027344 |
| 2.  0.1109926700592041 | 2.  0.005146503448486328 | 2.  0.006133556365966797 | 2.  0.000997304916381836 |
| 3.  0. 10541987419128418 | 3.  0.0031371116638183594 | 3.  0.005490541458129883 | 3.  0.001024484634399414 |
| 4.  0.12854552268981934 | 4.  0.003072500228881836 | 4.  0.0009970664978027344 | 4.  0.0003457069396972656 |
| 5.  0. 11135983467102051 | 5.  0.0038454532623291016 | 5.  0.001993417739868164 | 5.  0.0009984970092773438 |
| 6.  0.10294389724731445 | 6.  0.003711700439453125 | 6.  0.0027523040771484375 | 6.  0.000997304916381836 |
| 7.  0.10027074813842773 | 7.  0.0046100616455078125 | 7.  0.00482940673828125 | 7.  0.0010271072387695312 |
| 8.  0.10833001136779785 | 8.  0.0023534297943115234 | 8.  0.004335641860961914 | 8.  0.0009984970092773438 |
| 9.  0.1060645580291748 | 9.  0.0054302215576171875 | 9.  0.003045797348022461 | 9.  0.0009949207305908203 |
| 10.  0.10322237014770508 | 10.  0.00536799430847168 | 10.  0.0041484832763671875 | 10.  0.0009975433349609375 |
| Average  0.111964 | Average  0.004192 | Average  0.00409 | Average  0.000938 |

During the implementation and testing portion of the project, there was a lot of trial and error regarding the creation of the sorting algorithms. Specifically, Chandler initially chose using a small array of 5 numbers to sort before moving to an array of 500 numbers which is significant enough for the timer to compare the sorts. Originally, a random array generator was used at the beginning of the program with the importation of the numpy python library to create the array. However, after realizing that the generator would create a different array every time the code was ran, it was decided that a text file with a pre-randomized set of numbers would be more beneficial to the comparison process. This way, no matter which sorting algorithm the user decided to use, the same set of numbers would be sorted to compare more efficiently. The debugging process involved editing various segments of code until the arrays were sorted successfully. Also, during testing, the Unsorted and Sorted versions of the arrays were outputted to the console to be sure that all of the sorting algorithms were working properly before focusing on the time recording portion of the project.

Overall, the results of the program ranked the algorithms from fastest to slowest with Insertion Sort being 1st, Quick Sort being 2nd, Merge Sort being 3rd, and Bubble Sort being 4th. When reflecting back on the project now that it has been completed, personally I do believe that this would have been more easily accomplished in a group rather than on the individual level. However, I believe I learned a great deal about the python language and how the sorting algorithms work in relation to each other. I believe that this was a great project prompt and covered the topics well in preparation for our future in the computer science field.