November 27, 2021

YUNG, Pak Hong Patrick

Computer engineer department

CLZ2103

Comparison of Sorting Algorithms

Final Year Project

**Appendix**

**Page**

**1.0 Introduction:**

1.1 Motivation 3

1.2 Background Research 4

**2.0 Fair Testing:**

2.1 Software and Hardware Control Factors 5

2.2 Environment Control Factors 6

2.3 Uncontrollable Factors 6

2.4 Data Collection Factors 7

**3.0 Merge Sort:**

3.1 Merge Sort Background information 7

3.2 Merge Sort Versions 8

3.2.1 Top-Down and Bottom up Merge Sort 8

3.2.2 InsertionMerge Sort 9

3.2.3 Tim-Sort Merge Sort B 10

3.2.4 Bitonic Merge Sort 10

**4.0 Merge Sort Variant Methodology**

4.1Methodology 11

4.2 Design type for Merge Sort 13

**5.0 Merge Sort Data Collection**

5.1 Testing Result 14

5.2 Data Analysis for Merge Sort 15

5.3 Size of the Algorithm and Space Complexity 16

**6.0 Evaluation for Merge Sort**

6.1 Evaluation on Top-Down and Bottom Up Merge Sort 17

6.2 Evaluation Insertion Merge Sort/Tim Sort 18

6.3 Evaluation on Tim-Sort B 19

6.3.1 Extension on Tim-Sort B 20

6.4 Evaluation of Bitonic Sort 20

**7.0 Merge Sort Conclusion**

7.1 Overall Result 21

7.2 Best Merge Sort for C++ 22

7.3 Best Merge Sort Variant in Python 23

7.4 Conclusion 24

**8.0 Reference** 25

**9.0 Appendix** 27

**1.0 Introduction**

**1.1 Motivation**

           Since the invention of computers, different computer algorithm has been designed by computer scientists to solve a large quality or perform complex calculation and operations. For example, simple addition and subtraction for large quality are difficult for humans but could be coded into algorithms for the computer to process. However, not all algorithms have equal performance, despite accomplishing the same operations. Hence, computer science needs to investigate and design more efficient algorithms, that allow computers and systems to achieve more complex calculations more efficiently.

           Similar to designing a product, computer scientist needs to consider the following aspect when designing or improving an algorithm to the best of their ability.

• Design problem

• Algorithm’s

* + Data used
  + Type of language used to code

• Running time constancy

• Time complex/Running time: The number of computational complexities an algorithm requires to run and finish.

* + Best and worst-case
  + Average case

• Computer/system/hardware intend to use from

• Space Complexity

• Memory Complexity

Creating or improving an algorithm is a difficult process and requires innovative visualization or concept in approaching the design problem. Any slight changes may improve certain aspects, but potentially weaker in others. Hence, the best algorithms to solve an issue/problem is case-dependent and not always measured by their efficiency, but could be compared in real-life application against each other.

           For this investigation, I’m motivated to learn the different aspects a computer scientist needs to consider in designing an algorithm. A majorly of algorithms have similar or same time complexity in achieving the same task but does not make the algorithms have equal performance. Hence, each algorithm needs to investigate and compare details under real-life situations. Ultimately, we wish to compare the strength of different algorithms and attempt to merge certain aspects/concepts from another algorithm to build a better version.

**1.2 Background research**

One of the most discussed issues is the quickest method to sort an array of elements or integers in ascending or descending order. Sorting numbers in ascending order is simple, but in the fields of mathematics, sciences, or technology the simpler operation holds great importance. For example, sorting an array of integers allows us to organize files, analyze patterns, data spread, etc. A real-life situation could be as simple as finding a book in a library.

Before computers, most librarians and humans would sort elements by comparing each element with each other, then placing the elements at the correct location. This method is known as bubble sort, which compares each element with all others. However, this method requires n2 (n being the number of elements) number of comparisons and is difficult for humans to physically remember and sort. Nowadays, different computer scientist has designed different types and version of sorting algorithm as shown in figure 1 and table 1.1, with less number of comparison.Diagram

Description automatically generated

**Table 1.1 Commonly Used Sorting Algorithm with Time Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Time Complexity** | | |
| **Sorting Algorithms** | **Best Case** | **Average Case** | **Worst Case** |
| **Bubble Sort** | O(n) | O(n2) | O(n2) |
| **Selection Sort** | O(n2) | O(n2) | O(n2) |
| **Heap Sort** | O(nlog(n)) | O(nlog(n)) | O(nlog(n)) |
| **Merge Sort** | O(nlog(n)) | O(nlog(n)) | O(nlog(n)) |
| **Quick Sort** | O(nlog(n)) | O(nlog(n)) | O(n2) |
| **Insertion Sort** | O(n) | O(n2) | O(n2) |

Each sorting algorithm is capable of sorting an array of elements in ascending order, and each sorting algorithm consists of different alternative versions. For this investigation, I would investigate merge sort and its variants to determine the most efficient sorting algorithm.

**2.0 Fair testing**

           To ensure each sorting algorithm is tested and investigated fairly, certain factors and resources are controlled evaluated under real-life situations as closely as possible. The running time of an algorithm may vary due to differences in hardware, software, or environment control factors, and should be tightly controlled and minimized for this investigation. Hardware and software factors focus refer to the development of the algorithm and physical device used. Environment factors refer to the testing environment, equipment, or additional algorithm requirement, and each factor should be minimized to as best of my ability.

The bellow rules are applied in all algorithm’s methodology, code used and testing environment to ensure fair testing.

**2.1 Software and Hardware control factor**

• All coding and testing will be conducted and limited to the website. The sorting algorithm needs to be best suitable to be used in a different online platform to test.

•       [repit.com](http://repit.com) has over 50 languages and is trusted by Google, Facebook, stripe, etc.

• The version used would be 2021 version of [repit.com](http://repit.com)

• All algorithms are written by Yung Pak Hong Patrick. (See Appendix A for all algorithms used)

• C++ and Python languages would be used for this investigation.

• Besides time-related and sorting algorithm required module, no additional code or module would be used in the algorithm.

**2.2 Environment control factor**

• After each testing, all algorithm is required to print out the sorted algorithm to ensure successful testing.

• Time is measured only at the merge sort algorithm in a nanosecond.

• Each algorithm

* + Needs to be written in two languages
  + 1000 runs are required to determine the average time taken to sort an array
  + 1000 integers are used in the array must range between -1000 to 1000

           However, certain aspects in the testing environment are uncontrolled and an attempt to reduce the impact on testing results or assumptions would be made in regards to the issue. For example, the length and structure of code algorithms in different languages would affect the running time and result in certain languages having shorter running times for the specific sorting algorithm. Hence, two different languages(C++[ubtuntu0.18.04.1] and python 3.8.2 ) would be implanted and compared separately. Other factors of assumption or uncontrollable factors are listed below.

**2.3 Uncontrollable factors**

* Process ability of each languages are considered equally as efficiency as each other. (create temporary space, length, reading/writing/access array etc)
* Time module imported into the algorithm are accurate.
* Algorithm written by Yung Pak Hong Patrick are consider the most efficient method possible.

**2.4 Data Collection Factors**

* For this investigation, 10 test result(each contain the average running time for 100 runs) would be written down for each algorithm testing, to discover each sorting algorithm has the shortest running time.
* Variance would be calculated with testing results to determine the constancy of the algorithm.
* After each sorting algorithm, the user requires to print the result in the console to confirm its successful sorting algorithm.
* Please refer to appendix B for the data set used in this investigation its desire sorted outcome.

**3.0 Merge Sort**

**3.1 Merge Sort Background information:**

          Merge Sort is a divide and conquer sorting algorithm, and has a time complexity of O(nlog(n)) time. The core of merge sort focuses on dividing the unsorted array into sub-arrays until array size is less than 2 elements, often dividing the array into two halves(left array and right array). Afterward, merge sort compares the smallest integer in each array, and input it back to the original array. The process repeats until the array is sorted. The merge sort algorithm was invented by John Von Neumann in 1945. For a simple visual demonstration, please refer to appendix 1.

**Advantages of merge sort:**

• Given best, worst and advantage time complexly of merge sort being O(nlog(n)) time, the constancy makes the algorithm very efficient at dealing with random sorted data.

• Running time and constancy of merge sort would not be greatly affected by the size of integer array, due to its simple design structure of merge sort, running time. Hence, sorting large-size lists would not result in significant running time variance.

**Disadvantage of merge sort:**

• Space complexity of merge sort is O(n) due to the need to create a copy of the left and right array, so additional memory space is generated.

**3.2 Merge Sort Versions**

Similar to many different sorting algorithms, there are different types of merge sort. For example, 3-way merge sort divides the array into three smaller sub-arrays rather than two, etc. Therefore, for this paper, we would investigate five variants.

* Top-down merge sort
* Bottom-up merge sort
* Bitonic merge sort
* Insertion merge sort(Tim Sort)
* Tim merge sort B

Each of the above variants has a different unique method to approach the merge sort algorithm. Investigating different versions of merge sort is crucial, as real-life data situation often includes specific patterns, distribution models, or structures, and not always in an equal random distribution. Hence, different versions of merge sort may result in different running times and are considered separately in this investigation.

Merge sort methodology is separated into two sections, one operation function, and one structure function. The structure-function decides the size, and which two sorted sub-array are intended for the main operation function merge together. The operation function is responsible to receive the positions of two sub-arrays, and by repeatedly comparing the minimum value of both arrays, sort the array back to the original array.

**3.2.1 Top-Down and Bottom up merge sort**

Top-down and bottom-up merge sort use different structure-function. As shown in Figures 2, top-down often uses a recursive function to divide the array and only returns if the array size is less than 2. Then, proceeds to merge sort with the resulting position of both sub-array. Therefore, top-down merge sort would sort the array starting left most integer of the array and continues to sort in the power of 2. On the other hand, Bottom-up merge sort instead divides the array using the “for loop” function to isolate the array(figure 3:line 3-7). The bottom-up function would pair up integer/groups the array to perform merge sort, then double the paired size for each rotation.(ie 2->4->8->16) Hence, the entire array would only be sorted after the function is completed.

A screenshot of a computer

Description automatically generated with medium confidence

Although both top-down and bottom-up merge sort has different structure functions, the number of comparisons are the same. However, the main difference is top-down merge sort uses a recursive function, but top-down “for loop” function instead.

**3.2.2 Insertion Merge Sort**

Insertion merge or Tim sort designed by L. R Ford Jr and Selmer M. Johnson, incorporates insertion and merge sort but rotates based on array sizes. Secondary research suggests that insertion sort performs better in small array sizes, but is less efficient than most sorting algorithms for larger array sizes. Hence, insertion merge sort would alternate between merge sort or insertion if the array size is less than a certain threshold. For example, if the array size is less than 20 elements, the algorithm would perform insertion sort, but 20 elements or above would perform merge sort instead. The concept of insertion merges sort hopes of incorporating the advantage of insertion sort into merge sort.

The array size to decide performing merge sort or insertion sort is debatable for different languages and systems. Insertion sort would perform 22 comparisons for array size of 10, 26 comparisons for 11 elements, and 30 comparisons for 12 elements. Hence, is the increase of elements worth an increased number of comparisons? We won’t be sure as different languages may prefer if the threshold is larger or smaller. Hence for this investigation, if an array size is seven or fewer insertion sort would be used in insertion merge sort.

Please be noted that insertion merge sort is often coded using a bottom-up structure, but for this investigation fair testing, insertion merge sort would use a top-down structure.

**3.2.3 Tim-sort merge sort B**

Tim-sort merge sort B focuses on analyzing patterns within two arrays and incorporating binary search/insertion sort into merge sort’s operation function to reduce the running time. For example, if array [A] has smaller elements, the probability of the smallest integers among the same array increases. Hence, Tim-sort B would perform binary search/insertion search on array [B]’s smallest integer on array [A], in hopes to reduce the number of comparisons. However, Tim-sort’s B weakness is dealing with equal random uniform distribution, as both sub array would likely have an equal number of elements larger and smaller than the average value.

In a real-life situation, human behavior would affect the position of data input, and not always distributed uniformly. For example, hypothetical in America 2021 presidential election, elderly votes would often process at a later date, because it is more difficult for the elder to attend voting booth and often vote by mail. Hence, sorting the American voter by age group would benefit from the Tim sort B, because the elders(older age) would more likely be the end of the array.

The required amount of integer taken from a specific array to perform a binary search is debatable for different array sizes, languages, and targets. Hence, similar to insertion merge sort 7 elements taken from one array would trigger one insertion merge sort.

**3.2.4 Bitonic Merge Sort**

Bitonic merge sort is a merge sort variance that utility monotone sequence to improve its efficiency. A monotone sequence is define when an array of integer is all in increasing or decreasing order(xn<=x(n+1) or xn>=x(n+1) for all n values). Bitonic Merge sort has a worst-case, best-case, and average case of O(log2(n)) time. Bitonic Merge Sort aims to divide the array into small sub-array, with one half of the array sorted in ascending order and the other half in descending order. Afterward, compare the beginning of each array and perform a swap if ascending array element is larger than descending order element, then the next element in both arrays are compared again. The process repeats only for half the array size, but for every sub-array.

However, Bitonic search is applicable for array size in powers of 2 to ensure the array could be divided equally into sub-array. Hence, for this investigation, I modified the Bitonic search to be appliable for all array sizes.

Depending on the size of the array, the array would undergo Bitonic merge sort under different sizes and merge using top-down merge sort. The different sizes are determined by the largest power of 2 possible. For example, for 1000 integers the array is divided into 512, 256, 128, 64, 32, and 8 that sum to 1000. Each array would undergo Bitonic merge sort and be combined using top-down merge sort together. For an array size of 1, the modified Bitonic merge sort would isolate the last element and perform an insertion sort at the end of the algorithm.

**4.0 Methodology**

**4.1 Methodology**

Below is code structure that would be used as reference for topdown, bottom up and Tim merge sort. For the code used in this section, please refer to appendix A.

**Merge sort:**

1. Create copies of both left and right array
2. Compare the smallest integer between the left array and right array. Repeat until either one array is empty

**Tim Sort B:**

* If over 7 integers are taken from one array, perform insertion search on the other array smallest integer on the other array.
* Copy all integer until for result from the insertion search.

1. Copy any remainders integers from either left or right array
2. Return the array

**Top Down:**

1. Divide the array into two halves(left array and right array), repeat step one on both left array and right array until array size is less than 2.
2. Use methodology for merge sort on left and right array to sort array, repeat step 2 until all array is sorted

**Bottom Up:**

1. Create a ‘for’ loop that uses group 2 integer and perform merge sort. Repeat step 1 for all integer.
2. Repeat step 1, but instead double the group size. Repeat step 2, until group size is equal to the array size.

**Insertion Merge Sort:**

1. Divide the array into two halves(left array and right array), repeat step one on both left array
2. If the array size is less than 8 elements, perform insertion sort, else perform merge sort. Repeat step 2 until array is sorted.

**Bitonic Merge Sort:**

1. Check the array size is power of two (2, 4, 8, 16, 32 ……)

2.1: If the array size is power of two divide the array into equal array size in power of two

* 1. Have alternate array be sorted in ascending and descending order
  2. Compare an ascending and descending array first integer with each other, if ascending is smaller than descending, perform swap
  3. Repeat step 2.1.c for array size
  4. Repeat step 2.2.b for all array until all element is sorted

2.2: If array size is NOT a power of two, divide the array into different section each with array size of power of two.

1. Isolate/create a copy with unsorted integer with array size equal to maximum power of 2 from original.
2. Have alternate array be sorted in ascending and descending order
3. Compare an ascending and descending array first integer with each other, if ascending is smaller than descending, perform swap
4. Repeat step 2.2.c for array size
5. Repeat step 2.2.b for all array until all element is sorted
6. Repeat step 1 for the remaining unsorted array
7. 3If array size is odd, perform insertion search/sort on the last element.

**4.2 Designed data type for Merge Sort**

**Best Sorted Data Type:** An already sorted array with 1000 integer from 1 to 1000.

**Worst Sorted Data Type:** Although there are difference merge sort variants, a majorly of merge sort core concept compares two sorted array and until one array becomes empty. Hence, the worst sorted data type would require comparing and switches with all the element within both sorted arrays. Therefore, for this investigation I designed to the worst data type is alternating elements of a sorted array. For example as shown below:

Sorted Array = {1,2,3,4,5,6…….}

Divided Array:

{1,3,5,7,9….}[2n+1](odd elements) | {2,4,6,8,…..}[2n] (even elements)

Divided divided Array:

{1,5,9, 13 …} [4n+1](odd alternatively) | {3, 7, 11 , ….} (odd alternatively) |

{2, 6 ,10, 14….} (Even Alternatively) | {4, 8 ,12 , 16}(Even Alternatively)|

And so on.

………

Please see appendix B for the complete list for worst data set and appendix 2 of the method to obtain the worst data set.

**Random Data Set:** Random data set 1,2 and 3 is generated from random.org. Each integer could occur more than once. Each random data set is calculated for its mean and variance and yield the following result:

Random Data Set 1: Mean: -2.089, Variance:335470.323

* Average number of increasing monotone sequence length: 340
* Average length of increasing monotone sequence length: 1.46
* Average number of decreasing monotone sequence length: 502
* Average length of decreasing monotone sequence length: 1.48

Random Data Set 2: Mean: -0.696, Variance: 339118.878

* Average number of increasing monotone sequence length: 325
* Average length of increasing monotone sequence length: 1.48
* Average number of decreasing monotone sequence length: 519
* Average length of decreasing monotone sequence length: 1.59

Random Data Set 3: Mean: 1.875, Variance: 344288.697

* Average number of increasing monotone sequence length: 336
* Average length of increasing monotone sequence length: 1.48
* Average number of decreasing monotone sequence length: 502
* Average length of decreasing monotone sequence length: 1.49

**5.0 Merge Sort Data Collection**

**5.1 Testing Result**

Below is a simplified version of the data collected, please refer to appendix C for a more detailed version. Table 5.3 include only the random data set average running time and standard deviation.

**Table 5.1: Merge Sort variant Average Running Time in C++**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Merge Sort**  **(C++)** | **Average Running Time (Nano Seconds)** | | | | |
| **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Top-Down** | 396,036.30 | 573,398.40 | 485,352.10 | 496,020.30 | 467,409.70 |
| **Bottom Up** | 389,000.50 | 500,909.20 | 485,637.10 | 484,447.40 | 473,189.50 |
| **Tim Sort B** | 577,440.20 | 591,779.80 | 677,613.40 | 721,542.70 | 718,482.40 |
| **Insertion** | 75,420.20 | 803,308.60 | 758,705.50 | 765,666.40 | 751,007.70 |
| **Bitonic** | 229,974.20 | 267,670.50 | 319,712.70 | 314,913.90 | 314,289.30 |

**Table 5.2: Merge Sort variant Average Running Time in Python**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Merge Sort**  **(Python)** | **Average Running Time (Nano Seconds)** | | | | |
| **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Top-Down** | 4,723,775.40 | 13,353,773.91 | 7,584,893.40 | 6,938,214.10 | 6,653,348.80 |
| **Bottom Up** | 6,294,793.30 | 10,386,895.10 | 7,069,590.00 | 8,790,463.40 | 9,935,175.05 |
| **Tim Sort B** | 7,722,137.20 | 25,198,639.39 | 9,407,987.00 | 10,886,018.50 | 11,535,839.10 |
| **Insertion** | 288,909.36 | 15,401,141.96 | 9,491,590.65 | 11,635,734.60 | 12,100,085.00 |
| **Bitonic** | 19,890,773.95 | 22,372,882.30 | 21,046,949.10 | 21,284,967.70 | 20,420,681.90 |

**Table 5.3: Merge Sort variant Average Running Time in Random Data Set**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Merge Sort** | **Running Time (Nano Seconds)** | | | |
| **Average**  **(C++)** | **Average**  **(Python)** | **Standard Deviation (C++)** | **Standard Deviation (Python)** |
| **Top-Down** | 482,927.37 | 7,058,818.77 | 53,343.72 | 925,349.09 |
| **Bottom Up** | 481,091.33 | 8,598,409.48 | 27,712.75 | 2,129,892.15 |
| **Tim Sort B** | 705,879.50 | 10,609,948.20 | 58,834.37 | 7,501,835.77 |
| **Insertion** | 758,459.87 | 11,064,556.98 | 32,735.78 | 2,769,228.35 |
| **Bitonic** | 317,313.30 | 20,917,532.90 | 15,850.44 | 1,861,932.45 |

**5.2 Data Analysis for Merge sort**

From data collection, the shortest average running time in C++ for random data set is Bitonic Merge Sort with the smallest standard deviation in C++. However, the Top-down has the shortest running time and standard deviation in Python. Below table is the order of the shortest to longest running time for each merge sort variant in random data set.

**Table 5.4: Ranking of each Merge Sort Based on Average Running Time**

|  |  |  |  |
| --- | --- | --- | --- |
| **Shortest Running Time** | **C++** | **Python** | |
| **First** | Bitonic Sort | | Top-Down Merge Sort |
| **Second** | Bottom-Up Merge Sort | Bottom-Up Merge Sort | |
| **Third** | Top-Down Merge Sort | Tim Merge Sort B | |
| **Fourth** | Tim-Sort Merge Sort B | Insertion Merge Sort | |
| **Fifth** | Insertion Merge Sort | Bitonic Sort | |

Similar designed or concept such as top-down and bottom-up or Tim-Sort and Insertion merge sort have similar running time. Hence are required to be analysis more detail under the consideration of best and worst data set.

The below graph is under consideration that distribution of merge sort running time perform similar or follow to a normal distribution.

**Graph 5.1:Normal Distribution for Merge Sort Variant in C++**

Chart

Description automatically generated

Top-Down

Bottom-Up

Tim-Sort B

Insertion-Sort

Bitonic-Sort

Chart, line chart

Description automatically generated**Graph 5.2:Normal Distribution for Merge Sort Variant in Python**

Top-Down

Bottom-Up

Tim-Sort B

Insertion-Sort

Bitonic-Sort

**5.3 Size of Sorting Algorithm and Space Complexity**

Space complexity is the memory required for the algorithm to be free or used. Although today's computer, it's a less important factor in today's algorithm due to advancement in technology, it's a more significant factor in older computers or where memory space is limited. For example, most merge sort variant requires a space complexity of O(n) time, but Bitonic Sort has a space complexity of O(n) or 1.

Meanwhile, memory size is the space required to store the algorithm itself. Often, complex sorting algorithms require additional system memory space to store. Below demonstrate each merge sort variant memory size(excluding the comments) and its space complexity.

**Table 5.5 Number of times Space Complexity and Memory size for Merge Sort**

|  |  |  |  |
| --- | --- | --- | --- |
| **Merge Sort Variants** | **Space Complexity** | **Memory of algorithm** | |
| C++ | Python |
| Top-Down Merge Sort | O(n) | 1.29KB | 1.03KB |
| Bottom Up Merge Sort | O(n) | 1.21KB | 1.01KB |
| Tim Merge Sort B | O(n) | 2.54KB | 1.29KB |
| Insertion Merge Sort | O(n) | 1.12KB | 1.01KB |
| Bitonic Merge Sort | O(nlog2(n)) | 1.67KB | 1.41KB |

**6.0 Evaluation For Merge Sort**

**6.1 Evaluation on Top-Down and Bottom Up Merge Sort**

From data collection, Top-down Merge Sort has the third shortest running time but is most consistent in Python. While, bottom-up Merge Sort has a short running time, but has second-most consistent in C++.

Both versions of merge sort have a similar amount of comparison within all data set, but the illustration in bottom-up is shorter for both languages. As shown in figure 2:line 3-4, the Top-down merge sort has an additional “if” function to ensure array size is larger than 2 before returning but would increase the overall running time for each rotation/branch. However, although the additional illustration would create a small impact on a small-scale set of data, the impact would be more significant in a larger set of data.

On the other hand, the bottom-up merge sort has additional operations to determine if the array is either odd or even. In figure 3:line 5 a “min” function is responsible to determine the lowest value between the endpoint of the array and separating the original array into two subarrays. This ensures each integer is involved within an odd array size. However, operation running time may vary in different computer systems and create less constancy in running time, thus resulting in a higher standard deviation compared to top-down merge sort.

Research conducted by Arthur Kay on comparison between top-down and bottom-up merge sort has yielded similar results to this investigation. Through Kay’s experiments, the bottom-up merge sort has a shorter running time compared to the top-down merge sort, due to the multiple uses of recursive function leading to computing overhead. Computing overhead refers to the delay caused by calling a recursive function multiple times. Each recursive function requires the computer to record the current status, registers, and address onto a stack until a return is called. Extensive use of recursive function causes a large delay.

**6.2 Evaluation** **Insertion Merge Sort/Tim Sort**

Insertion Merge Sort competes with Tim-Sort B in terms of average running time for random data set but has the fourth largest standard deviation amongst all other variants.

Insertion Merge sort perform differently in C++ and Python. For example, in the worst data set, this variant holds the second longest-running time in Python, and the longest-running time in C++. Whilst, having also having shortest running time for best-case data set among the all merge sort variants, with a significant 38 times faster than random data set average running time in Python and 10 times faster in C++. If taken into consideration the average running time for worst and best data, insertion merge sort may hold the shortest average running time between the other variants.

As previously mentioned in 3.2.2, secondary research suggests insertion merge sort strength lies within its ability to include benefits of insertion sort for the small array. Avoiding certain parts of the merge sort worst data set to a certain degree, but the constant requirement to check array size for rotation results in becoming a less efficient sorting algorithm in a top-down structure. The only benefit is apparent only in C++ for worst data set, with its running time similar to its random data set(difference of 50,000ns).

As mentioned in 3.2.2, insertion merge sort is often written using a bottom-up structure to prevent repeat checking the requirement for array size, making it a more efficient and consistent algorithm. I have performance additional 100 runs using a bottom-up structure for insertion merge sort, and the average running time is around 40,000 ns. Making it potentially the second-best merge sort variant.

**6.3 Evaluation on Tim-Sort B**

Tim-sort B has the fifth longest-running time and highest standard deviation time in all data sets (table 5.3). The result in data collection emphasize multiple weakness, and difficulty the algorithm encountered in each data set.

One of the weaknesses of Tim-sort B includes the insertion search within the Tim-sort B condition being difficult to achieve in random data set. Without the condition for Tim-sort fulfilled, Tim-sort would become a normal bottom-up merge sort with additional useless code. For example, random data set from Tables 5.1 and 5.2 are often 15% longer than the best data set running time, as both best and worst data grantees activation of Tim-sort. To confirm this theory, additional testing on the number of times Tim-sort’s condition is fulfilled has been conducted and shown in table 5.4(condition 7 consecutive).

**Table 6.1 Number of times Tim-Sort B Condition is Achieved**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Number of Rotation Tim-sort Condition Fulfilled** | | | | | |
|  | **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Tim-Sort** | 564 | 0 | 266 | 316 | 390 |

The testing result from table 6.1 has indicated the random distribution of 1000 integers has only met the Tim-sort B requirement between 250 to 400 times, but the worst-case doesn’t even meet the requirement once. Even if trigged in best case data set, Tim-Sort B is still inferior to other variants. In addition, conditional for Tim sort B is checked and reset if the condition is not met per rotation, so Tim-sort B operation function has an additional comparison than other merge sort variants. A solution is increasing the probability of Tim-sort B being triggered, by a reduction in the number of integers require to initiate insertion sort would increase the probability in random data set.

Moreover, random data set used for this investigation starts with an average monotone both positive and negative sequence has a length of 1.48 to 1.59 shown in 4.2. Meaning the probability of meeting the requirement for Tim-Sort B at early rotation are unlikely. As previously mention in 3.2.3 Tim-Sort B benefit from a bias type data set, in which monotone sequence average length are longer with low number of average number of monotone sequence.

On the other hand, the effectiveness of insertion sort in Tim-sort B isn’t as effective for small and random data set. One insertion sort takes O(log(n)) time per search to achieve a lower amount of comparison than top-down or bottom-up merge sort, but sometimes would result from an opposite effect. For example, insertion sort may require 10 comparisons to end its search, but the top-down or bottom-up search may use only 5 comparisons to achieve the same effect. Therefore, small and random distribution data set reduces the probability for large comparison reduction, making a majorly of the effectiveness of Tim sort B equal or less than top-down and bottom-up merge sort.

Overall, the benefits of Tim-sort aren’t always achieving and beneficial to the user.

**6.3.1 Extension for Tim-Sort B**

For this investigation, the largest array to perform merge sort is two sub array with 500 elements each, giving less than or equal to 32.3% to perform insertion sort for each rotation. The probability only increases with a larger data set size, so the requirement to trigger Tim-Sort B should be interchangeable to maximize the efficiency of the sorting algorithm. For example, having a smaller requirement for Tim-Sort B for smaller array size, but larger requirement for larger array size would be more beneficial.

**6.4 Bitonic Sort**

In this investigation, Bitonic sort has the shortest running time in C++, but the longest in Python. Its standard deviation is considerably low in both languages, having the shortest in C++ and longest in Python. Investigating, the best and worst-case yield different results, such as worst-case data being faster than random data set in C++, but the opposite in Python. Overall, considering the average running time in table 5.3, Biotitic sort could be twice as fast as Insertion Merge sort(longest running time) in C++, or three times longer than Top-down merge sort(shortest running time).

As mentioned above, Bitonic sort worst-case data set average running time in C++ is faster than random set is occurred due to several reasons. Firstly, the worst-case data set was designed to maximize the number of comparisons and switches between two sorted arrays in ascending order. However, Bitonic sort-merge sorted one array in ascending order and another in descending order. Hence, the general worst-case data set for merge sort is not appliable for Bitonic sort, instead, the worst running time should also include alternating reverse order to achieve the worst running time possible.

Meanwhile, random data set used in this investigation is close to best case data set for Bitonic Sort. A unique feature for Bitonic best case data set is having the same average length and number of both positive and negative monotone sequence (ie average length=1, number=500). Switches are unavoidable unlike other variant, because its require to merge sort two sub array with different monotone sequence directions. Hence, having a balance positive and negative monotone sequence would provide the first rotation to be already sorted. (i.e 1,2,4 ,3, 5, 6, 7, 8……)

On the other hand, Bitonic sort has a unique requirement to be able to perform on array size in the power of 2. To achieve this requirement, the array is divided into smaller sections that sum to the array size of 1000(512 +

256 + 128 + 64 + 32 + 8), and require to import a math module to use the log2 and power 2 functions to divide the array as large as possible, then perform merge sort together. However, the additional calculation and merge sort function would increase the overall complexity of the function and several issues. For example, Bitonic sort divides and transverse the array into smaller equally size array to perform bitonic sort and then merge sort the different array size. However, each sub-array(512, 256, 128, 64, 32, 8) needs to transverse and divide the array, increasing the number of comparisons overall.

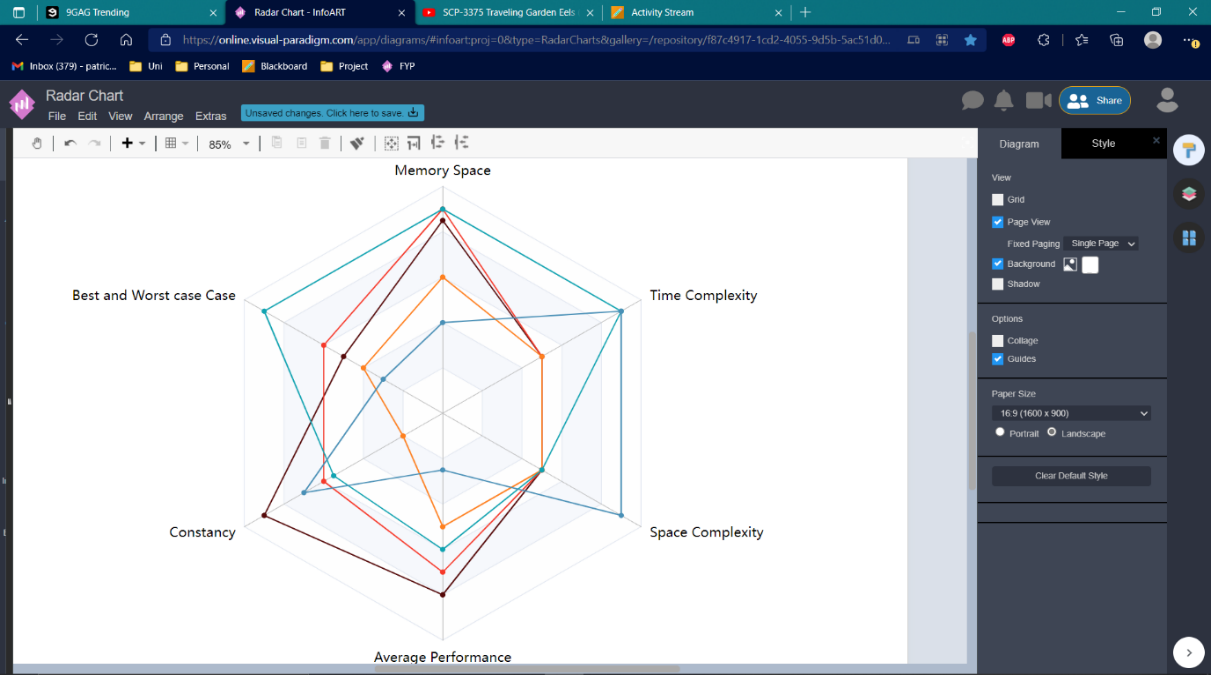
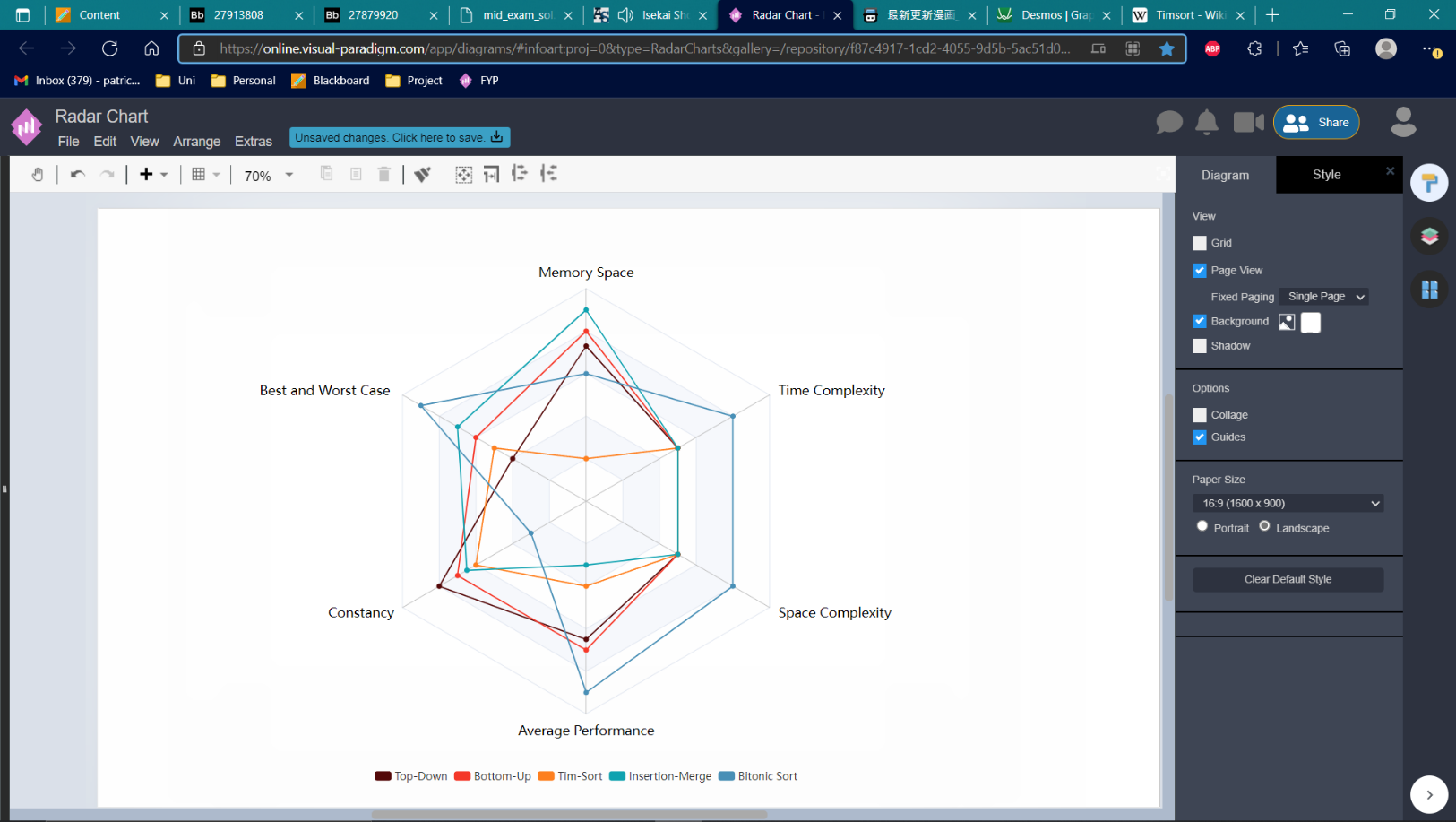
Therefore, it’s possible for array size in the power of 2 would have a shorter running time, such as having 1024 elements in an array rather than 1000 elements would have a shorter running time. Using this concept, an alternative method is adding "n" number of “int” elements each with value “-232”(largest negative int value), with “n” being the number of elements required for array size to become a power of two, then removing any value containing -232. In hopes, it reduces the running time by removing the merge sort part.

**7.0 Merge Sort Conclusion**

* 1. **Overall Result**
  2. To determine the most efficient the following aspect of sorting algorithms would be considered for each variant. However, each of the aspects is not equally as valued with today’s standards in developing an algorithm, and more aspects or changes in value may occur in the future.
* Time Complexity
* Memory Space
* Space Complexity
* Average performance for 1000 integers
* Constancy
* Best and Worst case difference((Best+Worst)/2 – average running time)

Below spider charts rank each of the above aspect from 1 to 100, with 1 being rated the lowest and 100 ranked as the best.

**Figure 4: Analysis of Merge Sort in C++ Figure 5: Analysis of Merge Sort in Python**



**7.2 C++ Best Merge Sort**

From the data collection, testing, and secondary research under the considered factors in 7.1, the most efficient merge sort sorting algorithm is bitonic Sort, followed by bottom-up merge sort and top-down merge sort. As mentioned in 3.1, merge sort could be divided into the structure, operation part and could be interchange and additional features could be added. Hence, I believe bottom up-insertion bitonic merge sort is the most efficient for C++, given the total memory space factor isn’t significate.

Bottom up insertion Bitonic Sort holds the following advantages and disadvantages

**Advantage:**

* Bottom-up structure reduce the overhead running time
* Bitonic sort has a space complexity of O(n) or O(1) based on size
* Both Bitonic sort and bottom up Sort has the shortest running time.
* Insertion sort prevents the impact of worst data type to a certain extend
* Insertion Sort is efficient in small array size
* Insertion Sort has high performance in bottom up structure

**Disadvantage:**

* Requires an additional operation to divide the array size of be a power of 2 for bitonic sort
  + Running time may vary based on array size, making it a high variance sorting algorithm for different sizes.
* Involves multiple operation, hence more likely to cause error
* Large memory size

**7.3 Python Best Merge Sort**

From this investigation, the top three efficient merge sort variant in python is Top-down, bottom-up, and Tim Sort version B. However, unlike in programs in C++, evident from table 5.2 point towards any additional features or operation changes would increase the running time. Top-down Tim-Sort B would include the instability of Tim-Sort B(high variant) and reduce the overall efficiency. Hence, the simplest top-down merge sort is the most efficient merge sort algorithm in Python. However, the bottom-up merge sort has a similar average running time than Top-down with only 1539590ns slower. There is evidence in 6.1 and secondary research that top-down merge sort would create overhead issues in sorting large amounts of elements. Hence, an consideration is deciding to use top-down merge sort or bottom-up merge sort based on array size.

Top-down MergeSort holds the following advantages and disadvantages

**Advantage:**

* Top-down structure is simple
* Order is sorted by left more element to right, hence could be interrupt mid program and part of the array would be still be sorted

**Disadvantage:**

* May create large amount of overhead, may consider switching to bottom up if array size passes a certain thresholds

**7.4 Conclusion**

In conclusion, python and C++ each have their own unique merge sort variant that performances the most efficiently, but as mentioned in 2.3 there are other factors impacts that may be under assumption or case dependent factors. From a time complexity perspective, each variant has the same performance, but with current real-life performance and issues with the algorithm. Hence, I believe the bottom-up insertion biontic sort should be the most efficient in the general case.

Under the assumption the time module used for Python and C++ are accurate, python performs poorly in general in comparison with C++, often having three to ten times longer running time. Suggesting, python isn’t compatible with complex calculations used for this investigation. However, I believe that future technology development would reduce and improve complex operations in general, like C++, python, and java program versions are constancy improved, new operation, sorting methods, etc. Each language excels in a different area, and some program requires it to be written in a specific language. Thus, although I believe bottom-up insertion bitonic sort has the best performance in the general case, each language should use its respective variant to sort the array. Alternatively, some languages contain modules that are capable of sorting arrays in a single line of code.

**8.0 Reference**

1. Arthur Kay(April 13, 2012), JavaScript Mergesort: Top-Down vs Bottom-Up. Receive from <https://www.akawebdesign.com/2012/04/13/javascript-mergesort-top-down-vs-bottom-up/> on July 25th.
2. Eamon Nerbonne(May 21, 2018), Why top down merge sort is popular for learning, while most libraries use bottom up?. Receive from <https://cs.stackexchange.com/questions/75216/why-top-down-merge-sort-is-popular-for-learning-while-most-libraries-use-bottom> on July 27th.
3. Dr Mads Haadr (October 2021), Random Integer Set Generator. Receive from [https://www.\operatorname{random}.org/integer-sets/ on](https://www.\operatorname%7brandom%7d.org/integer-sets/%20on) July 28th.
4. Mohd Arsalan(14 Jun, 2020), Merge Sort vs. Insertion Sort. Receive from [Merge Sort vs. Insertion Sort - GeeksforGeeks](https://www.geeksforgeeks.org/merge-sort-vs-insertion-sort/) on November 16
5. GeeksforGeeks (15 Nov, 2020), Time Complexities of all Sorting Algorithms. Receive from https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/ on November 16

**Appendix 1 Merge Sort Example**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | | | | | 8 | | | 15 | | | | | 10 | | | 4 | | | | | 16 | | | | | 5 | | | | | | |
|  | | |  | | | |  | | | |  | | | |  | |  | | | | | |  | | | | | |  | | | |
| 12 | | | 8 | | | | 15 | | | | 10 | | | |  | | 4 | | | | | 16 | | | | | | 5 | | | | |
|  | | |  | | | |  | | | |  | | | |  | |  | | | | | |  | | | | | |  | | | |
| 12 | | 8 | | | |  | | | 15 | | | 10 | | |  | | 4 | | 16 | | | | | | |  | | | | | 5 | |
|  | | |  | | | |  | | | |  | | | |  | |  | | | | | |  | | | | | |  | | | |
| 12 |  | | | 8 | |  | | | 15 |  | | | | 10 |  | | 4 |  | | | | | | 16 | | |  | | | | | 5 | |
|  | | |  | | | |  | | | |  | | | |  | |  | | | | | |  | | | | | |  | | | |
| 8 | | 12 | | | |  | | | 10 | | | 15 | | |  | | 4 | | | 16 | | | | |  | | | | | 5 | | |
|  | | |  | | | |  | | | |  | | | |  | |  | | | | | |  | | | | | |  | | | |
| 8 | | | 10 | | | | 12 | | | | 15 | | | |  | | 4 | | | | | 5 | | | | | | 16 | | | | |
|  | | |  | | | |  | | | |  | | | |  | |  | | | | | |  | | | | | |  | | | |
| 4 | | | | | 5 | | | 8 | | | | | 10 | | | 12 | | | | | 15 | | | | | 16 | | | | | | |

|  |  |
| --- | --- |
|  | Unsorted Array |
|  | Sorted Array |

**Appendix 2 Merge Sort Worst Data Set**

         For the worst data set, I have used the general concept of merge sort in the designing in the design process. Merge sort core concept is comparing the smallest element in two sorted arrays until one array is empty. Hence, merge sort performs the worst when require to compare two arrays with alternating monotone sequence. As shown in the diagram on the slide. From a sorted array, alternating elements in a sub array would be even and odd intergers, afterwards, alternating the sub array would be alternating even elements, and alternating odd elements. The process repeats itself until the sub array is less than one, then the alternating elements would switches position again to create a reverse order. The final result is the worst data set for the general merge sort.

Diagram

Description automatically generated

**Appendix A**

The bellow github link contains all the code used for this paper. Each algorithm is sorted by the sorting type and language used. There are comments in each file explaining details of each part of the code. Please feel to take reference from the code.(Uncomplete, await for university arrangement for upload code)(for code please contact [patrickyyung@gmail.com](mailto:patrickyyung@gmail.com) for request)

<https://github.com/patrick-yung/Sorting-Algorithum.git>

Graphical user interface, text, application, email

Description automatically generated

**Appendix B**

The bellow table is the data set used to run all the code in Appendix A and its desire output.

|  |  |
| --- | --- |
| **Test Case** | **Sorted** |
| **Best Time Possible**  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100 | 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100 |
| **Worst Time Possible**  100, 99, 98, 97, 96, 95, 94, 93, 92, 91,90, 89, 88, 87, 86, 85, 84, 83, 82, 81,80, 79, 78, 77, 76, 75, 74, 73, 72, 71,70, 69, 68, 67, 66, 65, 64, 63, 62, 61,60, 59, 58, 57, 56, 55, 54, 53, 52, 51,50, 49, 48, 47, 46, 45, 44, 43, 42, 41,40, 39, 38, 37, 36, 35, 34, 33, 32, 31,30, 29, 28, 27, 26, 25, 24, 23, 22, 21,20, 19, 18, 17, 16, 15, 14, 13, 12, 11,10, 9, 8, 7, 6, 5, 4, 3, 2, 1 | 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100 |
| **Random 1**  463, 627, 62, 616, 997, 958, -294, -721, 178, -552, -65, -998, 209, 174, -309, 311, 242, -730, 980, -609, -540, 991, -835, -970, -252, -514, 788, 310, 423, -267, 68, -784, 55, 115, 829, 585, 764, -985, 516, -482, -915, -935, -31, 886, -547, 720, -677, -494, 351, -685, 952, 842, 629, 654, 79, 474, -64, 488, 996, -661, -446, -341, -869, 530, 831, 120, 899, 552, 840, 353, 69, -326, 389, 8, 694, -602, -42, 950, 626, -322, 494, -371, 727, 885, -132, -333, -191, 13, 277, -630, -971, 978, -77, 138, -555, -270, -107, 709, -477, 319 | -998 -985 -971 -970 -935 -915 -869 -835 -784 -730 -721 -685 -677 -661 -630 -609 -602 -555 -552 -547 -540 -514 -494 -482 -477 -446 -371 -341 -333 -326 -322 -309 -294 -270 -267 -252 -191 -132 -107 -77 -65 -64 -42 -31 8 13 55 62 68 69 79 115 120 138 174 178 209 242 277 310 311 319 351 353 389 423 463 474 488 494 516 530 552 585 616 626 627 629 654 694 709 720 727 764 788 829 831 840 842 885 886 899 950 952 958 978 980 991 996 997 |
| **Random 2**  -648, -25, -168, -716, 885, -394, -399, 834, 184, -733, -167, 355, 597, 282, 624, 147, 234, 472, 156, 5, 828, 615, 694, 491, -994, 762, 116, -495, 135, -693, 186, -270, -564, -247, 511, 1, 392, 528, 102, -628, -786, -431, -645, -60, -968, 882, 461, 845, -472, -682, 893, -449, -300, -752, -174, 14, 581, 416, 134, 575, 217, 247, -959, -575, 3, -458, 467, -413, -954, 485, 339, 245, 289, 124, 982, -607, -678, -102, 578, -578, -711, -207, -352, -199, -303, -838, 630, -953, -187, 902, -570, 852, -613, -386, -818, -79, -749, 420, -732, 360 | -994 -968 -959 -954 -953 -838 -818 -786 -752 -749 -733 -732 -716 -711 -693 -682 -678 -648 -645 -628 -613 -607 -578 -575 -570 -564 -495 -472 -458 -449 -431 -413 -399 -394 -386 -352 -303 -300 -270 -247 -207 -199 -187 -174 -168 -167 -102 -79 -60 -25 1 3 5 14 102 116 124 134 135 147 156 184 186 217 234 245 247 282 289 339 355 360 392 416 420 461 467 472 485 491 511 528 575 578 581 597 615 624 630 694 762 828 834 845 852 882 885 893 902 982 |
| **Random 3**  573, -187, -400, -356, -100, 440, 355, -116, -9, 229, -5, 969, -179, 894, -664, 205, -753, -132, 271, 441, 976, -931, -67, -247, -382, 889, 496, -487, -998, -154, -573, 864, 962, -572, -676, -534, 478, -691, -555, -386, -868, -766, -660, 197, 742, 933, -943, 887, -201, -898, 971, -607, -260, 40, -148, 353, -449, 93, 52, -548, 174, 250, 96, -174, -231, 233, 286, 70, -372, -960, 358, 22, -981, -554, -562, 558, 746, 490, -621, -822, -86, 696, -642, -93, -873, 411, 619, -926, 780, -203, 420, -553, -523, -447, -68, 786, 645, 674, -383, 448 | -998 -981 -960 -943 -931 -926 -898 -873 -868 -822 -766 -753 -691 -676 -664 -660 -642 -621 -607 -573 -572 -562 -555 -554 -553 -548 -534 -523 -487 -449 -447 -400 -386 -383 -382 -372 -356 -260 -247 -231 -203 -201 -187 -179 -174 -154 -148 -132 -116 -100 -93 -86 -68 -67 -9 -5 22 40 52 70 93 96 174 197 205 229 233 250 271 286 353 355 358 411 420 440 441 448 478 490 496 558 573 619 645 674 696 742 746 780 786 864 887 889 894 933 962 969 971 976 |

**Appendix C**

Due to the amount of data collected please refer the google sheet, link below for the full detail time for each trial run. The average for each sorting algorithm version is displayed with its variance.

<https://docs.google.com/spreadsheets/d/1cI4CexnblpUPqF-EViVGJo3WUcEG4Qs5s8_39Kc-EHc/edit?usp=sharing>