[Date]

YUNG, Pak Hong Patrick

Computer engineer department

CLZ2103

Comparison of Sorting Algorithms

Final Year Project

**Appendix**

**Introduction:**

1.1 Motivation

1.2 Background Research

1.3 Objectives

1.4 Fair Testing

1.4.1 Software and Hardware Control Factors

1.4.2 Environment Control Factors

1.4.3 Uncontrollable Factors

1.4.4 Data Collection Factors

**Merge Sort:**

2.1 Merge Sort Background information

2.2 Merge Sort Versions

2.2.1 Top-Down and Bottom up Merge Sort

2.2.2 Tim Merge Sort

2.3 Merge Sort Structure 2.4 Merge Sort Data Collection

2.4.1 Data Analysis for Merge Sort

2.5 Merge Sort Evaluation

2.5.1 Evaluation for Top-down and Bottom up Merge Sort

2.5.2 Evaluation for Insertion Merge Sort

2.5.2.1 Extension for Tim Merge Sort

2.5.3.1

**Introduction:**

**1.1 Motivation**

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

Similar in designing a product, computer scientist needs to consider the following aspect when designing or improving an algorithm to 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

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

For this investigation, I’m motivated to learn the different aspect 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 detailed under real-life situations. Ultimately, we wish to through 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 or mathematics, sciences or technology the simpler operation holds great importance. For example, sorting an array of integer allows us to organized files, analysis patterns, data spread etc. A real-life situation could be as simple as finding a book in a library.

Before computers, most librarian and human would sort elements by comparing each element with each other, then placing the elements at the correct location. This method known as bubble sort, that compares each element with all other. However, this method requires n2 (n being the number of elements) number of comparison 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) |

With each sorting algorithm capable of sorting an array of elements in ascending order, and within 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.

**1.4 Fair testing**

To ensure each sorting algorithm is tested and investigated fairly, certain factors and resources are controlled evaluated under real life situation 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 minimize for this investigation. Hardware and software factors focus refer to the development of the algorithm and physically device used on. 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.

**1.4.1 Software and Hardware control factor**

* All coding and testing will be conducted and limited to the website. Sorting algorithm needs to best suitable to be used in 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 is 2021 version of [repit.com](http://repit.com)
* All algorithm are written by Yung Pak Hong Patrick.(See Appendix A for all algorithm used)
* C++ and Python languages would be used for this investigation.
* Beside time related and sorting algorithm required module, no additional code or module would be used in the algorithm.

**1.4.2 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 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 time for the specific sorting algorithm. Hence, two different languages(C++[ubtuntu0.18.04.1] and python 3.8.2 ) would be implanted and compare separately. Other factors of assumption or uncontrollable factors are listed below.

**1.4.3 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.

**1.4.4 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.

**Merge Sort**

**2.1 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 element, often dividing the array into two halves(left array and right array). Afterward, merge sort compares the smallest integer in each array, and input back to the original array. The process repeats until the array is sorted. 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 at random sorted data.
* Running time and constancy of merge sort would not be greatly affected from the size of integer array, due to its simplicity design structure of merge sort, running time. Hence, sorting large size list 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 left and right array, so additional memory space is generated.

**2.2 Merge Sort Versions**

Similar to many different sorting algorithms, there are different types of merge sort, such as, 3-way merge sort that divides the array into three small arrays rather than two etc. Therefore, for this paper we would investigate 5 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 important, as real-life data situation often includes certain 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 should be considered as a distinctive sorting algorithm.

Merge sort is divided into two sections, the main operation function, and the structure-function. The main operation function is to receive input argument for the positions of two arrays and the original array and perform merge sort of left and right array back into the original array. The structure function decides the position, order, and size of each merge sort arrays intended for the main operation function.

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

Top-down and bottom-up merge sort use different structure-function. As shown in figure 2 and 3, top-down 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 left and right 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. 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. 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 compared to are the same. However, navigation within the top-down merge sort recursive function, or top-down “for loop” function may still cause a difference in running time.

**2.2.2** **Insertion Merge Sort**

Insertion merge or Tim sort designed by L. R Ford Jr and Selmer M. Johnson, incorporates insertion and merge sort together but rotates in between for different array sizes. Secondary research has shown that insertion sort performs better in small array sizes, but perform less efficient than most sorting algorithm in larger array size in practices. 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 merge sort hopes of incorporate the advantage of insertion sort into merge sort.

The array size to decided performing merge sort or insertion sort is debatable for different languages and system. 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 in increased number of comparisons? We won’t be sure as different language may prefer if the threshold is larger or smaller. Hence for this investigation 8 element or less elements would use for insertion merge sort.

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

**2.2.3 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, comparing the smallest integer between two sorted arrays in merge sort, if more integers are taken from array [A], 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 distribution, as random distribution would prevent Tim-sort’s insertion sort function be trigged.

In a real-life situation, specific human behavior would affect the position of data input, and not always distributed uniformly. For example, in the voting poll for the 2021 America 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, 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 and targets. Secondary research suggests 7 integers taken from the same array should be sufficient to perform a binary search.

**2.2.4 Bitonic Merge Sort**

Bitonic merge sort is a merge sort variance that utilities 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. Afterwards, comparing the beginning of each array and perform swap if ascending array element is larger than descending order element, then the next element in both array is 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 power of 2 to ensure the array could be divided equally into sub-array. Hence, for this investigation I modified Bitonic search to be appliable for all array size.

Depending on the size of the array, the array would undergo Bitonic merge sort under different sizes and merge together 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, each undergoing Bitonic merge sort and combined using top-down merge sort together. For odd size array, the modified Bitonic merge sort would isolate the last element and perform insertion sort for one element at the end. The top-down merge sort process is more likely have shorter running time, given the shortest array size requires to be sorted only.

**2.3 Merge Sort Structure:**

**2.3.1** **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)

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. If array size is odd, perform insertion search/sort on the last element.

**2.3.2 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.

**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

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

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

**2.4 Merge Sort Data Collection**

Below is a simplified version of the data collected, please refer to appendix C for a more detailed version. Table 2.3 include only the random data set average running time and standard deviation.

**Table 2.1: 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 2.2: 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 2.3: 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 |

**2.4.1 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 2.4: 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 2.1: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 2.2:Normal Distribution for Merge Sort Variant in Python**

Top-Down

Bottom-Up

Tim-Sort B

Insertion-Sort

Bitonic-Sort

**2.4.2 Size of Sorting Algorithm**

Memory size and space complexity is important in sorting algorithm, because for larger size array would also require additional memory to store temporary values(register). For example, in most merge sort variant requires a space complexity of O(n) time, but Bitonic Sort has a space complexity of O(nlog2(n)). In additional, more complex sorting algorithm require additional system memory space to store the algorithm itself. Below demonstrate the memory size of each merge sort variants(excluding the comments) and its space complexity.

**Table 2.5 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 |

**2.5 Evaluation For Merge Sort**

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

From data collection, Top-down Merge Sort has the third shortest running time but has the smallest standard deviation among all other versions of merge sort in Python. While, bottom-up Merge Sort has a short running time, but the has the second shortest standard deviation in C++.

Both versions of merge sort have the 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, Top down merge sort has an additional “if” function to ensure array size is larger than 2 before returning but would increases the overallrunning 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 used to calculate and compare the lowest value between the endpoint of the array, and the position of the dividing point of the array. This ensures each integer is involved within merge sort, depict unable to divide equally in an odd size array. 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.

Similar research conducted by Arthur Kay on comparison between top-down and bottom-up merge sort has yielded alike result to this investigation. Through Kay’s experiments, the bottom-up merge sort has a shorter running time compared to the top-down merge sort, because the recursive function lead to computing overhead in practical use. Computing overhead refers to calling the function that would require a computer to assign a memory location before conducting, yet excessive recursive function would lead to performance issues. Each recursive function requires the computer to record the current status, registers and address onto a stack under the recursive returns. Whilst Katajainen and Larsson Traff argues, bottom up merge sort actually uses O(nlog(n)+n) memory, and require computer to generate more temporary values address to calculate. Highlighting certain computer operations or languages should perform better in top-down merge sorts

**2.5.2 Evaluation** **Insertion Merge Sort/Tim Sort**

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

Insertion Merge sort perform differently in C++ and Python. For example, in 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 the taken in consideration of the average running time for worst and best data, insertion merge sort may holds the shortest average running time between the other variants.

As previously mentioned in 2.2.3, secondary research suggest insertion merge sort strength lies within its ability to include benefits of insertion sort for small array. For example, insertion sort requires performing (n-1) number of comparisons for any time of data set. Avoiding small size merge sort worst data set to a certain degree, but the constant requirement to check array size for rotation result in becoming a less efficiency sorting algorithm in a top-down structure. The only beneficial on the advantage on worst data set only apparent in C++, with its worst data set running time similar to its random data set(difference of 50,000ns).

As mentioned in 2.2.3, insertion merge sort are often written using a bottom up structure to prevent repeat checking the requirement for array size, making it a more efficient and consistency 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.

**2.5.3 Evaluation on Tim-Sort B**

Between all merge sort variants, Tim-sort B has the fifth longest running time and highest standard deviation time in all data sets (table 2.3). 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 2.1 and 2.2 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 2.4(condition 7 consecutive).

**Table 2.6 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 2.4 has indicated the random distribution of 1000 integers has only meet the Tim-sort requirement between 250 to 400 rotation, but the worst-case doesn’t even meet the requirement once. Even if trigged in best case data set, Tim-Sort is still inferior to other variants. In addition, conditional for Tim sort is checked and reset if the condition is not meet per rotation, so Tim-sort operation function has an additional comparison than other merge sort variant. To increase the probability of Tim-sort being triggered, a reduction in the number of integers require to initiate insertion search would increase the probability in random data set.

On the other hand, the effectiveness of insertion sort in Tim-sort B isn’t as effective for small and random data set. Each insertion sort takes O(log(n)) time per search, in hopes 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 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, and often or not reduces the running time.

**2.5.3.1 Extension for Tim-Sort B**

For this investigation, the largest array to perform merge sort is 500 elements in an array, giving less than or equal to 32.3% to perform insertion sort for each rotation. The probability only increases with the 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 and vice versa.

**2.5.4 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 result, such as, worst case data being faster than random data set in C++, but the opposite in Python. Overall, consider the average running time in table 2.3, 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 design to maximized the number of comparison and switches between two sorted array 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 worst running time possible.

On the other hand, Bitonic sort has a unique requirement to able to perform on array size in power of 2. To achieve this requirement, the array is divided into smaller section 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 function 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 array size together. However, for each sub array(512, 256, 128, 64, 32, 8) need to transverse and divide the array, increase the number of comparison overall.

Therefore, it’s possible for array size in power of 2 would have a shorter running time, such as having 1024 elements in an array rather than 1000 element would have a shorter running time. Using this concept, an alternative method is adding n number of “int” element each with value “-232”(largest negative int value), with “n” being the number of element require for b array size being the power of two, then removing at the end of Bitonic sort may result reduce running time.

**2.6 Merge Sort Conclusion**

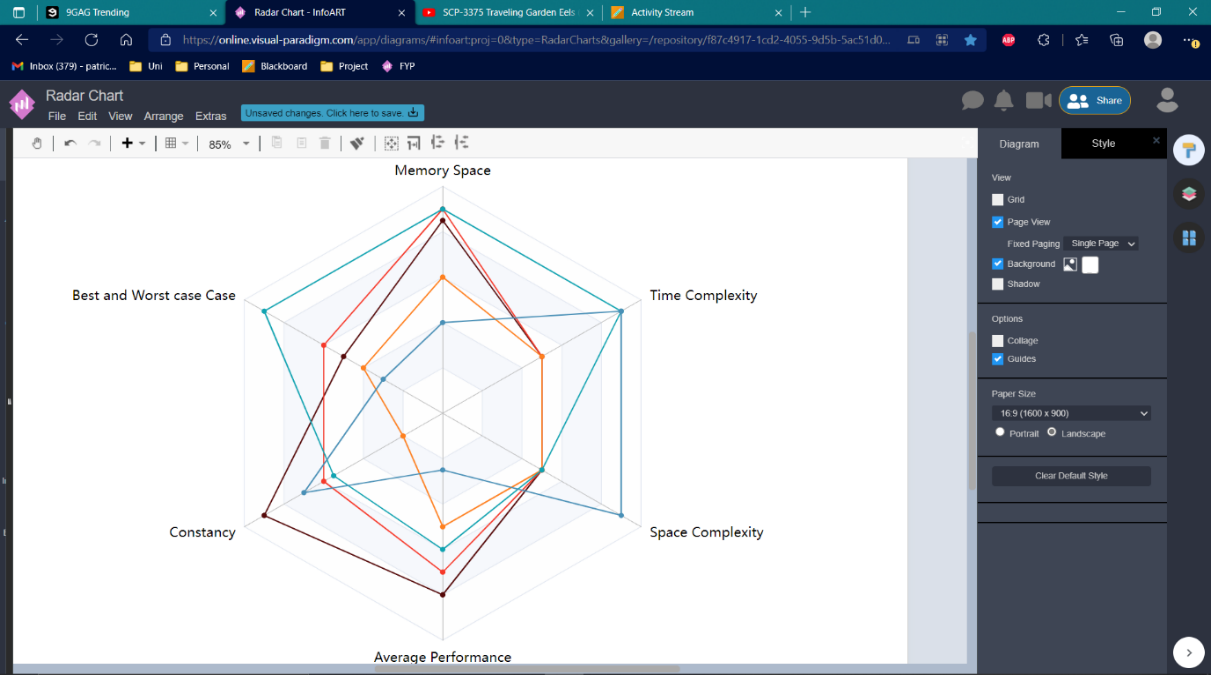
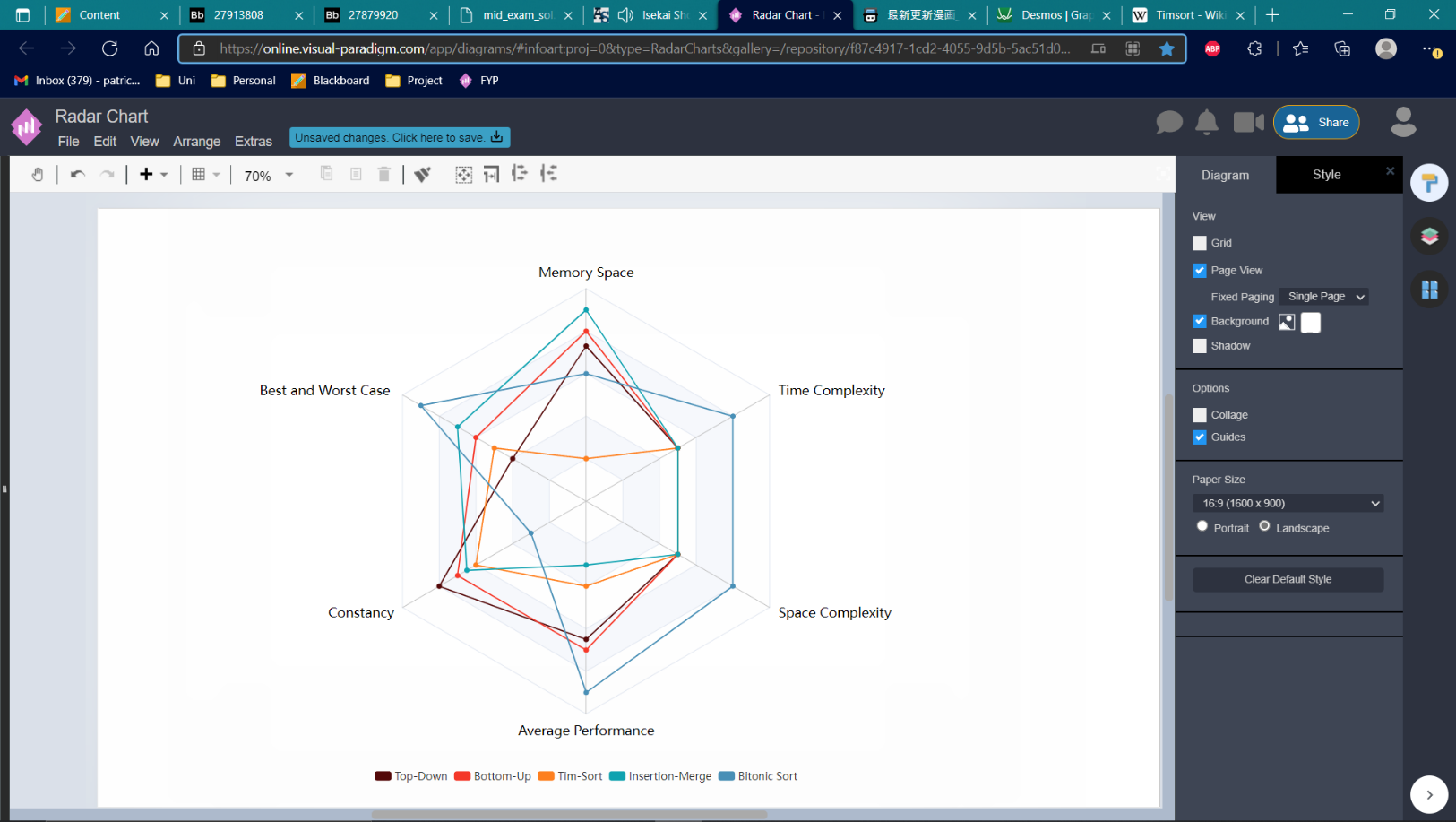
**Overall Result**

To determine the most efficient the following aspect of sorting algorithms would be considered for 2.6.1, 2.6.2 and 2.6.3. However, each of the aspect is not equally as valued with today’s standards in developing algorithm, and more aspect or change in value may occur in the future.

* Memory Space
* Time Complexity
* 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**



**2.6.1 C++ Best Merge Sort**

From the data collection, testing and secondary research under the considered factors in 2.6, the most efficient merge sort sorting algorithm is bitonic Sort, followed by bottom up merge sort and top-down merge sort. As mentioned in 2.2 merge sort could be divided into 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(nlog2(n))
* 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.

**2.6.2 Python Best Merge Sort**

From this investigation, the top three of efficient merge sort variant in python is Top-down, bottom up and Tim Sort version B. However, unlike in 2.6.1, 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, bottom-up merge sort has a similar average running time than Top-down with only 1539590ns slower. There is evidents in 2.5.1 and secondary research that top-down merge sort would create overhead issues in sorting large amounts of elements. Hence, an consideration is deciding the uses 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

**2.6.3 Most efficient Best Merge Sort Variant**

In conclusion, python and C++ each has its own unique merge sort variant that performances the most efficient, but as mentioned in 1.4 there are other factors impacts that may were under assumption or case dependent factors. From a time complexity perspectives, each variant has the same performance, but with current real life performance and issues with algorithm. Hence, I believe bottom up insertion biontic sort should be the most efficient in general case.

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

**C**

**\*Remarks**

<https://www.geeksforgeeks.org/building-heap-from-array/>

<https://courses.cs.washington.edu/courses/cse373/18wi/files/slides/lecture-14-ann.pdf>

<https://www.geeksforgeeks.org/heap-sort/>

<https://www.happycoders.eu/algorithms/heapsort/>

<https://courses.cs.washington.edu/courses/cse373/18wi/files/slides/lecture-14-6up.pdf>

**Reference**

**Merge Sort**

* 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/> at July 25th.
* 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> at July 27th.

**Quick Sort**

* Shrinu Kushagra, Co Writers: Alejandro López-Ortiz, J. Ian Munro, Aurick Qiao(May 2014), Multi-Pivot Quicksort: Theory and Experiments. Receive from <https://www.researchgate.net/publication/289974363_Multi-Pivot_Quicksort_Theory_and_Experiments> at August 23rd.