­­­

March 28, 2022

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

2.5 Design Data Types 7

**3.0 Merge Sort:**

3.1 Merge Sort Background information 9

3.2 Merge Sort Versions 10

3.2.1 Top-Down and Bottom up Merge Sort 10

3.2.2 InsertionMerge Sort 11

3.2.3 Tim-Sort Merge Sort B 12

3.2.4 Bitonic Merge Sort 12

**4.0 Merge Sort Variant Methodology**

4.1Merge Sort Methodology 13

**5.0 Merge Sort Data Collection**

5.1 Merge Sort Testing Result 15

5.2 Data Analysis for Merge Sort 16

5.3 Space Complexity and Memory size for Merge Sort 18

**6.0 Evaluation for Merge Sort**

6.1 Evaluation on Top-Down and Bottom Up Merge Sort 19

6.2 Evaluation Insertion Merge Sort/Tim Sort 20

6.3 Evaluation on Tim-Sort B 21

6.3.1 Extension on Tim-Sort B 22

6.4 Evaluation of Bitonic Sort 23

6.5 Merge Sort Conclusion 24

**7.0 Quick Sort:**

7.1 Quick Sort Background information

7.2 Quick Sort Variants

7.2.1 Choosing a pivot point

7.2.2 Quick Sort Schemes

7.3 Methodology for Hoare Partition

**8.0 Quick Sort Data Collection**

8.1 Quick Sort Testing Result

8.2 Data Analysis for Quick Sort

8.3 Space Complexity and Memory size for Quick Sort

**9.0 Evaluation for Quick Sort**

9.1 Last/First Element Pivot Point Quick Sort Evaluation

9.2 Middle Pivot Point Quick Sort Evaluation

9.3 Random, medium of three and k-element Quick Sort Evaluation

9.4 Quick Sort Math Analysis

9.5 Quick Sort Conclusion

**10.0 Overall Conclusion**

**11.0 Reference**

**­­­**

**12.0 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

Sorting Algorithm

**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 capable of sorting an array of elements in ascending order, and each sorting algorithm consists of different alternative versions. However, most secondary resources published on similar topic focus mainly on comparing different types of sorting algorithms, and not focused on the type of variant against one and another. Does the variant of an sorting algorithm largely affects its performance, and if so how would we be able to make a fair comparison between sorting algorithm? (i.e its best performance variant, commonly used, based on personal needs etc). For this investigation, I would investigate merge sort and quick sort 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 Microseconds.

• Each algorithm

* + Needs to be written in two languages
  + 100 runs are required to determine the average time taken to sort an array
  + 100,000 integers are used in the array must range between -100,000 to 100,000
  + The reason for the total number of runs and range would be explained in 2.4 design type for merge sort

           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 10 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 A for the data set used in this investigation its desire sorted outcome.

**2.5 Designed data type**

For this investigation, we tried to analysis each variant with the largest data set possible to reduce the effects from human error for the methodology in writing the program. Through prior testing, its discovered majorly of merge sort variant requires an average of 2.4 sec run for 1 million element data set in a 0.2- 0.5 vCPUs(Repit). An array with over 1 million element, or using multiple loops per run would exceeded Repit.it limit of 500MB and crash the program(i.e signal killed error). Therefore, to maintain a balance between achieving the largest data set possible with enough data/runs to measure its constancy. For this investigation, 100,000 size data set with 100 hundred runs would be used for this investigation.

The data set size used and method to generate the random data set is often used by other researcher and university that conduct study on comparison of sorting algorithms.

**Best Data Set:** An already sorted array with 100,000 integer from 0 to 99,99.(Remark: one of quick sort variant has a different best data set )

**Random Data Set:** Random data set 1,2 and 3 is generated from random module from C++. 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: 50120.3965, Variance: 28849.10797

* Average number of increasing monotone sequence length: 33236
* Average length of increasing monotone sequence length: 1.50
* Average number of decreasing monotone sequence length: 50183
* Average length of decreasing monotone sequence length: 1.509

Random Data Set 2: Mean: 49890.5785, Variance: 28875.47767

* Average number of increasing monotone sequence length: 33475
* Average length of increasing monotone sequence length: 1.492
* Average number of decreasing monotone sequence length: 50052
* Average length of decreasing monotone sequence length: 1.49518

Random Data Set 3: Mean: 49913.01253, Standard Deviation : 28854.426

* Average number of increasing monotone sequence length: 1.49
* Average length of increasing monotone sequence length: 33383
* Average number of decreasing monotone sequence length: 1.49
* Average length of decreasing monotone sequence length: 50031

**Merge Sort Worst Data Set:** 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 A for the complete list for worst data set and appendix 2 of the method to obtain the worst data set.

**Quick Sort Worst Data Set:** Quick sort worst data type is dependent on the data structure and its pivot point. In any case, quick sort performs the best if the pivot point is chosen to the middle elements within its respective sub array for the least amount of comparison and switches. An example, include choosing the pivot point as the first or last element in a already sorted array would perform achieve the worst time complexity. Secondary research suggest quick sort worst time complexity would achieve quadratic time complexity(O(n2)). Alternatively, quick sort would achieve worst case time complexity if all elements are identical. Hence, for the respective different method to choose the pivot point, I designed the data to ensure the worst case data is achieved. Appendix 3 contains the details for each quick sort variant worst case and best case data set

**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 Merge Sort 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.

**5.0 Merge Sort Data Collection**

**5.1 Merge Sort Testing Result**

Below is a simplified version of the data collected, please refer to appendix B 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 (Microseconds)** | | | | |
| **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Top-Down** | 6,421,554.90 | 7,456,710.70 | 7,264,459.10 | 7,199,669.60 | 7,227,191.80 |
| **Bottom Up** | 70,955.00 | 93,565.70 | 91,941.30 | 87,908.50 | 87,920.00 |
| **Tim Sort B** | 77,167.70 | 101,923.10 | 95,953.50 | 96,683.30 | 95,502.20 |
| **Insertion** | 22,673,053.30 | 38,247,637.20 | 33,308,277.00 | 30,151,074.90 | 31,292,718.90 |
| **Bitonic** | 179,496.20 | 190,456.20 | 215,529.60 | 205,432.10 | 233,290.00 |

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Merge Sort**  **(Python)** | **Average Running Time (Microseconds)** | | | | |
| **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Top-Down** | 2,866,281.77 | 3,967,953.67 | 3,414,111.49 | 3,303,450.30 | 3,279,713.25 |
| **Bottom Up** | 3,385,337.74 | 4,510,502.48 | 4,191,498.52 | 3,970,320.07 | 3,696,418.35 |
| **Tim Sort B** | 3,708,712.46 | 4,867,797.99 | 4,556,441.57 | 4,532,089.39 | 5,152,761.14 |
| **Insertion** | 502,147.82 | 1,501,661.39 | 1,314,262.19 | 1,327,630.69 | 1,319,212.38 |
| **Bitonic** | 11,604,463.60 | 14,792,533.76 | 15,837,551.94 | 12,944,907.27 | 12,352,091.17 |

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Merge Sort** | **Running Time (Microsecond)** | | | |
| **Average**  **(C++)** | **Average**  **(Python)** | **Standard Deviation (C++)** | **Standard Deviation (Python)** |
| **Top-Down** | 942,560 | 3,332,425 | 1,204,859 | 506,116 |
| **Bottom Up** | 89,256 | 3,952,745 | 6,700 | 444,865 |
| **Tim Sort B** | 96,046 | 4,747,097 | 8,853 | 1,261,556 |
| **Insertion** | 31,584,023 | 1,320,368 | 10,126,934 | 174,621 |
| **Bitonic** | 210,480 | 13,711,516 | 15,262 | 3,485,198 |

**5.2 Data Analysis for Merge sort**

From data collection, the shortest average running time in C++ for random data set is bottom up Merge Sort with the smallest standard deviation in C++. However, the insertion merge sort instead 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** | Bottom-Up Merge Sort | | Insertion Merge Sort |
| **Second** | Tim-Sort Merge Sort B | Top-Down Merge Sort | |
| **Third** | Bitonic Sort | Bottom-Up Merge Sort | |
| **Fourth** | Top-Down Merge Sort | Tim-Sort Merge Sort B | |
| **Fifth** | Insertion Merge Sort | Bitonic Sort | |

Similar designed or concept such as top-down and bottom-up 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 with low confidence

Top-Down

Bottom-Up

Tim-Sort B

Insertion-Sort

Bitonic-Sort

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

Chart

Description automatically generated with low confidence

Top-Down

Bottom-Up

Tim-Sort B

Insertion-Sort

Bitonic-Sort

**5.3 Space Complexity and Memory Size for Merge Sort**

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 Space Complexity and Memory size for Merge Sort Variant**

|  |  |  |  |
| --- | --- | --- | --- |
| **Merge Sort Variants** | **Space Complexity** | **Memory of algorithm** | |
| C++ | Python |
| Top-Down Merge Sort | O(n) | 790B | 690B |
| Bottom Up Merge Sort | O(n) | 816B | 768B |
| Tim Merge Sort B | O(n) | 1.54KB | 1.26KB |
| Insertion Merge Sort | O(n) | 1017B | 899KB |
| Bitonic Merge Sort | O(n) or O(1) | 1.44KB | 1.39KB |

**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 second shortest running time Python, but the second most slowest in C++. Meanwhile, bottom-up Merge Sort has a shortest running time and smallest variant in C++, but third fastest in Python

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.

However, the effect of overhead is not as apparent in Python, as top-down merge sort still outperforms bottom-up merge sort by 50,000 micro-seconds in random data set. As previously mention in fair testing, all other operation types variants would incorporate top-down merge sort as its structure and would encounter overhead issues similar to top-down merge sort. However, tim-Sort B and bionic sort seems capable to reduce the running time further than top-down merge using its advantages.

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

Insertion Merge Sort has opposite performs in C++ and Python. Being the fastest and most constant program in Python, but the slowest and least consistence in C++.

In extreme cases(best and worst case data set), insertion merge sort has 15-20% increased running time for worst case data set compared to random, and has either has a 28% decrease in C++ or 60% decreases for best case average running time in python. Insertion merge sort also holds the longest running time for best and worst case data set in C++, but also the shortest average running time for best and worst cases in python. Based on table 5.3, insertion merge sort could either be 3 times faster in comparison to other variant in python, or 30 times slower in C++.

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. This benefit is apparent in Python and C++, with worst data set running time similar to its random data set(difference of 500,000ms).

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 in C++, and the average running time is around 22,000,000 ms. Although, still inferior to other variants in C++, it could it suggest the insertion merge sort could achieve even a shorter running time in python.

**6.3 Evaluation on Tim-Sort B**

Tim-sort B has the fourth ranking in Python in table 5.3, but the second fastest in C++. Its performance almost matches with bottom-up merge sort with only 7000 microseconds slower in C++, but if tim-sort B were to incorporate bottom-up merge sort structure, it may even out perform bottom-up merge sort. Making tim-sort B the fastest merge sort variant. However, the result in Python emphasize several 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, in both languages random data set average running time is often 6% faster than worst case data set, but 19% slower than best case data set. Random data set doesn’t always achieve similar effect compared to best case data, but closer to worst case data set instead. 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 Effectiveness of Tim-Sort B Condition is Achieved**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Number of Rotation Tim-sort Condition Fulfilled** | | | | | |
|  | **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Number of time meet requirements** | 9887 | 0 | 5651 | 11668 | 11780 |
| **Number comparison Reduce** | 586675 | 0 | 347037 | 357550 | 351121 |
| **Number comparison require** | 96858 | 0 | 228706 | 219244 | 222084 |
| **Total comparison Saved** | 489817 | 0 | 118331 | 138306 | 129037 |

The testing result from table 6.1 has indicated the random distribution of 100,000 integers has only met the Tim-sort B requirement between around 5,000 to 10,000 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 1.50 shown in 4.2 Data set. Meaning the probability of meeting the requirement for Tim-Sort B at early rotation are unlikely. In a data set of 100,000, a normal merge sort would make around 1.6 million comparison in random data set, but incorporating Tim-Sort effectively reduce the number of comparison. 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 achieved 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 50,000 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 third shortest running time in C++, but the longest in Python. Similarly, its constancy corresponsive to its performance in respective languages. However, interesting bitonic sort performance in best and worst case is different from other variants. For example, its worst-case data being faster than random data set in C++, but the opposite in Python. Biontic sort is uniquely different compared to other variants, because removes some of its unique features from typical merge sort(i.e space complexity of O(n)). Hence, parts of testing aren’t as suited for bionic sort compared to other variants.

For example, 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. However, bionic sort is slower than other variant, even if not design for worst time complexity.

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 100,000(65536 + 32768 + 1024+512+128 + 32 ), 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(65536, 32768, 1024, 512, 128 , 32) 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 131072 elements in an array rather than 100,000 e. 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.

Moreover, bitonic sort can process better than merge sort variant under a memory limited environment(repit.t). Given bitonic sort doesn’t require to create multiple different array size compared to other variants, bitonic sort could loop tens time more in one runs for 100,000 before encountering signal killed error. Hence, although bitonic sort performs average in C++ and poorly in Python, its has a unique feature that other variants doesn’t have.

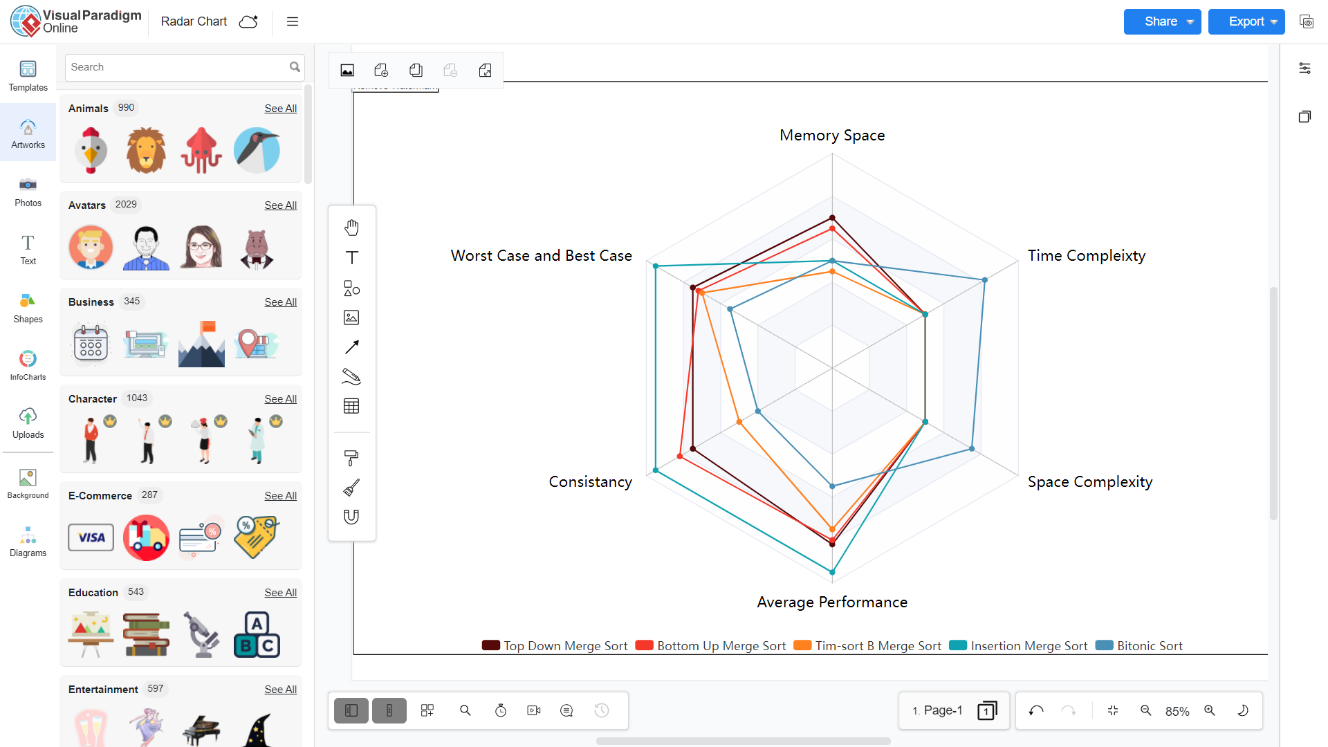
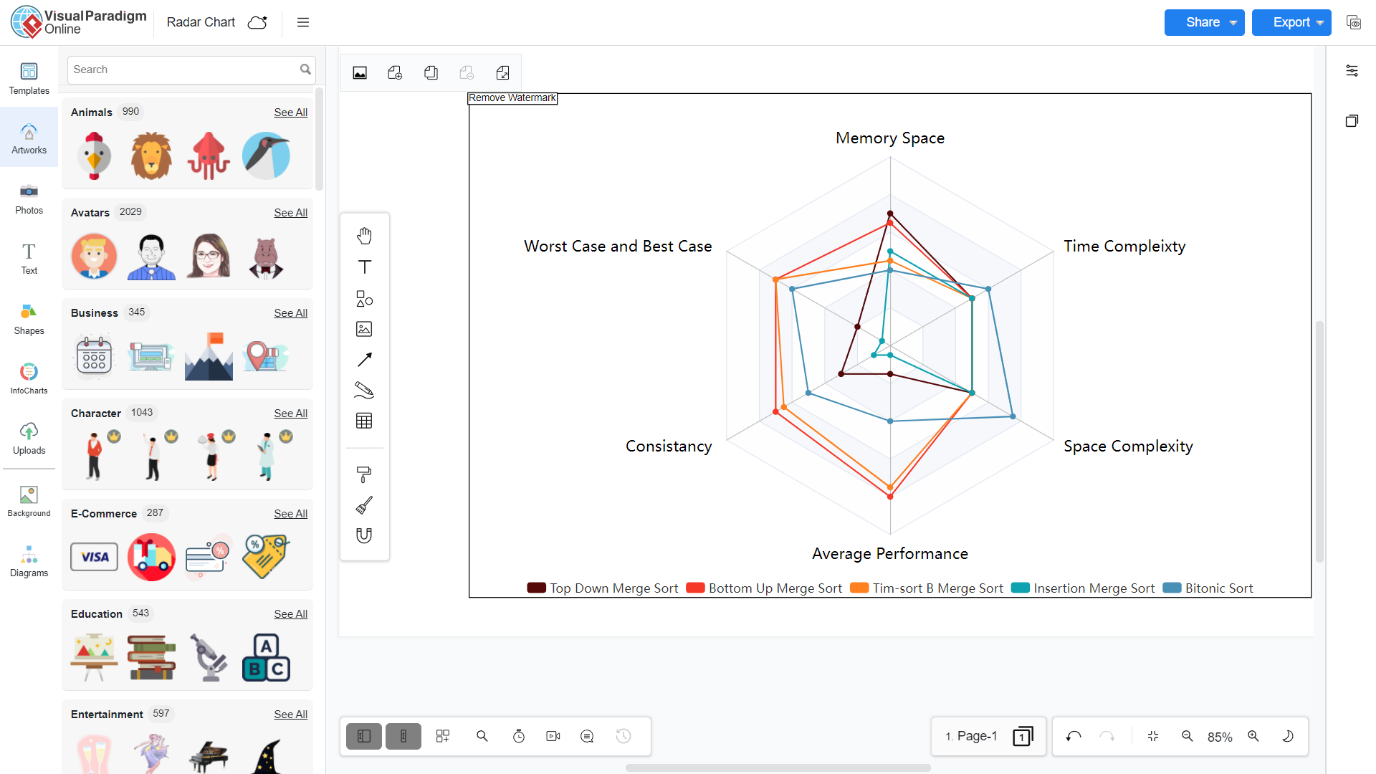
**6.5 Merge Sort Conclusion**

To determine the most efficient the following aspect would be used to evaluated each merge sorting 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(Table 11.1)
* Space Complexity
* Average performance for 10,000 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. For calculation process please refer to appendix 4.

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



**Quick Sort**

**7.1 Quick Sort Background information:**

Quicksort is a type of divide and conquer sorting algorithm, and has a running time of O(nlog(n)) time. Quicksort core concept chooses a pivot point in an array of unsorted integers, and places all integers larger than the pivot integer on the left, and smaller on the right. This process is repeated for on left and right array until the size of the array is less than 2. The scheme to choose pivot point affects the running time and constancy of the algorithm. The scheme includes randomly decided or repeat until a certain condition is met. The array is invented by Tony Hoare in 1959 and has become one of the most efficient commonly used sorted arrays. Appendix 2 demonstrates an example of quicksort.

**Advantages of Quick sort:**

* No additional storage/memory are required, as majorly of array sorts by within the array.
* Best and advantage case for quick sort is O(nlog(n)) time, making any sort of data set capable to achieve the best running time.

**Disadvantage of Quick sort:**

* If quick sort is not implanted properly may lead to time complexity of worst case of O(n2), because of improperly condition for pivot point.
* Quick sort is an unstable sorting algorithm, because the swap is based on the pivot position and the data set uniqueness.
* Quick sort may lead to a large variance if random pivot choosing is used. However, the variance may reduce with proper condition for choosing the pivot point.
* If elements are already sorted, bubble sort would be quicker.

**7.2 Quick Sort Versions**

Quick sort has one of the most varies implantation methods among all sorting algorithm. Computer scientist has theorized different methods in choosing the pivot point, and schemes to performing switches without required to generate additional memories. However, currently there isn’t a method to select the ideal pivot point for every data set, without any additional calculation perform.

**7.2.1 Choosing a Pivot Point**

In choosing the pivot point for quicksort, computer scientists investigated multiple solutions to reduce the number of comparisons and switches with the pivot point to maximize efficiency and time. Some of the methods include, always picking the first/last elements, the middle element of the array, or randomly. The common aim is picking a pivot point that divides the array into a 1:1 ratio for each rotation, but achieving the perfect ratio is difficult given each element is likely to equal. Hence, a method to choose the pivot point has its benefits and disadvantages in a different case.

Based on secondary sources, picking the first/last element as the pivot point has the highest probability to yield the worst time complexity O(n2) among the other two methods. The quicksort worst time complexity is derived from making the maximum number of switches with pivot point and dividing the array into one partition for each rotation. Hence, the worst-case time complexity would perform similar to equal of bubble sort(O(n2)), with each rotating being O(n)->O(n-1),->O(n-2)….O(1) time. Alternatively, randomly picking without condition may result in a similar performance, but the probability is low and could be averted by setting certain conditions.

On the other hand, randomly picking the pivot point requires additional operations to generate a number between the array, as PRNG(Pseudorandom number generator) is relatively slow for certain languages. Hence, picking the middle element of the array would create a similar effect as picking random, but choosing a random pivot is statistically more likely to be close or equal to the median.

To overcome the weakness of random and middle element quick sort, alternative versions such as medium of three, medium of four(Yarosalvisty), medium of five, etc overcomes the issues. Three or more pivot point is randomly picked between the array, then compared to determine the medium among the pivot points. Hence, avoiding the worst time complexity case and yield a closer effect as random quick sort, but requires two comparisons per rotation. Multi-Pivot Quick Sort investigation conducted at the University of Waterloo, discovered 1-pivot point often has O(2nln(n)) comparison with (0.333nln(n)) swaps, but the medium of 3 has O(1.71nlog(n)) comparison with O(0.343nln(n)) swaps. However, more pivot point doesn’t always lead to reduce the number of swaps and comparison. Ultimately, different quick sort versions would perform better for certain data set, but random, first/last element pivot point, median element pivot point, median of three and k-pivot point pivot point quick sort would be mainly investigated for this paper.

**7.2.2 Quick Sort Schemes**

There are multiple different implantations in performing quick sort switches in coding languages. For example, we could create two stack data types, then any element larger than the pivot point be a push to one stack and a smaller element on the other. The process repeats on both stacks until the size is less than 1, and pop the value from the smallest stack out. However, quicksort is well known for its space complexity being O(1), but the above-suggested method requires additional memory to store the stack. Hoare partition suggests by Tony Hoare, and Lomuto partition schemes designed by Bentley uses switches within the array to reduce space complexity.

Both Hoare and Lomuto partition schemes creates two pointers(pointer A, pointer B) that converges towards the pivot point, and stop if the below conditions are meet:

* Pointer A stops if element is larger than the pivot point
* Pointer B stops if element is smaller than the pivot point
* Both pointers stop if point A and pointer B passes each other

If above condition 1 and 2 is meet the two elements at each pointer performs a swap, and each pointer continues to converge towards the pivot point.

Lomuto partition schemes typically choose the first/last element as the pivot point, and both pointers at the other end. While Hoare partition schemes are more flexible with their pivot points, and chooses the pointers at opposite ends. Secondary research suggests the Hoare partition scheme has better performance than the Lomuto patriots scheme, because

* Hoare partition statically performs three times less swap compared to Lomuto patriots scheme
* If all elements are equal, Hoare partition scheme time complexity is O(nlog(n)), but Lomuto partition schemes instead takes O(n2).

Hence, Hoare partition scheme implantation would be only investigated for quick sort for this paper.

**7.3 Methodology for Hoare Partition**

Below code, the structure would be used for reference for quick sort Hoare Partition that uses random, middle, and last element as the pivot point. For full quick sort code please reference appendix A.

1. Choose a pivot point integer via
   * + 1. Random generated
       2. Middle element
       3. First or last integer
       4. Choose three random pivot point, and use the median as the pivot point

Based on array size, take k number of pivot point and take median pivot point, the larger the array the more pivot point would be taken

1. Create a position tracker/pointer on the start(left) and end(right) of the array
2. Move the left position tracker toward the pivot until reach to an integer larger or equal than the pivot point integer
3. Move the right position tracker toward the pivot until reach to an integer smaller or equal than the pivot point integer
4. Swap position between right position tracker with left position tracker
5. Repeat step 3 until left position tracker or right position tracker reach the same position.
6. Perform step 1 to 6 for the array on the left and right, until array size is less or equal to 1.

**8.1 Quick Sort Data Collection**

Below is a simplified version of the data collected, please refer to appendix B for a more detailed version.

**Table 8.1: Average Running Time Quick Sort in C++**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Quick Sort (C++)** | **Average Running Time (Microseconds)** | | | | |
| **Pivot Point** | **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Last element** | 2,169,264 | 338,692,502 | 603,442 | 654,887 | 652,536 |
| **Middle Element** | 385,463 | 1,694,525,769 | 699,530 | 653,026 | 551,686 |
| **Randomly** | 1,943,111 | 37,147,300 | 2,182,479 | 2,276,020 | 2,260,350 |
| **Medium of Three** | 1,980,230 | 2,042,672,366 | 2,632,556 | 2,890,555 | 2,650,164 |
| **K-Random** | 1,773,496 | 1,927,313,855 | 2,943,431 | 2,870,164 | 2,591,915 |

**Table 8.2: Average Running Time Quick Sort in Python**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Quick Sort (Python)** | **Average Running Time (Microseconds)** | | | | |
| **Pivot Point** | **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Last element** | 931,316 | NA | 1,448,864 | 1,404,648 | 1,158,544 |
| **Middle Element** | 943,134 | NA | 1,235,976 | 1,323,017 | 1,333,623 |
| **Randomly** | 2,093,068 | NA | 391,410 | 370,169 | 312,235 |
| **Medium of Three** | 222,381 | NA | 273,748 | 291,555 | 291,200 |
| **K-Random** | 228,543 | NA | 250,277 | 253,361 | 259,054 |

**Table 8.3: Average Running Time for Quick Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Quick Sort** | **Running Time (Microseconds)** | | | |
| **Average**  **(C++)** | **Average**  **(Python)** | **Standard Deviation (C++)** | **Standard Deviation (Python)** |
| **Last element** | 636,955 | 1,337,352 | 220,680 | 407,259 |
| **Middle Element** | 634,747 | 1,297,538 | 222,589 | 550,990 |
| **Randomly** | 2,239,616 | 357,937 | 211,407 | 63,857 |
| **Medium of Three** | 2,724,425 | 285,500 | 371,057 | 42,127 |
| **K-Random** | 2,801,837 | 254,230 | 351,814 | 6,410 |

**8.2 Data Analysis for Quick sort**

From the data collection, choosing the middle element as pivot point is fastest in C++, but is the second slowest in Python. On the other hand, random pivot point instead is the fastest in Python, but the third slowest in C++. The difference in performance and ranking may due to the difficulty for C++ to generate a random pivot point, and the average running time increases with each increased number of pivot point for medium of three and k-element pivot point. Python method to generate random pivot point is more stable and faster compared to C++ as shown in table 8.3, allowing the advantage from random pivot point more apparent being approximately three times faster than last/first and middle pivot point quick sort. However, without proper used in specific language it may lead to 4 times slower as shown in C++. The ratio between random data set and best case for both language is approximately between 1.8-1.2 : 1, emphasizing the difference between best time complexity and average time complexity. Below table is the order of the shortest to longest running time for each merge sort variant in random data set.

**Table 8.4: Ranking of each Merge Sort Based on Average Running Time**

|  |  |  |  |
| --- | --- | --- | --- |
| **Shortest Running Time** | **C++** | **Python** | |
| **First** | Middle Element Pivot Point | | Random Pivot Point |
| **Second** | Last Element Pivot Point | Medium of Three Pivot Point | |
| **Third** | Random Pivot Point | K-Random Element Pivot Point | |
| **Fourth** | Medium of Three Pivot Point | Middle Element Pivot Point | |
| **Fifth** | K-Random Element Pivot Point | Last Element Pivot Point | |

**Graph 8.1:Normal Distribution for Quick Sort Variant in Python**

Chart

Description automatically generated with low confidence

Random

Middle

Last

Median of three

K-random elements

**Graph 8.2:Normal Distribution for Quick Sort Variant in C++**

Chart

Description automatically generated

Random

Middle

Last

Median of three

K-random elements

**8.3 Space Complexity and Memory Size**

Space complexity and memory size used to investigate each quick sort variant is O(1) and equal or around 533B, as the main differences between the each variant is the method to choose the pivot point.

**Table 8.5 Space Complexity and Memory size for Quick Sort Variant**

|  |  |  |  |
| --- | --- | --- | --- |
| **Quick Sort Variants** | **Space Complexity** | **Memory of algorithm** | |
| C++ | Python |
| Last/First Quick Sort | O(1) | ≈363B | ≈553B |
| Median Quick Sort |
| Random Quick Sort |
| Median of Three Quick Sort |
| K-Pivot Point Quick Sort | ≈1.02KB | ≈997B |

**9.0 Evaluation For Quick Sort**

**9.1 Last/First element Pivot Point Quick Sort Evaluation**

From the data collection, choosing the last/first element as the pivot point would be in par the short-running time variant in C++. Unlike other variants, the pivot point is directly obtained neglecting the process require to calculate middle random pivot point or generate the pivot. Hence, resulted in a smallest standard deviation in C++ and similar performance for random data set as to middle pivot point.

However, choosing the last or first element for worst case data set(a sorted data set) requires exponential time, as it perform similar to bubble sort(O(n2)). As a result, the average running time for worst case data set in almost 37 sec per run in c++, 1700 times longer than its average running time for random data set depending on the language. This issues for exponential time is also applicable to a partially sorted array, making it difficult in certain real-life situation (i.e cross checking). An alternative worst case data set for any quick sort array is when array is identical elements. For this investigation, the mono-tone sequence length and size is designed to be uniform and random, semi-ideal data set for this variant as its unlikely to choose the last or first element each time.

Surprising best-case data set for last/first quick sort is outperform by random data set depict picking the medium element every time in C++, but not in python. As mentioned in appendix 4, quick sort aims to pick the medium element to divide the array into equal size, but the pivot point requires to perform switch with half of the array. In comparison with other variants data set, medium of three, middle pivot point and k-element pivot point quick sort doesn’t require perform switches. I calculated that best case data set performs around 2295146 switches and random data set performs 1938565, 18% more switches despite achieving the medium pivot point. Hence, last/first quick sort for best case date is instead random data set with a balance of switches and probability to pick medium pivot point.

A remark note is running the worst case lead to maximum recursion dept for python(1000), and despite to add an additional change dept recursive function still lead to segment fault(core dump) error. However, changing the maximum recursive dept may yield to stack overflow issues and create overflow issues as shown with top-down merge sort in 6.1.

Given it weakness for choosing the last/first element for quick sort, the problem overcome with a prior checking to determine if the array already or partially sorted. Then, evaluated to determine the right sorting algorithm to used. Making this variant depict its flaw one of the most used variants for quick sort.

**9.2 Middle Pivot Point Quick Sort Evaluation**

Choosing the middle pivot point is similar to in performance as if choosing first/last element quick sort, with equal probability of the medium element being the first, last, middle element of the array in a uniform random distributed array. To differentiate the difference in performance between first/last element quick sort, we must take in consideration of different for worst case data set type and the cost to calculate the middle pivot point.

Best case data set for middle pivot point quick sort is a sorted array in ascending or descending, and leading middle pivot point to have the faster average time complexity for best-case data set. This type of best data set allows middle pivot point to served as a cross check for sorted array and less impact in partially sorted array. However, the worst case data set has instead lead to a 2669 times longer in comparison to its average running time, due to the additional function require to calculate the middle element position. In comparison for the application, last/first pivot point quick sort worst data set has less functionality. For example, adding random data onto sorted data is one of the common practices. Middle pivot point worst case requires a specific pattern that is uncommon relative to a practically sorted array.

On the other hand, middle pivot point has a slight higher standard deviation in relative to first/last pivot point quick sort from the calculate the middle pivot point(start array + (end array- start array)/2). Thus, as the array size increase the difference in consistency between first/last pivot point quick sort would theoretically increase.

**9.3 Random, Median of Three and K-elements Quick Sort Evaluation**

Any method to pick pivot point involving random modules performs poorly in C++ but excellent in python. An increased number of pivot point would often lead to increase in average running time in python, but opposite in C++. In both languages, increasing number of pivot point also increase its consistency.

The difference in performance between medium of three and k-element isn’t significant relative to difference in other variants. In addition, k-element requires an additional quick sort to determine the medium element among the pivot point, leading to potential overhead issues demonstrated in merge sort top-down. In this investigation, two additional pivot point would be generated for per 5000 elements in each sub array. Hence, at the beginning with 100,000 elements a total of 20 pivot point would be generated, and only one pivot point would be used for 5000 element or below. The reason for the 5000 is from secondary research by Angela Carpenter determine quick sort perform poorly compared to merge sort after 5000 or more. The method to scale the number of pivot point could be adjusted using different methods.

The worst-case data set for all three random related module pivot point has a much more significant impact compared to last/first and middle pivot point quick sort. Being almost 600-1700 times longer compared to its average running time for C++ random data set, but each increased sample pivot point seeming reducing its difference. Having more pivot point allows more likeliness to able to choose a medium element to divide the array equally.

Computer scientists theorized random pivot point would likely choose the middle element in random data set compared to other variants, as mention in 9.2.1. To prove this theory, I tested 1000 times to calculate the average distance is the chosen pivot point from its actual medium point in C++ for random data set for all variants.

**Table 9.1 Average Difference from pivot point to medium**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Number of elements difference on average** | | | |
|  | **Random Set 1** | **Random Set 2** | **Random Set 3** | **Average** |
| **Last element** | 14.64 | 39.78 | 15.70 | 23.37 |
| **Middle Element** | 15.64 | 16.74 | 15.37 | 15.92 |
| **Randomly** | 14.71 | 15.57 | 15.12 | 15.13 |
| **Medium of Three** | 13.72 | 13.54 | 14.48 | 13.91 |
| **K-Random** | 0.483 | 0.483 | 0.484 | 0.5 |

Table 9.1 support my hypothesis that random operation involving in choosing the pivot point and including multiple would yield a higher likely chance to obtain a medium pivot point. K-random quirt sort almost always achieving middle pivot point every time.

However, due to its random pivot point nature, its worst case and best case is harder to achieved compared to choosing the first/last element or middle element as its pivot point. Hence, random quick sort, medium of three and k-element quicksort only have worst case data set if every element is the same, making it a more consistence against other types of data set. The inconsistence mainly comes from random generated modules and could be further reduced to improve its accuracy with certain requirement. For example, pivot point would randomly generate until the element is between 1/3 to 2/3 of the medium array, or using last, first and middle element as pivot point for medium of three quick sort. As demonstrated in Python, additional pivot point does benefit in larger array size, but shouldn’t be obtain with random modules.

For this investigation, we did not restrict the random pivot point condition, but the further investigation could be investigated for different restricted random quick sort would yield the shortest and stable sorting algorithm.

**9.4 Quick Sort Math Analysis**

Within this investigation, uniform random distribution data set was the mainly benchmark to evaluate and test the performance of each quick sort variant. However, there are certain patterns that are commonly observed in certain practices/sequence types. Some common sequence types seen in real life are

* Uniform random distribution: Data distributed
* Sorted/partial sorted
* Exponential increase/decrease distribution
* Gamma distribution
* Linear increase/decrease distribution
* Normal distribution
  + High/low/average mean
  + Large/small standard deviation
* Cosine Wave distribution
* K-monotone sequence
* K restricted sequence
* etc

The distribution is represented with the x axis in proportional to the position of the data within the array, and the y axis is the average value from neighboring data value around the x position. An example is when x =100, while y value is the average of x-50, x-49, x-48…. x+48, x+49, x+50. The sample range for each x value may adjust. Besides the listed distribution above, if we sum all the difference patterns it would ultimately result in a uniform distribution. Nevertheless, sorting algorithm main objective is converting any of type of distribution to either increase/decrease linear or exponential distribution.

In most quick sort variant, the pivot point hopes to choose the medium element(y-axis not too high or too low) to reduce the number of dept(recursive/overhead). However, some quick sort variant detriment against more patterns types than others, such as last/first pivot point in exponentially and linear increase or decrease would always result repeatly result in the worst distribution/sequence. However, medium pivot point would convert an normal distribution sequence into an increase and decrease exponential sequence, even with an unflavored first pivot point as shown below.

Certain distribution /sequence pattern reflects the best and worst case for specific quick sort variant, while others would undergo transformation to improve its performance. However, we could predict its performance for infinite size array with certain commonly distribution patterns seen in real life situation. Below table is a prediction on some of the commonly seen types

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Distribution Patterns** | **Quick Sort Variants** | | | | |
|  | **Last/First** | **Middle** | **Random** | **Medium of three** | **K-Elements** |
| **Uniform** | No Change | | | | |
| **Exponential/Linear increase/decrease** | No change(Worst Case) | No change (Best Case) | No Change(but performs closer like middle pivot point quick sort) | | |
| **Normal Distribution** | Normal distribution to Exponential distribution | 1. Normal distribution with exponential distribution 2. Increasing and Decreasing Exponential distribution 3. No change(Only possible with random, medium of three, k-elements) | | | |
| **Cosine Wave** | Repeating chi-squared distribution | No change | | | |
| **Random** | No change | | | | |

Time complexity of each variant is slightly different for each variant. The calculation process discovered by Hoare in 1962 are shown as follows

C: Constant Value

P: Distribution

E: Estimate/Integrate over [0,1] on p

If there are more than one pivot point to find medium of three the equation is instead with k being the medium element for 2k+1 pivot point.

Using the above equation, last/first, middle and random pivot point has an approximation time complexity of 1.386nlog(n), with an 38% performing slower for average case compare to best case. Meanwhile, medium of three has 1.88nlog(n) time complexity, but with a 3 percent increase in total number of swap. Although the above equation suggest more pivot point leads to smaller time complexity, data collection has highlight several issues in implantation for real life application.

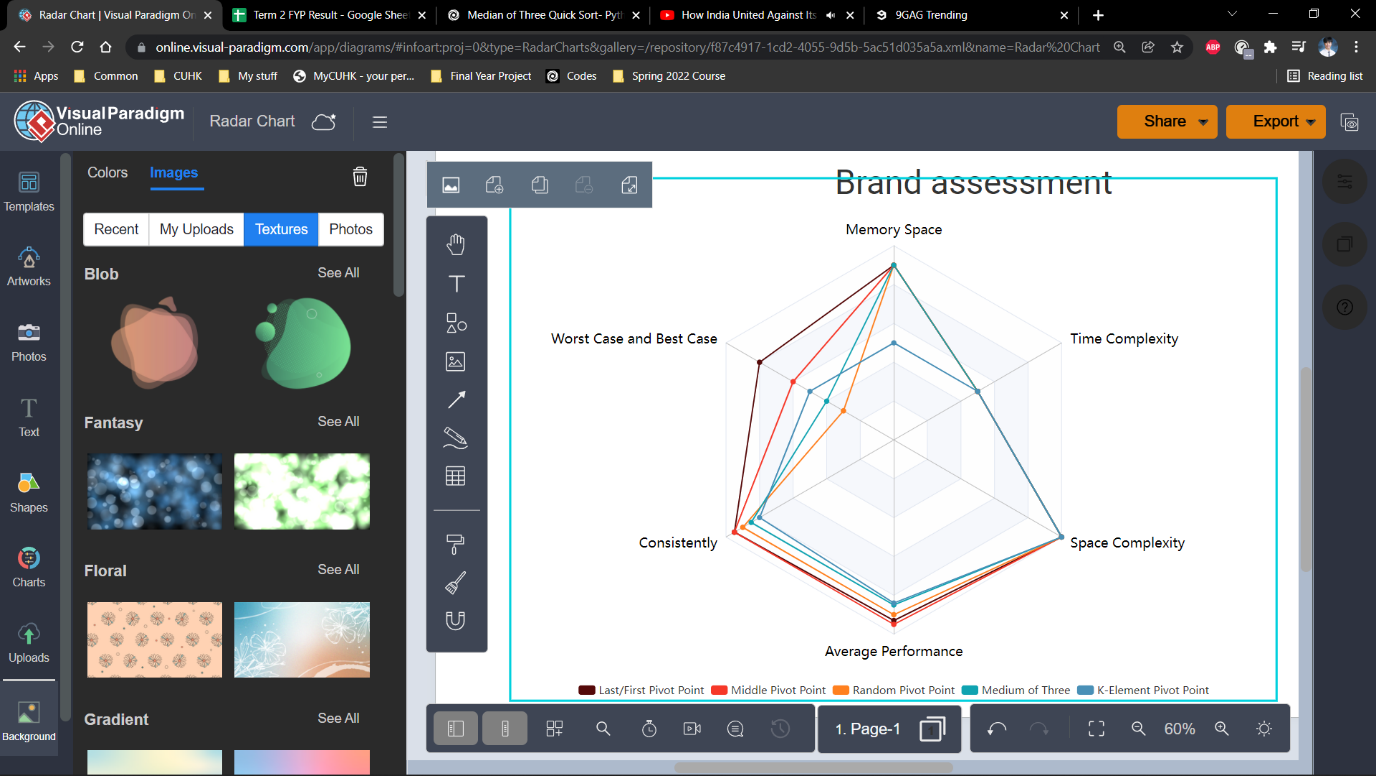
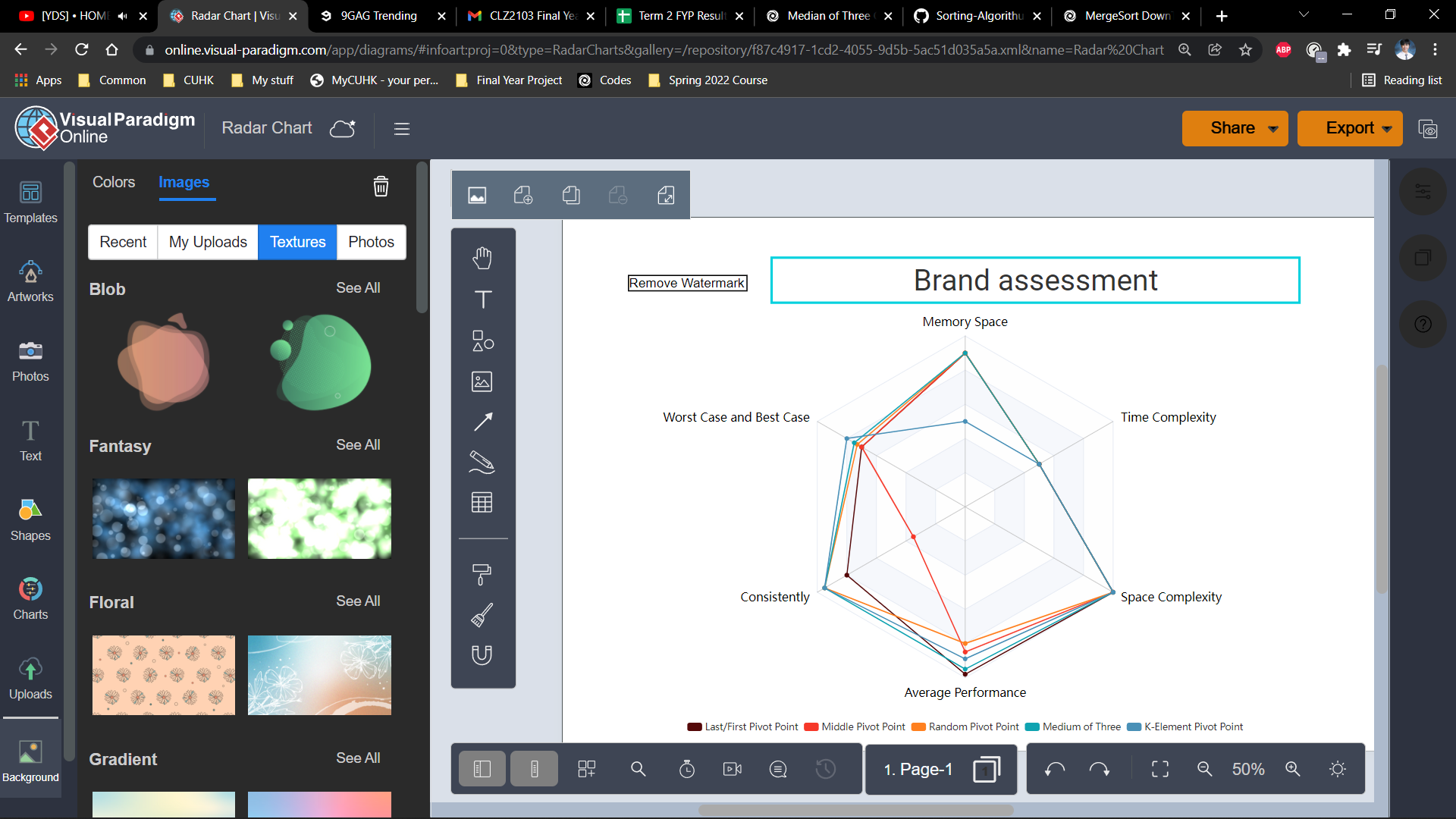
**9.5 Quick Sort Conclusion**

To determine the most efficient the following aspect would be used to evaluated each quick sorting variant. Unlike merge sort, quick sort evaluation doesn’t have space complexity, because all variant have a space complexity of O(1) time. However, space complexity would still be include to allow better comparison with merge sort’s spider diagram.

* Time Complexity
* Memory Space(Table 11.1)
* Average performance for 100,000 integers
* Constancy
* Best case data set

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. The calculation process is calculated using appendix 4.

**Figure 6: Analysis of Quick Sort in C++ Figure 7: Analysis of Quick Sort in Python**



**10 Conclusion**

Python and C++ each have their own unique sort algorithm 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 its current real-life performance may vary. However, some may prefer certain sorting algorithm due to its unique advantages.

|  |  |
| --- | --- |
| Merge Sort   * Space Complexity: O(n)[except bitonic sort] * Best, worst, average time complexity: O(nlog(n)) * Better for larger array size * Better in link types of array | Quick Sort   * Space Complexity: O(1) * Worst Case Time complexity:O(n2) * Best and average time complexity: O(nlog(n)) * Better for smaller array size * Better in array store elements |

From my data collected, merge sort variant is typically faster than quick sort variant for array size 100,000, matching with research conducted by others for large array size, but isn’t always the case for all variants. Some of merge sort variant even the popular used top-down merge sort is slower than most quick sort variants. Below table is in comparison the fastest merge sort and quick sort variant for C++ and Python.

**Table 11: Fastest Sorting Algorithm Variants Investigated**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sorting Algorithm** | **Running Time (Microseconds)** | | | |
| **Average**  **(C++)** | **Average**  **(Python)** | **Standard Deviation (C++)** | **Standard Deviation (Python)** |
| **K-Random Quick Sort** | 2,801,837 | 254,230 | 351,814 | 6,410 |
| **Middle Pivot Point Quick Sort** | 634,747 | 1,297,538 | 222,589 | 550,990 |
| **Insertion Merge Sort** | 31,584,023.60 | 1,320,368.42 | 10,126,934.49 | 174,621.26 |
| **Bottom Up Merge Sort** | 89,256 | 3,952,745 | 6,700 | 444,865 |

On the other hand, results conducted in this investigation emphasizes several key importance in making comparison of algorithms. Research paper and online paper on similar topic often conducted comparison between different types of sorting algorithm(merge sort, quick sort, counting sort etc), but not always labeling the type of variant used to reach its conclusion. Often claiming certain types of sorting algorithm is superior to other. For example, research in comparison sorting algorithm by Angela Carpenter determined that quick sort is faster than merge sort for smaller array size, but opposite for array size exceeding size 75,000 through testing in python. However, Angela explored/conducted test with top-down merge sort against a first/last pivot point quick sort in python, and only briefly mention some quick sort variants only. Ultimately concluded that merge sort(top-down) average running time is 54.037ms, and quick sort(last pivot point) average running time 67.664ms for an array size of 10,000. Certain aspect of this investigation oppose the result obtain by Angela in python, but it may be cause by the different application used, persuade code, data set etc(Angela used NumPy and Pandas).

First, different variants could lead an exponential increase or decrease in sorting algorithm performance and leads to the question “what is best methodology to determined and compare sorting variants efficiency?”. Majorly of current computer company prefers quick sort over merge sort due the significant less space complexity, but there exist a merge sort variant with a space complexity same as quick sort(bitonic sort, if configured correctly). Although, bitonic sort average running time isn’t the fastest, bitonic sort has it own advantages. Moreover, different variants could combine to become a more efficient program. Hence, should comparison between sorting algorithm be conducted only for personal needs, application, equipment etc?

Alternatively, this investigation couldn’t confidently claim which sorting algorithm is superior, because there are other factors that wasn’t fully explored. i.e

* Different array size
* Other alternative variants
* A mixture of variants together
* Types of data set
* Etc

Secondary, application and theory of sorting algorithm isn’t always applicable to each variant. An example includes quick sort best case data set for last/first pivot point performing worse than random data set. Most over quick sort variant has a shorter average running time from having the medium element as pivot point, but last/first pivot point quick sort suffers from the number of switches and is slower. However, picking the medium element instead as pivot point is beneficial for other variants that performs no switches. Time complexity, space complexity, theories isn’t always applicable for each variant in the same sorting algorithm or in real life application.

With different variants performing differently from commonly represented sorting algorithm(ie. Quicksort: Last/first pivot point Quick Sort or Merge Sort: Top-down Merge sort). Certain sorting algorithm variants shouldn’t be label as a branch of the original algorithm, given it removes certain unique aspect/features of the original. For example, bitonic sort has a space complexity of O(1) even though it’s a merge sort variant. Therefore, labeling sorting algorithm by its specific names, theories applicable to which variants, assumption made, etc, is crucial to be clear in reports and teaching involving algorithm.

**11.0 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, 2021.
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, 2021.
3. 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
4. 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, 2021.
5. Angela Carpenter(2021), Benchmarking Sorting Algorithms in Python. Receive from [CTA Benchmarking Project Report final (angela1c.com)](https://www.angela1c.com/projects/cta_benchmarking/ctabenchmarkingproject#Python-Benchmarking-application-code) on January 4th, 2022

**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 3 Quick Sort Worst Data Set**

         For quick sort, each variant performs the worst when require the pivot point picked is either the first or last element, and perform the best when the pivot point is the middle pivot point.

**Last/First Element Pivot Point**

Best Case: Mid point starting from left, for example in a 100,000 size array:

50000, 25000, 12500, 6250, 3125, 1562, 781, 390, 195, 97, 48, 24, 12, 6, 3, 1……

An example is to input 100,000 in a balance binary tree and reading it in a preorder traversal.

Worst Case: An already sorted array in ascending, descending order and array with identical elements

**Middle Element Pivot Point**

Best Case: Already sorted array in ascending order

Worst Case: An array with identical elements

**Random Element Pivot Point**

Best Case: No answer, due to the inconsistency of random module, can’t ensure each pick would be either the last/first element or the middle element. However, an sorted array would be closest to best case.

Worst Case: An array with identical elements

**Appendix 4 Spider Diagram calculation**

Best and worst case is calculated by adding the average best and worst case running time and measured using the average performance on its respective languages. Memory space is same for both languages

|  |  |  |  |
| --- | --- | --- | --- |
|  | **For C++** | | |
|  | **Average Performance(103)** | **Consistency(103)** | **Memory Space(Bytes)** |
| **100** | 0-45 | 0-20 | 0-2 |
| **90** | 45-90 | 20-40 | 2-4 |
| **80** | 900-135 | 40-80 | 4-6 |
| **70** | 135-180 | 80-160 | 6-8 |
| **60** | 180-225 | 160-320 | 8-10 |
| **50** | 225-270 | 320-640 | 10-12 |
| **40** | 270-315 | 640-1,280 | 12-14 |
| **30** | 315-360 | 1,280-2,560 | 14-16 |
| **20** | 360-405 | 2,560-5,120 | 16-18 |
| **10** | 405-450 | 5,120-10,240 | 18-20 |
| **0** | >450 | >10,240 | >20 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **For Python** | | |
|  | **Average Performance(103)** | **Consistency(104)** | **Memory Space(Bytes)** |
| **100** | 0-1 | 0-2 | 0-2 |
| **90** | 1-2 | 2-4 | 2-4 |
| **80** | 2-4 | 4-8 | 4-6 |
| **70** | 4-8 | 8-16 | 6-8 |
| **60** | 8-16 | 16-32 | 8-1 |
| **50** | 16-32 | 32-64 | 1-12 |
| **40** | 32-64 | 64-128 | 12-14 |
| **30** | 64-128 | 128-256 | 14-16 |
| **20** | 128-256 | 256-512 | 16-18 |
| **10** | 256-512 | 512-1024 | 18-2 |
| **0** | >512 | >1024 | >2 |

**Appendix A**

The bellow github link contains all the code and data set 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**

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>