[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 Tim Merge Sort

2.5.2.1 Extension for Tim Merge Sort

**Quick Sort:**

3.1 Quick Sort Background information

3.2 Quick Sort Versions

3.2.1 Choosing a Pivot Point

3.2.2 Quick Sort Schemes

3.3 Methodology for Hoare Partition

3.4 Quick Sort Data Collection

3.4.1 Data Analysis for Quick Sort

2.5 Quick Sort Evaluation

2.5.1 Last/First Element Pivot Point Evaluation

2.5.2 Middle and Random Element Pivot Point Evaluation

2.5.3 Random Element Pivot Point Extension

**Introduction:**

**1.1 Motivation**

Since the invention of computers, computer scientist has been designing different computer algorithm to solve a large quality or perform complex calculation and operations. For example, a computer’s additional and subtraction for large quality are considered algorithms. However, the performance of different algorithms could be compared and measured. Nowadays, computer science is investigating and design more efficient algorithms, to create better computers and systems to achieve more complex calculations.

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

To create or improve an algorithm is a difficult process and requires innovative visualization or concept in approaching the design problem. A different version of the algorithm is superior in certain aspects but potentially weaker in others. Hence, the best algorithms to solve an issue/problem is case dependent but could be compared in real-life application against each other.

For this investigation, I’m motivated to learn the different aspect 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**

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

Today, different computer scientist has designed different types and version of sorting algorithm as shown in figure 1 and table 1.1.

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) |

**1.4 Fair testing**

To ensure each sorting algorithm is tested and investigated fairly, certain factors and resources are controlled to prevent any advantage of one algorithm over another. 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 on the development of the algorithm, and their impact is consistent in each algorithm. Environment factors refer to the testing environment, equipment, or additional algorithm requirement, and each factor should be minimized to as best of our 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)
* A MacBook Pro, MacOS Catalina version 10.15.6 would also be used for this investigation.
* 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
  + Ten runs are required to determine the average time taken to sort an array
  + 100 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, so certain languages may result in a shorter running time for the specific sorting algorithm. Hence, two different languages(C++ and python) 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 considered to be 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 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 type of divide and conquer sorting algorithm, and has a running time of O(nlog(n)) time. The core of merge sort focuses on dividing the unsorted array into smaller arrays to less than 2 integers, 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 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. Therefore, for this paper we would investigate top-down merge sort, bottom-up merge sort and Tim merge sort, each with 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.

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

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.

Top-down and bottom-up merge sort use different structure-function. As shown in figure 1, 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 comparison and integers 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 Tim-sort merge sort**

Tim-sort 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 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 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.

**2.3 Merge Sort Structure:**

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:**

* 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 one 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.

**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 2.1: Average Running Time for Top-down, Bottom up and Tim-sort Merge Sort in C++**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Merge Sort**  **(C++)** | **Average Running Time (Nano Seconds)** | | | | |
| **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Top-Down** | 26,284.5 | 30,901.1 | 34,336.7 | 32,212.1 | 34.391.1 |
| **Bottom Up** | 25,587.9 | 28,097.5 | 39,657.9 | 31,485.7 | 31,210.5 |
| **Tim Sort** | 25,529.3 | 27,083.4 | 40,500.2 | 35,771.7 | 36,032.2 |

**Table 2.2: Average Running Time for Top-down, Bottom up and Tim-sort Merge Sort in Python**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Merge Sort**  **(Python)** | **Average Running Time (Nano Seconds)** | | | | |
| **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Top-Down** | 622,318.0 | 632,163.2 | 695,007.9 | 727,236.2 | 639,646.2 |
| **Bottom Up** | 491,676.1 | 526,617.0 | 653,825.9 | 632,825.9 | 615,642.9 |
| **Tim Sort** | 594,351.7 | 668,217.0 | 876,070.2 | 1,056,105.5 | 842,427.7 |

**Table 2.3: Average Running Time for Top-down, Bottom up and Tim-sort Merge Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Merge Sort** | **Running Time (Nano Seconds)** | | | |
| **Average**  **(C++)** | **Average**  **(Python)** | **Standard Deviation (C++)** | **Standard Deviation (Python)** |
| **Top-Down** | 31,685.1 | 663,274.5 | 3,395.0 | 45,556.0 |
| **Bottom Up** | 31,207.9 | 583,998.6 | 5,308.0 | 70,745.0 |
| **Tim Sort** | 32,983.4 | 807,434.4 | 6,402.0 | 182,042.0 |

**2.4.1 Data Analysis for Merge sort**

From the result in data collection, the running time for bottom-up merge sort is often faster than both top-down and Tim-sort in C++ and python. However, the top-down merge sort has the lowest standard deviation in both languages, thus making the algorithm more stable. On the other hand, the running time taken for all random sets in all versions of merge sort has a more significant amount of time compared to the best and worst-case sets.

Chart, line chart

Description automatically generated

Overall, results suggest the bottom-up merge sort has the shortest running time, followed by the top-down merge sort with 1.53% slower in C++ or 13.6% slower in python, and Tim-sort merge sort with 5.7% slower in C++ and 38.3% slower in python.

**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 second shortest running time but has the smallest standard deviation among all other versions of merge sort in both languages. While, bottom-up Merge Sort has a short running time, but the has a second shortest standard deviation.

Both versions of merge sort have the same 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 has an additional “if” function to ensure array size is larger than 2 before returning but would increases the running time for each rotation/branch. However, although the additional illustration would create a small impact on a small-scale set of data(such as the data set used for this investigation), 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 2:line 5, the function uses a min function to calculate and compare the lowest value between the endpoint of the array, or 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 similar data 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 may 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. Whilst, Christian Rinderknecht argues top-down merge sort average cost is lower than bottom-up merge sort. Highlighting certain computer operations or languages should perform better in top-down merge sort.

**2.5.2 Evaluation on Tim-Sort Merge Sort**

Between top-down and bottom-up merge sort, Tim sort has the 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 includes the insertion search within the Tim-sort 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 40%-50% 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.

**Table 1.3 Number of times Tim-Sort Condition is Achieved**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Number of Rotation Tim-sort Condition Fulfilled** | | | | | |
|  | **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Tim-Sort** | 11 | 7 | 1 | 3 | 3 |

The testing result from table 2.4 has indicated the random distribution of 100 integers has only met the Tim-sort requirement between 1 to 3 rotation, but best and worst-case meets the requirement over 5 rotation. 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 top-down or bottom-up merge sort. 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 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.

**2.5.2.1 Extension for Tim-Sort**

Tim-Sort adds features into the operating system with Top-Bottom merge sort as its structure operation. However, additional research/investigation could be conducted, such as using Bottom-up merge sort as its operation or changing the requirement for insertion sort across different array sizes. For this investigation, the largest array to perform merge sort is 50 elements in an array, giving less than or equal to 32.3% to perform insertion sort for each rotation. The probability only increases and decreases with the size of the data set, so the requirement to trigger Tim-Sort should be interchangeable to maximize the efficiency of Tim-Sort. Smaller requirement for smaller array size and vice versa.

**Quick Sort**

**3.1 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.

**3.2 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.

**3.2.1 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 only random, first/last element and middle element quick sort would be mainly investigated for this paper.

**3.2.2 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.

**3.3 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 A.

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

**3.4 Quick Sort Data Collection**

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

**Table 3.1: Average Running Time Quick Sort in C++**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Quick Sort (C++)** | **Average Running Time (Nano Seconds)** | | | | |
| **Pivot Point** | **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Last element** | 1,244,233.2 | 1,362,325.0 | 465,535.8 | 528,751.2 | 467,761.4 |
| **Middle Element** | 5,186.7 | 6,289.0 | 13,626.8 | 16,995.9 | 13,406.5 |
| **Randomly** | 100,828.5 | 104,872.0 | 117,227.7 | 119,540.5 | 111,150.0 |

**Table 3.2: Average Running Time Quick Sort in Python**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Quick Sort (Python)** | **Average Running Time (Nano Seconds)** | | | | |
| **Pivot Point** | **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Last element** | 3,450,438.0 | 4,690,482.2 | 427,699.7 | 421,936.2 | 362,577.8 |
| **Middle Element** | 199,140.9 | 238,556.4 | 374,024.7 | 318,073.9 | 354,704.0 |
| **Randomly** | 278,332.6 | 321,756.0 | 382,812.8 | 424,785.2 | 358,804.5 |

**Table 3.3: Average Running Time for Quick Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Quick Sort** | **Running Time (Nano Seconds)** | | | |
| **Average**  **(C++)** | **Average**  **(Python)** | **Standard Deviation (C++)** | **Standard Deviation (Python)** |
| **Last element** | 813,721.3 | 1,870,626.8 | 449,565.3 | 2,055,622.9 |
| **Middle Element** | 11,101.0 | 296,900.0 | 5,113.2 | 75,330.4 |
| **Randomly** | 110,723.7 | 353,298.2 | 7,943.2 | 56,184.0 |

**3.4.1 Data Analysis for Quick sort**

From the result in data collection(table 3.3), choosing the middle element as the pivot point has the shortest running time, followed by randomly and choosing the last element in both C++ and Python language. On the other hand, randomly chosen pivot point has the lowest variance in Python, but choosing the middle element has the lowest variance in C++. I have converted the result in table 3.3 into normal distribution as shown in graphs 2.1 and 2.2 for comparison.

From the data collection, choosing the middle element is typically faster than the other methods, being ten times faster than choosing a random element in C+++, but only 19% faster in python. On the contrarily, choosing the last element to perform poorly in all data set(table 3.1 and 3.2), and has the largest running time, especially in the best and worst data set being around 110 times slower than the average running time for the middle element quick sort in C++. While, choosing a random pivot point in Python yields a similar running time result(table 3.2), but it's 20 times slower in C++(table 3.1). Choosing a random pivot excels only at constancy in Python being 19,146 nanoseconds smaller than choosing the middle pivot point.

Timeline

Description automatically generated with medium confidence

Chart

Description automatically generated

**3.5 Evaluation For Quick Sort**

**3.5.1 Last/First element Pivot Point Evaluation**

From the data collection, choosing the last or first element as the pivot point would often result in the longest-running time among other quick sort versions. This version performs poorly in sorted or partially sorted data, resulted in 10 times slower in Python and 3 times slower in C++(Table 3.1 and 3.2). As mentioned in 3.2, choosing the last/first element as pivot point may divide into O(n-1) and O(1) array size, and make maximum comparison similar to bubble sort. In addition, the large standard deviation for the last element in table 3.3 reflects the lack of constancy in dealing with different types of data, because of the difference between the best and worst data set compared to random data set.

**3.5.2 Middle and Random Pivot Point Evaluation**

On the other hand, choosing a random pivot point or middle pivot point yield the shortest running time is well supported in data collection, but the middle pivot point is typically a bit faster in both C++ and python. However, the variance for these versions varies in different languages. For example, random pivot point has 19,146 nano-second smaller standard deviations compared to the middle pivot point in Python, but 2,830.2 larger standard deviations in C++. This is explained by the random generated module and randomness against the operation to calculate the middle element of an array.

Firstly, the randomly generated module that operates in each code language is different and has non-identical running time and variance. Although computer scientists theorized random pivot point would yield better average running time, they are under the assumption generating random integers running time is similar to a dividing operation(O(1)). However, generating random integers in practices has a long-running time, and is inconsistent in different languages. Calculating the middle pivot point on the other hand is a constant operation, and makes the running time more stable and shorter running time.

Secondly, the chosen random pivot point should have a larger variance than the middle pivot point, because each trial/run would be chosen a different pivot point from the previous. However, the random pivot point for python in each trail run in Appendix B may potentially not be uniformly distributed, but clustered together and cause similar running time. Thus, random pivot point may appear more stable than middle pivot point in Python in table 3.3.

The prove my theory, I compared the running time to generate a random integer and calculate to middle element between 0 to 99 in C++ and Python.

**Table 3.4.1 Compare Random Operation against Calculation Middle Operation**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Running Time(nanoseconds)** | | | | | | | | | | | |
| **C++** | **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Trial 6** | **Trial 7** | **Trial 8** | **Trial 9** | **Trial 10** |
| **Random** | 1961 | 1916 | 1666 | 1208 | 1274 | 1942 | 1105 | 1529 | 1085 | 1202 |
| **Middle** | 157 | 119 | 111 | 138 | 170 | 151 | 135 | 221 | 178 | 110 |
|  |  |  |  |  |  |  |  |  |  |  |
| **Python** | **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Trial 6** | **Trial 7** | **Trial 8** | **Trial 9** | **Trial 10** |
| **Random** | 5873 | 8869 | 15716 | 12299 | 4952 | 5063 | 8244 | 9888 | 9009 | 7131 |
| **Middle** | 4923 | 3073 | 5136 | 3969 | 5162 | 3418 | 3465 | 4288 | 5203 | 4080 |

**Table 3.4.2 Average Compare Random Operation against Calculation Middle Operation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Running Time (Nano Seconds)** | | | |
|  | **Random Element (Average)** | **Middle Element (Average)** | **Random Variance** | **Middle Variance** |
| **C++** | 1488.8 | 149.0 | 129053.1 | 1190.7 |
| **Python** | 8704.4 | 4271.7 | 11294350.0 | 641805.8 |

Table 3.4.1 and 3.4.2 support my hypothesis that random operation has longer running time and variance compared to calculating the middle element. The difference in running and variance would increase the larger the array becomes(more pivot point need).

Overall, the middle pivot point has more stability outweighs the statistical advantages from a random pivot point in practice. The random pivot point may become more applicable than the middle pivot point in future technology development, if the random generation module running time could be equal or less than an operation.

**3.5.3 Random Pivot Point Extension**

As previously mentioned in 3.2.1, a random quick sort pivot point could be configured to further reduce the probability of worst time complexity, running, and improve constancy. For example, a randomly generated integer must be between 1/3 to 2/3 of the array to prevent the pivot point from being either end of the array. Create a similar performance to a medium of 3 and maintain the element of a random pivot, but the additional operation/limitation may also lead to a longer running time.

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.

**4.1 Heap Sort Background information**

Unlike Merge sort or Quick Sort(uses divide and conquer method), Heap Sort is a comparison based sorting algorithm with a tree data structure(see appendix XXXX for reference). Quick Sort’s has an average, best and worst time complexity of O(nlog(n)), and operates by divided the array into sorted and unsorted parts and focus on reducing the unsorted areas. The core concept to perform Heap Sort is converting the data set into a tree data structure, and perform swaps with data to achieve a balance binary tree binary as the final product. This sorting algorithm is designed by John William Joseph Williams in 1964. Please refer appendix 4.1 for a binary tree reference.

**Advantage of Heap Sort**

* Has best, worst and average time complexity of O(nlog(n))
* Heap Sort has consistent performance given it equal performance in best, worst and average time complexity
* Memory uses is minimal compared to other sorting algorithm(in-place sorting algorithm).

**Disadvantage of Heap Sort**

* Additional space and process requires to read the sorted data, as data set is rearranged as a tree type data structure.
* May occur a mistake when sorting multiple equal elements(not a stable sort).

**4.2 Heap Sort Version**

Heap sort is divided into two different parts. Part A or heapify is rearranging the array into a heap data structure. The heap data structure could be either a min-heap or max-heap complete binary tree(depending the order to sorted by). See appendix 4 for an example. In a max heap sort, each integer represents a node(parent) within a tree, and any number smaller than the node is linked to the left and right node(child). A parent has a maximum of two node, and without a child is called the root.

Part B involves repeatedly removing the smallest/largest element of the element(top node) into the array, and perform heap reconstruction until all element is removed.

**4.2.1 Williams and Floyd Constructing Heap Data Structure**

In the same year Williams designed heap sort, Robert W. Floyd has made a small improvement towards heapifying the array. Williams version involves performing shift up and down on all the element, but Floyd heap sort perform shift up for half of the array’s elements(appendix 4.2). Overall, Floyd heap sort remove the process shift down and reducing the number of elements require to perform heapifying the array, making the time complexity to O(n) time.

In additional to Floyd improvement, often Floyd heap sort would often combined with a bottom up approach. From the investigation on Merge Sort Section 1, bottom up approach reduces computer overhead, and statically reduces the number of comparisons in heap sort. Top-down approach on average requires 2nlog(n) + O(n) comparison time, because f or each rotation in part B of Heapsort, after the swap the heapify require two comparison to find minimum node between its children. However, bottom up aims to find the largest children and perform one comparison per level, resulting in nlog(n)+O(1) comparison time.

Hence, a bottom up Floyd heap sort would be one of the investigated heap sort variant.

**4.2.2 Ternary Heapsort and Memory-optimized heapsort**

Each parent in a heapsort binary tree has a maximum of two children node, but Ternary heapsort allows each parent to have three children node. Having three child node would bring the bellowed advantages in heap sort:

* Three comparison per rotation, but one swap required only
* Reduce the number of levels within the tree, so less rotations would be performed.

The process of adding additional node towards the parent is called d-ary heap, and four child node per parent could be implanted into Heapsort(Memory-optimized heapsort). However, implanting additional node per parent increases the complexity of the algorithm, such as calculating the position of child, store the child element location, stored value, etc. Secondary research highlights ternary heapsort statically perform better than binary Heapsort, but additional nodes would reduce the advantage and efficient. Hence, the advantage of additional node need to balance with increase complexity of the algorithm to achieve the shortest running time. For simplicity, ternary heapsort and memory-optimized heapsort(four child node per parent) would be investigated for this paper.

* 1. **Methodology of Heap Sort**

**Part A: Convert the data set into a heap/Heapify**

1. Check the current element array with its child element
   * Start with checking with the middle element(n/2)
   * Child node with current element is calculated by [n\*2+1], [n\*2+2],…..[n\*2+k], where k is the number of child node per parent node.
2. If any of the child node is larger than the current parent node, perform swap with the smallest child node
3. If a swap has occurred in step 2, repeat step 1,2 and 3 for the swapped parent position, until reach to tree’s roots or no swap has occurred.
4. Repeat step 1 and 2 until current parent is the last element, else decrease current element by 1.

**Part B: Reading the heap binary tree**

1. Swap position between the nth element(starts with last element) with the first element.
2. Perform heapify with the first element only with array size – n only
3. Repeat step 1 and 2 until n become less or equal to 0.
   1. **Data Collection for Heap Sort**

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

**Table 4.1: Average Running Time Heap Sort in C++**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Heap Sort (C++)** | **Average Running Time (Nano Seconds)** | | | | |
| **Type** | **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Floyd** | 13,221.4 | 11,333.1 | 13,068.5 | 12,976.2 | 13,000.2 |
| **Ternary** | 13,080.4 | 11,140.8 | 12,089.00 | 12,688.7 | 12,848.4 |
| **Memory** | 11,745.8 | 10,131.7 | 12,192.8 | 11,251.3 | 11,241.3 |

**Table 4.2: Average Running Time Heap Sort in Python**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Heap Sort (Python)** | **Average Running Time (Nano Seconds)** | | | | |
| **Type** | **Best Case** | **Worst Case** | **Random 1** | **Random 2** | **Random 3** |
| **Floyd** | 602,108.7 | 448,853.9 | 521,650.0 | 568,977.4 | 536,118.6 |
| **Ternary** | 568,412.3 | 418,811.4 | 456,301.0 | 453,919.9 | 485,430.3 |
| **Memory** | 488,086.5 | 431,683.8 | 446,102.4 | 451,984.9 | 438,858.7 |

**Table 4.3: Average Running Time for Heap Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Quick Sort** | **Running Time (Nano Seconds)** | | | |
| **Average**  **(C++)** | **Average**  **(Python)** | **Standard Deviation (C++)** | **Standard Deviation (Python)** |
| **Floyd** | 12,719.88 | 535,541.72 | 781.10 | 57,569.95 |
| **Ternary** | 12,369.46 | 476,574.98 | 778.62 | 56,509.80 |
| **Memory** | 11,312.58 | 451,343.26 | 769.05 | 21,910.63 |

* + 1. **Date Analysis for Heap Sort**

From the data collection, Memory heap sort has the shortest running time with 11,312.58ns in C++ and 451,343.25ns in Python, followed by Ternary and Floyd Heap Sort. Table 4.3 indicate each additional child node leads a reduction in running time and standard deviation, but the amount decreased differ for each language. For example, Ternary heap sort has 2.86% reduction from Floyd in Python running time, but only a 11.02% reduction in C++ running time. However, additional node in memory heap sort only leaded to a decrease of 8.6% in C++ and 9.5% in Python from Ternary to Memory. Whilst rate of increased efficiency seems to improve with each additional node for C++, and rate of increased efficiency seems to decrease in Python.

On the other hand, the worst-case data set perform the most efficient among the other data set(table 4.1 and 4.2). Unlike other sorting algorithm, Heap Sort doesn’t directly sort the array, but sorting into a particular structure(Max-heap binary tree) and uses a specific read program to rearrange. Hence, in Heapsort perform the best if data set is similar to worst data set, and perform worst in sorted or partially sorted data set.

Heapsort’s advantages being a constant sorting algorithm is shown in table 4.3. The standard deviation for all heapsort version is smallest compared to merge sort or quick sort. Especially with each increase child node, the standard deviation decreases even further.

**Graph 3.1: Average Running Time for Heap Sort in Normal Distribution for C++**

Running time

Probability



Memory Heap Sort

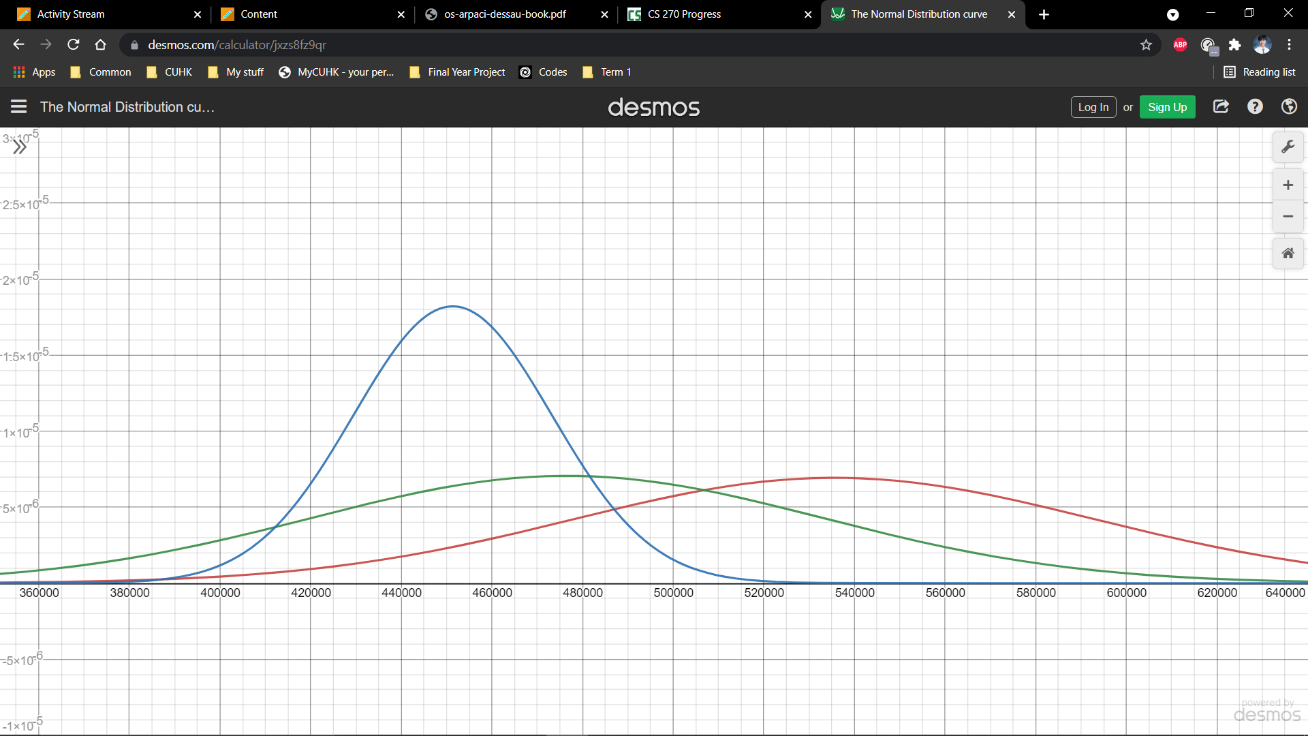
Ternary Heap Sort

Floyd Heap Sort

**Graph 3.2: Average Running Time for Heap Sort in Normal Distribution for Python**

Running time

Probability



Memory Heap Sort

Ternary Heap Sort

Floyd Heap Sort

**4.5 Evaluation for Heap Sort**

**4.5.1 Floyd Heap Sort Evaluation**

The designed for Floyd Heap Sort with only two child per parent performed the worst among the two other version of Heap Sort, having the longest running time and standard deviation. Although, the number of comparisons amongst all heap sort versions remains the same, calling the function(heapify), number of swaps, creating variables, etc, lead to an increase the running time and inconsistence.

Floyd high standard deviation may be explained by one of the reasoning below

* Calling heapify function
  + Requires the system to search/scan the code and relies on the CPU process power to locate.
  + Repeated called function may cause a cache to miss(function not loaded into CPU cache), and requires to access function from main RAM.
  + Each rotate at minimum require to create two or more or more temporary variables to store the position of its child. Creating temporary variables and delating may vary based on dif
  + ferent languages.
  + Require the CPU to register and store the input and output parameters
* Requires call heapify compared to other sorting version of heap sort

**4.5.2 Ternary and Memory Heap Sort**

Both Ternary and Memory heap sort performed more efficiently than Floyd heapsort. As previously mentioned in 4.2.2, Ternary and Memory heapsort has additional child per parent to reduce number of tree levels, but each rotation increase complexity of the algorithm. Hence, deciding the number of child node per parent is needs to balance with the complexity of the algorithm. I hypothesis each increased child node would decreases the running time and variance until it reaches a certain threshold, and any additional child node would lead to a reduce performance. The threshold are unique for different code languages as calling function or creating variables is more efficient for certain languages.

To support this hypothesis, using Floyd heapsort in this data set has a minimum of 7 levels with 50 rotations in the heapify process. Ternary heapsort has 5 level with 32 rotations and Memory heap sort has 5 level with 24 rotations. Ternary and Memory heap tree structure has the same number of tree levels, but only a difference of 8 rotation. An additional child node toward memory Heap Sort would further reduce to 4 tree levels with 19 rotations. Hence, the benefit from reduced number of rotation and levels becomes less significant, and the increase complexity becomes more impactful. In real life application, data set are finite and ideal number of child node may vary.

On the other hand, if the number of child is equal to the array size, heapify require one rotation but perform similar or equal to bubble sort(O(n2)). Hence, to identify the ideal number of child node for different sizes, further testing would be require identify the max threshold of child node per languages.

**\*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.