**Quick Sort**

**8.1 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.

**8.2 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.

**8.2.1 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.

**8.2.2 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.

**8.3 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.

**8.4** **Quick Sort Data Collection**

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

**Table 8.1: 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** | 1,244,238.2 | 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 8.2: 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 | 318,078.9 | 354,704.0 |
| **Randomly** | 278,332.6 | 321,756.0 | 382,812.8 | 424,785.2 | 358,804.5 |

**Table 8.3: 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 | 5,118.2 | 75,330.4 |
| **Randomly** | 110,728.7 | 353,298.2 | 7,948.2 | 56,184.0 |

**8.4.1 Data Analysis for Quick sort**

From the result in data collection(table 8.3), 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 8.3 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 8.1 and 8.2), 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 8.2), but it's 20 times slower in C++(table 8.1). 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

**8.****5 Evaluation For Quick Sort**

**8.5.1** **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 8.1 and 8.2). As mentioned in 8.2, 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 8.3 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.

**8.5.2 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 8.8.

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 8.4.1 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 8.4.2 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 | 129058.1 | 1190.7 |
| **Python** | 8704.4 | 4271.7 | 11294350.0 | 641805.8 |

Table 8.4.1 and 8.4.2 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.

**8.5.3 Random Pivot Point Extension**

As previously mentioned in 8.2.1, 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 | 11,338.1 | 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 | 448,858.9 | 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 | 431,688.8 | 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 | 451,348.26 | 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 451,348.25ns 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 4.8. 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 8.1: Average Running Time for Heap Sort in Normal Distribution for C++**

Running time

Probability



Memory Heap Sort

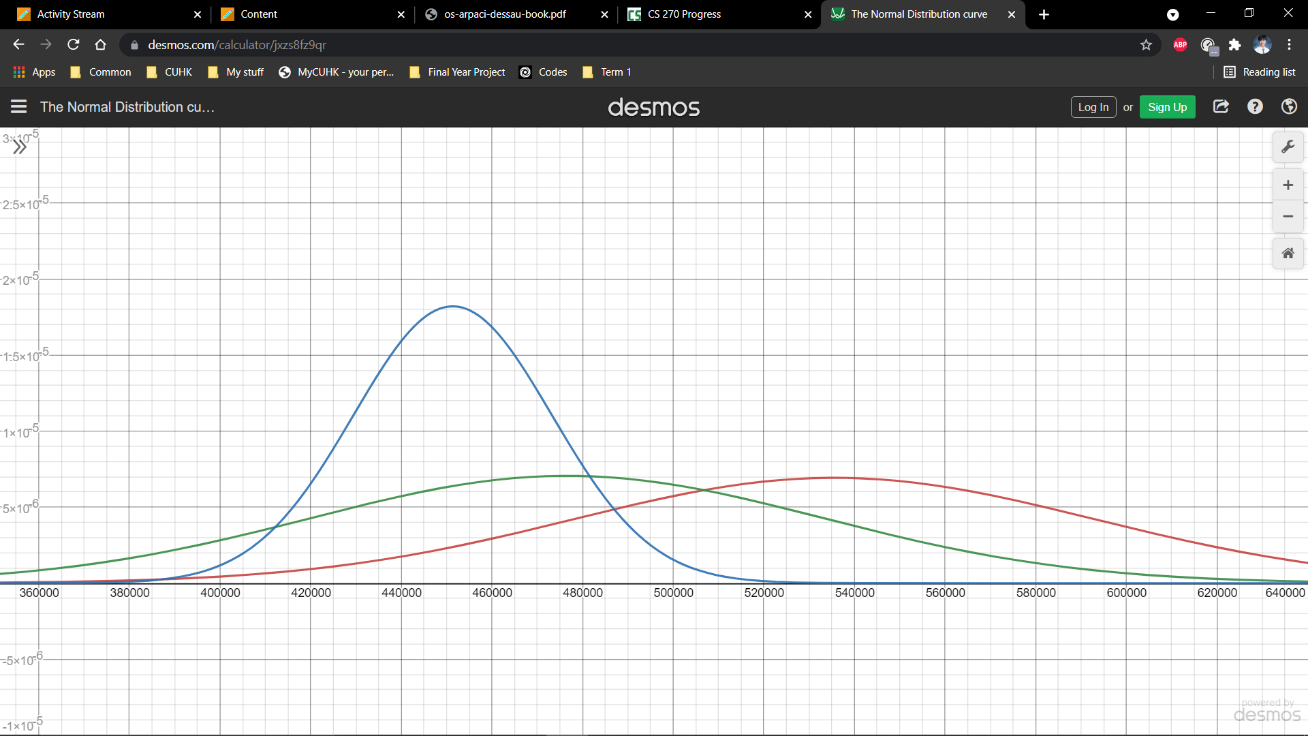
Ternary Heap Sort

Floyd Heap Sort

**Graph 8.2: 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.