**CS 584: Algorithm Design and Analysis**
**Final Project**
**Haomin He Cao**

# Comparison of Sorting Algorithms

## Introduction

In this project, there are three sorting algorithms that I implement in Java programming language, Quicksort, Counting Sort, Timsort. Their complexities are analyzed under the best case and the worst case scenarios. Each algorithm is tested on a variety of input types, which leads us to explore pros and cons of the algorithms according to their computation times and memory used for the calculation.

Quicksort is a divide and conquer algorithm. Compare to Insertion Sort, Quicksort is much more efficient. It picks an element as pivot and partitions the input array around the picked pivot into two smaller sub-arrays. Quicksort can then recursively sort the sub-arrays. Reordering the array recursively, all elements with values less than the pivot come before the pivot, all elements with values greater than the pivot come after it.

Linear time sorting algorithm, counting sort, operates by counting the number of objects that have each distinct key value(like hashing), and using arithmetic on those counts to determine the positions of each key value in the output sequence.

Timsort is derived from merge sort and insertion sort. The algorithm finds subsequences of the data that are already ordered, and uses that knowledge to sort the remainder more efficiently. This is done by merging an identified subsequence, called a run, with existing runs until certain criteria are fulfilled.

## Algorithms - Quicksort

This algorithm achieves sorted array by sorting smaller segments of the array. As mentioned in the introduction, the first step of divide process is picking a pivot element, then divide the array into two subarrays. Its purpose is to rearrange the array, so that all elements before the pivot are less than it and all elements after are greater than or equal to it. The conquer process is sorting the partitioned subarrays recursively with quicksort. From here, we find the pivot for each subarrays until all subarrays contains only one element.

There are plenty of strategies to pick a pivot. In our pseudocode we pick the middle element as our pivot repeatedly. Split the array into left subarray and right subarray.

Variable left pointer points to the low index, right pointer points to the high index of the array. While left pointer is less than pivot, increment it by 1. While right pointer is greater than pivot, decrement it by 1. If the left pointer is less than or equal to the right pointer, we swap them. Then we call quicksort on both left and right partitions repeatedly until we reach the base case, left pointer is greater or equal to right pointer.

*Pseudocode:*

```
QuickSort (int[] array, int left, int right){
       // base case
       if (left >= right) return;
       int i = left;
       int j = right;
       int pivotValue = array[(left + right)/2];  // Pivot is at midpoint
       while (i < j)
           while (array[i] < pivotValue) i++;
           while (pivotValue < array[j]) j--;
           if (i <= j)
               int temp = array[i];
               array[i] = array[j];
               array[j] = temp;
               i++;
               j--;
       QuickSort (array, left, j);
       QuickSort (array, i, right);
   }
```

*Complexity:*
Dividing the array into subarrays less than and greater than the pivot takes O(n) time because the algorithm needs to scan through the array, which has n elements. The worst case would occur when the array is already sorted in increasing or decreasing order. And unfortunately, the partition process always picks the largest or smallest element of the array as pivot, which takes T(n - 1) time to solve the subproblems because there will be (n - 1) recursive calls to create subarrays of size (n -  1) and size 0, and so on so forth.

$$T(n) = T(n - 1) + O(n)$$
$$T(n) = O\left(n^2\right)$$

The best case is we have midpoint of the array as our pivot, which means the problem size is being halved at each recursive call. Since both subarrays have the same length, it takes $2T(\frac{n-1}{2})$ time to solve the subproblems.

$$T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$$

$$T(n) = O(n \log n)$$

Quicksort has a very slow worst-case running time, but a fast best-case running time.

**Algorithms - Counting Sort**

There are three types of arrays that are used in counting sort: INPUT[0, 1, …, n], OUTPUT[0, 1, …, n], and TEMPCOUNTER[0, 1, …, k]. Counting sort assumes that each of the n input elements in an array has key values k. Counting sort starts by going through INPUT array. For each element in INPUT, it goes to the index of TEMPCOUNTER that has the same value as INPUT[i], and increments the value of TEMPCOUNTER[INPUT[i]] by one. The purpose of TEMPCOUNTER is to keep track of how many elements in INPUT there are that have the same value of a particular index in TEMPCOUNTER. In other words, the values in TEMPCOUNTER correspond to the total number of times that a value in INPUT appears in INPUT. We modify TEMPCOUNTER[i] so it includes the number of elements before it. To do so, replace each TEMPCOUNTER[i] value with TEMPCOUNTER[i] + TEMPCOUNTER[i - 1]. From here we can do some arithmetic to calculate the position of each element in the OUTPUT array.

*Pseudocode:*

```
CountingSort(int[] arr) {
       int maxNum = Arrays.stream(arr).max().getAsInt();
       int minNum = Arrays.stream(arr).min().getAsInt();
       int rangeNum = maxNum - minNum + 1;
       int counter[] = new int[rangeNum];
       int output[] = new int[arr.length];
       for (int i = 0; i < arr.length; i++)
          //calculate the total number of times that a value in INPUT array
           counter[arr[i] - minNum]++;
       for (int i = 1; i < counter.length; i++)
           // add up previous value
           counter[i] += counter[i - 1];
       for (int i = arr.length - 1; i >= 0; i--)
           // calculate the OUTPUT array
           output[counter[arr[i] - minNum] - 1] = arr[i];
           counter[arr[i] - minNum]--;
       for (int i = 0; i < arr.length; i++)
           arr[i] = output[i];
```

*Complexity:*
Looping through INPUT array(n elements) to get TEMPCOUNTER array takes O(n) time. Looping through TEMPCOUNTER to add up previous value takes O(k) time. The third loop iterates through INPUT array to get OUTPUT array also takes O(n) time. Therefore, the counting sort algorithm has the worst and best running time of O(n + k).

**Algorithms - Timsort**
Timsort is a hybrid sorting algorithm, which combines a merge sort and an insertion sort to achieve more optimal sorting. Timsort uses RUN(64) to determine what method to use. If an array has fewer than 64 elements in it, Timsort will execute an insertion sort since it is fast and effective on small arrays. Insertion sort looks at elements one by one, and builds up sorted array by inserting the element at the correct location.

If the array has larger than 64 elements in it, an input array is divided into different subarrays, count of elements inside a subarray is defined as a RUN. With obtained sorted runs in an array, we make sure they are strictly ascending or descending by swapping elements.

Next, timsort performs mergesort to merge the runs together. In order to reduce comparisons, timsort makes assumptions that the different arrays have a pre-existing structure. Making that assumption, it allows timsort to check the first element of each array to see where they belong within the chain of another and quickly merge them. For example, if we have element A[0], we can do a binary search in array B to see if where A[0] belong in array B. If A[0] is the start of array B, then we know array B belongs at the end of array A and viceversa for array B belong at the end of array A. As stated, this is all based on the assumption that a pre-existing internal structure is in place, which is where the performance gain is.

*Pseudocode:*
```
RUN = 64
TimSort(int[] array, int n) {
        // Sort individual subarrays of size RUN
        for (int i = 0; i < n; i += RUN)
            insertionSort(array, i, Math.min((i + RUN - 1), (n - 1)));
        // start merging from size RUN (or 64).
        for (int size = RUN; size < n; size = 2 * size)
            // pick starting point of left sub array
            // After every merge, increase left by 2*size
            for (int left = 0; left < n; left += 2 * size)
                // find ending point of left sub array
                // mid+1 is starting point of right sub array
                int mid = left + size - 1;
                // make sure mid is not out of bound
                if (mid > n)
                    mid = n - 1;
                int right = Math.min((left + 2 * size - 1), (n - 1));
                mergeSort(array, left, mid, right);
    }
insertionSort(int[] array, int left, int right) {
        int currentVal;
        int j;
        for (int i = left + 1; i <= right; i++)
            currentVal = array[i];
            // previous
            j = i - 1;
            // make sure index is greater than zero
            while (j >= 0 && array[j] > currentVal && j >= left)
                array[j + 1] = array[j];
                j--;
            array[j + 1] = currentVal;
    }
mergeSort(int[] array, int leftPoint, int midPoint, int rightPoint) {
        int leftPart = midPoint - leftPoint + 1;
        int rightPart = rightPoint - midPoint;

        int[] left = new int[leftPart];
        int[] right = new int[rightPart];
        for (int count = 0; count < leftPart; count++)
            left[count] = array[leftPoint + count];
        for (int count2 = 0; count2 < rightPart; count2++)
            right[count2] = array[midPoint + 1 + count2];

        int templeft = 0;
        int tempright = 0;
        int target = leftPoint;
        // after comparing, we merge those two arrays in a larger sub array
        while (templeft < leftPart && tempright < rightPart)
            if (left[templeft] <= right[tempright])
                array[target] = left[templeft];
                templeft++;
            else
                array[target] = right[tempright];
                tempright++;
            target++;
        // copy remaining elements
        while (templeft < leftPart)
            array[target] = left[templeft];
            target++;
            templeft++;
        while (tempright < rightPart)
            array[target] = right[tempright];
            target++;
            tempright++;
    }
```

*Complexity:*
The worst case occurs when RUN is a truly random sequence. In this case, mergesort has to go through every element and merge them into one big array, which takes O(n log n). The best case happens when we have a sorted input. The presorting of runs costs just linear number of comparisons and movements incurred by merging are prevented by preliminary searches. Thus, the best case time complexity is O(n).

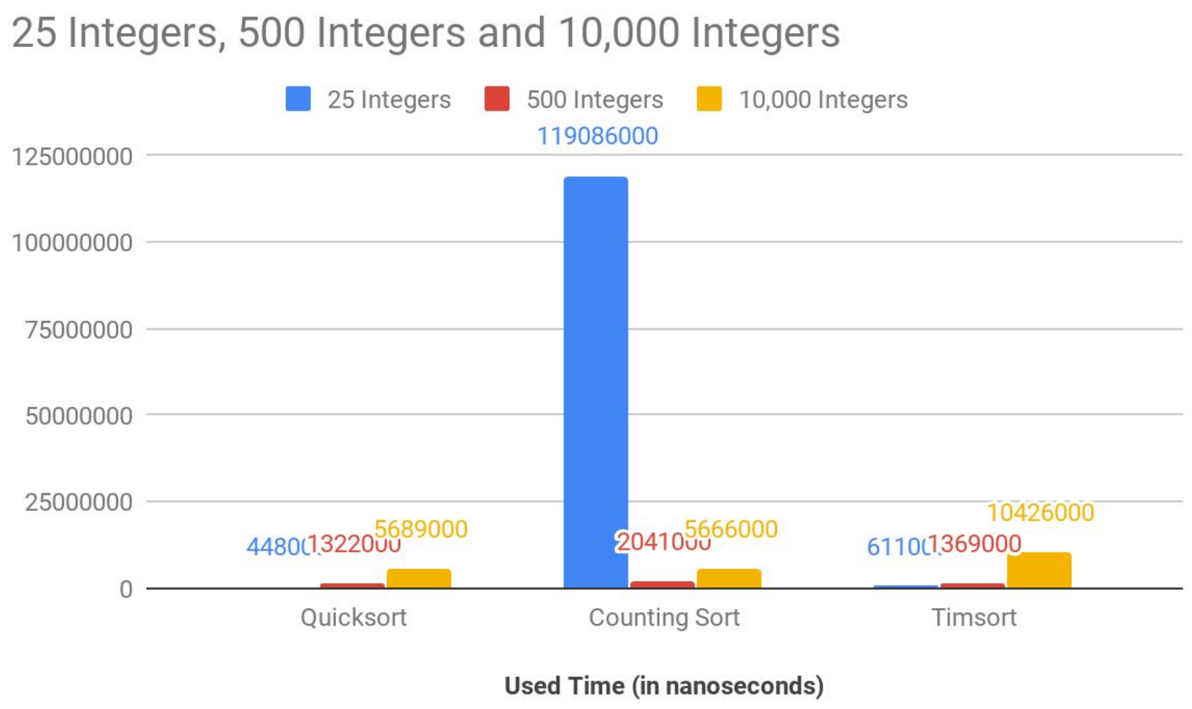
| *Algorithm* | *Best-case* | *Worst-case* |
|---|---|---|
| *Quicksort* | O(n log n) | O( $n^2$ ) |
| *Counting Sort* | O(n + k) | O(n + k) |
| *Timsort* | O(n) | O(n log n) |

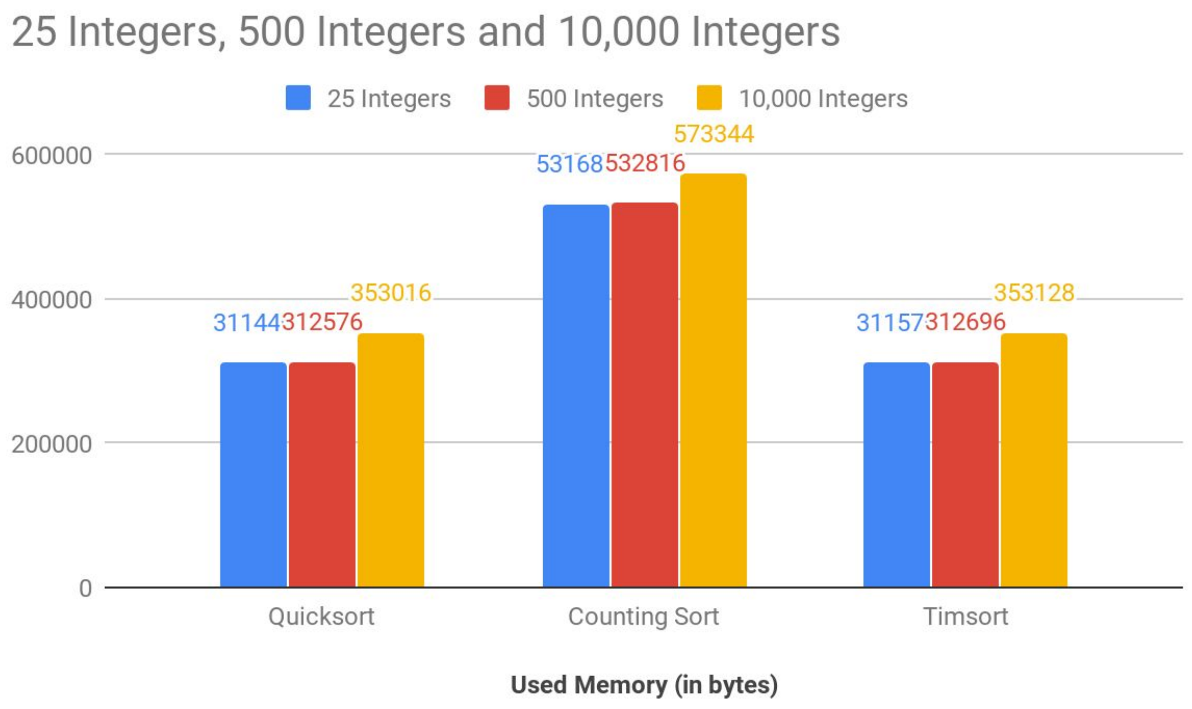
**Experiment**
Running time complexity is an important thing to consider when selecting a sorting algorithm since efficiency is often thought of in terms of speed. Quicksort, counting sort, and timsort has different best and worst complexities. This experiment shows how these algorithms react to various size arrays. We will look at their efficiency based on computation time and memory used.

There are three arrays: small, medium and big size arrays, which contains 25, 500, and 10,000 random generated integers. To take negative numbers into consideration, random generated numbers ranges from -5,000 to 5,000. Recording computation time by utilizing `System.nanoTime()` function, to get most accurate time calculation. Java runtime garbage collector is called after implementation of sorting algorithms, which can give us used memory for the operation (total memory minus freed memory). After calling sorting algorithm(quicksort, counting sort, and timsort), `isSorted()` function is implemented to check whether the specified array is sorted according to natural ordering or not.

*According to Appendix - Commands and Program Results:*

| *Algorithm* | | *25 Integers* | *500 Integers* | *10,000 Integers* |
|---|---|---|---|---|
| *Quicksort* | Used Time | 448000.0 | 1322000.0 | 5689000.0 |
| | Used Memory | 311440 | 312576 | 353016 |
| *Counting Sort* | Used Time | 1.19086E8 | 2041000.0 | 5666000.0 |
| | Used Memory | 531680 | 532816 | 573344 |
| *Timsort* | Used Time | 611000.0 | 1369000.0 | 1.0426E7 |
| | Used Memory | 311576 | 312696 | 353128 |





**Discussion**
As you can see from graphs above, for the small size array, quicksort runs the fastest with 448000.0 nanoseconds and used the least amount of memory, 311440 bytes. Conversely, counting sort runs the slowest (119086000 nanoseconds) in small size array. The reason could be the elements in the array ranges from -5,000 to 5,000, and when the algorithm looping through TEMPCOUNTER to add up all the previous values, it actually computes tons of unnecessary arithmetic operations. In small array case, counting sort tries to find only 25 integers out of 10,000, but it couldn't find and sort the integers immediately because it has to go through the process of creating a TEMPCOUNTER and calculating OUTPUT from TEMPCOUNTER, which leads to extreme high cost.

For the big size array, counting sort computes the array the fastest with 5666000 nanoseconds, but it used much more memory compare to other algorithm (573344 bytes). Quicksort used the least amount of space to do the calculation (353016 bytes). On the other hand, timsort spends the longest time to sort the big size array (10426000 nanoseconds). The big size array is randomly generated between -5,000 to 5,000. However, timsort works the best on arrays with pre-existing internal structure. This is the reason why timsort performs poorly with random array. This weakness especially shows in a large array.

In conclusion, counting sort is not good choice for a small array with large range of data. But it works very well on a big array with large range of data. In other words, the ratio of the number of elements in the array and elements value range. Small size array has the ratio of ( $\frac{25}{10,000}$ ) performs poorly compare to the big size array ratio ( $\frac{10,000}{10,000}$ ). Additionally, use timsort on array with sorted subarrays. It works poorly on pure random array.

**Appendix - Commands and Program Results**

1. Quicksort

```
javac QuickSortTest.java
java QuickSortTest
```

```
*** This program sorts an array of random integers between -5000 to
5000 by using Quicksort algorithm.

*** QuickSort Test 1: Small size array. There are 25 integers in the
array.
QuickSort on small size array SUCCESSFULLY performed.
Used time in nanoseconds: 448000.0
Used memory in bytes: 311440

*** QuickSort Test 2: Medium size array. There are 500 integers in
the array.
QuickSort on medium size array SUCCESSFULLY performed.
Used time in nanoseconds: 1322000.0
Used memory in bytes: 312576

*** QuickSort Test 3: Big size array. There are 10,000 integers in
the array.
QuickSort on big size array SUCCESSFULLY performed.
Used time in nanoseconds: 5689000.0
Used memory in bytes: 353016
```

2. Counting Sort

```
javac CountingSortTest.java
java CountingSortTest
```

```
*** This program sorts an array of random integers between -5000 to
5000 by using Counting Sort algorithm.

*** Counting Sort Test 1: Small size array. There are 25 integers in
the array.
Counting Sort on small size array SUCCESSFULLY performed.
Used time in nanoseconds: 1.19086E8
Used memory in bytes: 531680

*** Counting Sort Test 2: Medium size array. There are 500 integers
in the array.
Counting Sort on medium size array SUCCESSFULLY performed.
Used time in nanoseconds: 2041000.0
Used memory in bytes: 532816

*** Counting Sort Test 3: Big size array. There are 10,000 integers
in the array.
Counting Sort on big size array SUCCESSFULLY performed.
Used time in nanoseconds: 5666000.0
Used memory in bytes: 573344
```

3. Timsort

```
javac TimSortTest.java
java TimSortTest
```

```
*** This program sorts an array of random integers between -5000 to
5000 by using Timsort algorithm.

*** TimSort Test 1: Small size array. There are 25 integers in the
array.
TimSort on small size array SUCCESSFULLY performed.
Used time in nanoseconds: 611000.0
Used memory in bytes: 311576

*** TimSort Test 2: Medium size array. There are 500 integers in the
```

```
array.
TimSort on medium size array SUCCESSFULLY performed.
Used time in nanoseconds: 1369000.0
Used memory in bytes: 312696

*** TimSort Test 3: Big size array. There are 10,000 integers in the
array.
TimSort on big size array SUCCESSFULLY performed.
Used time in nanoseconds: 1.0426E7
Used memory in bytes: 353128
```

## References

Awdesh. “Timsort: The Fastest Sorting Algorithm for Real-World Problems.” The DEV Community, 14 Sept. 2018, dev.to/s_awdesh/timsort-fastest-sorting-algorithm-for-real-world-problems--2jhd.

“Counting Sort.” Wikipedia, Wikimedia Foundation, 13 Apr. 2019, en.wikipedia.org/wiki/Counting_sort.

“Counting Sort.” Brilliant Math & Science Wiki, brilliant.org/wiki/counting-sort/.

“Counting Sort.” GeeksforGeeks, 5 Mar. 2019, www.geeksforgeeks.org/counting-sort/.

Hulın, Matej. “Performance analysis of Sorting Algorithms”

“Quicksort.” Wikipedia, Wikimedia Foundation, 29 May 2019, en.wikipedia.org/wiki/Quicksort.

“Quick Sort.” Brilliant Math & Science Wiki, brilliant.org/wiki/quick-sort/.

“QuickSort.” GeeksforGeeks, 19 May 2019, www.geeksforgeeks.org/quick-sort/.

Skerritt, Brandon, and Brandon Skerritt. “Timsort - the Fastest Sorting Algorithm You've Never Heard Of.” Hacker Noon, Hacker Noon, 30 June 2018, hackernoon.com/timsort-the-fastest-sorting-algorithm-youve-never-heard-of-36b28417f399.

“Sorting Algorithms.” Brilliant Math & Science Wiki, brilliant.org/wiki/sorting-algorithms/.

“Timsort.” Wikipedia, Wikimedia Foundation, 27 Mar. 2019, en.wikipedia.org/wiki/Timsort.

“TimSort.” GeeksforGeeks, 2 Apr. 2019, www.geeksforgeeks.org/timsort/.

Tutorialspoint.com. “Data Structures and Algorithms Quick Sort.” www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm.