COMPUTER SCIENCE & ENGINEERING

CSE2046 - ANALYSIS OF ALGORITHMS

HOMEWORK #2 - REPORT

Abbas Göktuğ YILMAZ - 150115061

Enes GARİP - 150116034

Veysi Öz - 150116005

Sorting Algorithms

Insertion Sort

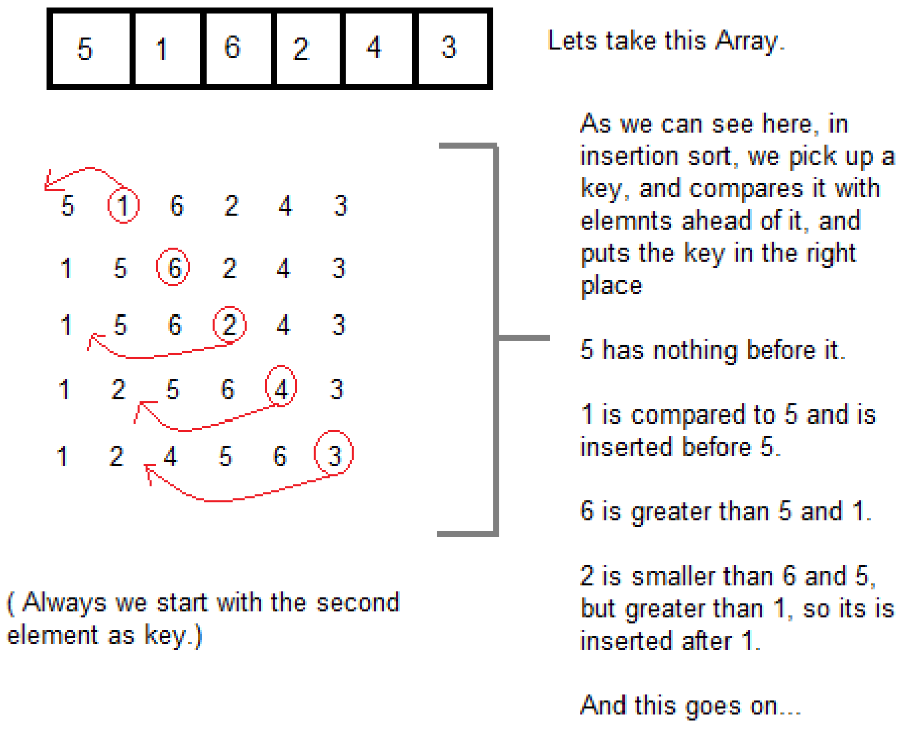
Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:

- Iterate from arr[1] to arr[n] over the array.
- Compare the current element (key) to its predecessor.
- If the key element is smaller than its predecessor, compare it to the elements before.

Move the greater elements one position up to make space for the swapped element.



Lets take this Array.

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

(Always we start with the second element as key.)

Pseudocode

```
Algorithm insertion_sort(A[0...n-1])
        for i = 1 to n-1
                key ← A[i]
                j ← i - 1
                while j >= 0 and A[j] > key
                        A[j+1] ← A[j]
                        j ← j - 1
                end while
                A[j+1] ← key
end for
```

Time Complexity

The Best Case

The best case of Insertion Sort Algorithm is that the given input
sequence is already sorted. It gives the linear running time complexity O(n). During each iteration, only one comparison is done.

$$c_{op} = 1$$

$$C_{best}(n) = \sum_{i=1}^{n-1} = n - 1$$

$$T(n) = c_{op} \times C_{best}(n) \ = \ 1 \times (n-1)$$

$$T(n) \ = \ n - 1 \ \varepsilon \ O(n)$$

The Worst Case

The worst case of Insertion Sort Algorithm is that the given input
array is sorted by descending order. The set of all worst-case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases, every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n2)).

$$c_{op} = 1$$

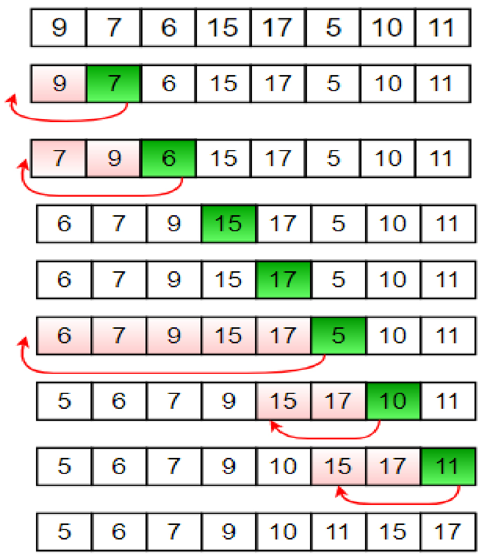
$$C_{best}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} = \frac{n^2+n}{2}$$

$$T(n) = c_{op} \times C_{best}(n) \ = \ 1 \times \frac{n^2+n}{2}$$

$$T(n) \ = \frac{n^2+n}{2} \ \varepsilon \ O(n^2)$$

Binary Insertion Sort

Steps of Binary Insertion Sort are similar to Insertion Sort. But to insert a new element into a sorted subarray, we use binary search algorithm to find the suitable position for it, instead of iterating all elements backward from i to 0 position.



Pseudocode

```
Algorithm insertion_sort(A[0...n-1])
      for i = 1 to n-1
            key ← A[i]
            ##find position to instert A[j] using binary search##
            j ← i - 1
            while j >= 0 and A[j] > key
                  A[j+1] ← A[j]
                  j ← j - 1
            end while
            A[j+1] ← key
end for
```

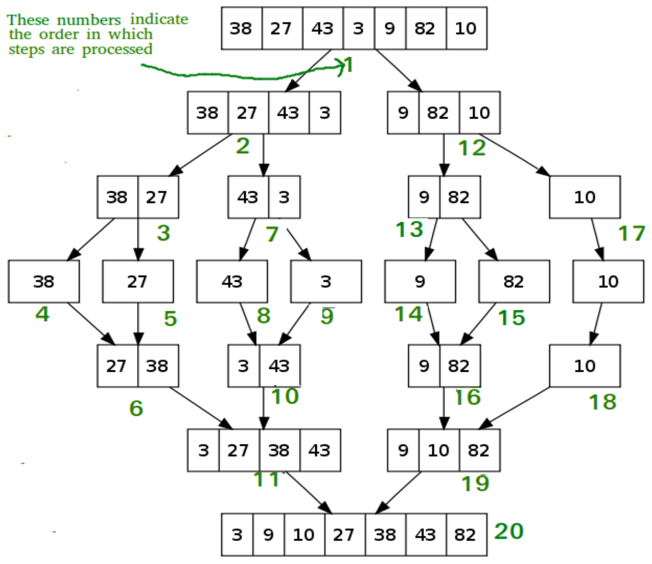
Time Complexity

Binary insertion sort employs a binary search to determine the correct location to insert new elements, and therefore performs [log2 n] comparisons in the worst case, which is O(n log n). The algorithm as a whole still has a running time of O(n2) on average because of the series of swaps required for each insertion.

Merge Sort

Merge sort is a method by which you break an unsorted array down into two smaller halves and so forth until you have a bunch of arrays of only two items. You sort each one, then merge each with another two-item array, then you merge the four-item arrays and so on all the way back up to a fully sorted array.

The operation of the algorithm on the list 38, 27, 43, 3, 9, 82, 10 is illustrated in Figure below



Pseudocode

```
Algorithm mergesort(A[0..n-1])
// A is an array that will be sorted
// B is an array that is left-hand part of array A
// C is an array that is right-hand part of array A
      If n>1
            B ← A[0..n/2-1]
            C ← A[n/2..n-1]
            mergesort(B)
            mergesort(C)
            merge(B, C, A)
Algorithm merge(left[0..p-1], right[0..q-1], A[0..p+q-1])
// Merges two sorted array into one sorted array
      i ← 0
      j ← 0
      k ← 0
      while i < p and j < q
            If left[i] ≤ right[j]
                  A[k] ← left[i]
                  i ← i + 1
            Else
                  A[k] ← right[j]
                  j ← j + 1
            k ← k + 1
      if i = p
            A[k..p+q-1] ← C[j..q-1]
      else
            A[k..p+q-1] ← B[i..p-1]
```

Time Complexity

Best Case

Merge sort's best case is when the largest element of one sorted sub-list is smaller than the first element of its opposing sub-list, for every merge step that occurs. Only one element from the opposing list is compared, which reduces the number of comparisons in each merge step to N/2.

$$T(n) = 2T(n/2) + n/2$$
$$T(n) = 2[2T(n/4) + n/4] + n/2 = 4T(n/4) + n/2 + n/2$$
$$T(n) = 4[2(n/8) + n/8] + n/4 + n/2 = 8T(n/8) + n/2 + n/2 + n/2$$

$$Let\ n = 2^k and\ k = log_2 n$$
$$T(2^k) = 2^i T(2^{k-i}) + i2^{k-1}$$

A single element array is already sorted.

$$T(1) = 0 \Rightarrow 2^{k-i} = 1 \Rightarrow k = i$$
$$T(2^k) = 2^k T(1) + k2^{k-1}$$
$$T(2^k) = 0 + k2^{k-1}$$
$$T(n) = (log_2 n)n/2 \in O(nlogn)$$

Worst Case

The worst case scenario for Merge Sort is when, during every merge
step, exactly one value remains in the opposing list; in other words, no comparisons were skipped. This situation occurs when the two largest values in a merge step are contained in opposing lists. When this situation occurs, Merge Sort must continue comparing list elements in each of the opposing lists until the two largest values are compared.

$$T(n) = 2T(n/2) + n - 1$$
$$T(n) = 4T(n/4) + n - 2 + n - 1$$
$$T(n) = 8T(n/8) + n - 4 + n - 2 + n - 1$$

$$Let\ n = 2^k and\ k = log_2 n$$
$$T(2^k) = 2^i T(2^{k-i}) + i2^k - \sum_{t=0}^{i-1} 2^t$$
$$T(2^k) = 2^i T(2^{k-i}) + i2^k - 2^i + 1$$

A single element array is already sorted.

$$T(1) = 0 \Rightarrow 2^{k-i} = 1 \Rightarrow k = i$$
$$T(2^k) = 2^k T(1) + k2^k + 2^k + 1$$
$$T(2^k) = 0 + k2^k + 2^k + 1$$
$$T(n) = log_2 n\,(n+1) + 1 \in O(nlogn)$$

Quick Sort (Pivot is the first element)

Quicksort doesn't swap the pivot into its correct position in that way, but it lies on the hypothesis that each recursive call sorts the sub-array and then merging sorted sub-arrays would provide a completely sorted array: ... pivot←pick() picks a pivot, in your case it's always the first element in V.

Pseudocode

```
procedure quickSort(arr[], low, high)
```

```
    arr = list to be sorted
    low – first element of the array
    high – last element of array
begin
    if (low < high)
    {
       // pivot – pivot element around which array will be
partitioned
        pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);  // call quicksort
recursively to sort sub array before pivot
        quickSort(arr, pivot + 1, high); // call quicksort
recursively to sort sub array after pivot
    }
end procedure
//partition routine selects and places the pivot element into its
proper position that will partition the array.
//Here, the pivot selected is the last element of the array
procedure partition (arr[], low, high)
begin
    // pivot (Element to be placed at right position)
    pivot = arr[high];
     i = (low - 1)  // Index of smaller element
    for j = low to high
    {
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
end procedure
```

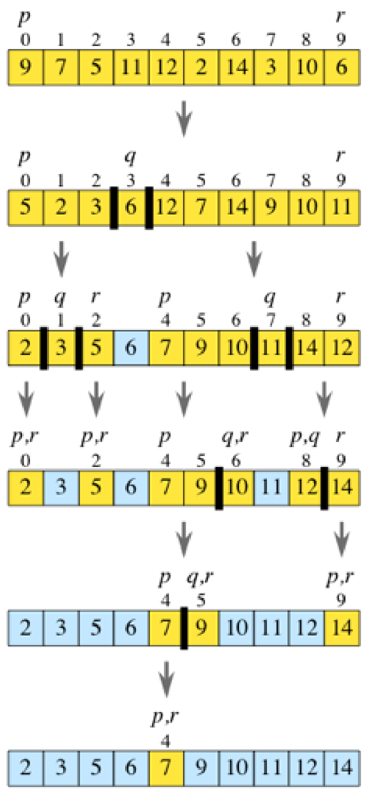
Time Complexity

The worst case time complexity of a typical implementation of QuickSort is O(n2). The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot.

Quick Sort (Pivot is the median of three)

In computer science quickselect is an algorithm to select the nth smallest element of an array, based on the quicksort algorithm.

Quicksort with median-of-three partitioning functions nearly the same as normal quicksort with the only difference being how the pivot item is selected.



Pseudocode

```
Algorithm medianOfThree(A[0..n-1], left, right, key)
    if left + 10 > right
    insertionSort(A)
    return
pivot ← findPivot(A, left, right)
if pivot = A[key] return
partionPoint ← findPartition(A, left, right, pivot)
if key <= partitionPoint
    medianOfThree(A, left, partitionPoint - 1, key)
if key > partitionPoint + 1
    medianOfThree(A, partitionPoint + 1, right, key)
Algorithm findPivot(A[0..n-1], left, right)
    mid ← (left + right) / 2
    if A[mid] < A[left]
        swap(A[mid], A[left])
    If A[right] < A[left]
        swap(A[right], A[left])
If A[right] < A[mid]
        swap(A[right], A[mid])
// Place the pivot position right-1
swap(A[mid], A[right-1])
return A[right-1]
Algorithm partition(A[n..n-1], left, right, pivot)
    l ← left, r ← right
    while(true)
```

```
          while(A[++l] < pivot);
          while(A[--r] > pivot);
          if( l>=r) break
          swap(A[l], A[r])
      swap(A[l], A[right-1])
      return l
```

Time Complexity

The worst case for Quicksort using median-of-3 method is when the
selected pivot reduces the problem size by the smallest possible amount. By avoiding the farthest elements in the list to choose our partition, we can minimize our deviation from the best/average case run time of O(nlogn). But, the worst case is O(n^2).

Heap Sort

Heap sort is a comparison sorting technique based on Binary Heap data structure. To understand the heap sort algorithm, we need to know what Binary Heap structure is.

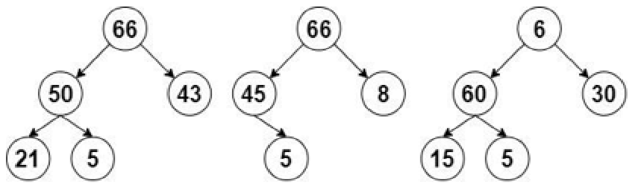
A Binary Heap is a complete binary tree in which every level except possibly the last, is completely filled, and all nodes are as far left as possible. A binary tree is defined a special tree with two additional constraints:

- Shape property: As we mention it, a binary heap is a completely binary tree;that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

- Heap property: t he key stored in each node is either greater than or equal to

(≥) or less than or equal to (≤) the keys in the node's children, according to
some total order.

Heaps where the parent key is greater than or equal to the child keys are called
max-heap.



For example, consider the trees of figure as shown on the left side. The first tree is a heap, but not the second and the third. Because the second tree violates the shape property, and the third three violates heap property with key 6.

Figure: Illustration of the definition of heap:
only the left is a heap.

The Heapsort algorithm involves preparing the list by first turning it into a max heap. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap. This repeats until the range of considered values is one value in length.
The steps are:

- Call the buildMaxHeap() function on the list. Also referred to as heapify(), this builds a heap from a list in O(n) operations.
- Swap the first element of the list with the final element. Decrease the considered range of the list by one.

- Call the siftDown() function on the list to sift the new first element to its appropriate index in the heap.
- Go to step (2) unless the considered range of the list is one element.

The buildMaxHeap() operation is run once, and is O(*n*) in performance. The siftDown() function is O(log *n*), and is called *n* times. Therefore, the performance of this algorithm is O(*n* + *n* log *n*) = O(*n* log *n*).

Pseudocode

```
Algorithm maxHeapSort(A[0..n-1])
    buildHeap(A[0..n-1])
    for i ← n-1 to 0
        swap(A[0], A[1])
        heapify([A[0..n-1], i, 0)
Algorithm buildHeap(A[0..1])
    for i ← n/2-1 to 0
        heapify(A[0..i], n, i)
Algorithm heapify(A[0..n-1], n, i)
    max ← i
    left ← 2*i + 1
    right ← 2*i + 2
if (left < n and A[left] > A[max])
    max ← left
if (right < n and A[right] > A[max])
    max ← right
if (max != i )
    swap(A[i], A[max])
    heapify(A[0..n-1], n, max)
```

Time Complexity

The buildMaxHeap() operation is run once, and is O(n) in performance. The siftDown() function is O(log n), and is called n times. Therefore, the performance of this algorithm is O(n +n log n) = O(n log n).

Counting Sort

Counting sort is a sorting technique which is based on the range of input value. It is used to sort elements in linear time. In Counting sort, we maintain an auxiliary array which drastically increases space requirement for the algorithm implementation

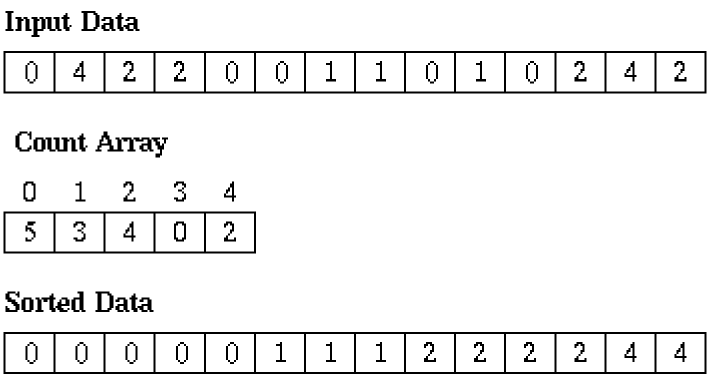
It works just like hashing, first, we calculate the max value in the input array, the array to be sorted. Then we count the number of occurrences of each array element from 0 to length-1 and assign it into the auxiliary array. This array is used again to retrieve the sorted version of the input array

It actually has linear time complexity but we can’t say that it’s the best algorithm because the space complexity is quite high and it is only suitable to use in a scenario where input array element range is close to the size of the array.

Algorithm

- Iterate the input array and find the maximum value present in it.
- Declare a new array of size max+1 with value 0

- Count each and every element in the array and increment its value at the corresponding index in the auxiliary array created
- Find cumulative sum is the auxiliary array we adding curr and prev frequency
- Now the cumulative value actually signifies the actual location of the element in the sorted input array
- Start iterating auxiliary array from 0 to max
- Put 0 at the corresponding index and reduce the count by 1, which will signify the second position of the element if it exists in the input array
- Now transfer array received in the above step in the actual input array



Input Data

0	4	2	2	0	0	1	1	0	1	0	2	4	2

Count Array

0	1	2	3	4
5	3	4	0	2

Sorted Data

0	0	0	0	0	1	1	1	2	2	2	2	4	4

Pseudocode

```
CountingSort(A)
//A[]-- Initial Array to Sort
//Complexity: O(k)
for i = 0 to k do
    c[i] = 0
//Storing Count of each element
//Complexity: O(n)
for j = 0 to n do
    c[A[j]] = c[A[j]] + 1
// Change C[i] such that it contains actual
//position of these elements in output array
////Complexity: O(k)
for i = 1 to k do
    c[i] = c[i] + c[i-1]
//Build Output array from C[i]
//Complexity: O(n)
for j = n-1 downto 0 do
    B[ c[A[j]]-1 ] = A[j]
    c[A[j]] = c[A[j]] - 1
end func
```

Time Complexity

Counting sort takes O(n + k)O(n+k) time and O(n + k)O(n+k) space, where nn is the number of items we're sorting and kk is the number of possible values.

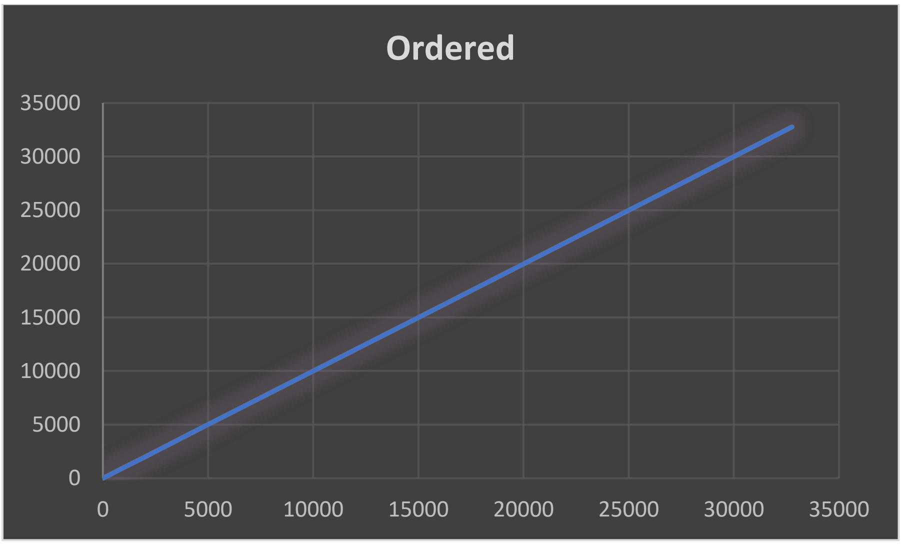
We iterate through the input items twice—once to populate counts and once to fill in the output array. Both iterations are O(n)O(n) time. Additionally, we iterate through counts once to fill in nextIndex, which is O(k)O(k) time.

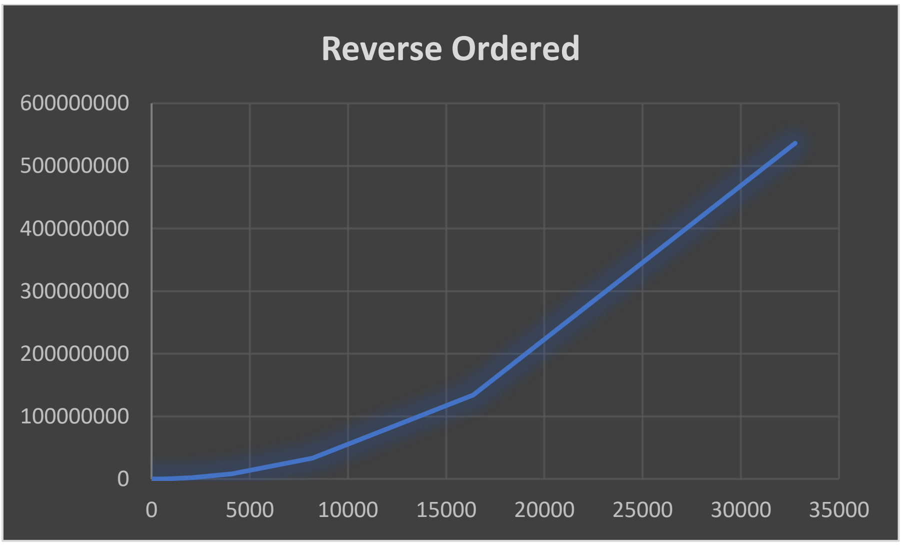
The algorithm allocates three additional arrays: one for counts, one for nextIndex, and one for the output. The first two are O(k)O(k) space and the final one is O(n)O(n) space.

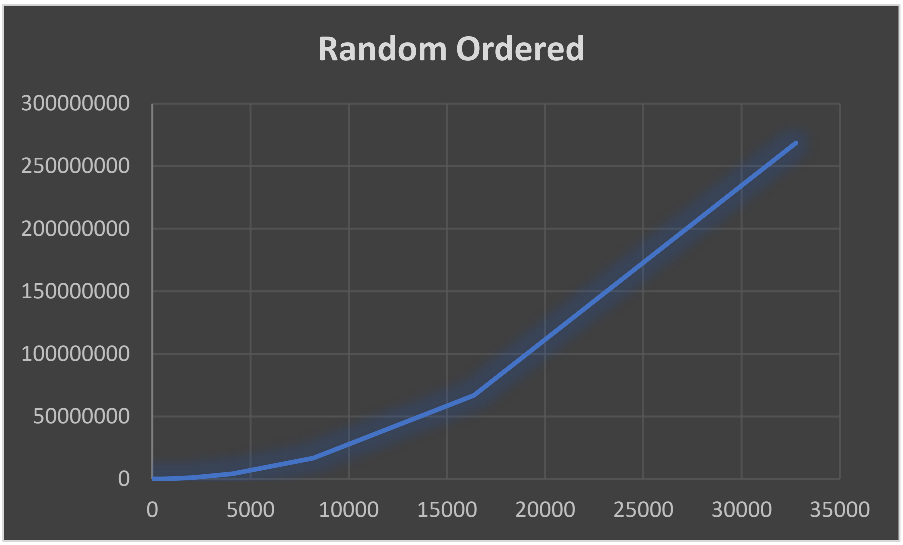
In many cases cases, kk is O(n)O(n) (i.e.: the number of items to be sorted is not asymptotically different than the number of values those items can take on. Because of this, counting sort is often said to be O(n)O(n) time and space.

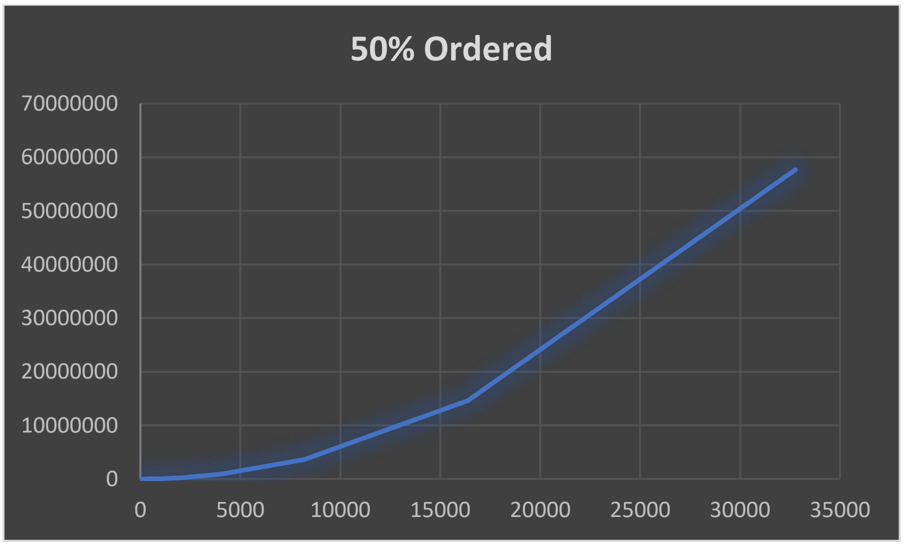
Analysis of Outputs

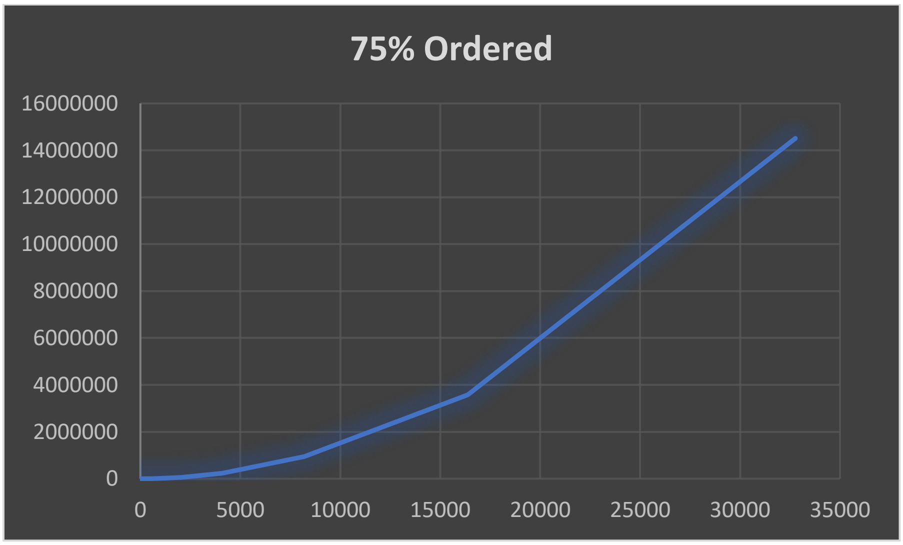
Insertion Sort



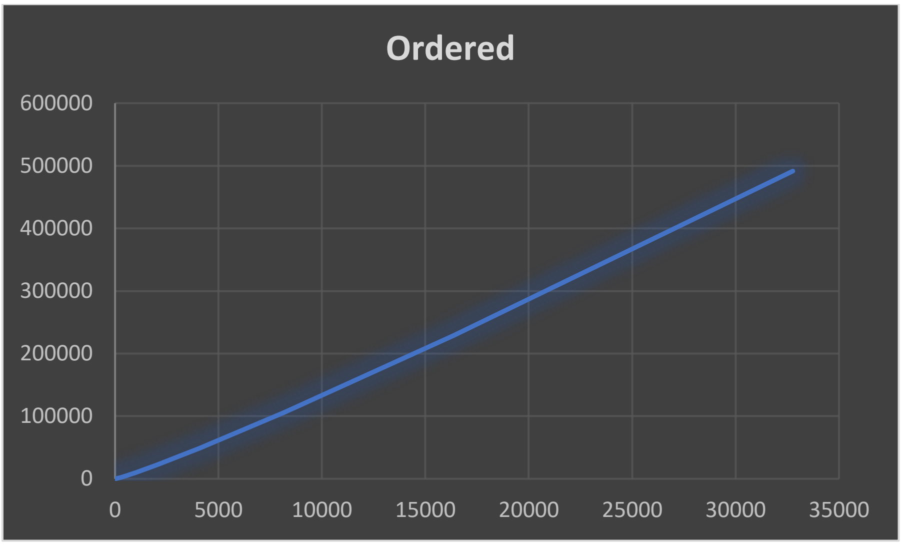


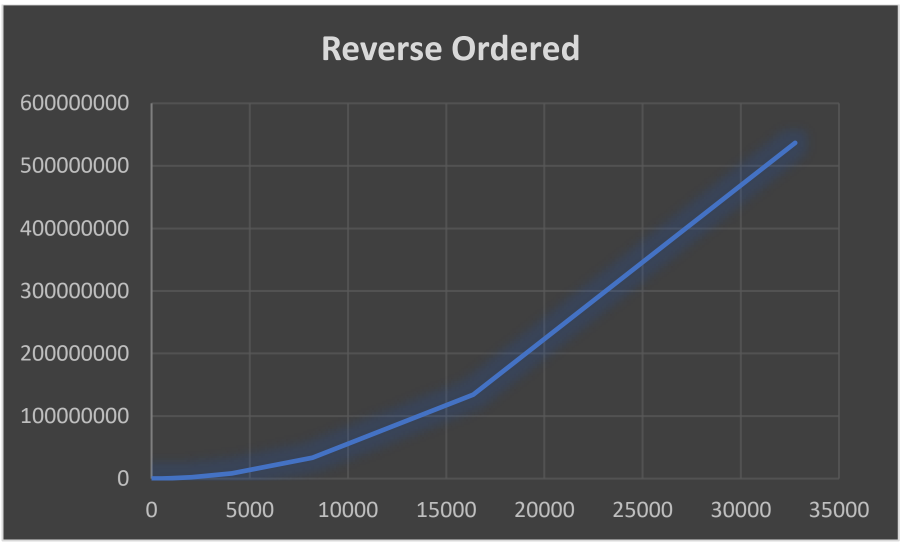

Random Ordered
300000000
250000000
200000000
150000000
100000000
50000000
0
0
5000
10000
15000
20000
25000
30000
35000

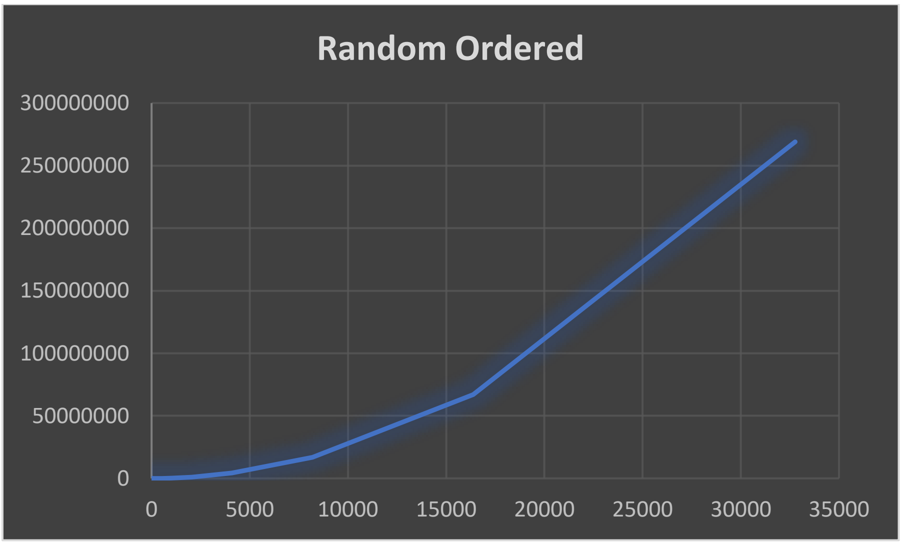

50% Ordered
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
0
5000
10000
15000
20000
25000
30000
35000

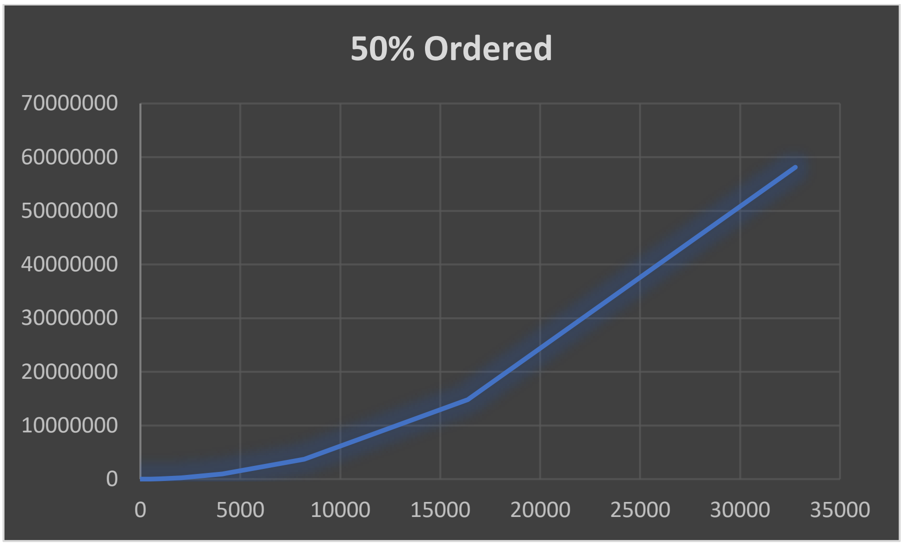

75% Ordered
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
0
5000
10000
15000
20000
25000
30000
35000

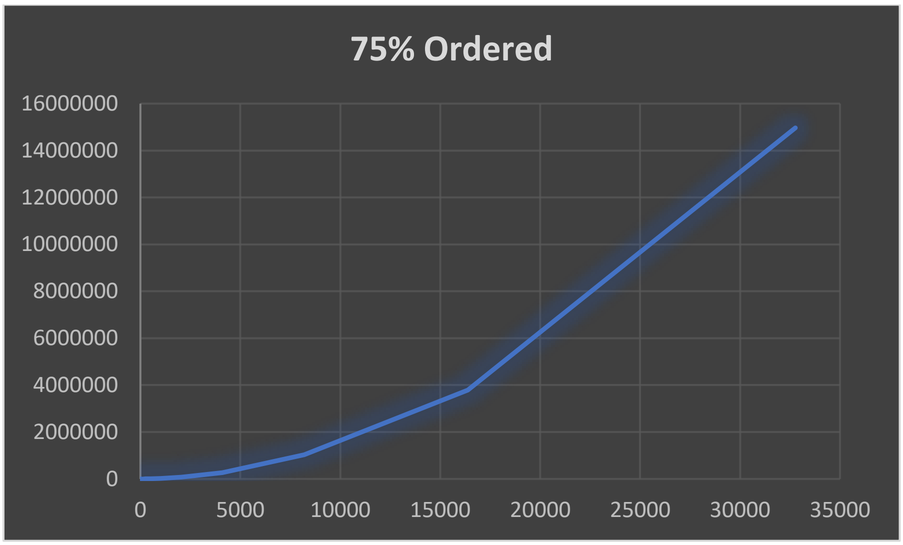
Binary Insertion Sort



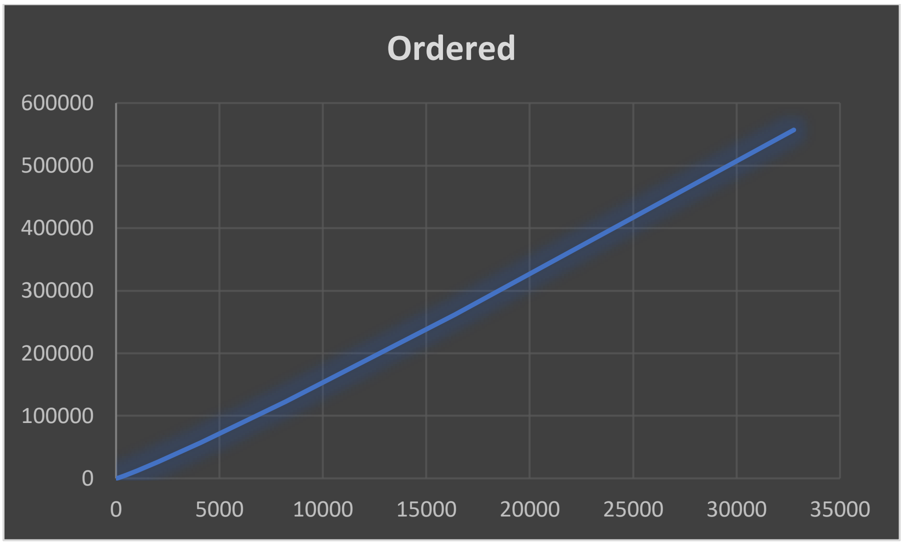


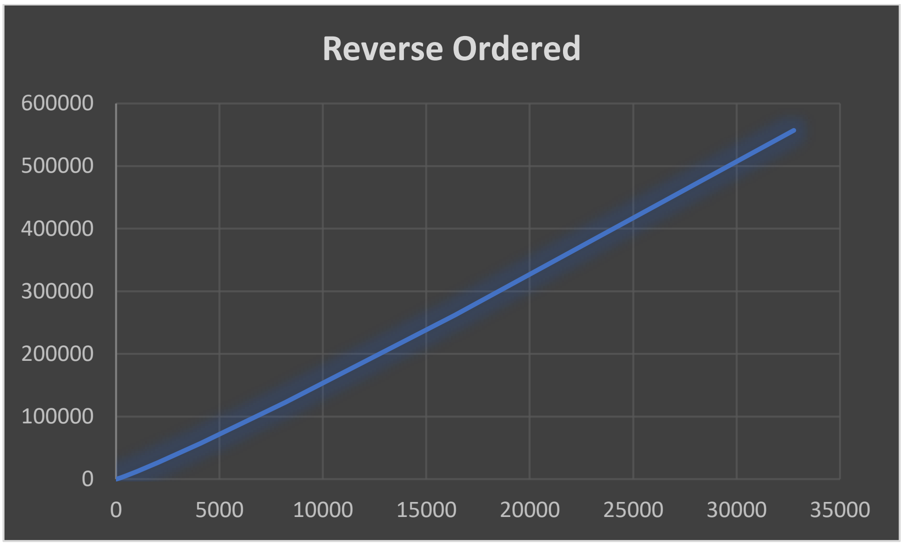


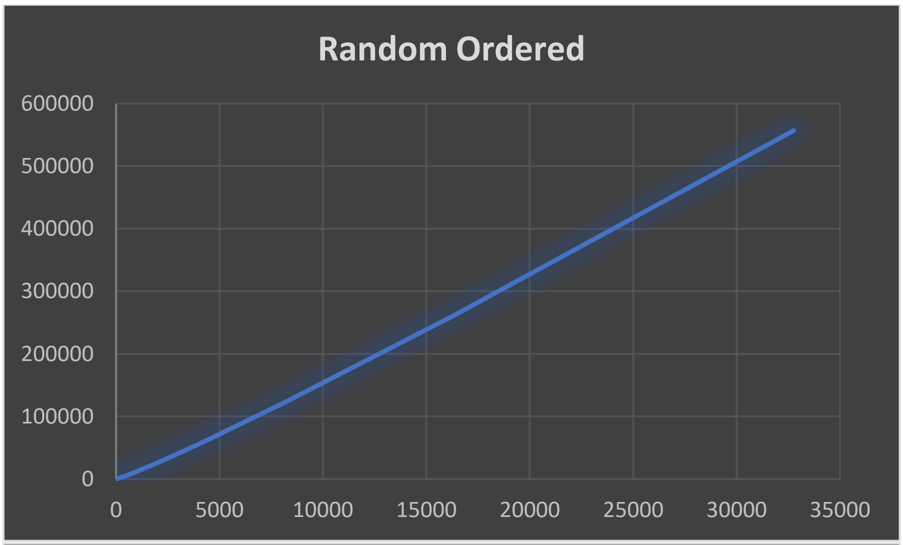

50% Ordered
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
0
5000
10000
15000
20000
25000
30000
35000

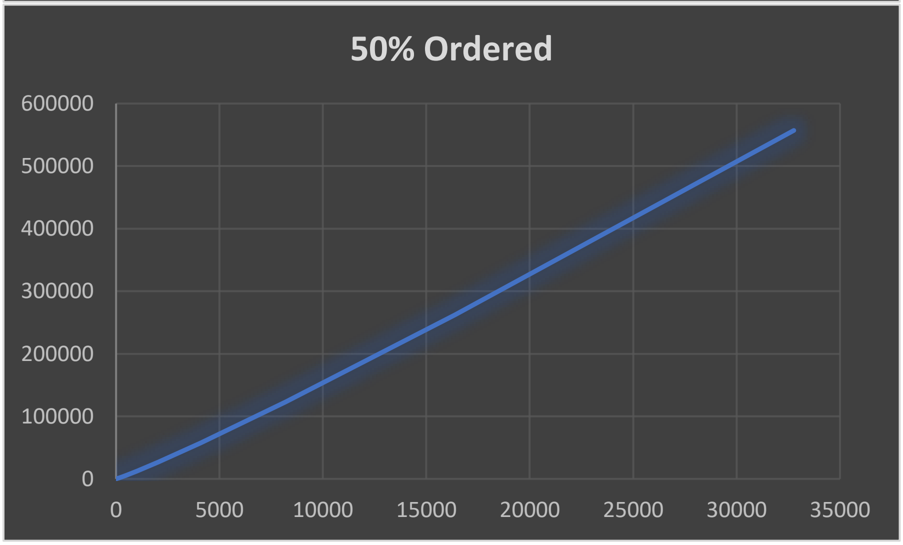

75% Ordered
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
0
5000
10000
15000
20000
25000
30000
35000

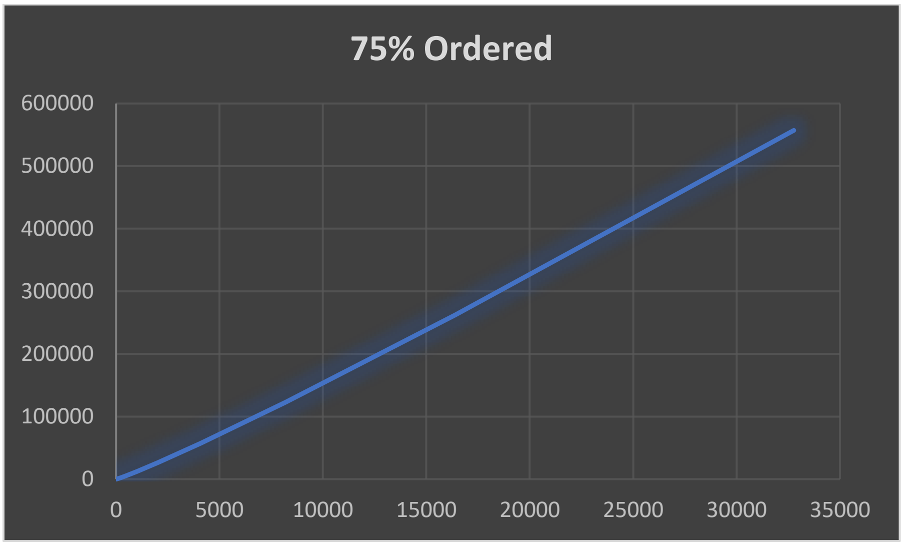
Merge Sort


Ordered
600000
500000
400000
300000
200000
100000
0
0
5000
10000
15000
20000
25000
30000
35000

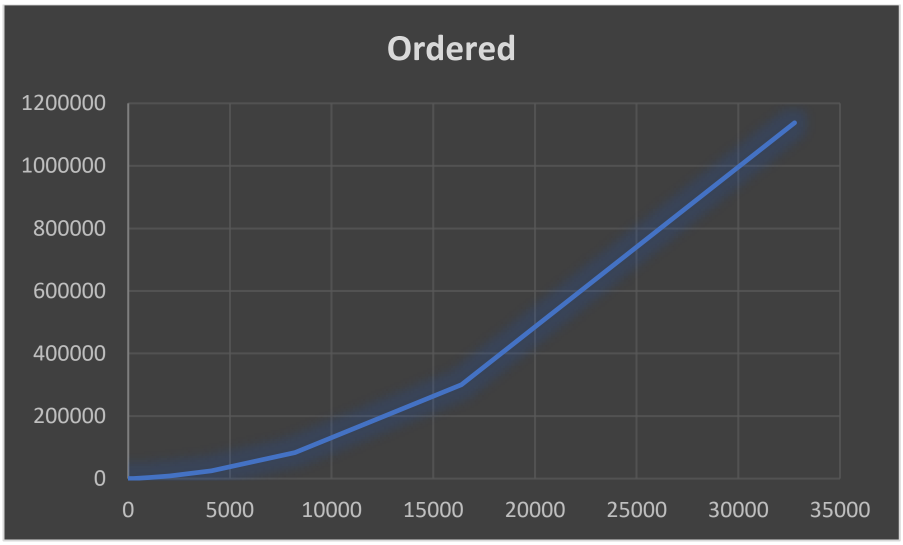

Reverse Ordered
600000
500000
400000
300000
200000
100000
0
0
5000
10000
15000
20000
25000
30000
35000

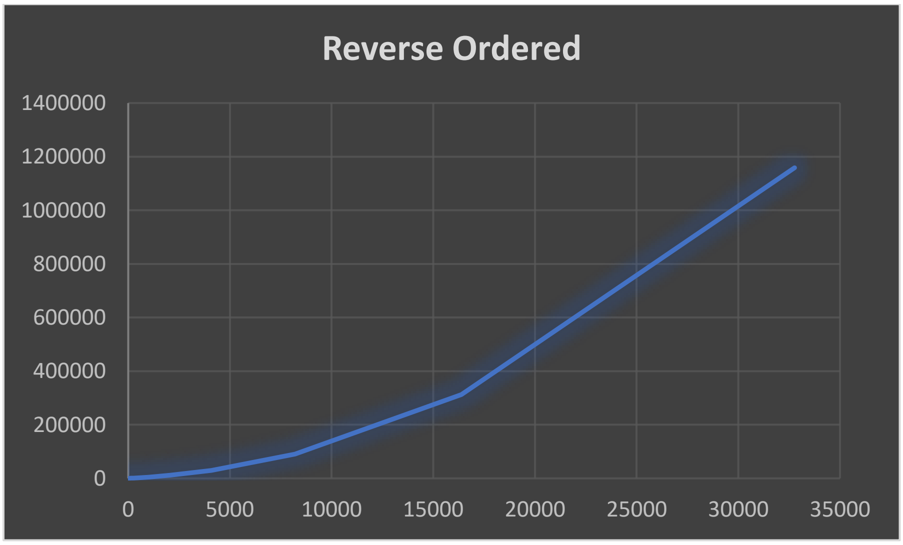

Random Ordered
600000
500000
400000
300000
200000
100000
0
0
5000
10000
15000
20000
25000
30000
35000

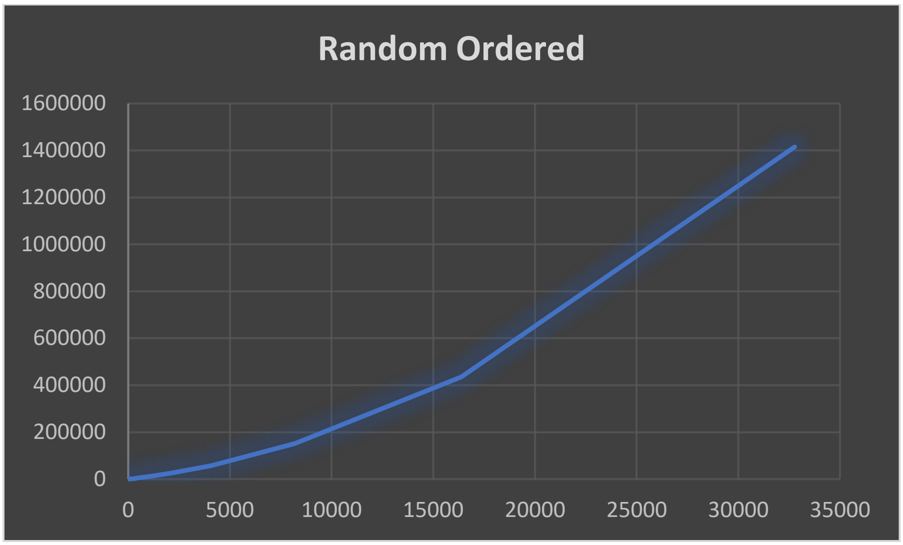

50% Ordered
600000
500000
400000
300000
200000
100000
0
0
5000
10000
15000
20000
25000
30000
35000

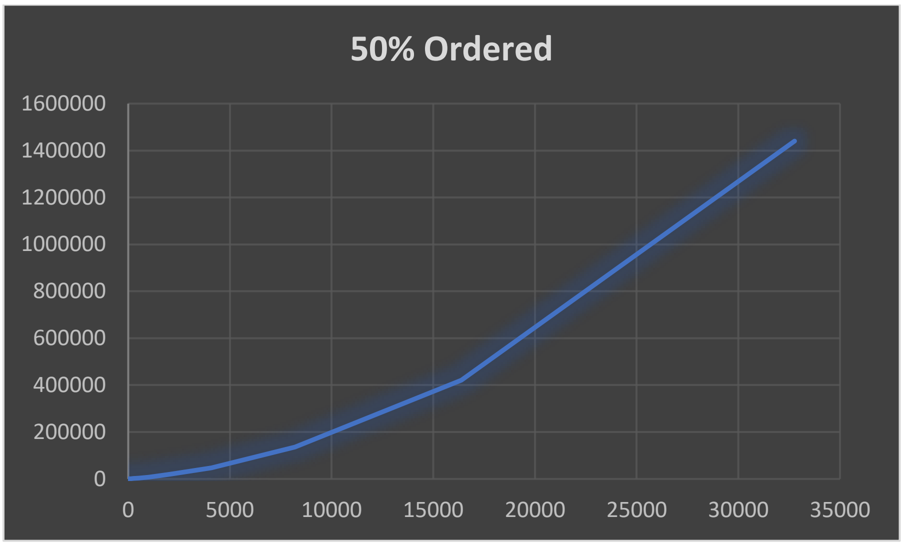


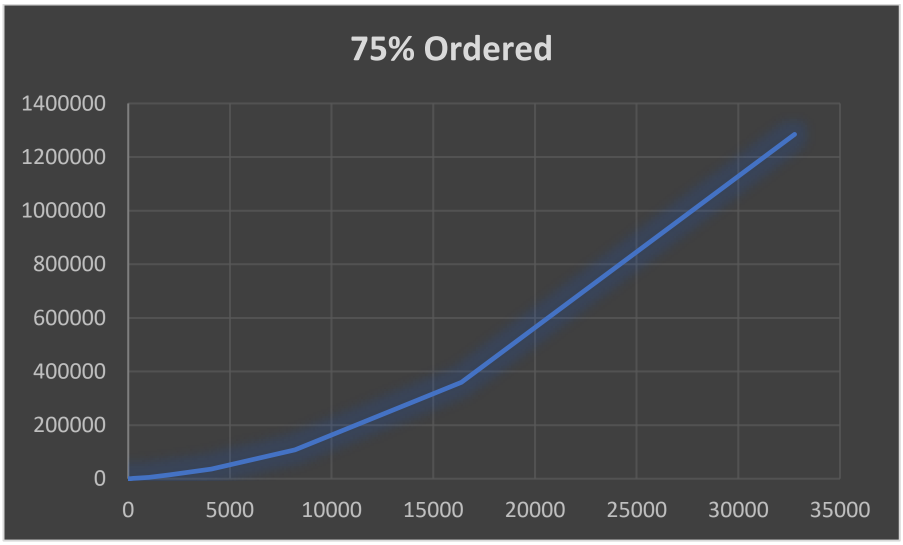
Quick Sort (Pivot is the first element)



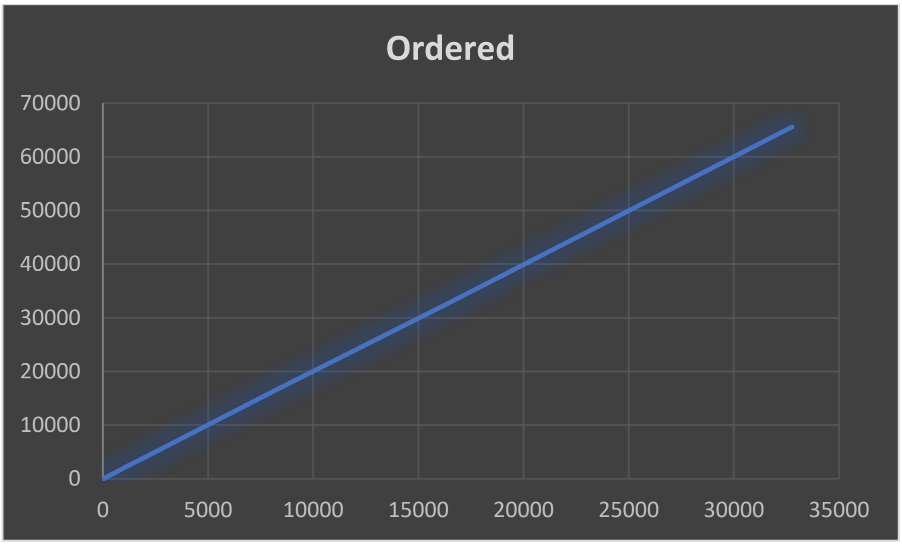


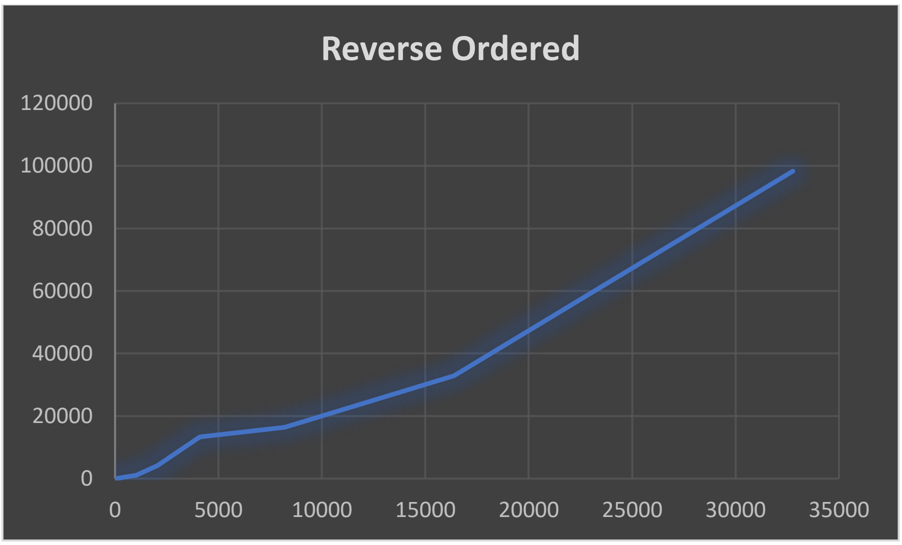

Random Ordered
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
0
5000
10000
15000
20000
25000
30000
35000

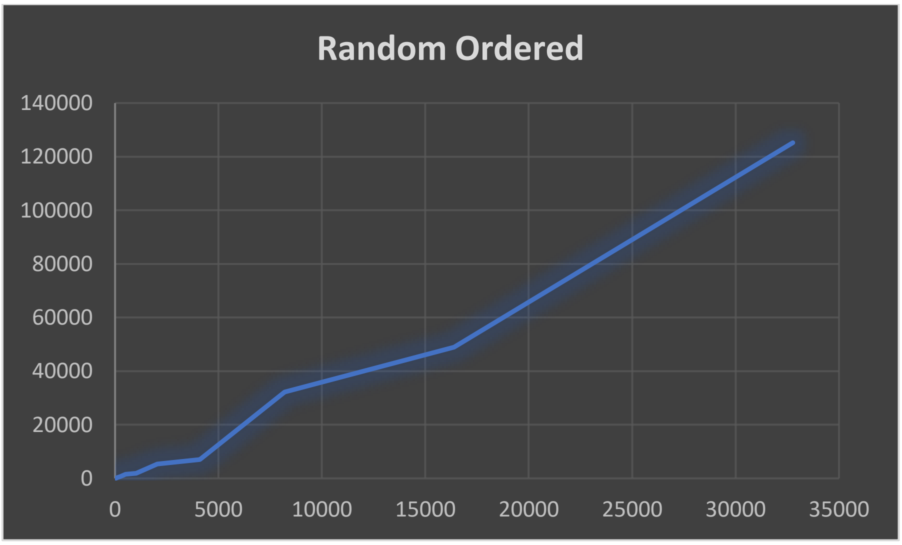

50% Ordered
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
0
5000
10000
15000
20000
25000
30000
35000

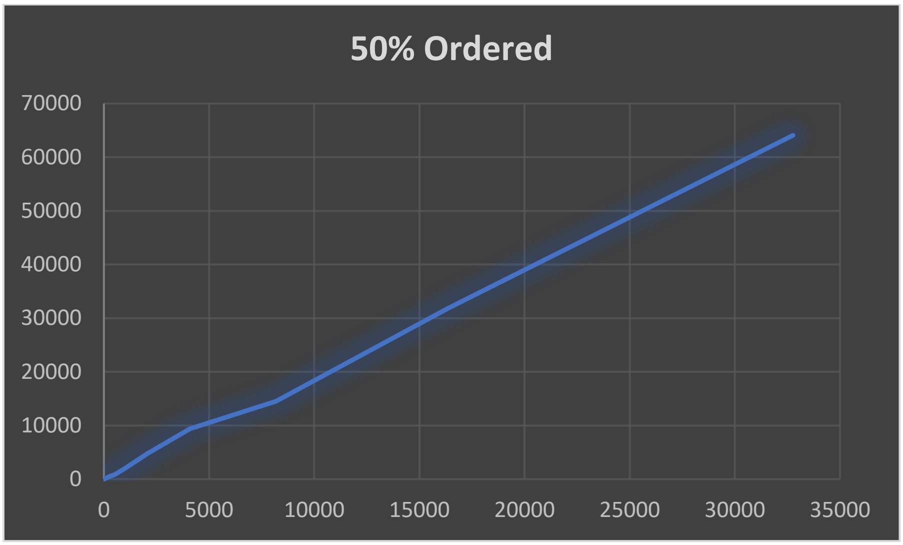

75% Ordered
1400000
1200000
1000000
800000
600000
400000
200000
0
0
5000
10000
15000
20000
25000
30000
35000

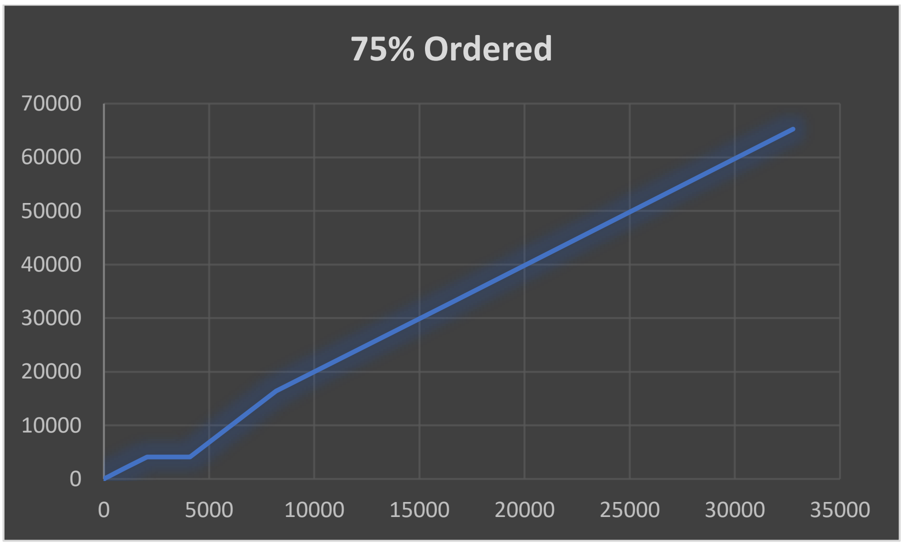
Quick Sort (Pivot is the median of three)



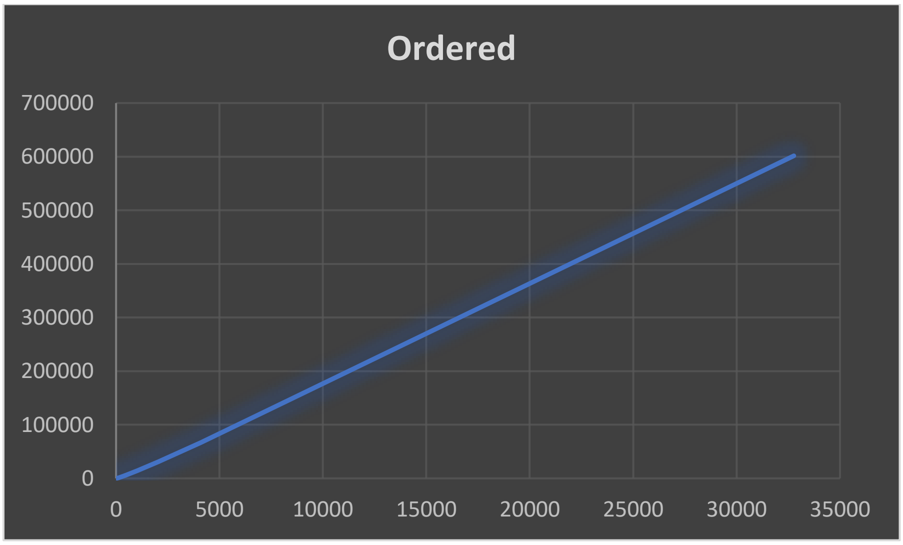


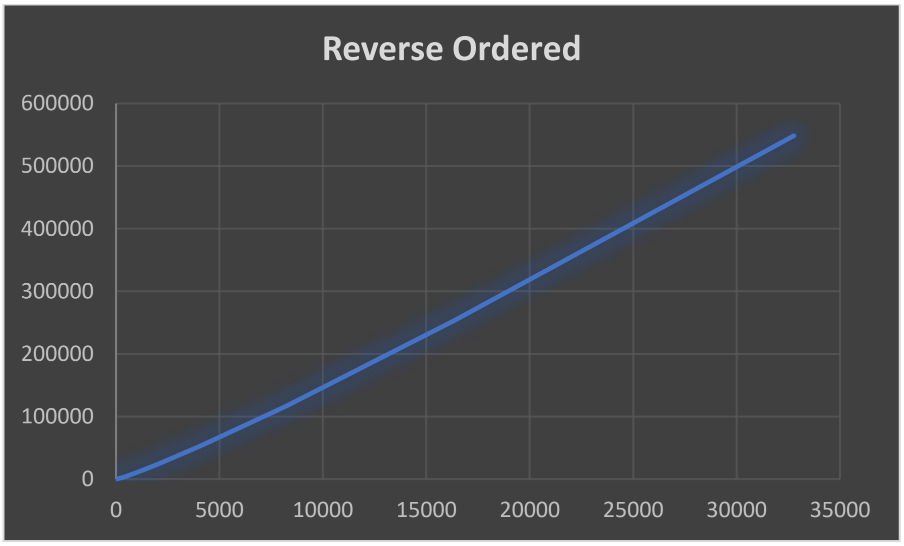


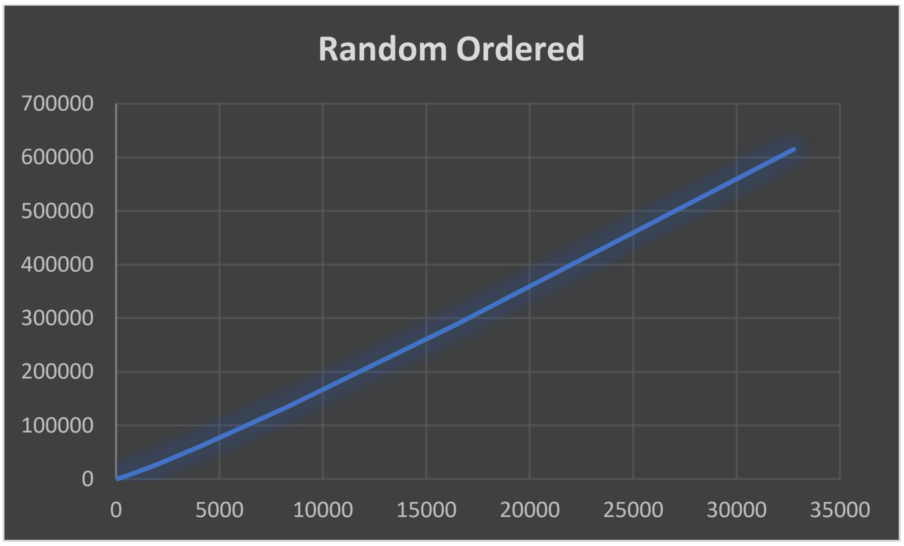

50% Ordered
70000
60000
50000
40000
30000
20000
10000
0
0
5000
10000
15000
20000
25000
30000
35000

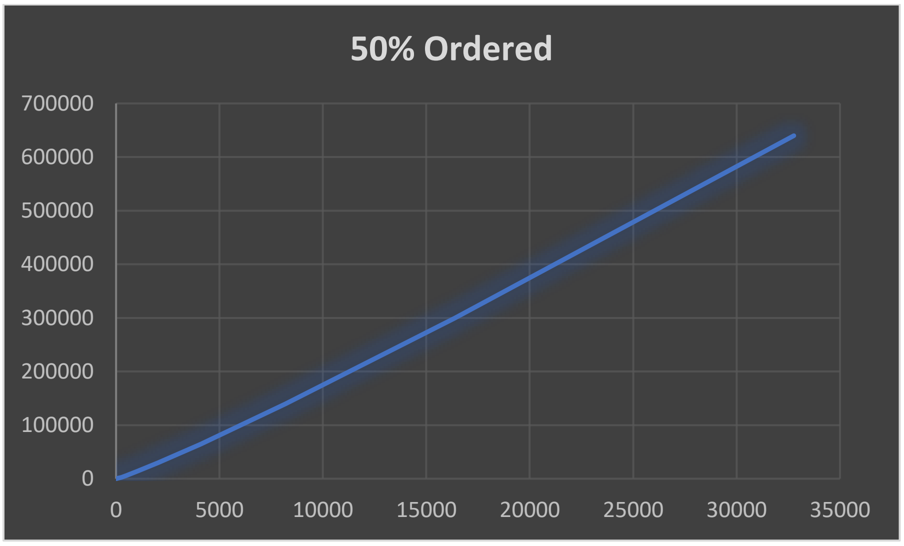

75% Ordered
70000
60000
50000
40000
30000
20000
10000
0
0
5000
10000
15000
20000
25000
30000
35000

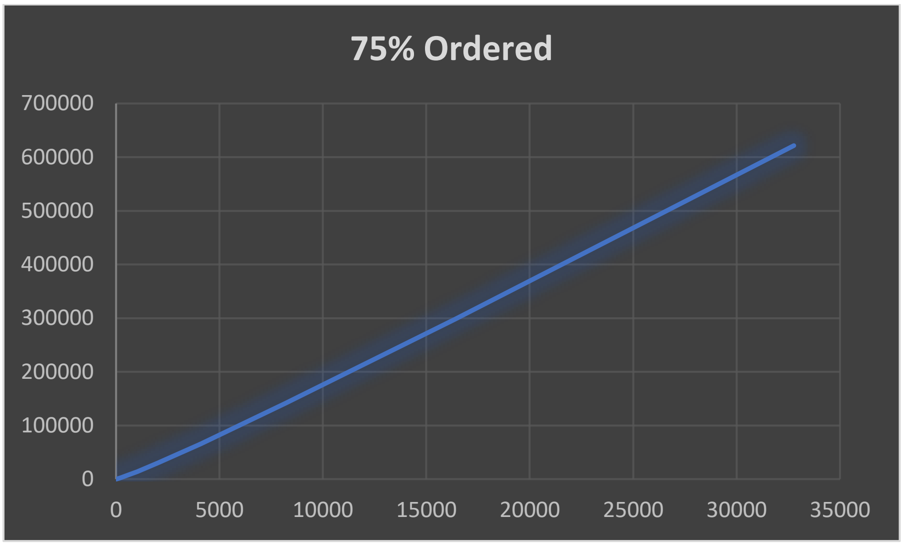
Heap Sort


Ordered
700000
600000
500000
400000
300000
200000
100000
0
0
5000
10000
15000
20000
25000
30000
35000

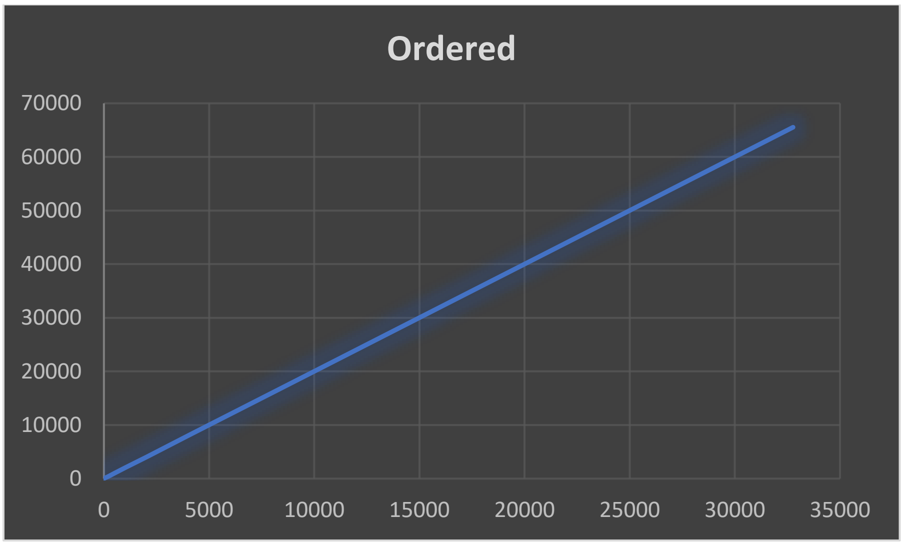

Reverse Ordered
600000
500000
400000
300000
200000
100000
0
0
5000
10000
15000
20000
25000
30000
35000

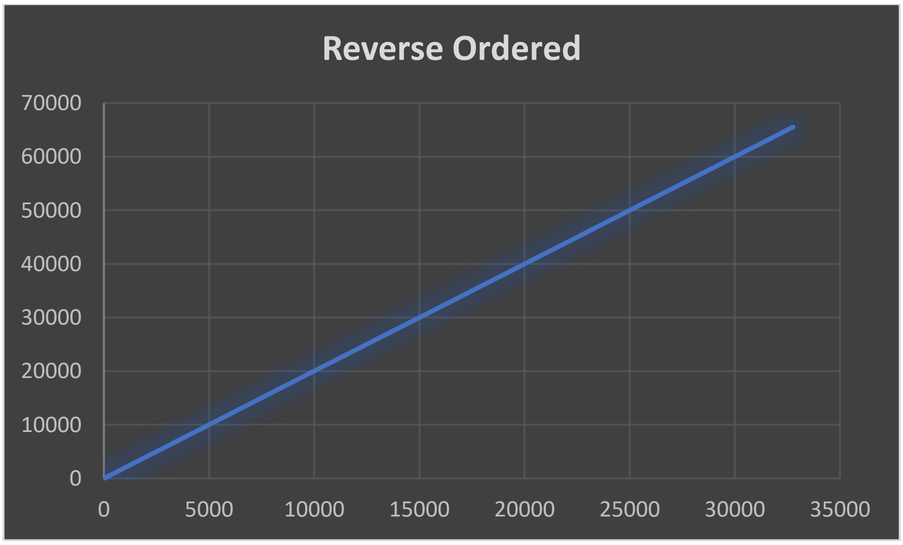

Random Ordered
700000
600000
500000
400000
300000
200000
100000
0
0
5000
10000
15000
20000
25000
30000
35000

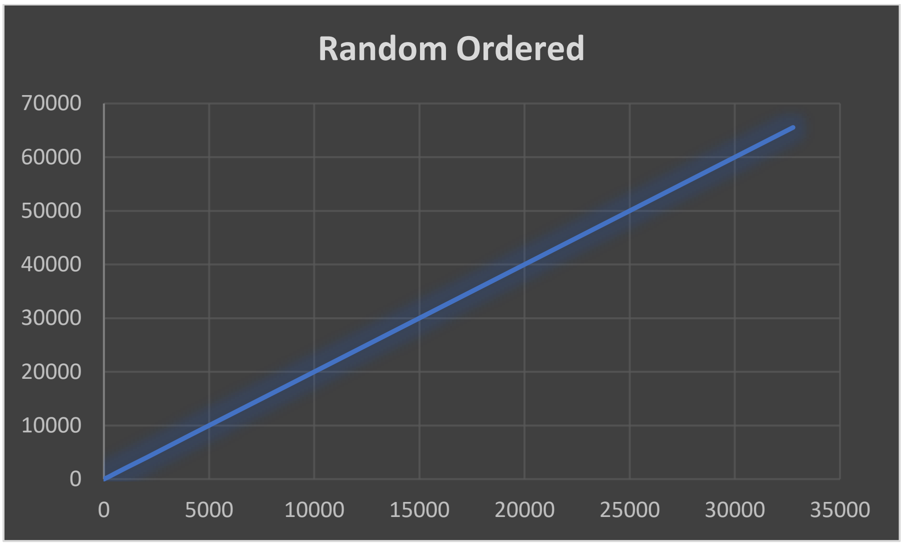

50% Ordered
700000
600000
500000
400000
300000
200000
100000
0
0
5000
10000
15000
20000
25000
30000
35000

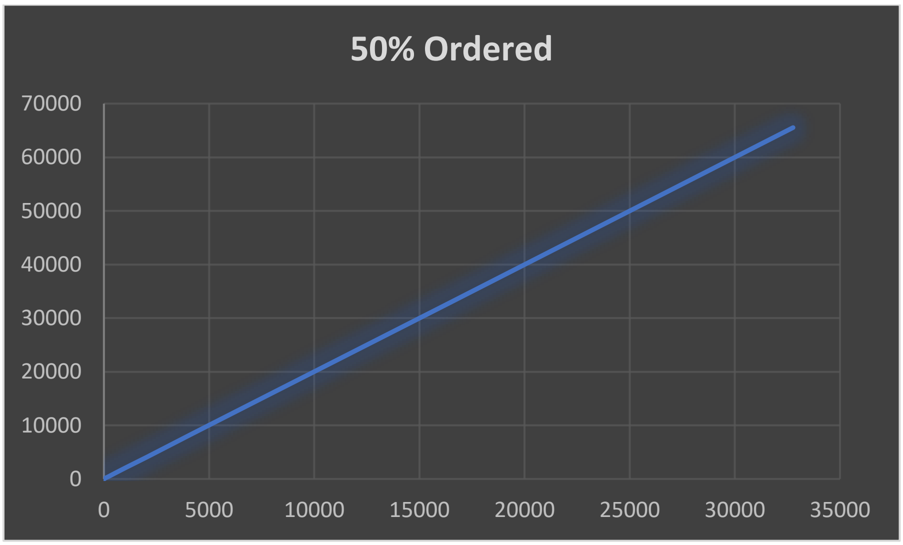


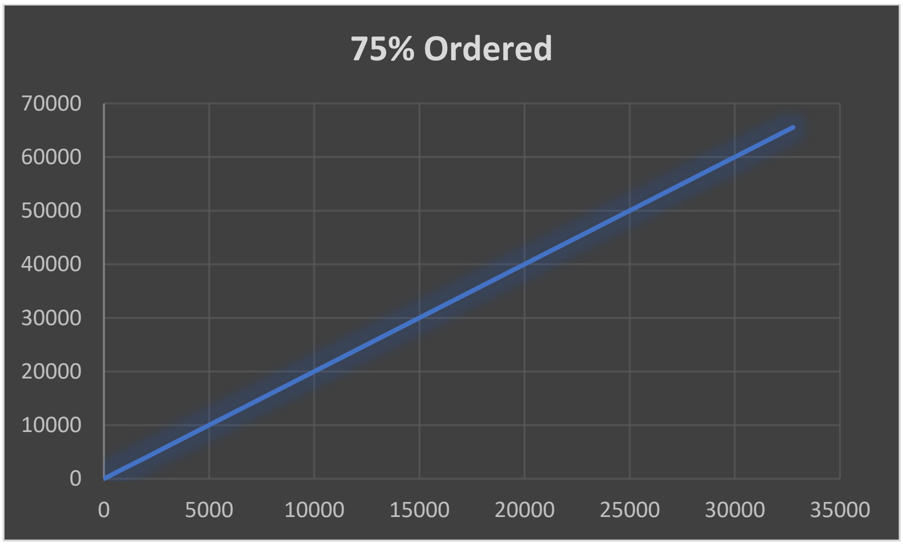
Counting Sort






Random Ordered
70000
60000
50000
40000
30000
20000
10000
0
0
5000
10000
15000
20000
25000
30000
35000


50% Ordered
70000
60000
50000
40000
30000
20000
10000
0
0
5000
10000
15000
20000
25000
30000
35000


75% Ordered
70000
60000
50000
40000
30000
20000
10000
0
0
5000
10000
15000
20000
25000
30000
35000