PERFORMANCE ANALYSIS REPORT ON INTROSORT ALGORITHM

SORTING

Given an array of size N, sorting can be done in O(NlogN) in average. The most frequently used sorting algorithms that can achieve this time complexity are quicksort, Heapsort and mergesort. They usually require O(logN), O(1) and O(N) working space, respectively. The worst-case time complexity of quicksort is $O(N^2)$. In practice, we combine quicksort and Heapsort to avoid worst-case performance while retaining the fast average speed. The resulting algorithm is called introsort.

INTROSORT

Introsort or introspective sort is a hybrid sorting invented by David Musser in Musser (1997). This algorithm that provides both fast average performance and (asymptotically) optimal worst-case performance. It begins with quicksort and switches to Heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted. This combines the good parts of both algorithms, with practical performance comparable to quicksort on typical data sets and worst-case O(NlogN) runtime due to the heap sort. Since both algorithms it uses are comparison sorts, it too is a comparison sort.
David Musser also introduced introselect, a hybrid selection algorithm based on quickselect (a variant of quicksort), which falls back to median of medians and thus provides worst-case linear complexity, which is optimal. [1]

Pseudocode:

```
procedure sort(A : array):
    maxdepth ← ⌊log(length(A))⌋ × 2
    introsort(A, maxdepth)

procedure introsort(A, maxdepth):
    n ← length(A)
    if n ≤ 1:
        return  // base case
    if n ≤ 16:
            insertionsort(array)
    else if maxdepth ← 0:
        Heapsort(A)
    else:
        p ← quicksortPartition(array) //this function does pivot selection
        introsort(right of pivot)
        set subarray to left of pivot
```

Analysis:

In quicksort, one of the critical operations is choosing the pivot: the element around which the list is partitioned. The simplest pivot selection algorithm is to take the first or the last element of the list as the pivot, causing poor behavior for the case of sorted or nearly sorted input.
Niklaus Wirth's variant uses the middle element to prevent these occurrences, degenerating to O(n^2) for contrived sequences. The median-of-3 pivot selection algorithm takes the median of the first, middle, and last elements of the list.
It sorts in place, except for $\theta(logN)$ stack space and is usually faster than other in-place algorithms such as heapsort, mainly because it does substantially fewer data assignments and other operations. However its worst case is $\theta(n^2)$, and although the worst case behavior appears to be highly unlikely, the very existence of input sequences that can cause such bad performance may be a concern in some situations. The best alternative has been to use heap sort with $\theta(NlogN)$ worst case as well as average case time bound but this results in computing time two to five times longer then quicksort's time on most inputs. Thus in order to protect completely against deterioration to quadratic time it would seem necessary to pay a substantial time penalty for the great majority of input sequences.

Introsort, a hybrid sorting algorithm almost behaves like median-of-3 quicksort for most inputs but which is capable of detecting when partitioning is tending toward quadratic behavior. By switching to heapsort in those situations introsort achieves the same O ($NlogN$) time bound as heapsort but is almost always faster than just using heapsort in the first place.
On a median-of-3 killer sequence of length 1,00,000 for example introsort switches to heapsort after 32 partitions and has a total running time less than 1/200th of that of median-of-3 quicksort . In this case it would be somewhat faster to use heapsort in the first place but for almost all randomly chosen integer sequences introsort is faster than heapsort by a factor of between 2 and 5.

Median-of-3 Killer Sequences

The simplest methods of choosing pivot elements for partitioning, such as choosing the first element, cause quicksort to deteriorate to quadratic time on already sorted sequences.
Since there are many situations in which one applies a sorting algorithm to an already sorted or almost sorted sequence (as in our case of address sorting,

where every time a new record is read, it is appended to the list that is already sorted), a more robust way of choosing pivot is needed. Ideal choice would be the median value but the amount of computation required is too large. Instead we can choose the first, middle, and last elements and then then choose the median of these three values. This method produces good partitions in most cases including the sorted or almost sorted cases that cause the simpler pivoting methods to blow up.
Nevertheless there are sequences that can cause Median-of-3 quicksort to make many bad partitions and take quadratic time.

Algorithm as used in the assignment

```
introsort (A, low, high, depthLimit)
        if(A.length<=1)
                return
        while (high-low > size_threshold)
                if (depthLimit == 0)
                        heapsort(A, low, high)
                        return
                depthLimit <-- depthLimit - 1
                int pivot <-- quicksortPartition(A, low, high, medianof3(a,
                introsort(A, pivot, high, depthLimit)
                high <-- pivot;
                insertionsort(a, lo, hi)
quicksortPartition(A, low, high, median)
        int i ← low,  j ← high
        while (true)
                while (a[i] < median)
                        i++
                        j ← j-1
                        while (median < a[j])
                        j ← j-1
                        if(!(i < j))
                              return i
                         exchange(a, i, j)
                         i++
```

In intorsort loop it is possible to omit the test for recursing on smaller half of the partition, since the depth limit puts an O(log n) bound on the stack depth anyway. This omission nicely offsets the added test for exceeding the depth-

limit, keeping the algorithm's overhead essentially the same as quicksort's. We cab skip the details of the heapsort algorithm, since they are unimportant for the time bound claimed for introsort, as long as heapsort, or any other algorithm used as the stopper has an O(*NlogN*) worst-case time bound.

Theorem: Assuming an O(*logN*) subproblem tree depth limit and an O(NlogN) worst case time bound for heapsort, the worst case computing time for INTROSORT is $\Theta(\mathrm{NlogN})$.

Proof: The time for partitioning is bounded by N times the number of levels of partitioning, which is bounded by the subproblem tree depth limit. Hence the total time for partitioning is O(*NlogN*). Suppose Introsort calls heapsort j times on sequences of length $n_1 \ldots n_j$.
Let c b e a constant such that $cN \log_2 N$ bounds the time for the heapsort on a sequence of length N; then the time for all the calls of heapsort is bounded by

$$\sum_{i=1}^{j} cn_i \log_2 n_i \leq c \log_2 N \sum_{i=1}^{j} n_i \leq cN \log_2 N$$

or O(NlogN) also. Therefore the total time for introsort is O(NlogN).
The lower bound is also $\Omega(\mathrm{NlogN})$ since a sequence such as an already sorted sequence produces equal length partitions in all cases, resulting in $\log_2 N$ levels each taking $\Omega(\mathrm{N})$ time.[2]

CONCLUSION: Time complexity for Introsort

	Best	Average	Worst
Introsort	***n* log *n***	***n* log *n***	***n* log *n***

Performance Analysis on Random Input Records

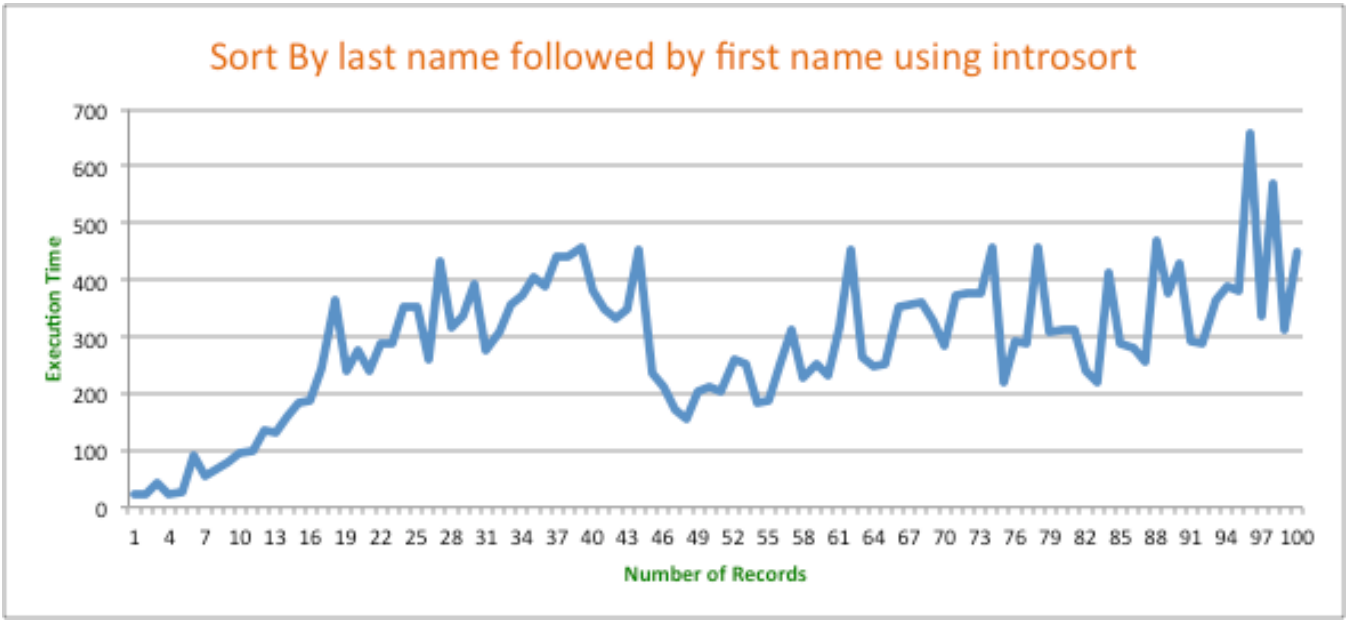

Fig 1: Address sorted by last name followed by first name using INTROSORT for 100 records

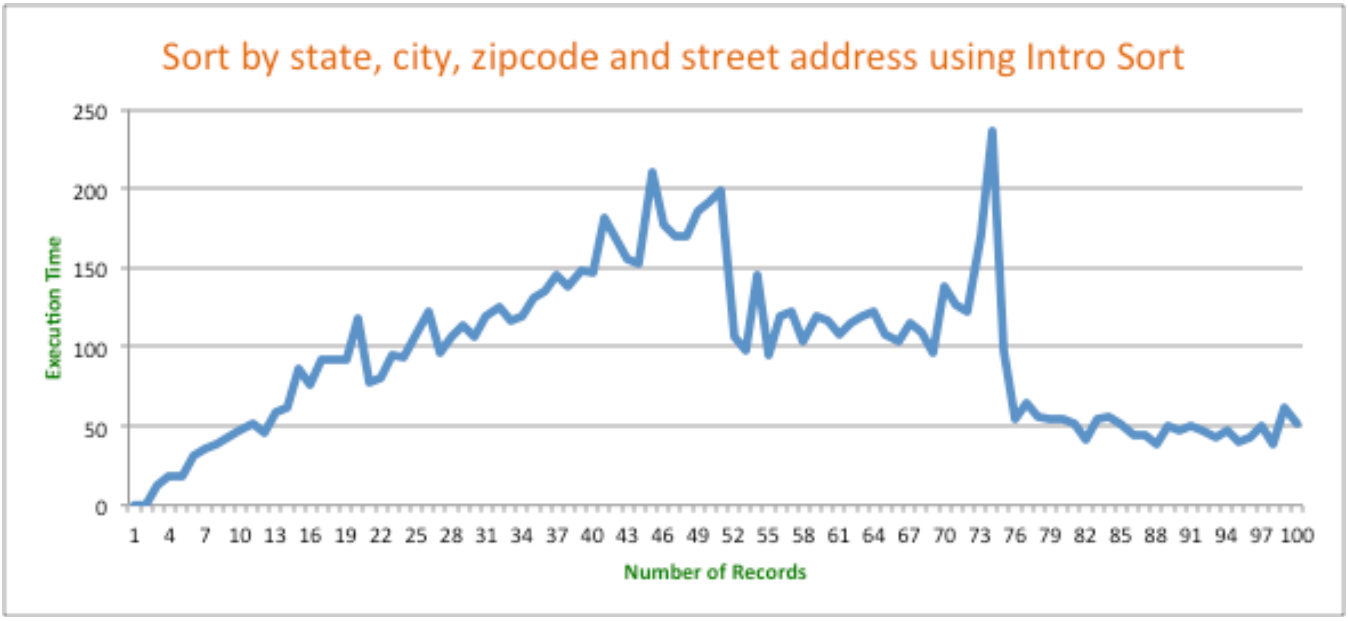

Fig 2: Address sorted by state, city, zip code followed by street address using INTROSORT for 100 records

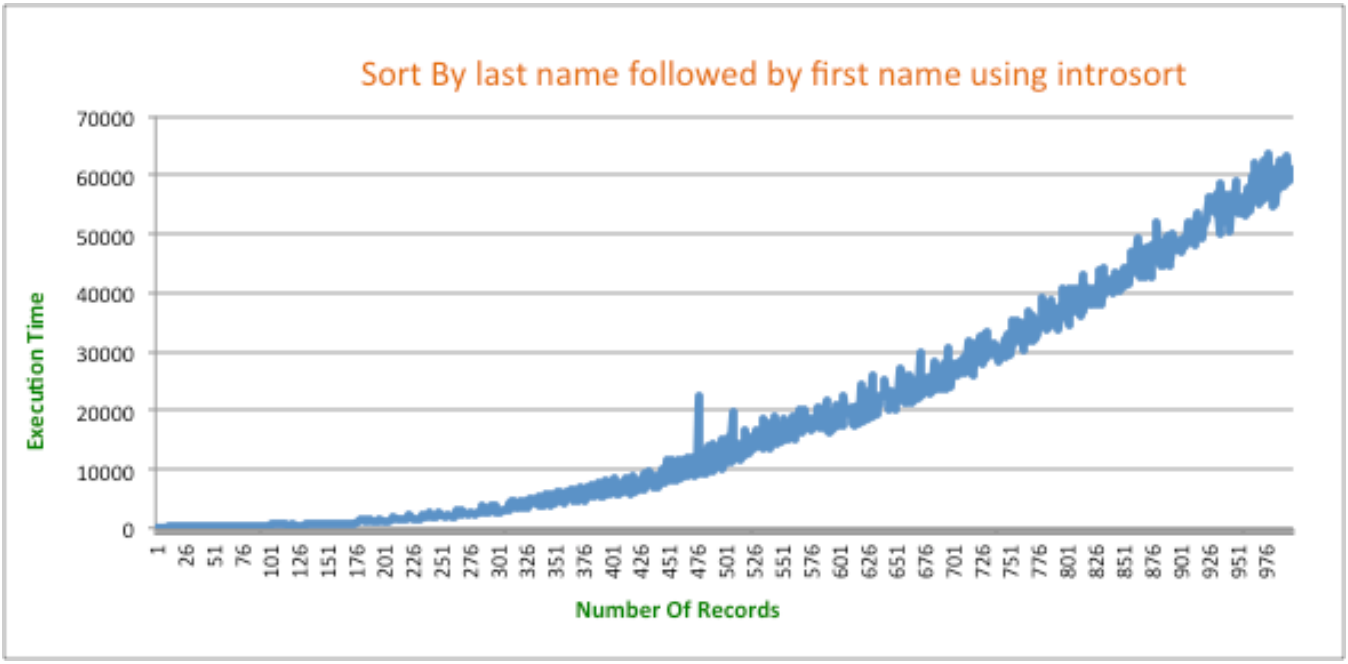

Fig 3: Address sorted by last name followed by first name using INTROSORT for 1000 records

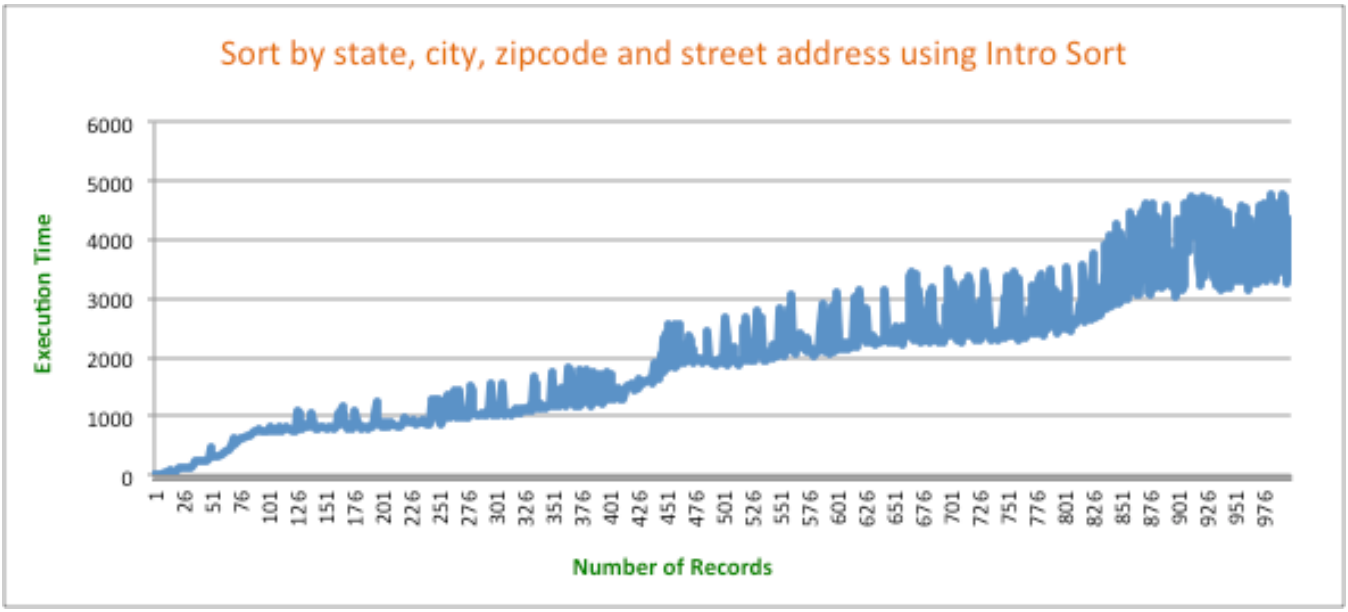

Fig 4: Address sorted by state, city, zip code followed by street address using INTROSORT for 1000 records

Observation:

The graph grows as per O(n log n) as expected.

Performance Analysis on Sorted Input Records

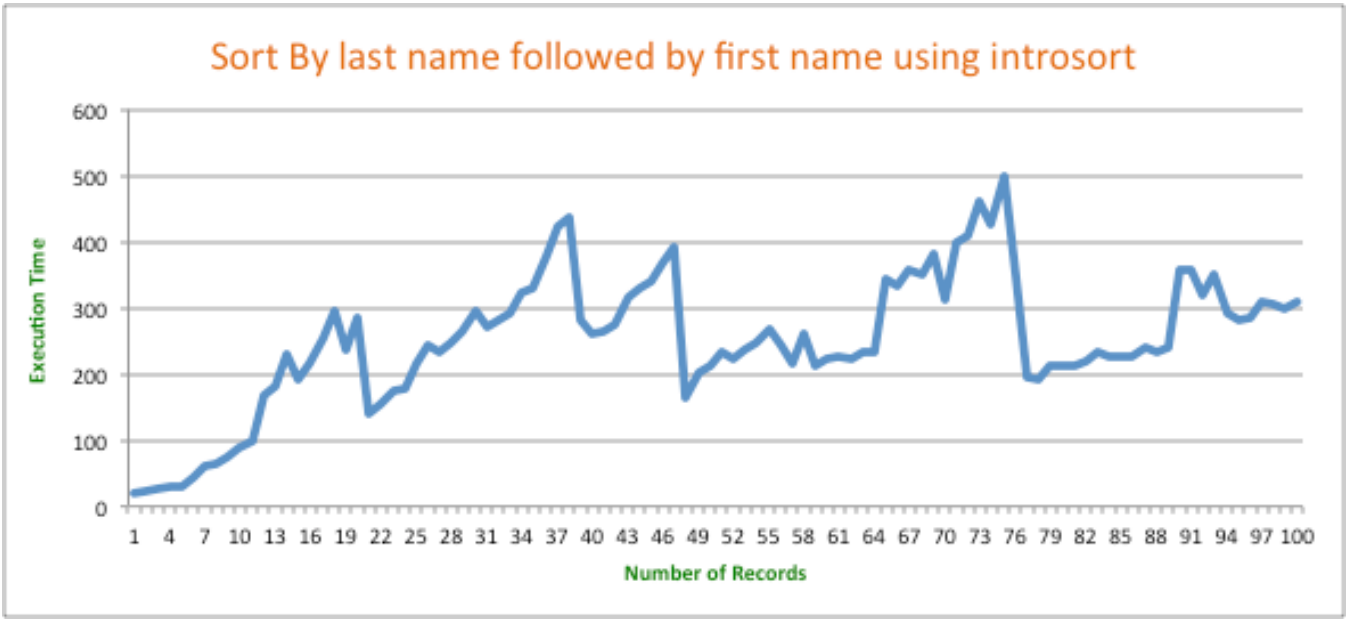

Fig 5: Address sorted by last name followed by first name using INTROSORT for 100 records

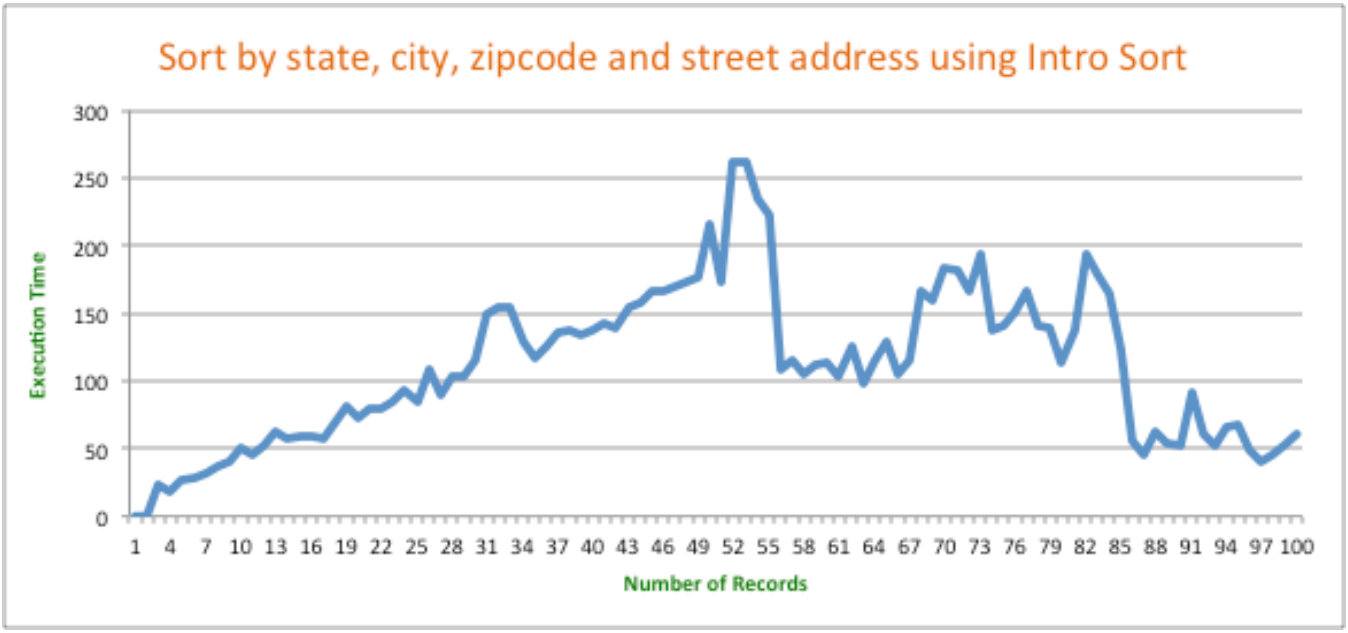

Fig 6: Address sorted by state, city, zip code followed by street address using INTROSORT for 100 records

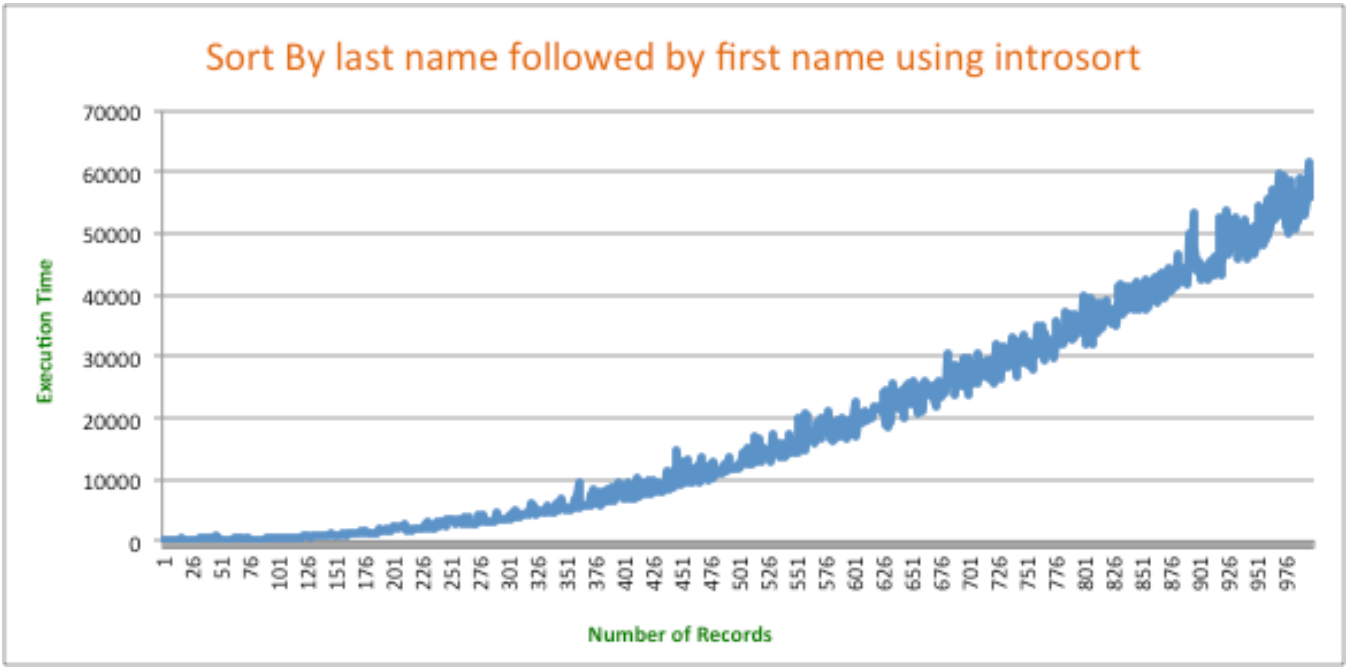

Fig 7: Address sorted by last name followed by first name using INTROSORT for 1000 records

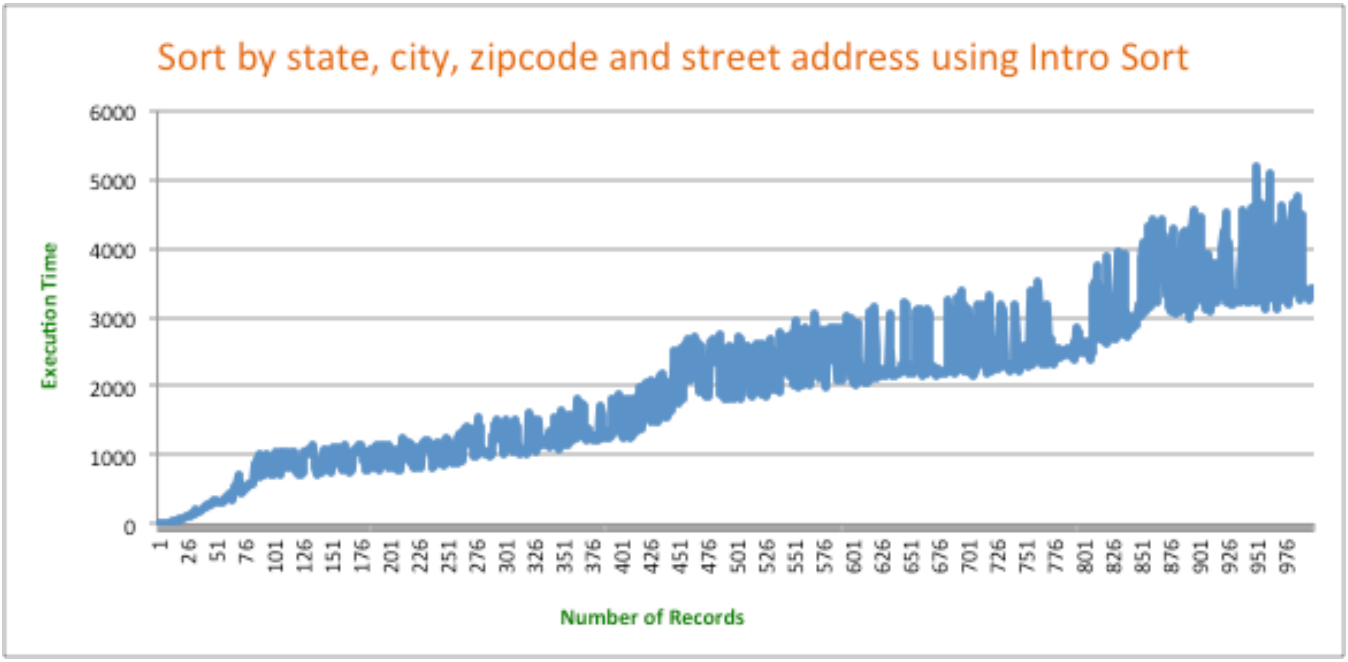

Fig 8: Address sorted by state, city, zip code followed by street address using INTROSORT for 1000 records

Observation:

The graph grows as per $O(n \log n)$ as expected.

Performance Analysis on Reversed Sorted Input Records

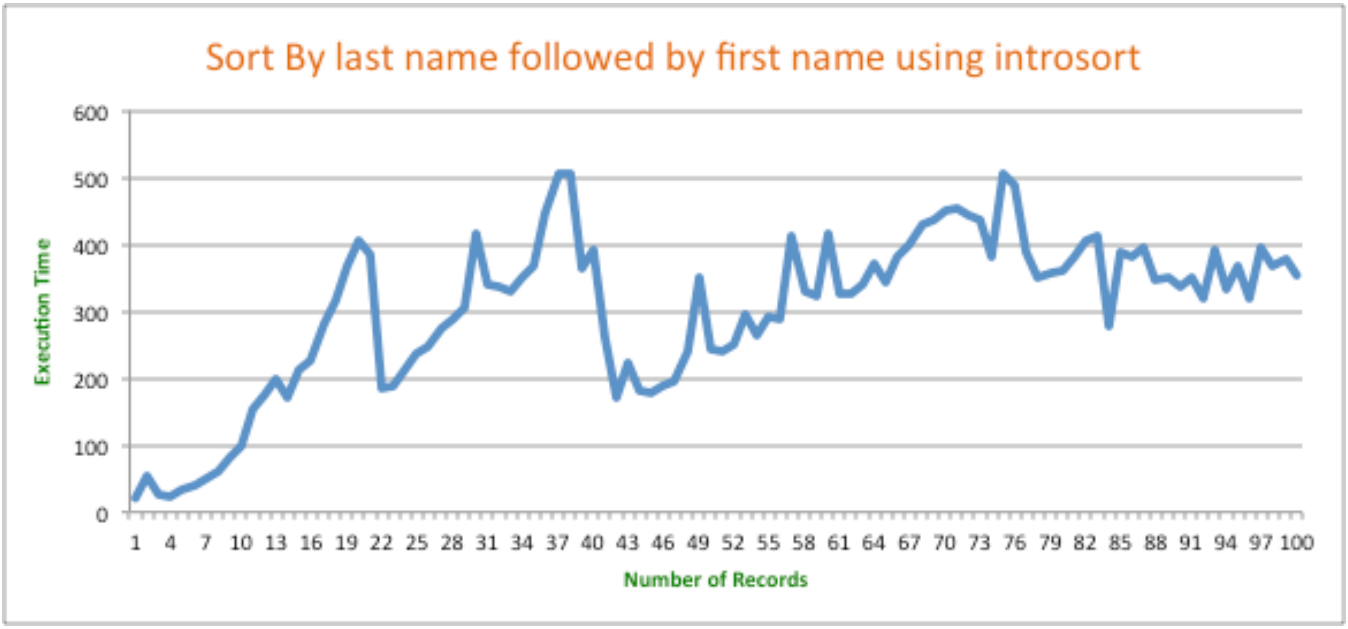

Fig 9: Address sorted by last name followed by first name using INTROSORT for 100 records

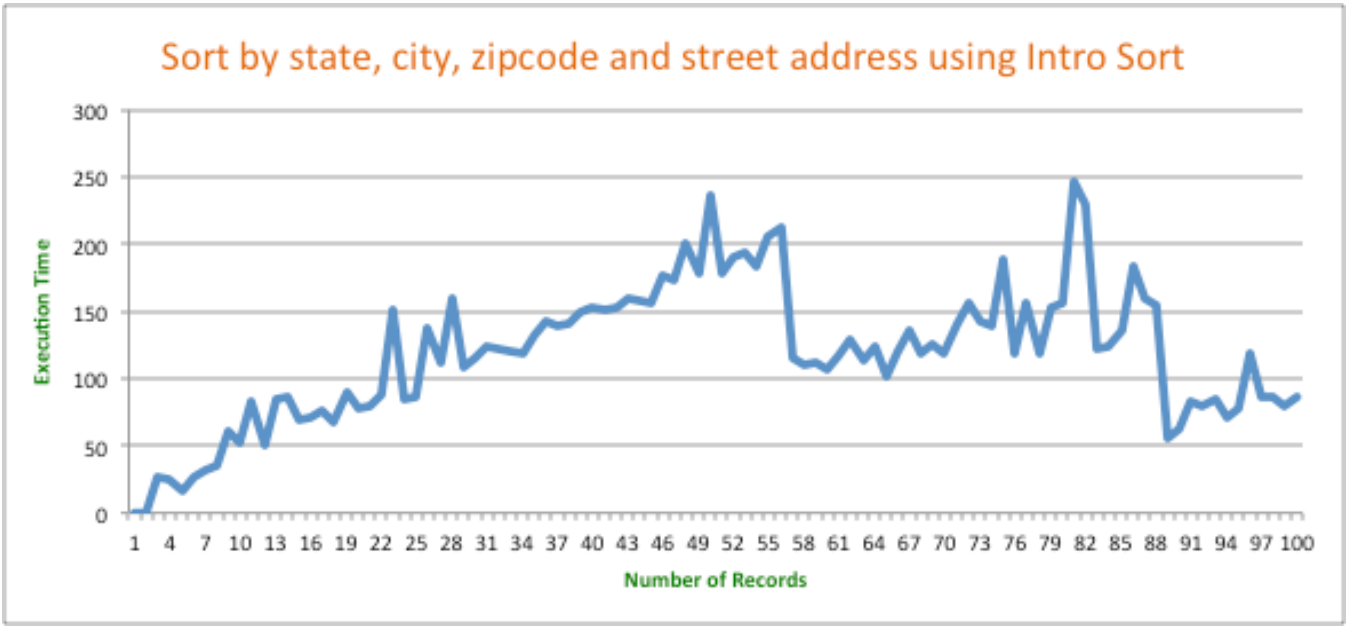

Fig 10: Address sorted by state, city, zip code followed by street address using INTROSORT for 100 records

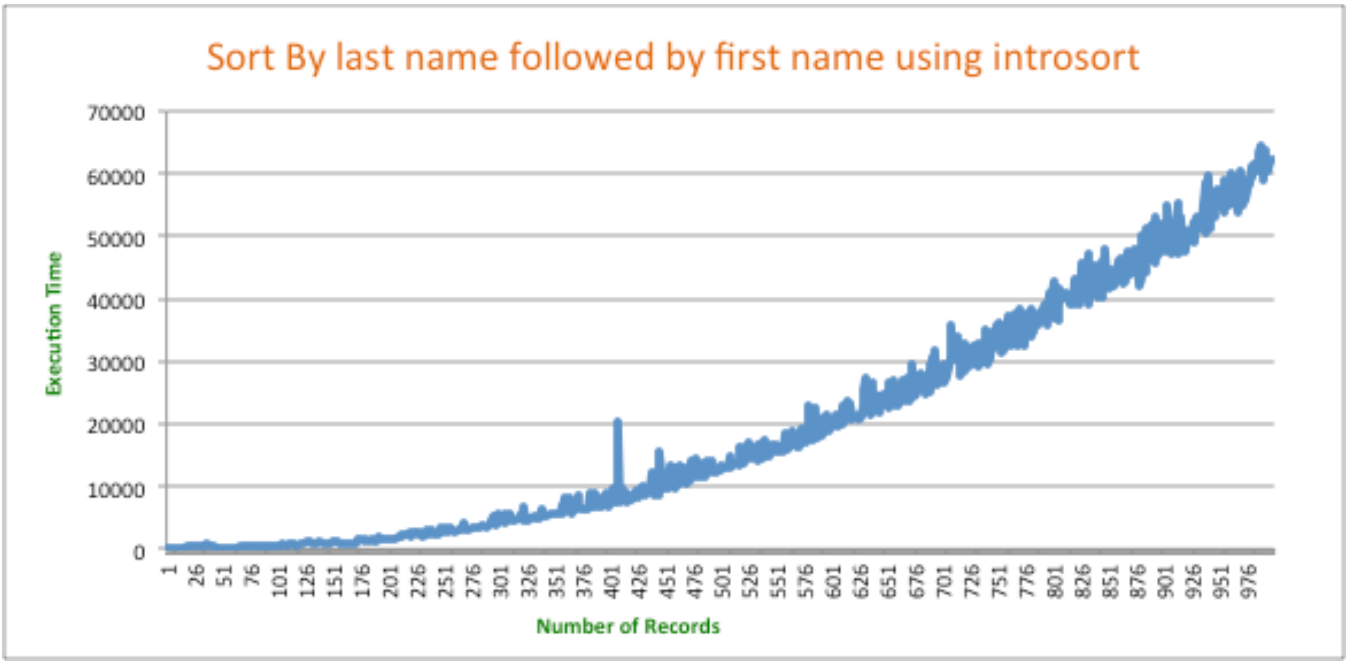

Fig 11: Address sorted by last name followed by first name using INTROSORT for 1000 records

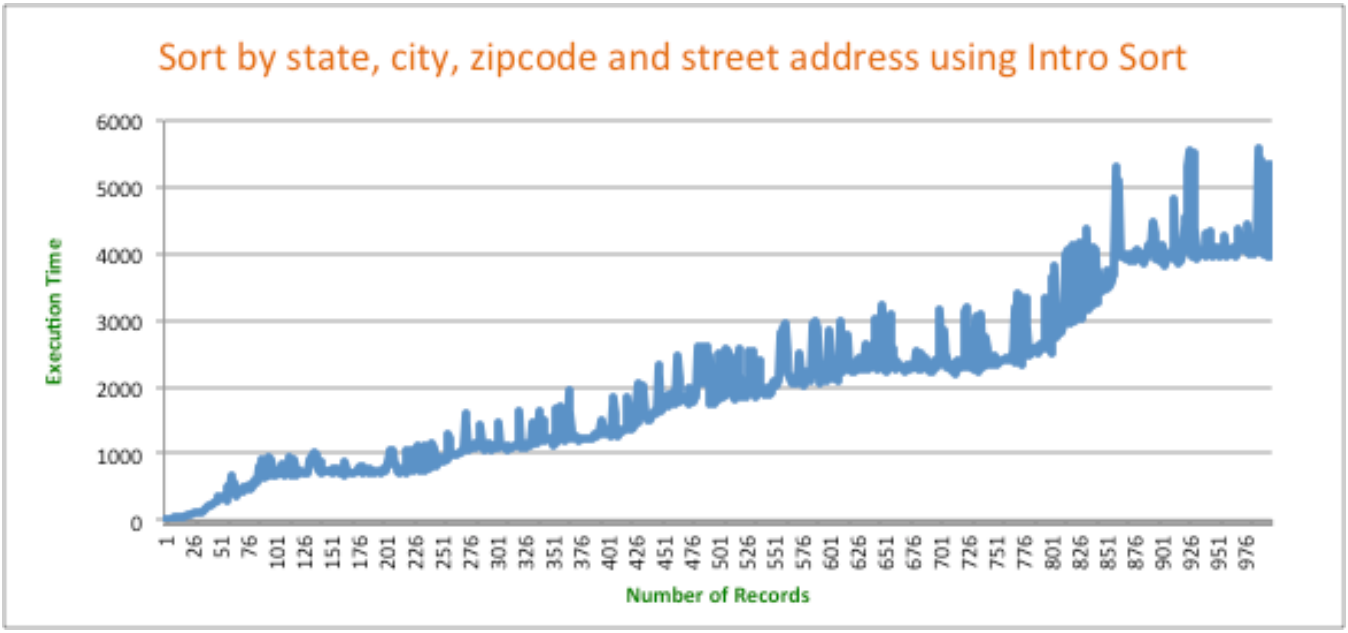

Fig 12: Address sorted by state, city, zip code followed by street address using INTROSORT for 1000 records

Observation:

The graph grows as per O(n log n) as expected.

References:

[1] : https://en.wikipedia.org/wiki/Introsort
[2] : http://www.cs.rpi.edu/~musser/gp/introsort.ps