Name : Parinaya Chaturvedi

Roll No. : B15120

LAB02 :: CS202 DSA

Selection Sort:

- Selection Sort takes **$O(n^2)$** time complexity. In this sort we find the positon of the minimum element and swap it with the element at position one.
- Then, we again find the minimum element in remaining array and swap it with element at position two . This process keeps on repeating till we reach the last element.
- This sorting algorithm is unstable as the order of equal valued elements might not be maintained after sorting is done.

Psuedo Code:

```
function SelectionSort(arr[],low,high)
    n = high-low+1
    for j=low+1 to n-1
        smallest=j
        i=j+1
        while(i<n)
            if (arr[i]<arr[smallest])
                smallest=i
        swap(a[j],a[smallest])
end function
```

Asymptotic Space Complexity and Time Complexity:

Best Case=$O(n^2)$

Worst Case=$O(n^2)$

Average Case=$O(n^2)$

Space Complexity=O(1)

Selecting the lowest element requires scanning all n elements (this takes $n - 1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires

scanning the remaining $n - 1$ elements and so on, for $(n-1) + (n - 2) + .. + 2 + 1 = n(n - 1) / 2 \in O(n^2)$ comparisons. Each of these scans requires one swap for $n-1$ elements (the final element is already in place).

Comparison with other sorting algorithms:

- Selection Sort is better than bubble sort and rank sort as the number of swaps in it is considerably lesser than bubble sort.
- It is very much similar to Insertion Sort as after ith iteration the first i elements are sorted. Insertion sort's only advantage is that it need to scan as many elements as it needs in order to place i+1 element whereas selection sort need to scan all the remaining elements.
- It is slower than Merge sort and Quick sort as the input size becomes large. However, in case the number of elements is less it might be better than the above said sorting algorithms by a negligible time.

Insertion Sort:

- Insertion sort has asymptotic time complexity of **$O(n^2)$.**
- Insertion sort removes one element from the input data in each iteration, finds the location to where it belongs as per the sorted list, and inserts it there.
- It repeats until no input elements remain. The sorted list grows in each iteration by one. It has the best case time complexity of **O(n)** if the entered array is already sorted.
- It is a stable Sorting algorithm.

Psuedo Code:

```
function InsertionSort(arr[], low , high)
    n =  high - low +1
    for i = 1 to n
        j = i
        while ( j > 0 and arr [j-1] > arr[j])
            swap ( A[j]  and A[j-1])
            j=j-1
end function
```

Asymptotic Space Complexity and Time Complexity:

Worst Case=$O(n^2)$

Best Case=O(n)

Average Case=$O(n^2)$

Space Complexity=O(1)

The best case input is when an array that is already sorted. In this case insertion sort has a linear running time of **O(*n*)**. During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The simplest worst case input is an array sorted in reverse 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 .

Comparison with other sorting algorithms:

- Insertion Sort is similar to selection sort. It is better than bubble and rank sort. It makes a fewer comparisons than selection sort but require more writes because inner loop can require shifting large number of elements of the sorted portion.
- As expected from it's order it is slower than divide and conquer sorts but can outperform them for smaller input size.

Rank Sort:

- Rank Sort is another inefficent sorting algorithm with order **$O(n^2)$.**
- Rank Sort is less effecient than insertion sort and selection sort. In rank sort an extra integer array is allocated which contains rank of each element.
- This extra array can be neglected by smart usage of memory.
- Rank of each element means the index of the element in the sorted array.
- After the array is obtained, the auxilliary array which is of same datatype as original array is used to contain elements in sorted order.
- The contents of this auxilliary array is then copied into the original array. Since all elements are pairwise compared, that's why it's complexity is **$O(n^2)$.**

Psuedo Code:

```
function RankSort( arr[ ] , high , low )
    n = high - low +1
    rank [n]
    for i = low to high
        rank [ i ] = 0
    for ( i = low  to  high )
        for ( j = i + 1 to high )
            if ( arr [ i ] < arr [ j ] )
                rank [ j ] =  rank [ j ]  + 1
    for ( i = low to high )
        Aux [ rank [ i ] ] =  arr [ i ]
    arr =Aux
end function
```

Asymptotic Space Complexity and Time Complexity:

Worst Case=$O(n^2)$

Best Case=$O(n^2)$

Average Case=$O(n^2)$

Space Complexity=$O(n)$

The time complexity for rank sort is **$O(n^2)$** as each element is compared pairwise with every other element. This is not an inplace sorting algorithm as an extra Auxilliary array is required to sort the elements.

Comparison with other sorting algorithms:

- This sorting technique is considerably slower than merge and quick sort.
- Insertion sort and Selection sort is also better than rank sort as they involve lesser comparisons and no extra space is required for these algorithms.
- Rank sort has better average time complexity than bubble sort due to lesser number of swaps.

Merge Sort:

- Mergesort is a divide and conquer algorithm.
- It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.
- The merge function is used for merging two halves. The merge function takes two sorted arrays as parameters and combine them to form a sorted array.
- The time complexity of Merge Sort comes out to be **O(n log (n)).**

Psuedo Code:

```
function MERGE-SORT(arr[ ], p , r)
     if( p <  r)
     MERGE-SORT(A, p, q)
         MERGE-SORT(A, q+1, r)
         MERGE(A, p, q, r)
end function

function MERGE (A, p, q, r)
     n1 = q – p +1
     n2 = r – q
     for i=1 to  n1
         L[ i ]  = A [ p + i -1]
     for j=1 to n2
               R[ j ]= A[ q + j ]
     L [n1 + 1 ] = ∞
     R [n2 + 1 ] = ∞
     i = 1
     j = 1
     for k = p to r
     if L[ i ] ≤ R [ j ]
         A[ k ] = L[ i ]
         i = i + 1
     else A[ k ] = R [ j ]
         j = j + 1
end function
```

Asymptotic Space Complexity and Time Complexity:

Worst Case=O(n log (n))

Best Case=O(n log(n))

Average Case=O(n log(n))

Space Complexity=O(n)

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + O(n)$

On solving this recuurence relation, time complexity of merge sort will come out to be O(n log(n)). It requires an additional extra space to merge the elements of sorted arrays.

Comparison with other sorting algorithms:

- Merge sort is faster than sorts like insertion sort, selection sort etc. as it follows a divide and conquer approach.
- Though in modern implementations an efficient quicksort can outperform mergesort yet merge sort is preferred over quicksort to sort linked lists.
- Slow random access of linked list elements make it difficult to implement quicksort on linked lists.

Bubble Sort:

- Bubble Sort is an inefficient sorting algorithm compared to others. It is comparable to other sorts like insertion sort, rank Sort etc. due to it's asymptotic time complexity is $O(n^2)$.
- Generally it is not preferred over other sorts of same order as it requires several swaps of the order $O(n^2)$.
- It's name itself suggests that smaller elements will "bubble" to the front. It is a stable sort i.e, repetitions will remain at their original position.
- In this sorting algorithm complete array is traversed and if an element is found which is greater than the following element then both those elements will be swapped. This step is repeated till whole array is sorted.

Psuedo Code(Original):

```
function bubblesort (arr[],low,high)
    k=0;
    while(k<(high-low))
        i=low
        while(i<high)
            if(arr[i]>arr[i+1])
            swap(a[i],a[i+1])
            i=i+1
    k=k+1
end function
```

In the original bubble sort,the outer while loop runs till 'k' is equal to index of the last element.If the array gets comletely sorted midway when k is less than size of array there is no provision for stopping the outer while loop. Also,the inner while loop is run for the whole length of array. There is no need to run it till last element as the last element in previous iteration is the biggest too.Thus, even if we give an ascendig sorted array the order of above algorithm will be $O(n^2)$.

Psuedo Code(Modified):

```
function bubblesort (arr[],low,high)
    flag=1
```

```
    k=0
    while(flag is not 0 and k<(high-low))
        flag=0
        i=low
        while(i<high-k)
            if(arr[i]>arr[i+1])
            swap(a[i],a[i+1])
            i=i+1
            flag=1
        k=k+1
end function
```

In the modified version of bubble sort a flag is kept there such that when the array gets completely sorted the condition in outer while loop becomes false and the control passes out of it. This flag checks whether a swap has been made in a single iteration. Thus, if we give an ascending sorted array the above algorithm will only take ***O(n)*** time to determine that it is sorted. If no swap has been made then the array is completely sorted. The inner while loop has also been modified so that the biggest element in previous iteration is not checked again.

Asymptotic Space and Time Complexity:

Worst Case(Both Original and Modified)=O(n²)

Best Case(Original)=O(n²)

Best Case(Modified)=O(n)

Average Case(Both Original and Modified)=O(n²)

Space Complexity=O(1)

Comparison with other sorting algorithms:

- It is less efficient than rank sort , insertion sort and selection sort as it requires larger no of swaps.
- It is less efficient than other sorts like merge and quicksort.

Quick Sort:

- QuickSort is a Divide and Conquer algorithm.
- It picks an element as pivot and partitions the given array around the picked pivot.
- There are many different versions of quickSort that pick pivot in different ways. In the original version usually first or last element is made pivot.
- The key process in quickSort is partition. Unlike merge sort the partition made here is not equal.

Psuedo Code(Original):

```
function Quicksort(A[], low , high )
         if (low < high)
              pivot_location = Partition(A,low,high)
              Quicksort(A,low, pivot_location)
              Quicksort(A, pivot_location + 1, high)

end function
function Partition(A[], low, high)
              pivot = A[low]
              leftwall = low

              for(i = low + 1 to high)

              if (A[i] < pivot)
                 swap(A[i], A[leftwall]
                leftwall = leftwall + 1

              swap(pivot,A[leftwall])

         return (leftwall)}

end function
```

Psuedo Code(Modified):

```
function Quicksort(A[], low , high )
         if (low < high)
              pivot_location = Partition(A,low,high)
              Quicksort(A,low, pivot_location)
              Quicksort(A, pivot_location + 1, high)

end function
```

```
function Partition(A[], low, high)
             pivot = rand()%(high - low + 1)+low
          swap(A[leftwall],A[pivot])
             leftwall = low

             for(i = low + 1 to high)

             if (A[i] < pivot)
                swap(A[i], A[leftwall]
               leftwall = leftwall + 1

             swap(pivot,A[leftwall])

          return (leftwall)}

end function
```

In modified version the pivot is randomised. The worst case complexity is still O(n) but after randomizing it the chances of getting the worst case complexity is extremely less. In original version the order is $O(n^2)$ if the array is already sorted.

Asymptotic Space and Time Complexity:

Best Case=O(n log(n))

Worst Case=$O(n^2)$

Average Case=O(n log(n))

Space Complexity=O(1)

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order.

Best Case:The best case occurs when the partition process always picks the middle element as pivot.

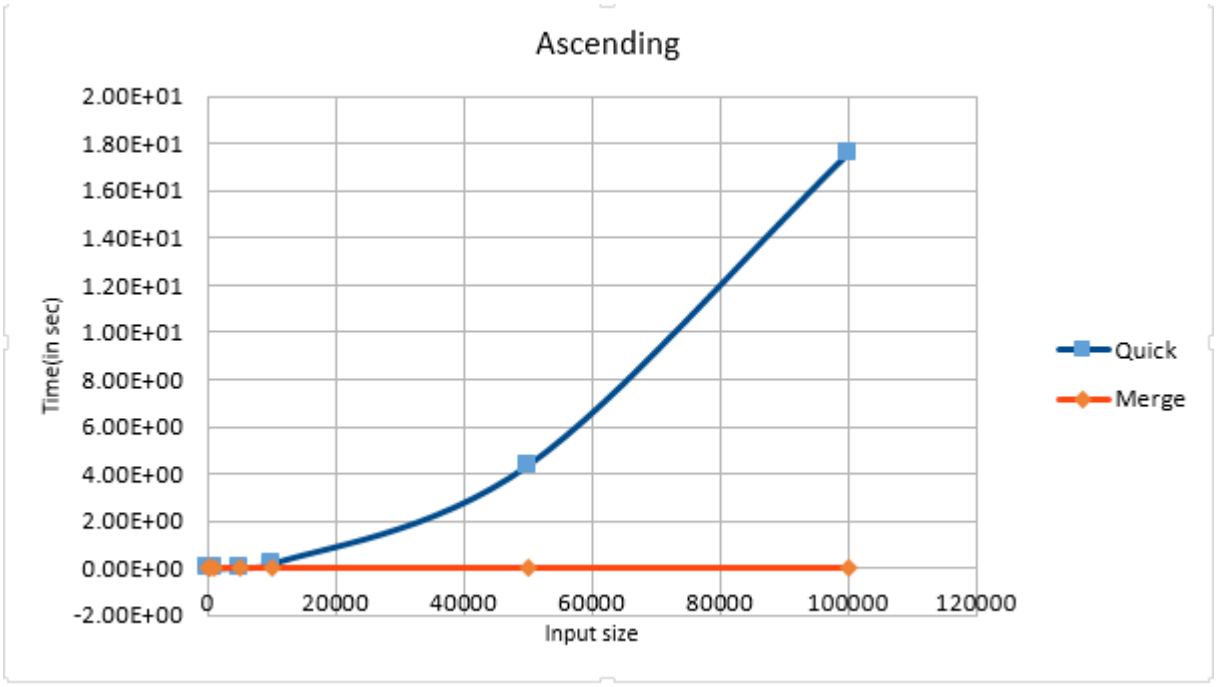
Average Case: The average case occurs when in one function call partitions are equal and in subsequent call they are skewed.

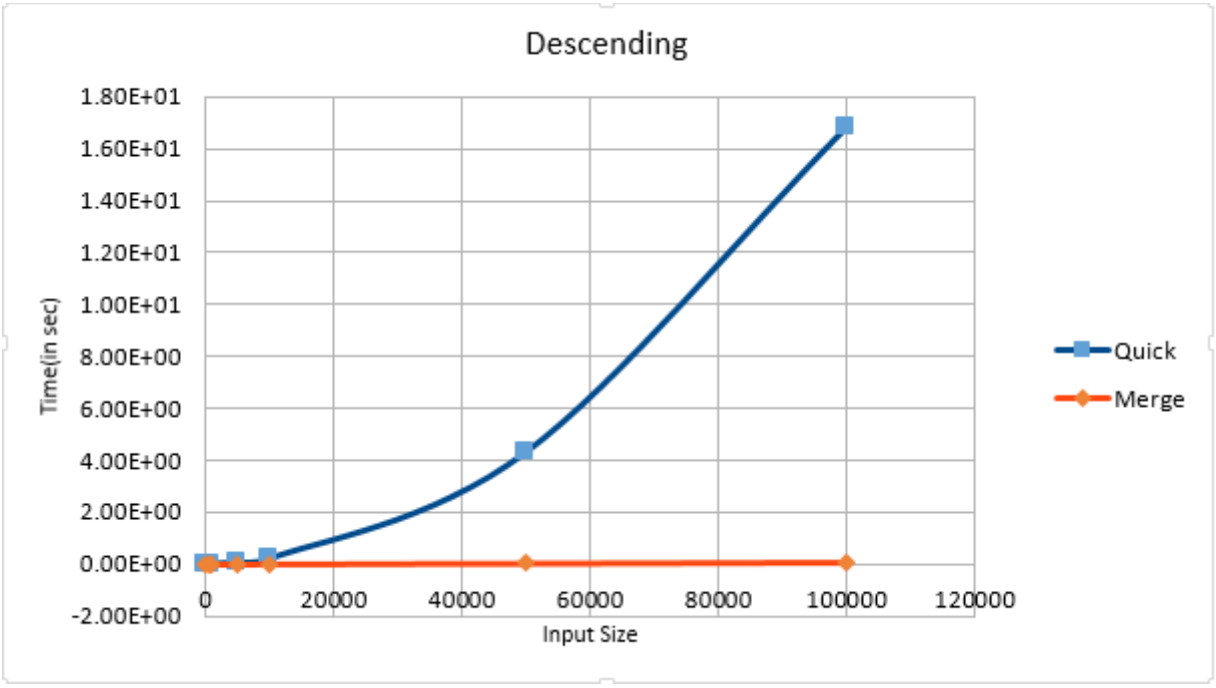
Comparison with other sorting algorithms:

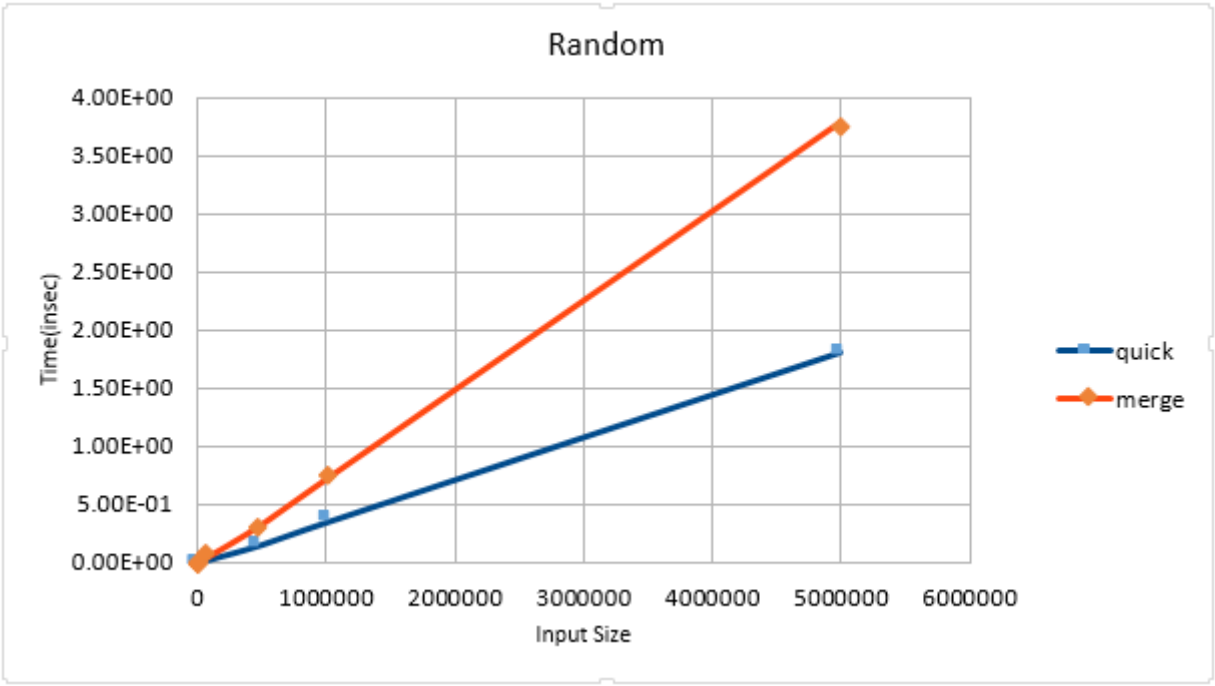
- Quick Sort is an in-place sorting algorithm whereas merge sort requires O(N) extra storage.
- Allocating and de-allocating the extra space used for merge sort increases the runtime of the algorithm.
- Comparing average complexity we find that both type of sorts have O(NlogN) but the constants obviously differ.
- For arrays, merge sort fails in comparison with other sorting algorithms due to the use of extra O(N) storage space. When using linked lists, merge sort is preferred.

Here are the visual graph comparisons :

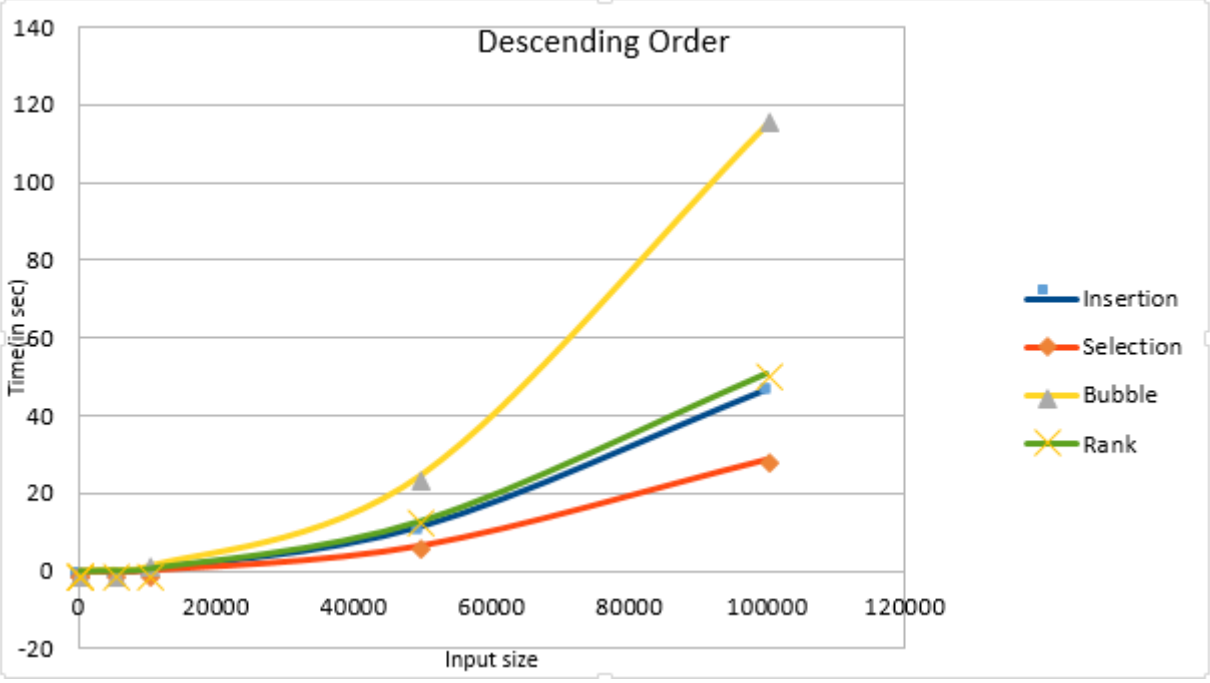
Between QuickSort and MergeSort :

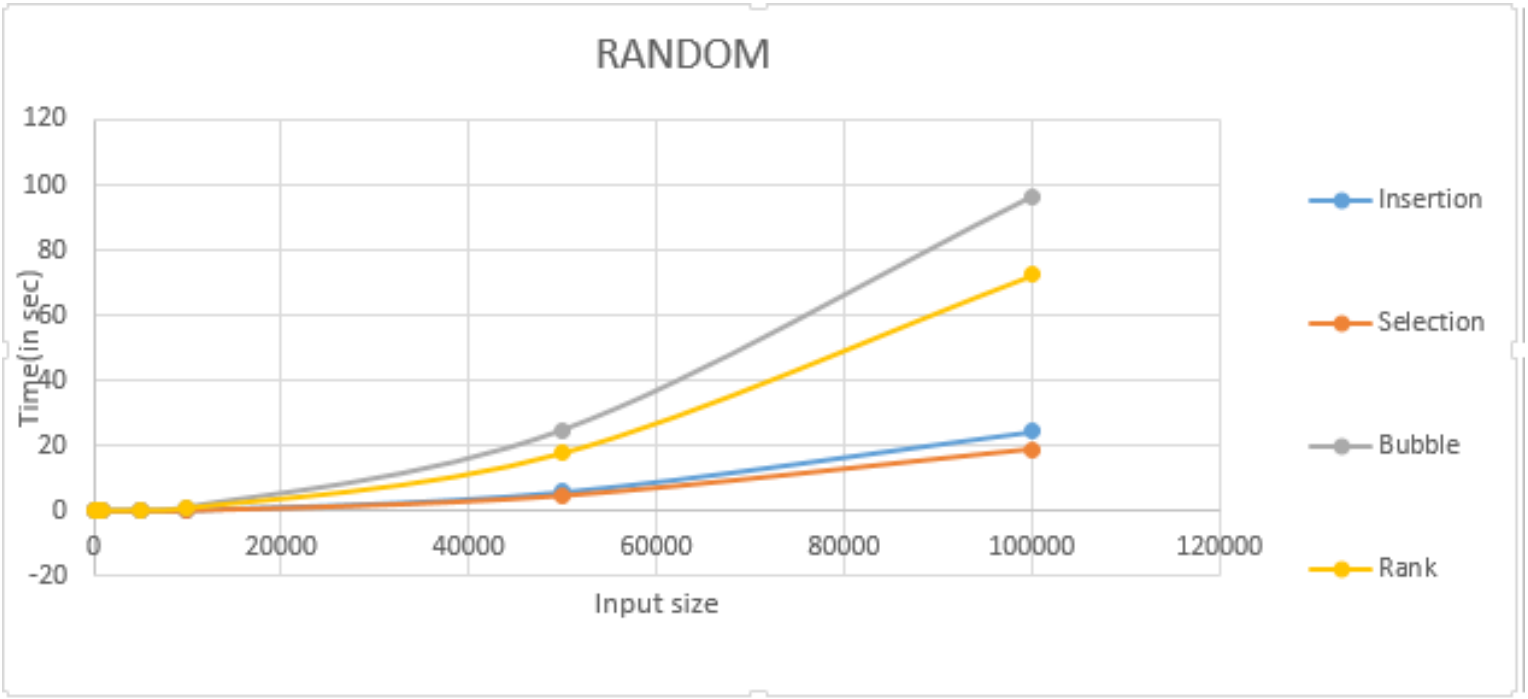





Random
4.00E+00
3.50E+00
3.00E+00
2.50E+00
2.00E+00
1.50E+00
1.00E+00
5.00E-01
0.00E+00
Time(insec)
0
1000000
2000000
3000000
4000000
5000000
6000000
Input Size
quick
merge

Between $O(n^2)$ Sorting Algorithms :





Web Resources :

- http://wikipedia.org
- http://geeksforgeeks.org
- http://hackerearth.com/codemonk