CS202

Analysis of different sorting algorithms

Contents

- ➔ Pseudocode
- ➔ Running time analysis of different sorting algorithms
- ➔ Graph comparison between sorting algorithms (Best, Worst and Average Case)

Vaibhav Agarwal(B15139)

06.04.2017

B.Tech , IIND YEAR CSE

BUBBLE SORT

Pseudocode

```
bubbleSort(array a)

      for i in 1 -> a.length - 1 do

            for j in 1 -> a.length - i - 1 do

                  if a[j] < a[j+1]

                        swap(a[j], a[j+1]);
```

N	ascending	descending	random
100	0	0	0
500	0	0	0
1000	0	0	0
5000	0.046875	0.203125	0.15625
10000	0.25	0.71875	0.65625
50000	6.57812	17.8125	17.6094
100000	26.2969	71.4375	69.5
500000	656.8438	1783.4872	1740.4655

In this sorting algorithm whole array is traversed and if an element is found which is greater than the following element then both those elements will be swapped. Generally it is not preferred over other sorts of same order because it requires a large number of swaps of the order ***O(n²).***

RANK SORT

Pseudocode

```
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 func
```

N	ascending	descending	random
100	0	0	0
500	0	0	0
1000	0	0	0
5000	0.078125	0.078125	0.07814
10000	0.34375	0.359375	0.35945
50000	9.01562	8.96875	9.01656
100000	35.8125	35.8594	35.8235
500000	897.3751	892.4263	896.7885

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.

In rank sort an extra integer array is allocated which contains rank of each element. Rank of each element means the number of elements lesser or greater than that element. To get rank array, all the elements are pairwise compared. 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

SELECTION SORT

Pseudocode

```
selectionSort(array a)
    for i in 0 -> a.length - 2 do
        maxIndex = i
        for j in (i + 1) -> (a.length - 1) do
            if a[j] > a[maxIndex]
                maxIndex = j
        swap(a[i], a[maxIndex])
```

N	random	ascending	descending
100	0	0	0
500	0	0	0
1000	0.015625	0.015625	0.015625
5000	0.078125	0.09375	0.15625
10000	0.28125	0.28125	0.265625
50000	6.1875	6.15625	6.14062
100000	24.7656	24.5781	24.6562
500000	2488.7585	611.4325	617.245

Comparison with other algorithms:

Selection Sort outperforms 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 kth iteration the first k elements are sorted. Insertion sort's only advantage is that it need to scan as many elements as it needs in order to place k+1 element whereas selection sort need to scan all the remaining elements.

It is considerably slower than Mergesort and Quicksort as the input size becomes large. However, in case the number of elements is less it might outperform the above said sorting algorithms by a negligible time.

In this sort first we find the positon of the minimum or maximum element depending on our implementation and swap it with the element at first position. Then, we again find the minimum or maximum element in remaining array and swap it with second element.

INSERTION SORT

Pseudocode

```
for i from 1 to N
    key = a[i]
    j = i - 1
    while j >= 0 and a[j] > key
        a[j+1] = a[j]
        j = j - 1
    a[j+1] = key
```

N	descending	ascending	random
100	0	0	0
500	0	0	0
1000	0	0	0.015625
5000	0.171875	0	0.078125
10000	0.71875	0	0.34375
50000	17.2969	0	8.67188
100000	69.3438	0	34.5312
500000	1735.4526	0.005265	861.7865

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 like 10-20 elements.

In each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

MERGE SORT

Pseudocode

Merge Sort

```
mergesort( A,p,r )
   if ( p < r ) then

      q = (p+r)/2
      mergesort( A, p, q )
      mergesort( A, q+1, r )
      merge( A, p, q, r )
end
```

Merge

```
merge (A, p, q, r)

   n1 = q - p + 1

   n2 = r - q

   A1[n1+1] = ∞

   A2[n2+1] = ∞

   i = 1

   j = 1

  for k = p to r

                  if A1[i] ≤ A2[j]

                        A[k] = A1[i]

                        i = i + 1

                  else

                        A[k] = A2[j]

                        j = j + 1

end
```

N	random	ascending	descending
100	0	0	0
500	0	0	0
1000	0	0	0
5000	0.015625	0.015625	0
10000	0	0	0.015625
50000	0.015625	0.015625	0.015625
100000	0.046875	0.046875	0.03125
500000	0.25	0.1875	0.1875
1000000	0.515625	0.390625	0.375
5000000	2.89062	2.14062	2.10938

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.

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.

QUICK SORT

Pseudocode

Partition Function

```
partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1

   while True do
      while A[++leftPointer] < pivot do
         //do-nothing
      end while

      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while

      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
      end if

   end while

   swap leftPointer,right
   return leftPointer

end
```

Quick Sort

```
quickSort(left, right)

   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if

end
```

N	random	ascending	descending
100	0	0	0
500	0	0	0
1000	0	0	0
5000	0	0.03125	0.0625
10000	0	0.15625	0.15625
50000	0.015625	3.21875	3.25
100000	0.03125	12.7812	13.0312
500000	0.140625		
1000000	0.265625		
5000000	1.42188		

Comparison with other sorting algorithms:

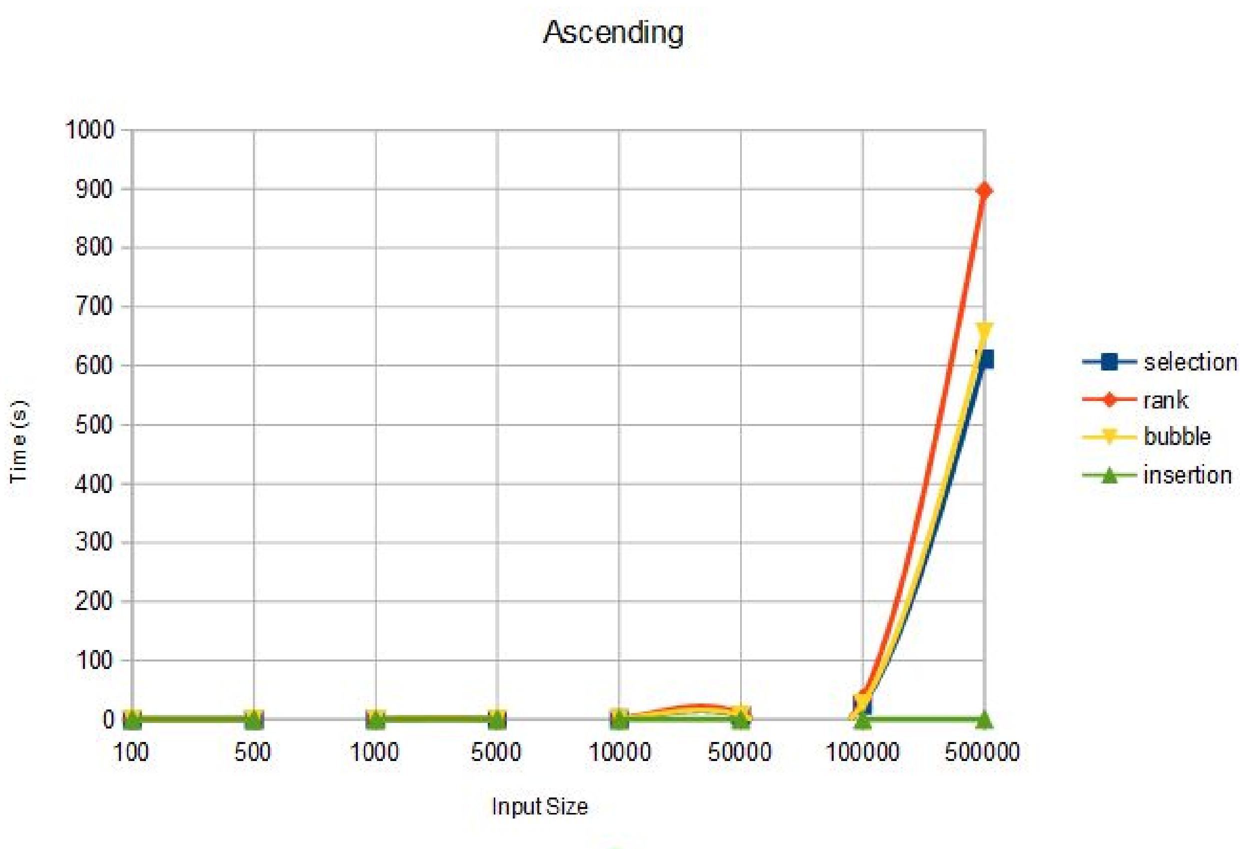
Quick Sort in its general form is an in-place sort whereas merge sort requires O(N) extra storage. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ.For arrays, merge sort loses due to the use of extra O(N) storage space. In linked lists, however , merge sort is preferred.

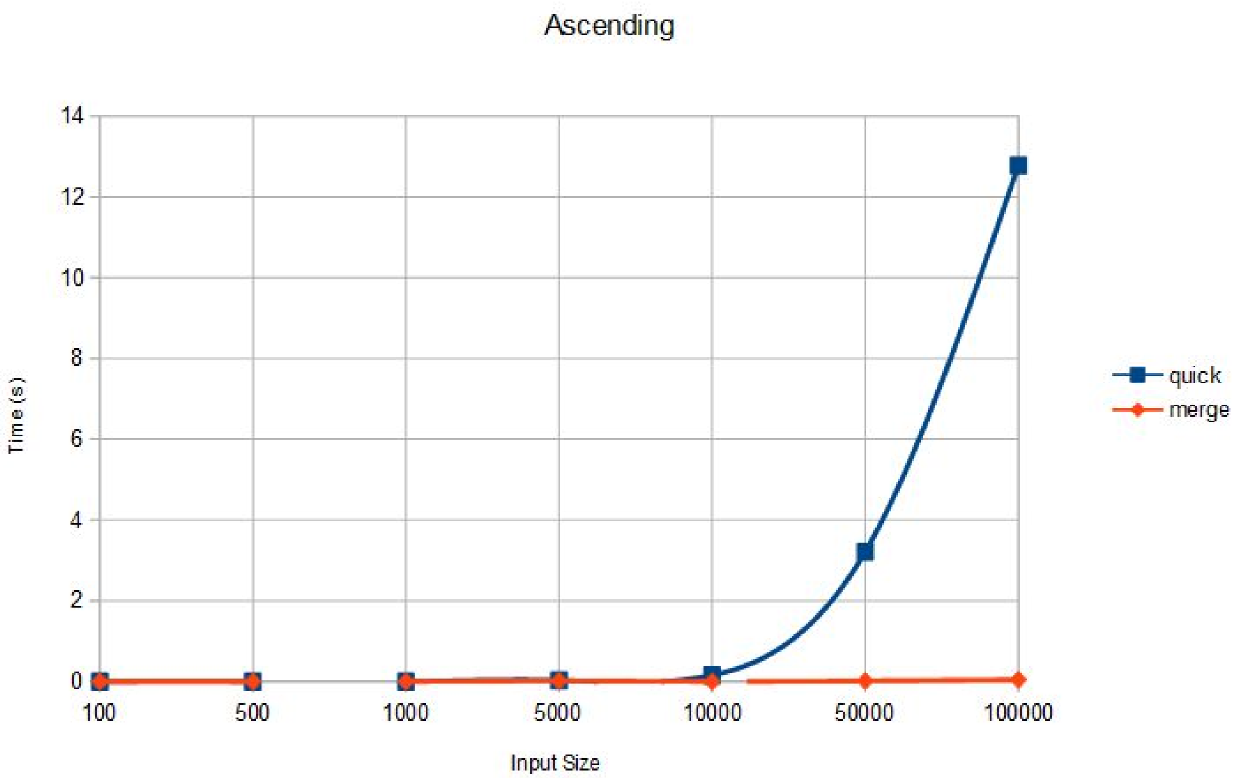
QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

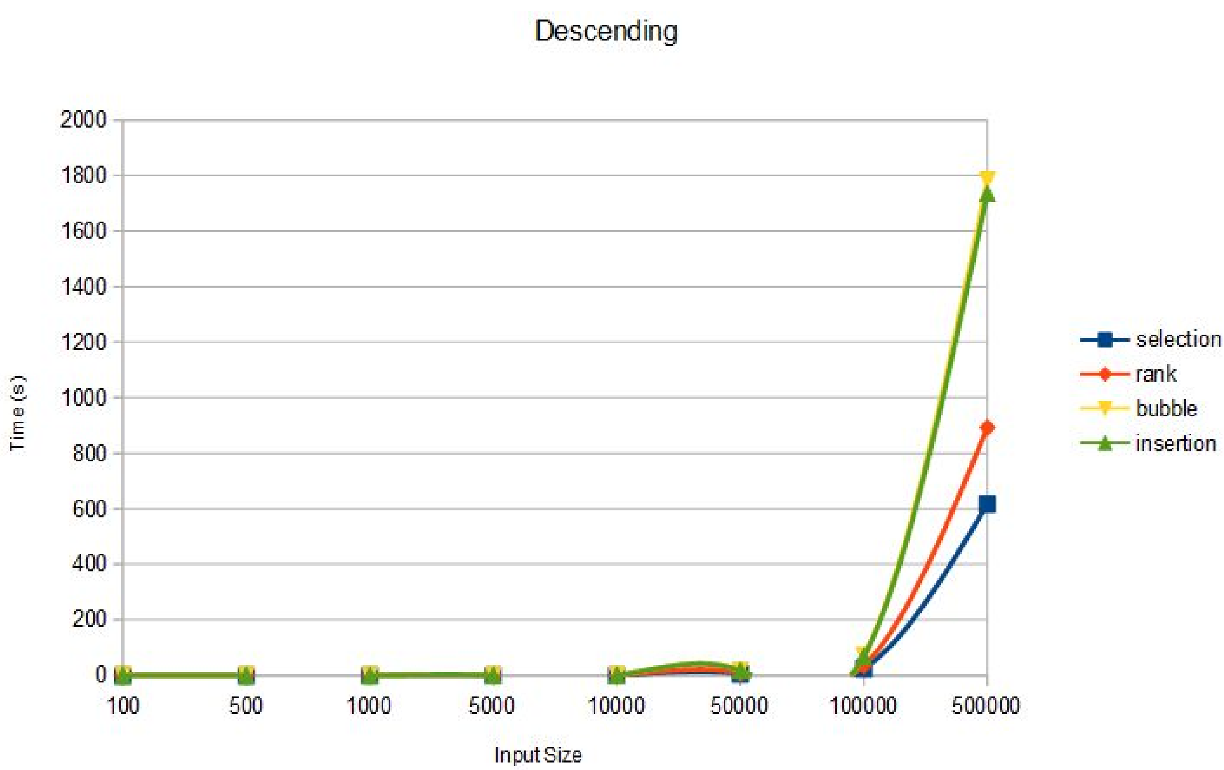
Time Complexity (Best, Worst and Average Case)

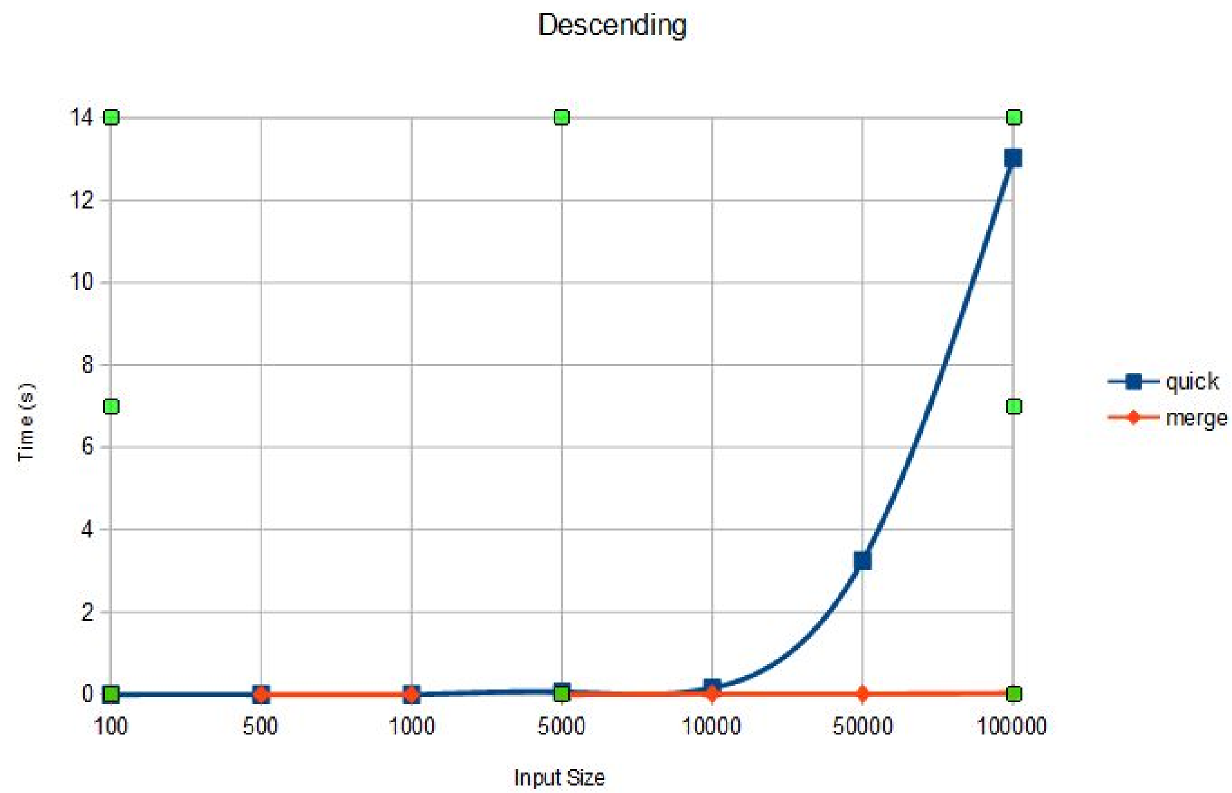
Algorithm	Best	Average	Worst
Quick sort	O(n log(n))	O(n log(n))	O(n^2)
Merge sort	O(n log(n))	O(n log(n))	O(n log(n))
Rank sort	O(n^2)	O(n^2)	O(n^2)
Bubble sort	O(n^2)	O(n^2)	O(n^2)
Insertion sort	O(n)	O(n^2)	O(n^2)
Selection sort	O(n^2)	O(n^2)	O(n^2)

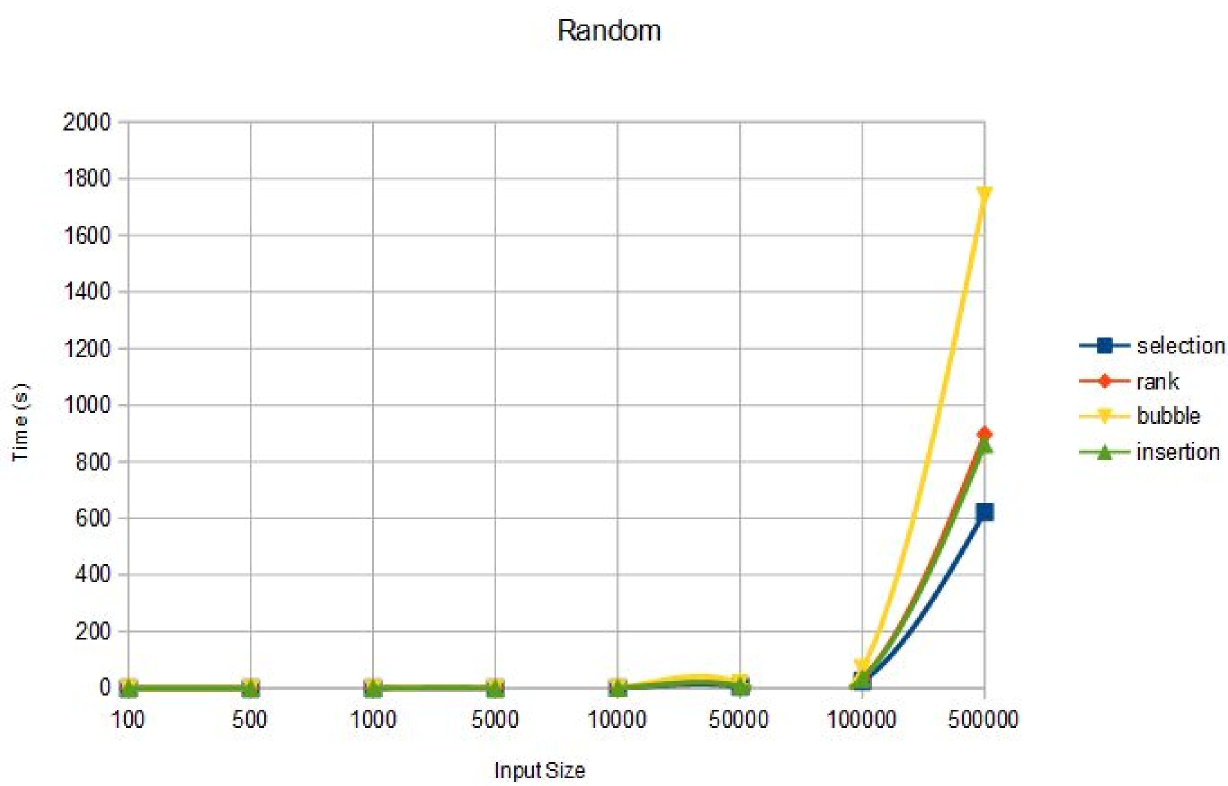
Graph

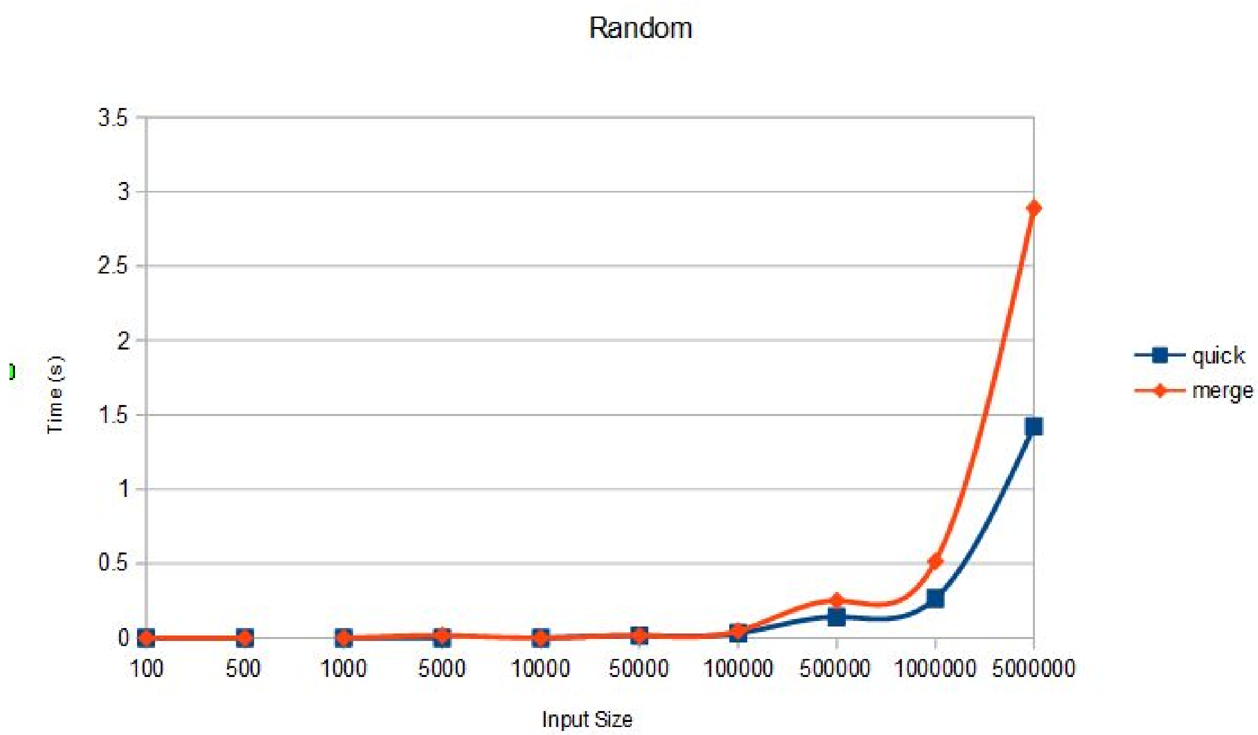

Ascending
1000
900
800
700
600
500
400
300
200
100
0
100
500
1000
5000
10000
50000
100000
500000
Time (s)
Input Size
selection
rank
bubble
insertion


Ascending
14
12
10
8
6
4
2
0
Time (s)
100
500
1000
5000
10000
50000
100000
Input Size
quick
merge


Descending
2000
1800
1600
1400
1200
1000
800
600
400
200
0
Time (s)
100
500
1000
5000
10000
50000
100000
500000
Input Size
selection
rank
bubble
insertion


Descending
14
12
10
8
6
4
2
0
Time (s)
100
500
1000
5000
10000
50000
100000
Input Size
quick
merge


Random
2000
1800
1600
1400
1200
1000
800
600
400
200
0
Time (s)
100
500
1000
5000
10000
50000
100000
500000
Input Size
selection
rank
bubble
insertion


Random
3.5
3
2.5
2
1.5
1
0.5
0
Time (s)
100
500
1000
5000
10000
50000
100000
500000
1000000
5000000
Input Size
quick
merge