CS202

ADVANCED DATA STRUCTURES AND ALGORITHMS

Dr. Dileep A.D.
Indian Institute of Technology Mandi

ASSIGNMENT-2

18 March 2016

REPORT

ON

ANALYSIS OF DIFFERENT SORTING ALGORITHMS

By

Yogendra Kumar Dhiwar
B14141

For all the following cases :
Input : A[1.....n] array of integers
Output : Sorted Array such that A[1]<=A[2]<=.............<=A[n]

Insertion_Sort(A)

Insertion sort is a simple sorting algorithm that insert an element into its correct position in previously sorted sequence like in arranging of pack cards.

It is an inplace algorithm , so does not requires any additional space. Thus space is required only for storing elements hence order n.

Psuedo Code :

```
for j=2 to n
    key = A[j]
        //Insert A[j] into the sorted sequence A[1...j-1]
        i=j-1
        while i>0 and A[i]>key
            A[i+1]=A[i]
            i=i-1
        A[i+1] = key
```

Asymptotic running time analysis :

1. Best Case : $T(n)=O(n)$
2. Worst Case : $T(n)=O(n^2)$
3. Average Case : $T(n)=O(n^2)$

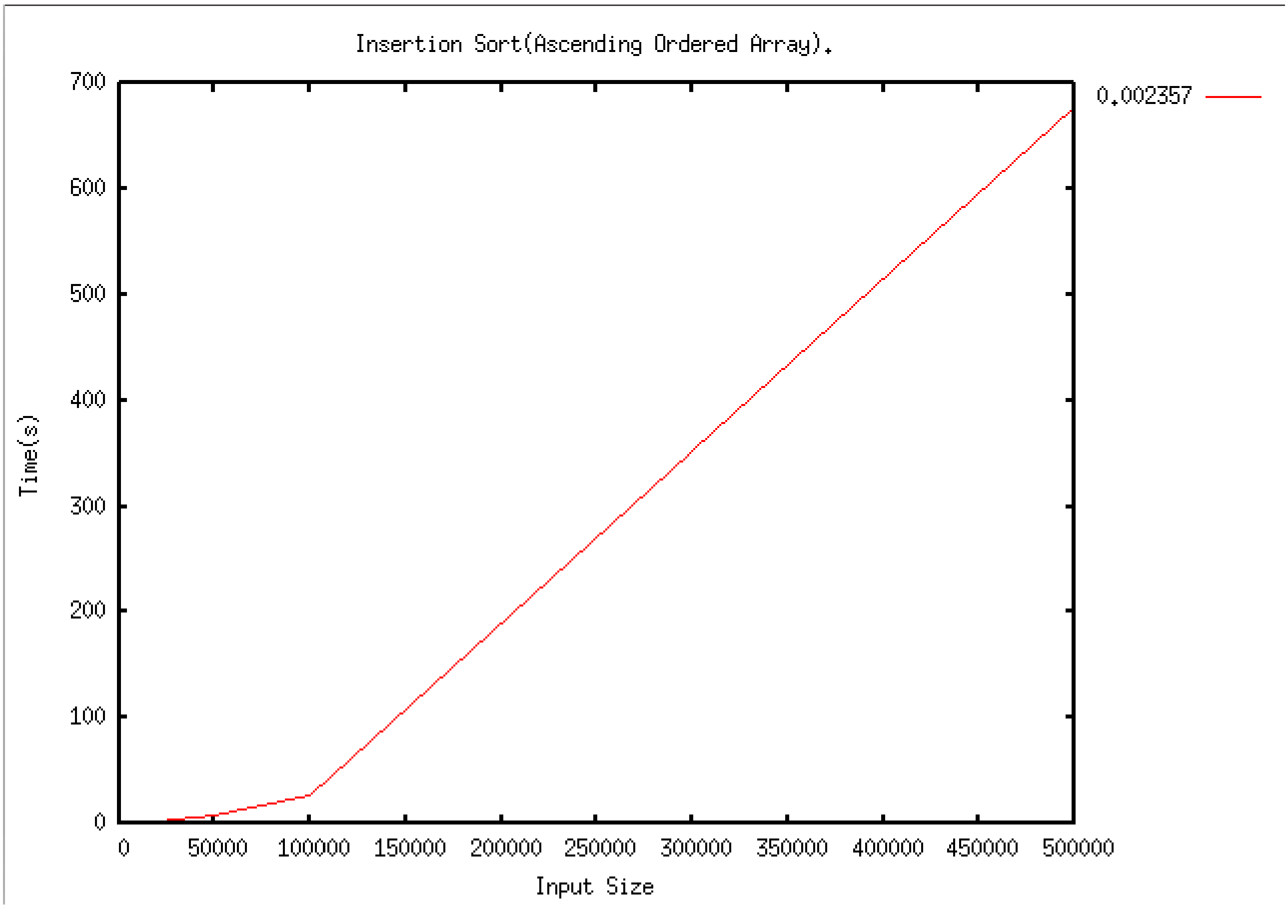
Actual running time analysis :

1). Ascending Data

size	time

500	0.000019
1000	0.000029
5000	0.000124
10000	0.000270
50000	0.000920
100000	0.001367
500000	0.005022

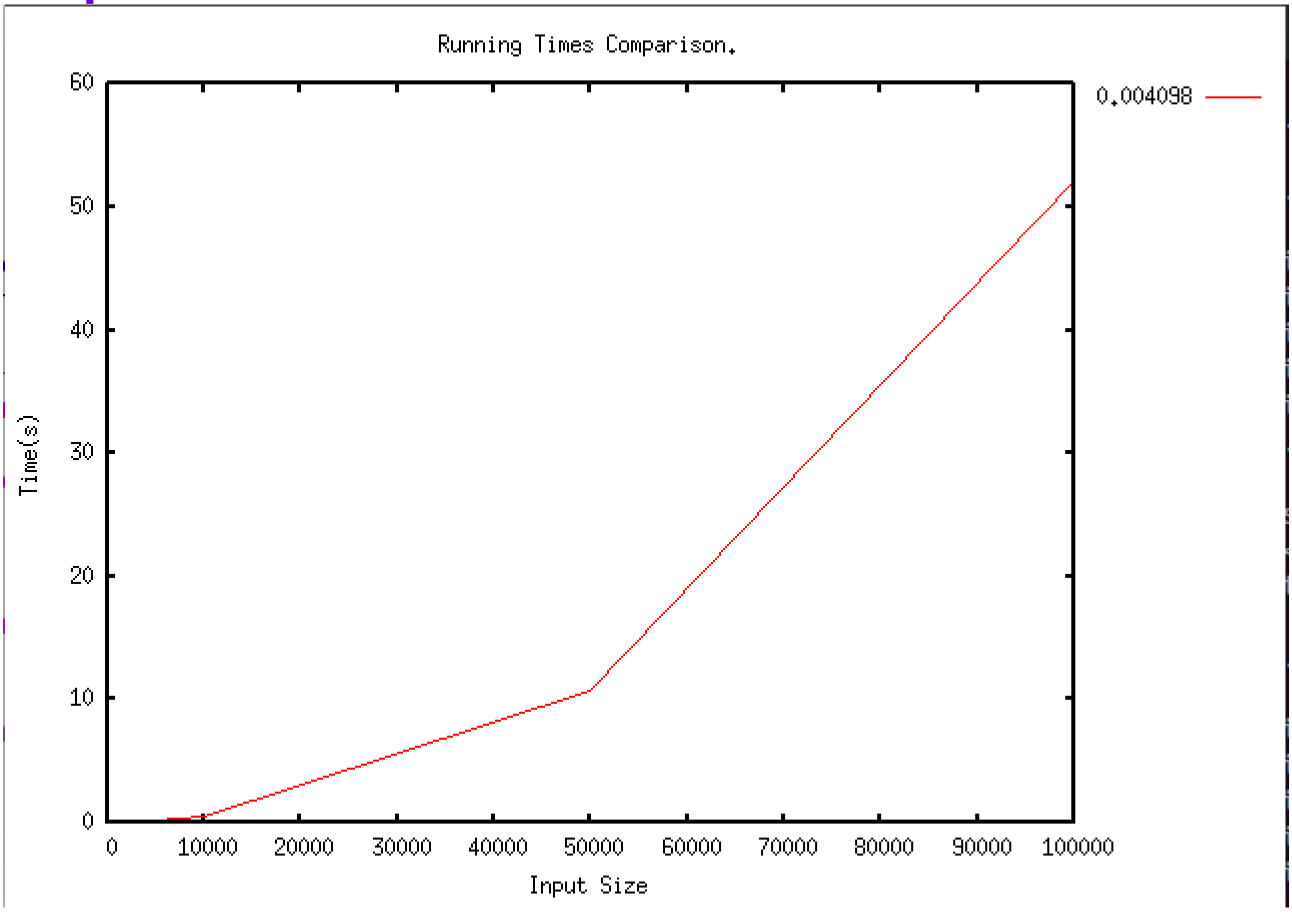
Graph :



2).Descending Data

size	time
500	0.004098
1000	0.008691
5000	0.122067
10000	0.510327
50000	10.7683
100000	52.0435

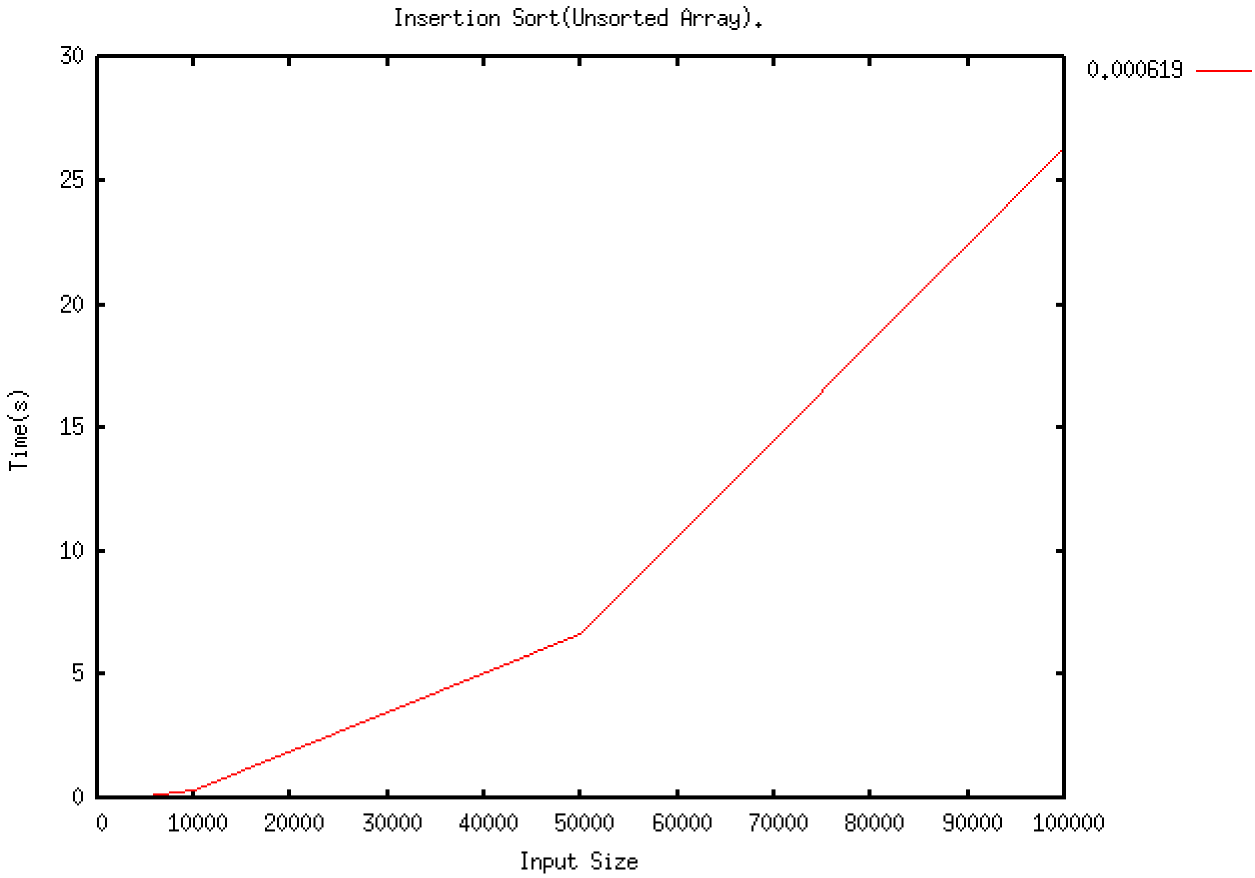
Graph :



3).Unsorted Data(Random)

size	time
500	0.000619
1000	0.003895
5000	0.104801
10000	0.288844
50000	6.67511
100000	26.36

Graph :



Insertion Sort(Unsorted Array).
0.000619
Time(s)
30
25
20
15
10
5
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Input Size

Rank_Sort(A)

Rank Sort works on the basis of giving ranking to each and every element of the list according to the ranking of the element in that array. It preserves the order of the same element in which they exist in the unsorted array.

It is not in-place algorithm and thus requires additional space to sort the list hence two additional array of size n is required hence space complexity is of order n.

Pseudo Code :

```
1. for j=1 to n
2.       R[j]=1
            //Rank the n elements in A into R
3. for j=2 to n
4.        for i=1 to j-1
5.            if A[i] <= A[j]
6.                R[j]=R[j]+1
7.             else
8.                R[i]=R[i]+1
            //Move to correct place in U[1......n]
9. for j=1 to n
10.      U[R[j]] = A[j]
           //Move the sorted entries into A
11. for j=1 to n
12.     A[j] = U[j]
```

Asymptotic Running time analysis

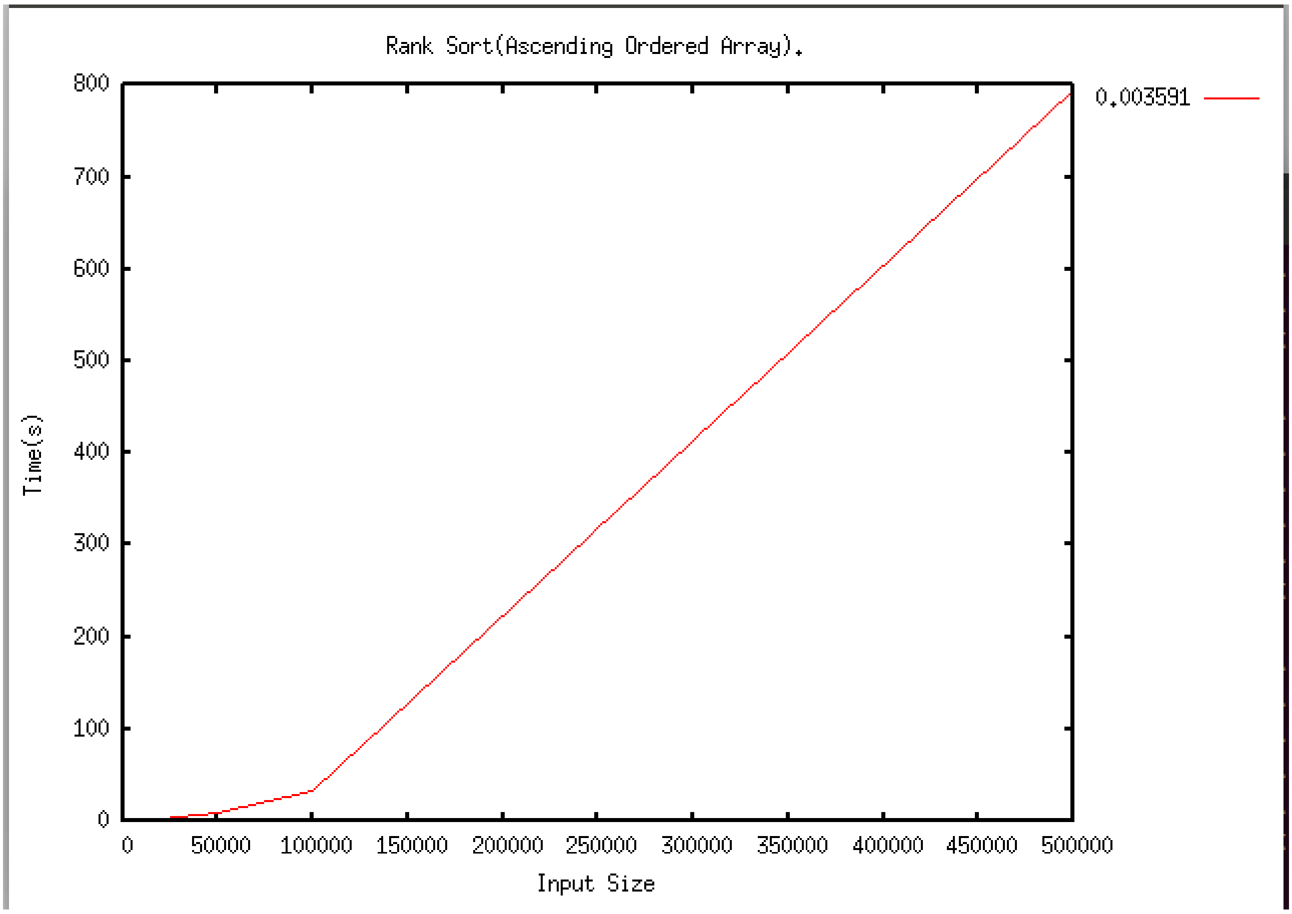
1. Best Case : $T(n)=O(n^2)$ When array is already sorted
2. Worst Case : $T(n)=O(n^2)$ When array is in Descending order
3. Average Case : $T(n)=O(n^2)$

Actual running time analysis :

1).Ascending Data

size	time
500	0.003591
1000	0.008510
5000	0.081267
10000	0.314393
50000	7.898296
100000	31.551628
500000	794.180916

Graph :

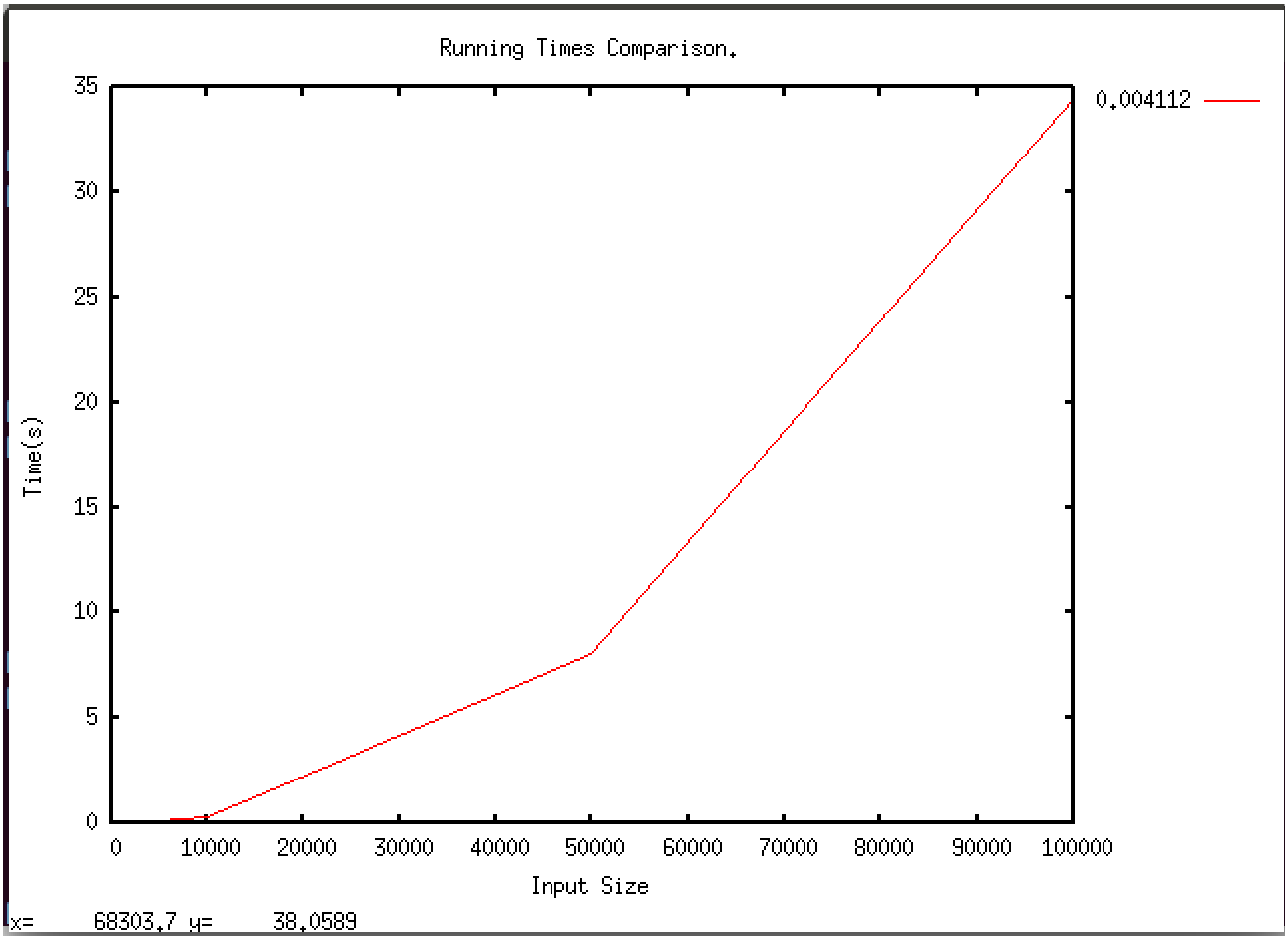


2).Descending Data

size	time
500	0.004112
1000	0.004087
5000	0.095907
10000	0.325427

50000	8.02484
100000	34.4911

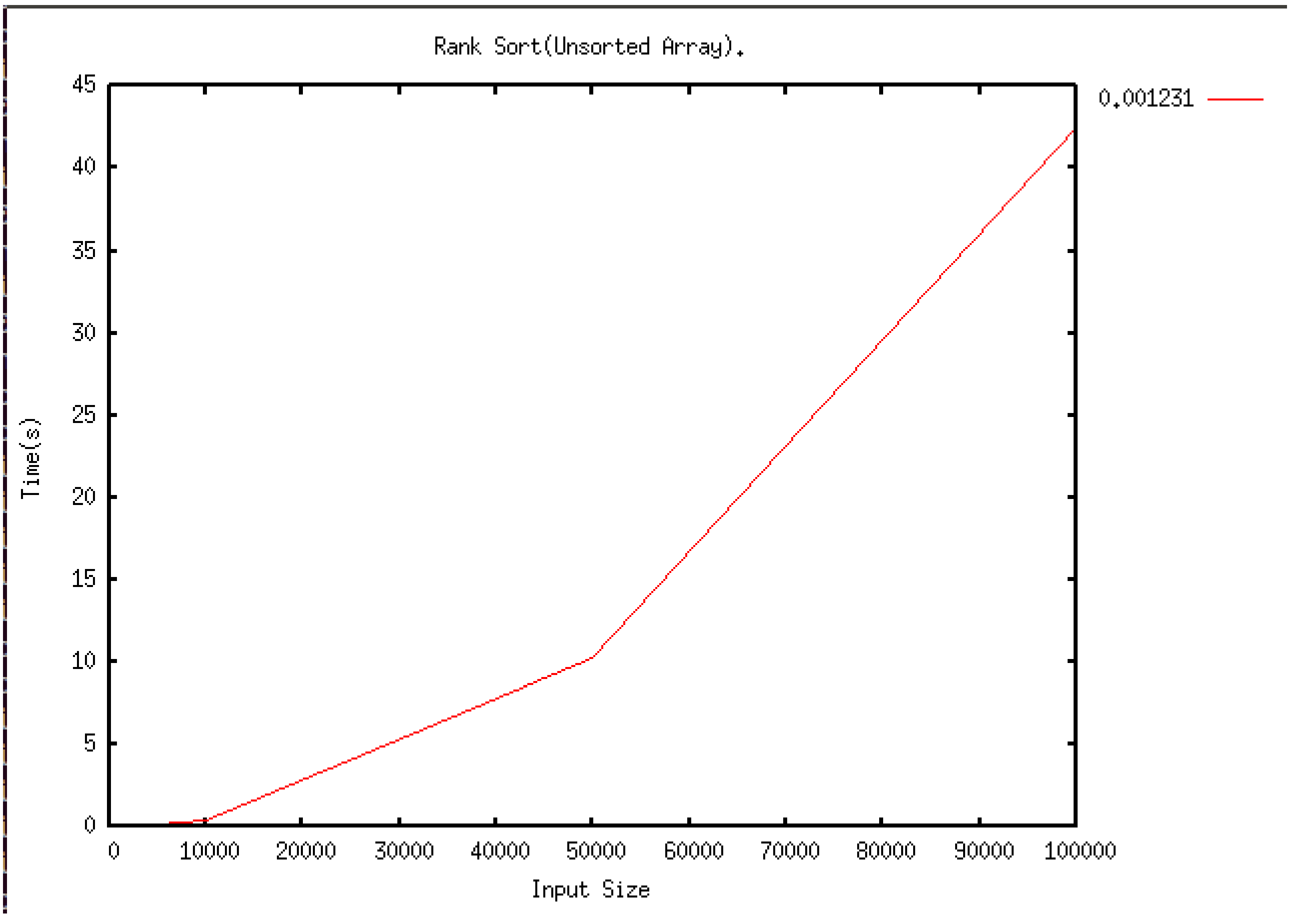
Graph :



3).Unsorted Data(Random)

size	time
500	0.001231
1000	0.006141
5000	0.156197
10000	0.410219
50000	10.2227
100000	42.4783

Graph :


Rank Sort(Unsorted Array).
0.001231
Time(s)
Input Size
0
5
10
15
20
25
30
35
40
45
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000

Improved_Bubble_Sort(A)

It is the most simple algorithm that sort the elements by swapping the adjecent elements if they are in wrong order. It is an in-place algorithm and it is stable i.e. it maintains the order of same elements.

It is an in-place algorithm, thus does not require additional space. It's space complexity is order of n.

Pseudo Code :

```
j=n
while j>=2&&swap==1
    //Bubble up the  smallest element to its correct position
    swap=0
    for i=1 to j-1
        if A[i] > A[i+1]
            temp=A[i]
            A[i]=A[i+1]
            A[i+1]=temp
            swap=1 //indicates that loop ran
    j=j-1
```

Asymptotic Running time analysis :

1. Best Case : $T(n)=O(n)$
2. Worst Case : $T(n)=O(n^2)$
3. Average Case : $T(n)=O(n^2)$

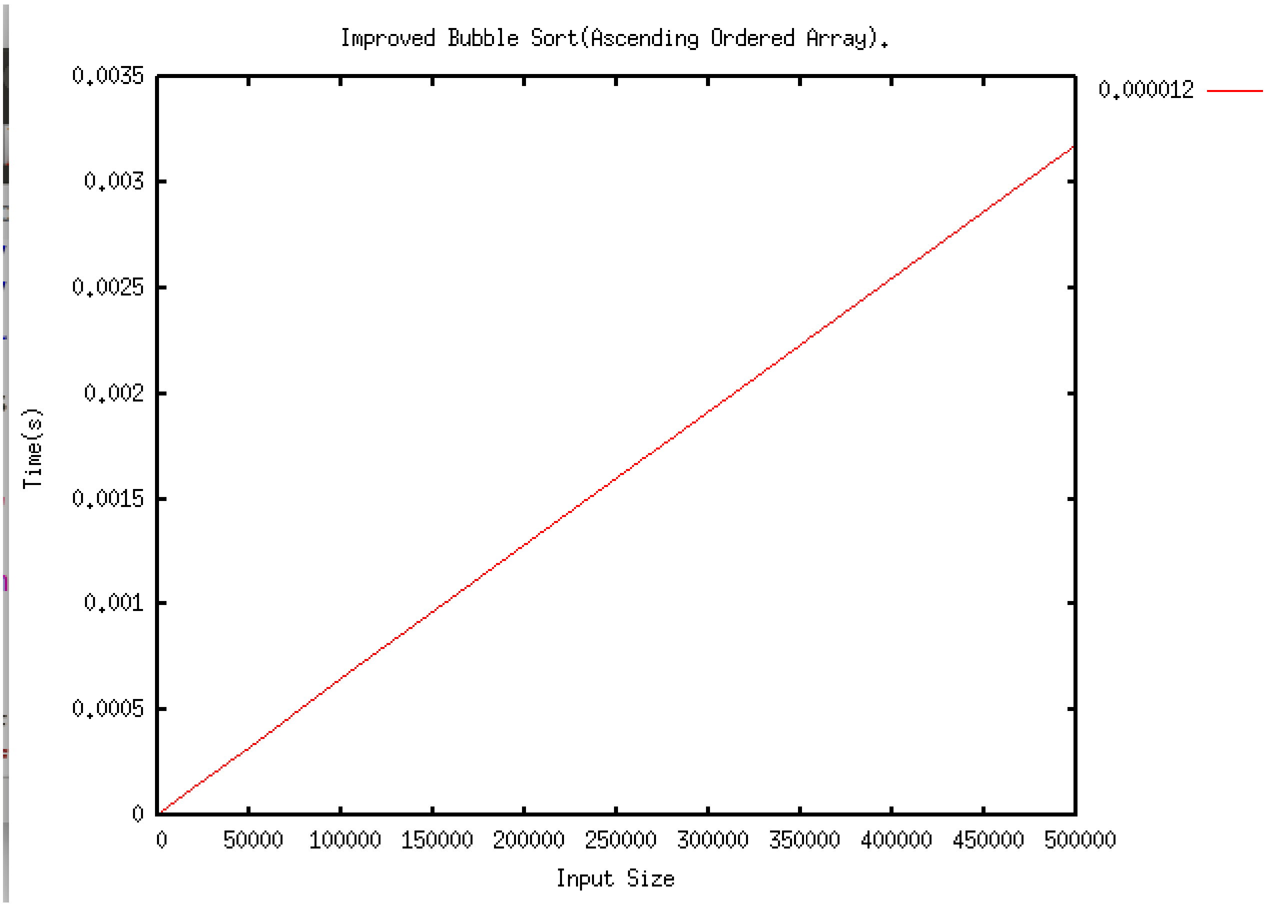
Actual Running time analysis :

1).Ascending Data

size	time
500	0.000012
1000	0.000013

5000	0.000038
10000	0.000077
50000	0.000323
100000	0.000650
500000	0.003181

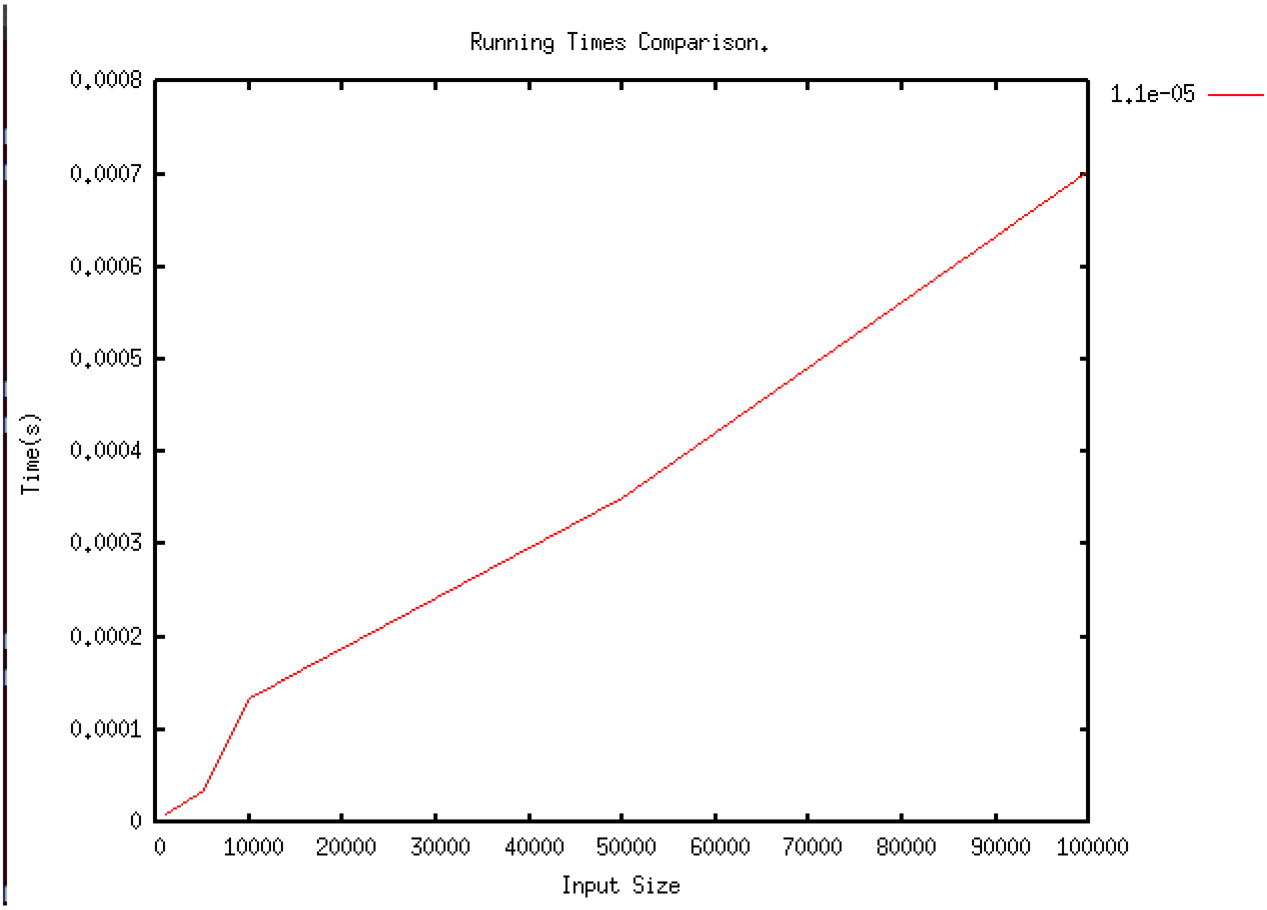
Graph :



2).Descending Data

size	time
500	1.1e-05
1000	9e-06
5000	3.5e-05
10000	0.000134
50000	0.000351
100000	0.000704

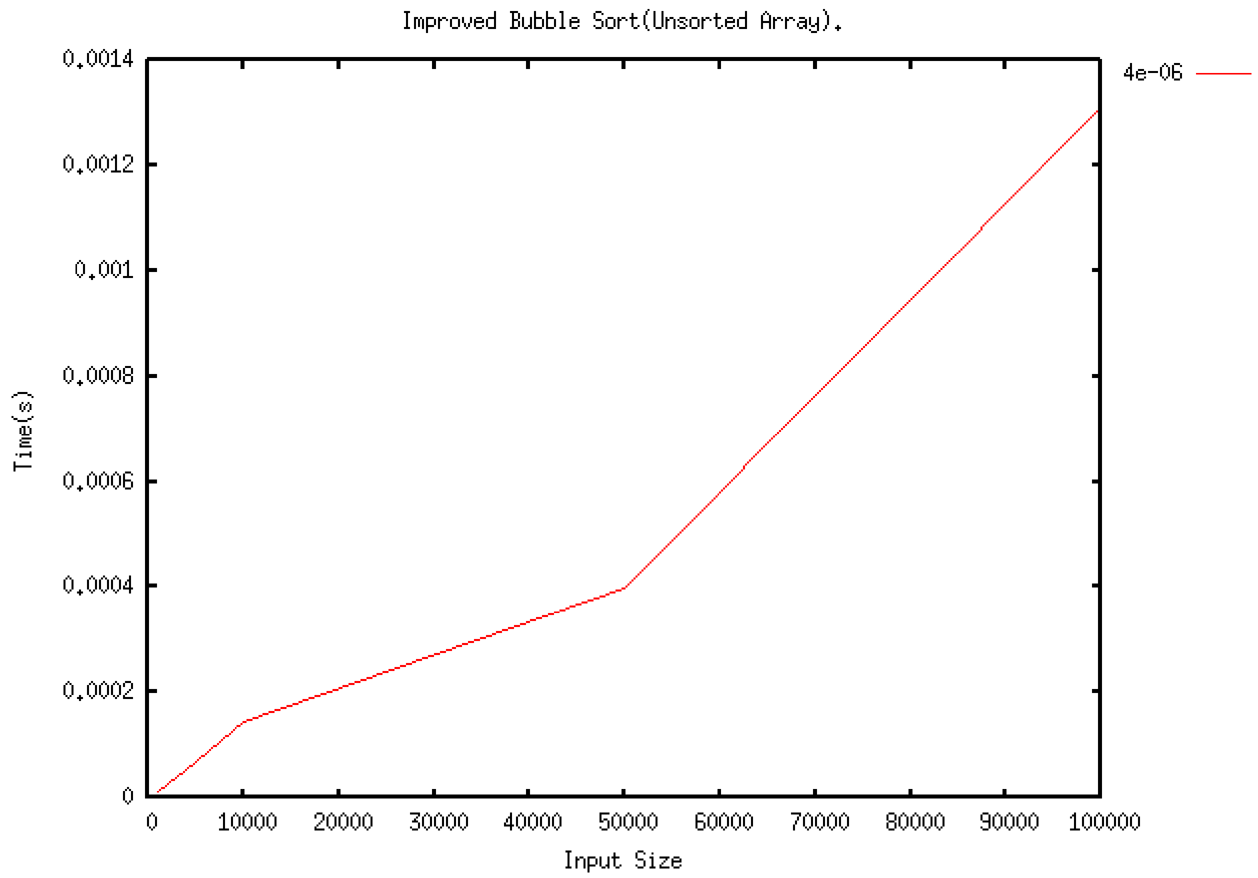
Graph :



3).Unsorted Data(Random)

size	time
500	4e-06
1000	1.3e-05
5000	6.7e-05
10000	0.000145
50000	0.000398
100000	0.00131

Graph :


Improved Bubble Sort(Unsorted Array).
4e-06
Time(s)
Input Size
0.0014
0.0012
0.001
0.0008
0.0006
0.0004
0.0002
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000

Selection_Sort(A)

The selection sort algorithm sorts an array by repeteadly selecting the maximum element from the unsorted array and put it into the last. Thus it maintain two sub array, the right sorted sub array and left unsorted sub array.

It's space complexity is order of n.

Pseudo Code :

```
sorted = false
j = n   // n is the number of element
while j>1 && sorted == flase
       pos = 1
       sorted = true
        // find the position of the largest element
              for I = 2 → j
                   if A[pos] <= a[i]
                          pos = I
               else
                          sorted = false      // Move A[j] to the
position of largest element by swapping

       temp = A[pos]
       A[pos] = A[j]
       A[j] = temp
        j = j-1
```

Asymptotic Running time analysis

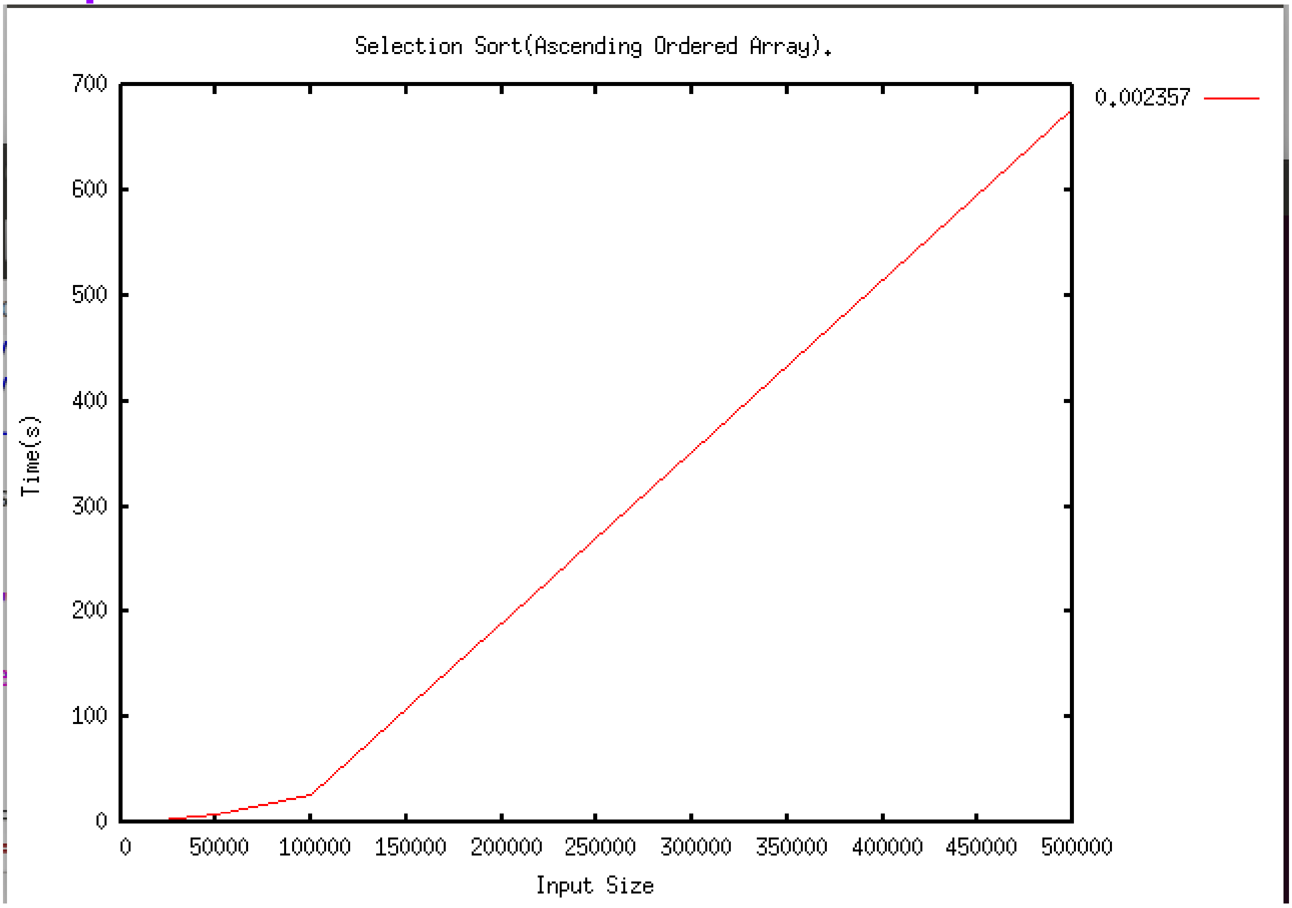
1. Best Case : $T(n)=O(n^2)$
2. Worst Case : $T(n)=O(n^2)$
3. Average Case : $T(n)=O(n^2)$

Actual Running time analysis :

1. Ascending Data

Size	Time
500	0.002357
1000	0.007609
5000	0.089332
10000	0.284157
50000	6.796366
100000	26.912760
500000	677.671686

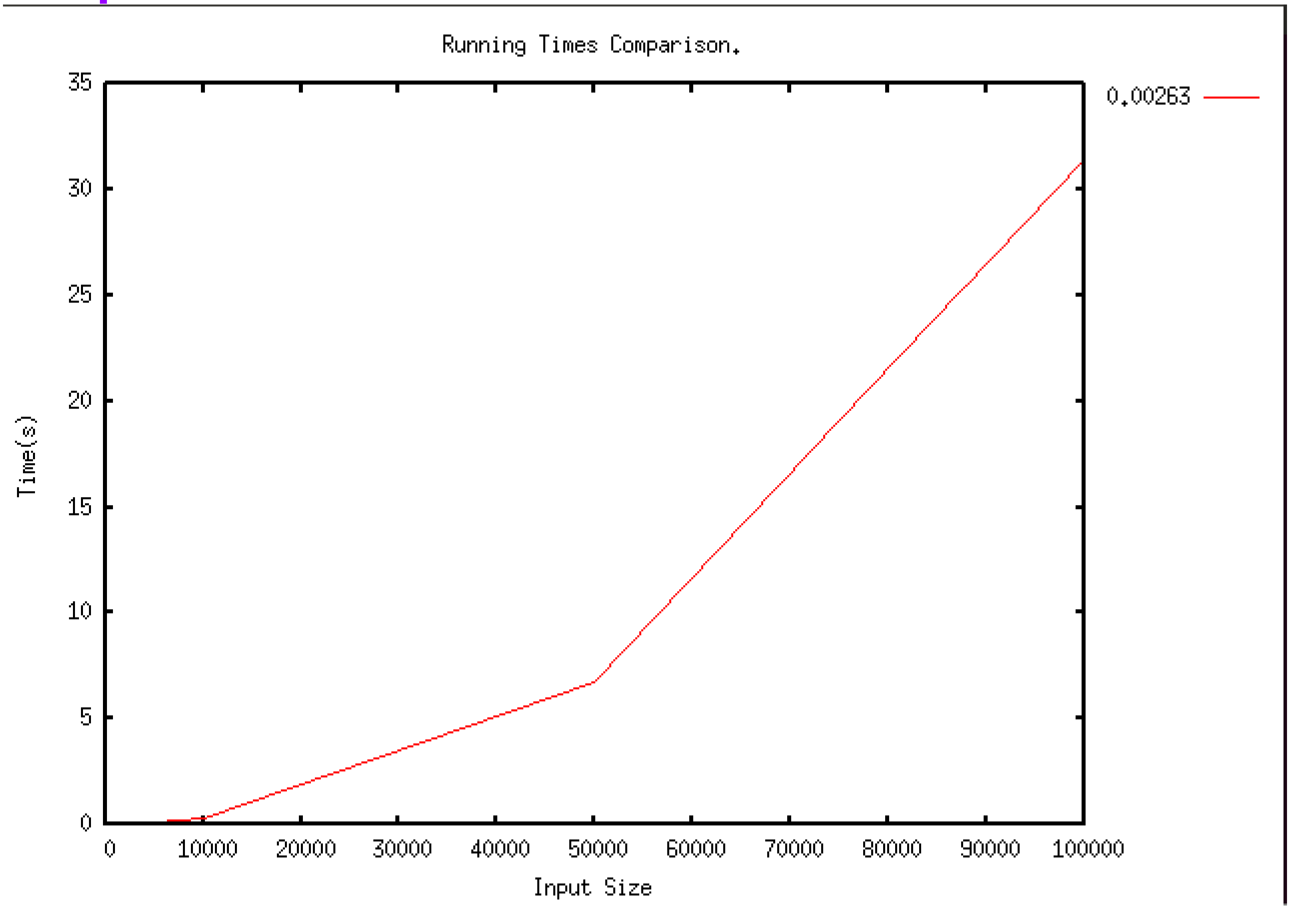
Graph :



2. Descending Data

Size	**Time**
500	0.00263
1000	0.005184
5000	0.082171
10000	0.291812
50000	6.70457
100000	31.3977

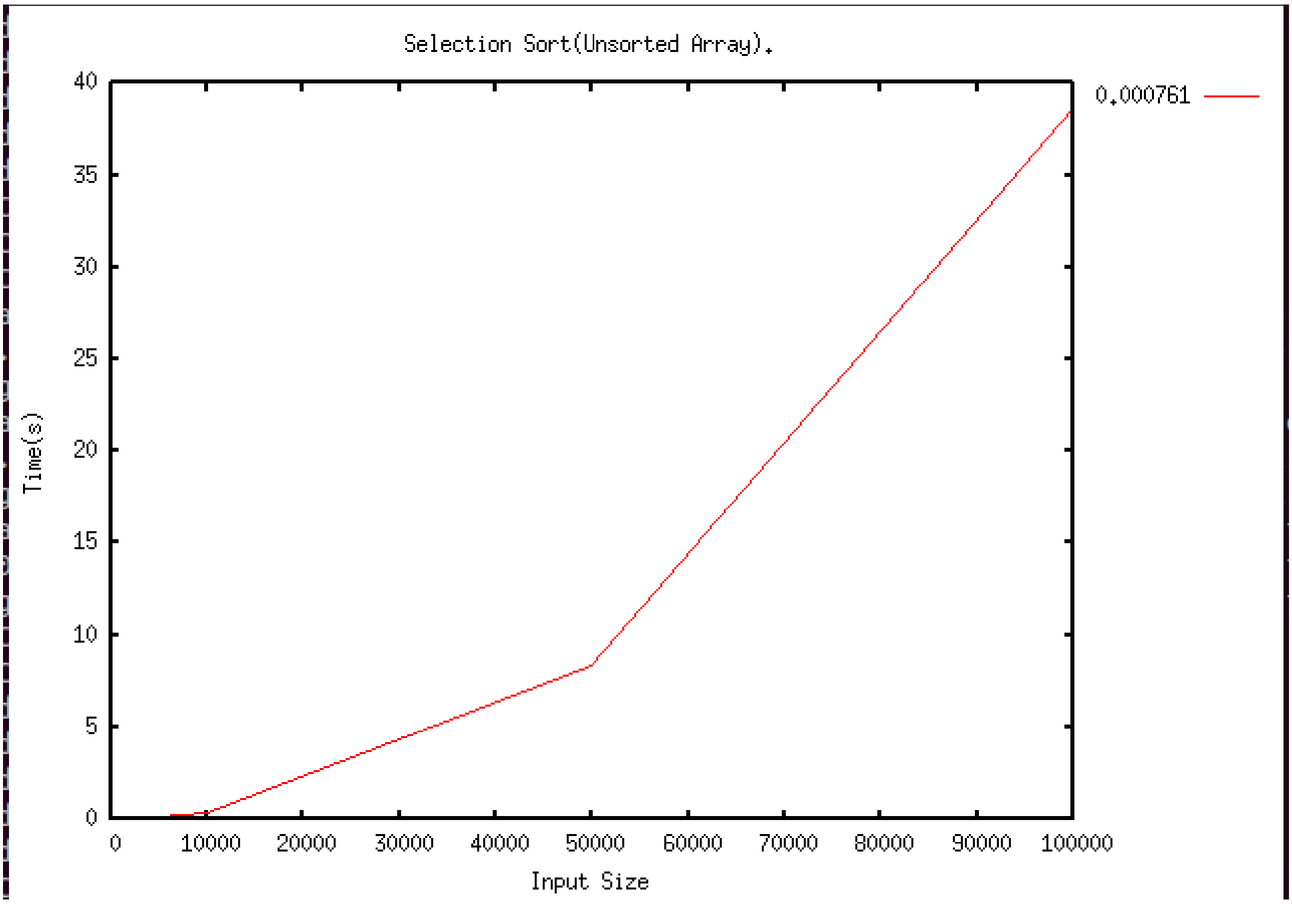
Graph :



3. Unsorted Data

Size	Time
500	0.000761
1000	0.004815
5000	0.126104
10000	0.357761
50000	8.38794
100000	38.6695

Graph :



Selection Sort(Unsorted Array).
0.000761
Time(s)
40
35
30
25
20
15
10
5
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Input Size

Bubble_Sort(A)

It is the most simple algorithm that sort the elements by swapping the adjecent elements if they are in wrong order. It is an in-place algorithm and it is stable i.e. it maintains the order of same elements.

It is an in-place algorithm, thus does not require additional space. It's space complexity is order of n.

Pseudo code :

```
1. j = n

2. while j>=2   // Bubble up the smallest element to its correct position
3.          for I = 1 to j-1
4.                   ifA[i] > A[i+1]  // swapping the element
5.                   temp = A[i]
6.                   A[i] = A[i+1]
7.                   A[i+1] = temp
8.  j = -1
```

Asymptotic Running time analysis

1. Best Case : $T(n)=O(n)$
2. Worst Case : $T(n)=O(n^2)$
3. Average Case : $T(n)=O(n^2)$

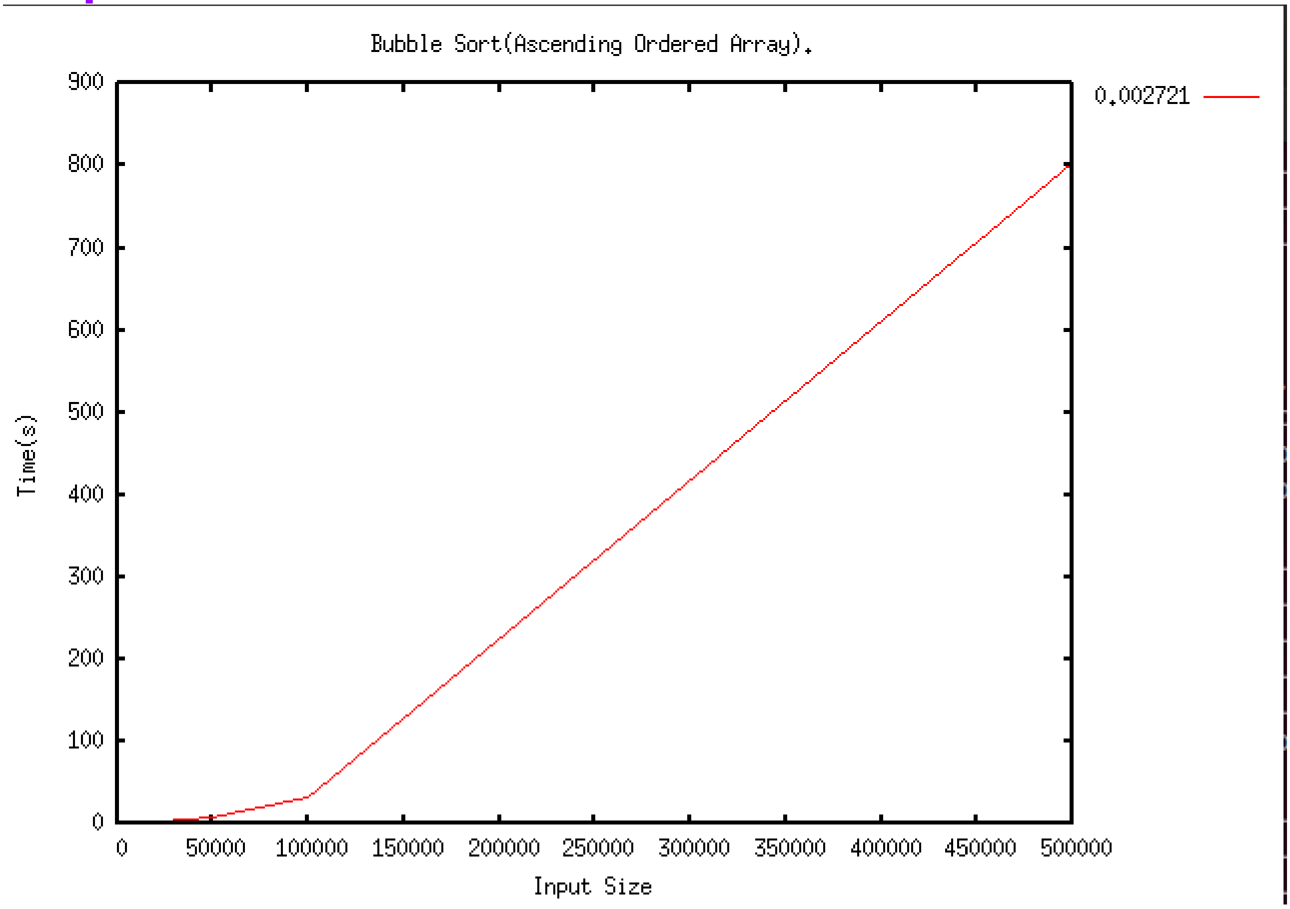
Asymptotic Running time analysis

1. Ascending Data

Size	Time
500	0.002721
1000	0.006099
5000	0.080256
10000	0.317477

50000	8.000231
100000	31.905337
500000	804.584724

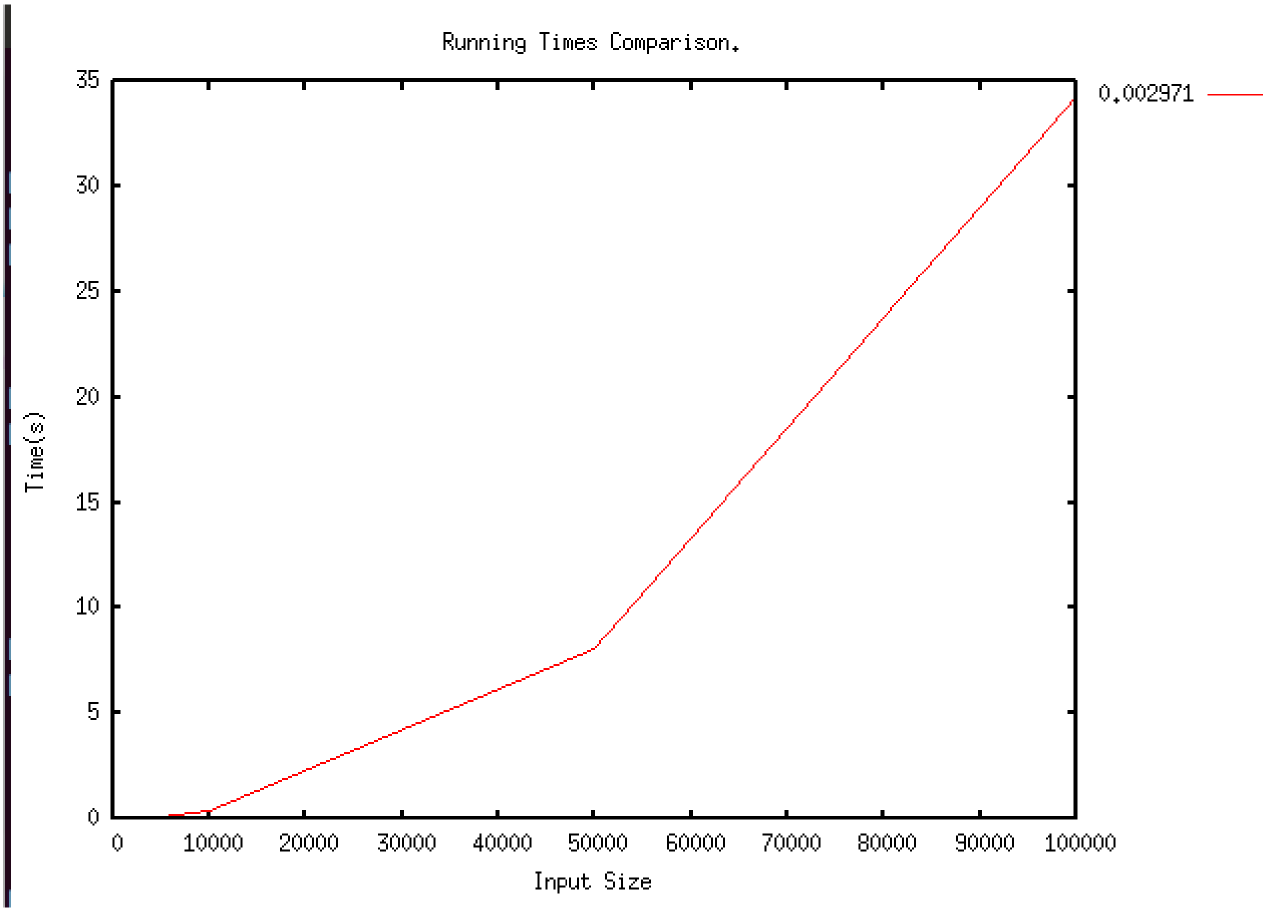
Graph :



2. Descending Data

Size	Time
500	0.002971
1000	0.005634
5000	0.096003
10000	0.340091
50000	8.01305
100000	34.2772

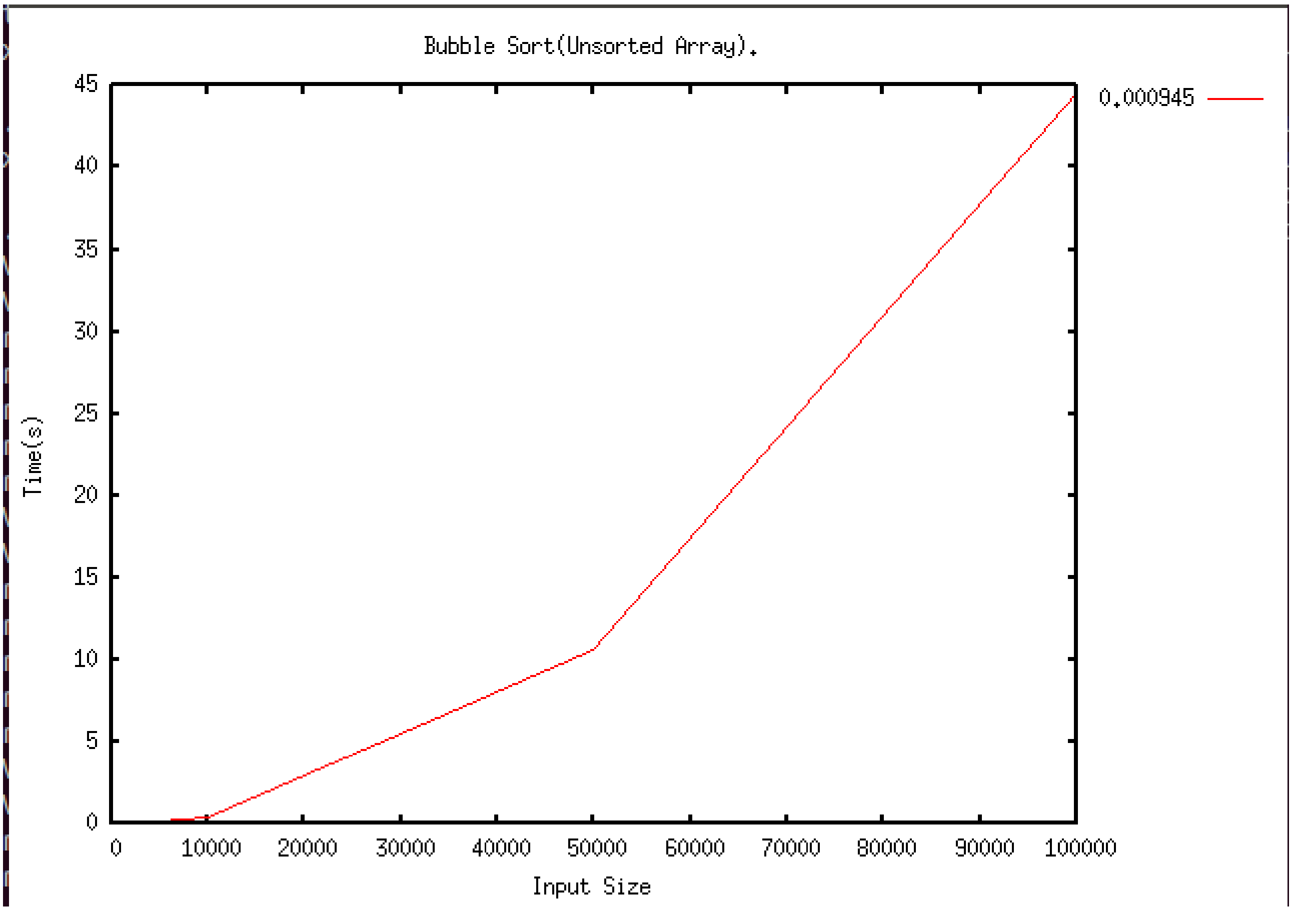
Graph :



3. Unsorted

Size	Time
500	0.000945
1000	0.006184
5000	0.139199
10000	0.416521
50000	10.6073
100000	44.5914

Graph :



Bubble Sort(Unsorted Array).
0.000945
Time(s)
Input Size
0
5
10
15
20
25
30
35
40
45
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000

Merge_Sort(A,p,r)

Algorithm of merge sort works on divide and conquer rule. It divides the unsorted array into two equal halves and call itself for two halves and then merge them. The merge() is used for merging two equal halves. At the stage where no more division is possible it sorts the elements in their correct order.

Space complexity is of order n.

Pseudo Code :

MERGE-SORT (A,p,r)

```
1.   if p<r  then
2.   q = [(p-r)/2]
3.   MERGE-SORT(A,p,r)
4.   MERGE-SORT(A,p,q,r)
5.   MERGE-SORT(A,q+1,r)
```

// Intial call : MERGE-SORT(A,p,r)

MERGE (A,p,q,r)

```
6.    n1 = q – p+1
7.    n2 = r – q8.
8.    A1[n1+1] = infinite
9.    A2[n2+1] = infinite
10.  j =1
11.  I =1
12.      for k=p to r
13.              ifA1[k] <= A2[J]
14.                         A[k] = A1[j]
15.                         I = I + 1
16.              else
17.                         A[k] = A2[j]
18.                         j = j + 1
```

Asymptotic Running time analysis

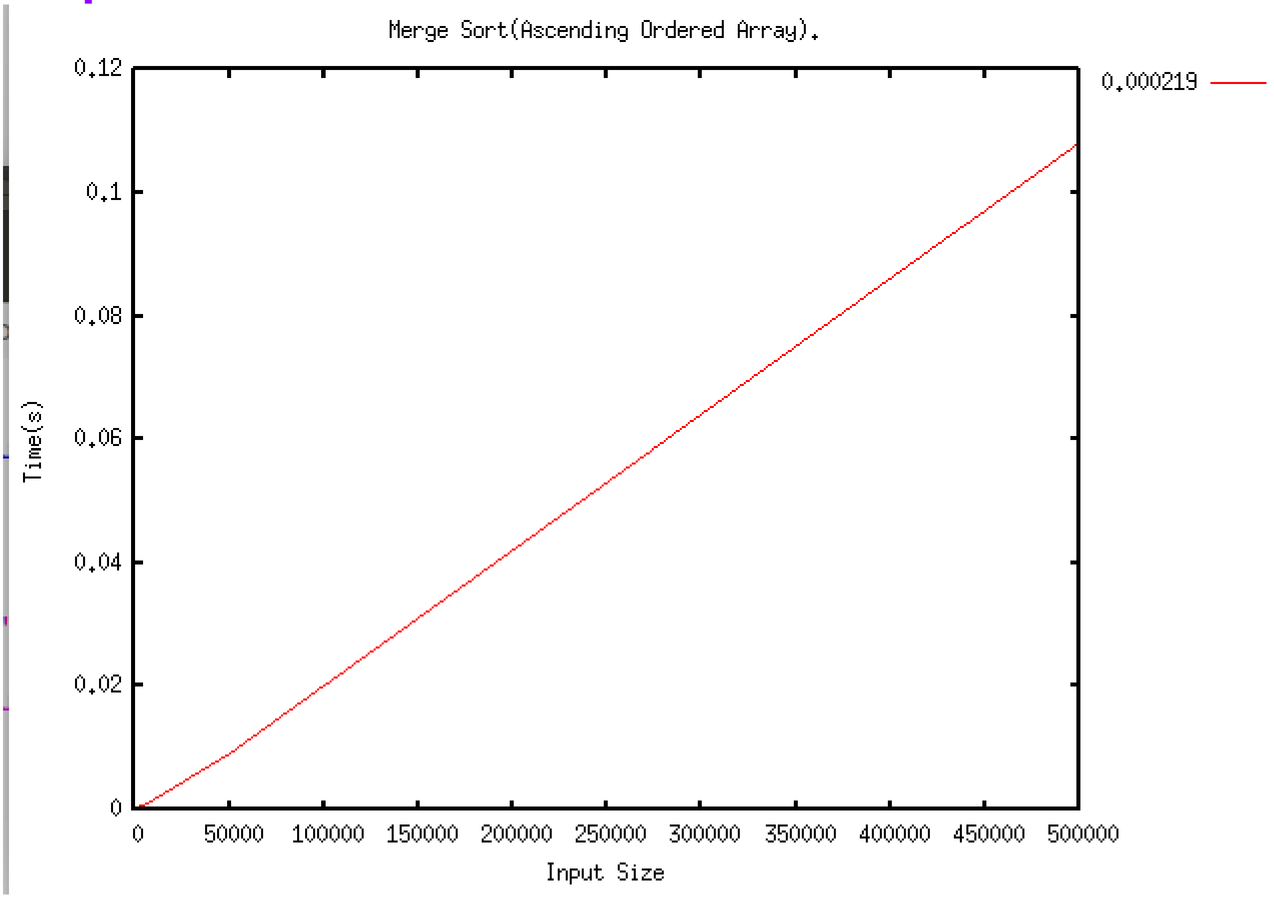
1. Best Case : T(n)=O(n*log(n))
2. Worst Case : T(n)=O(n*log(n))
3. Average Case : T(n)=O(n*log(n))

Actual Running time analysis

1. Ascending Data

Size	Time
500	0.000219
1000	0.000259
5000	0.000769
10000	0.001658
50000	0.009165
100000	0.019990
500000	0.108018

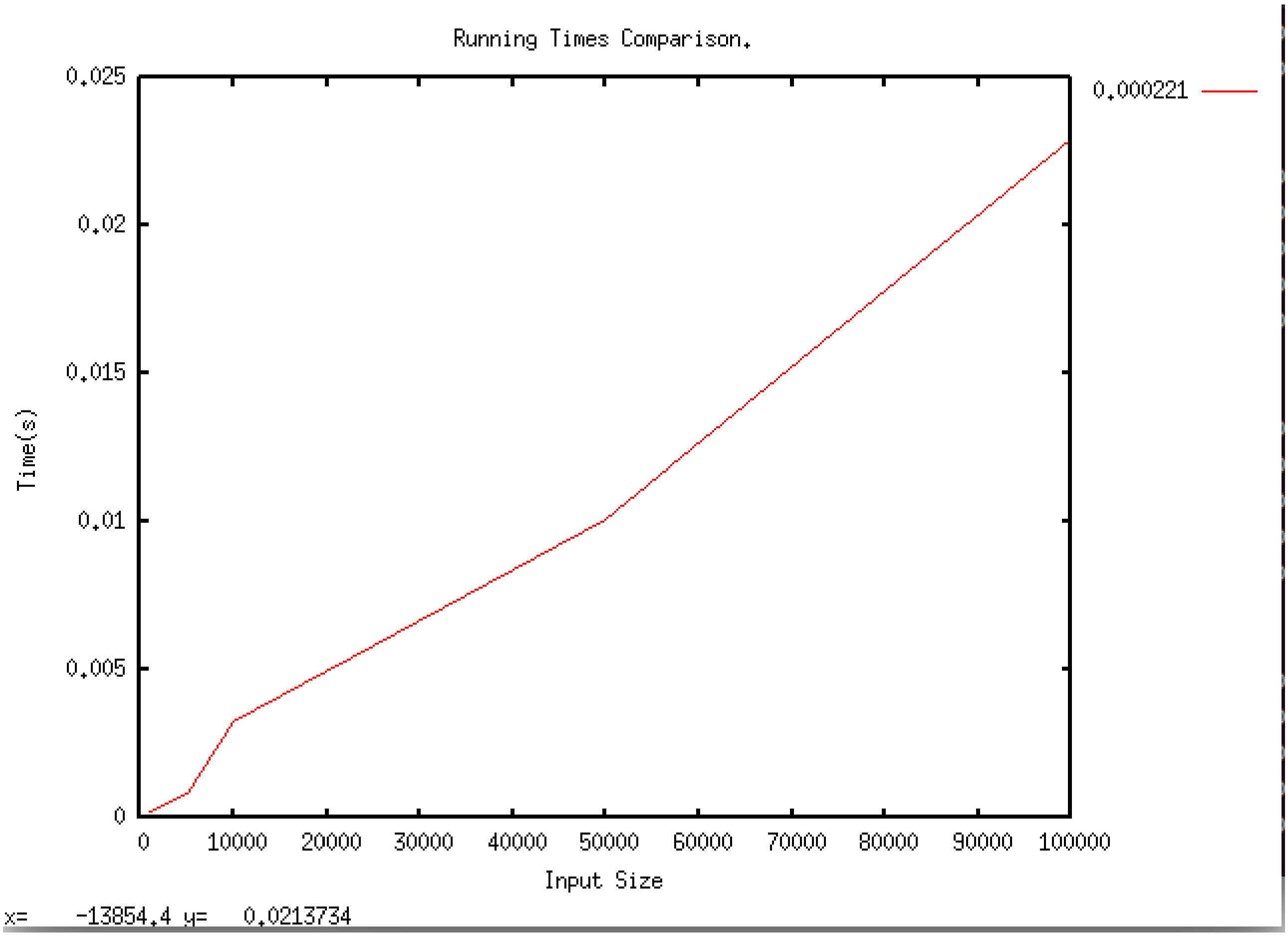
Graph :



2. Descending Data

Size	Time
500	0.000221
1000	0.000173
5000	0.000779
10000	0.003224
50000	0.010079
100000	0.022908

Graph :

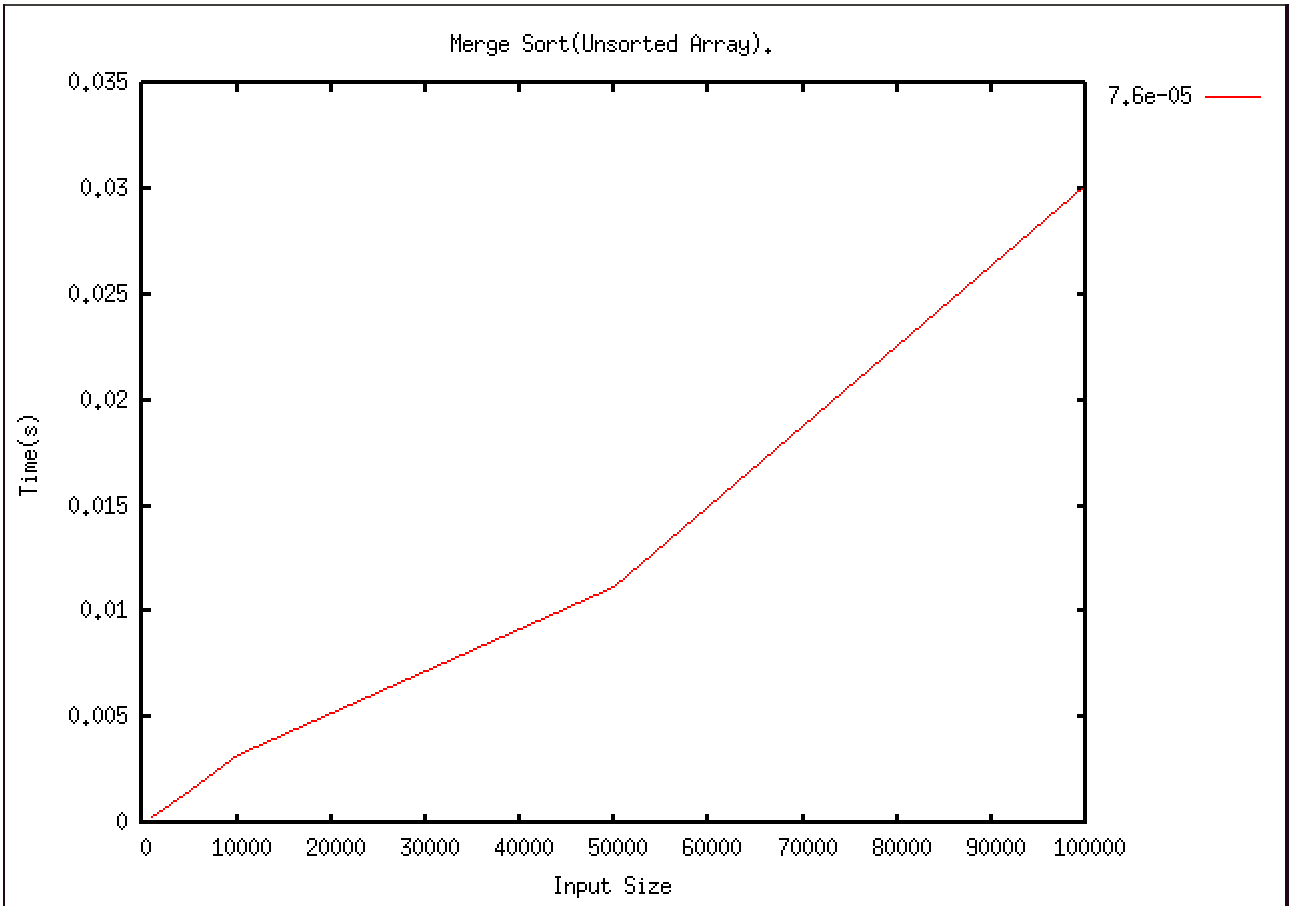


3. Unsorted

Size	Time
500	7.6e-05
1000	0.000259

5000	0.001523
10000	0.003211
50000	0.011204
100000	0.030196

Graph :



Quick_Sort(A,p,r)

It also follows devide and conquer method. It picks an element called pivot element and produce partiotion around that pivot element. It is very fast and efficient algorithm, in-place and stable. It takes place in three steps:

1. Divide : Partition around pivot element such that left sub part contains all elements that are smaller than pivot and right sub part contains larger elements.
2. Conquer : Sort left and right sub part recursively.
3. Combine : Combines two sub parts.

Space complexity is of order n.

Pseudo Code :

QUICK-SORT (A, p, r)

```
if p<r   than
      q = PARTITION (A,p,r)
      QUICK – SORT(A,p,q)
      QUCIK – SORT(A,q+1,r)

PARTITION(A,p,r)

pivot = A[r]
I = p – 1
j = q – 1
while TRUE
do
j = j – 1
      while A[j] > pivot
if j >i
exchange A[i] with A[j]
else if j = I
return j – 1
else
return j
```

Asymptotic Running time analysis

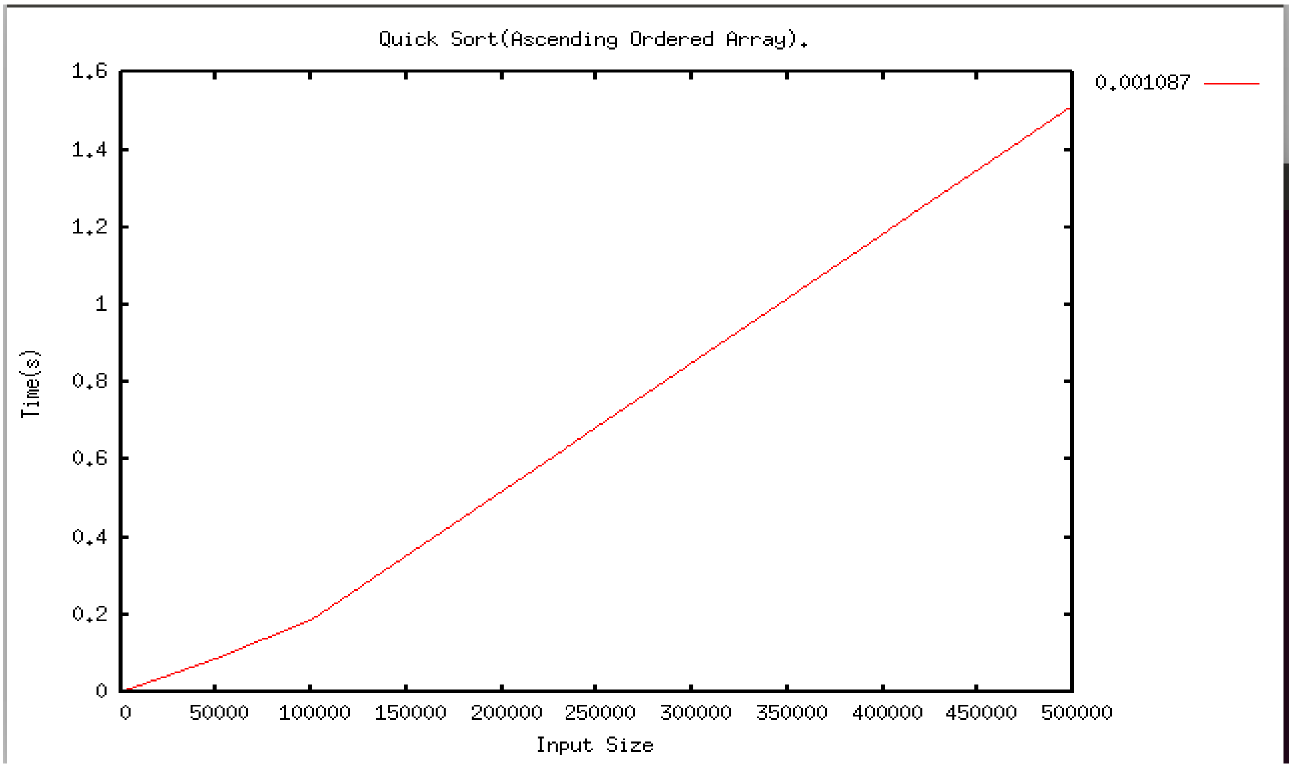
1. Best Case : T(n)=O(n*log(n))
2. Worst Case : T(n)=O(n*log(n))
3. Average Case : T(n)=O(n^2)

Actual Running time analysis

1. Ascending Data

Size	Time
500	0.001087
1000	0.001971
5000	0.007716
10000	0.016131
50000	0.085686
100000	0.184845
500000	1.513213

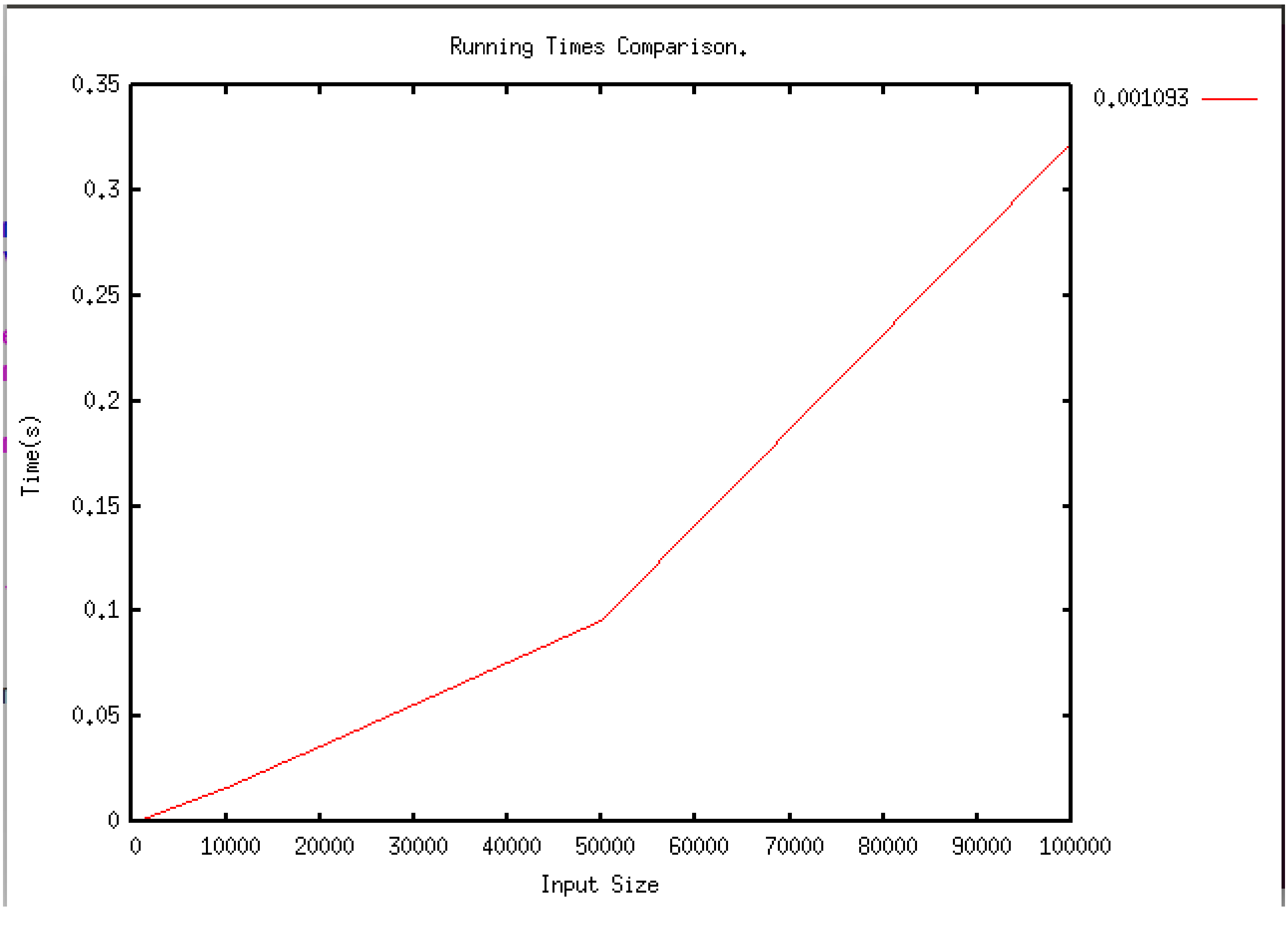
Graph :



2. Descending Data

Size	Time
500	0.001093
1000	0.001333
5000	0.007769
10000	0.016368
50000	0.095256
100000	0.323414

Graph :

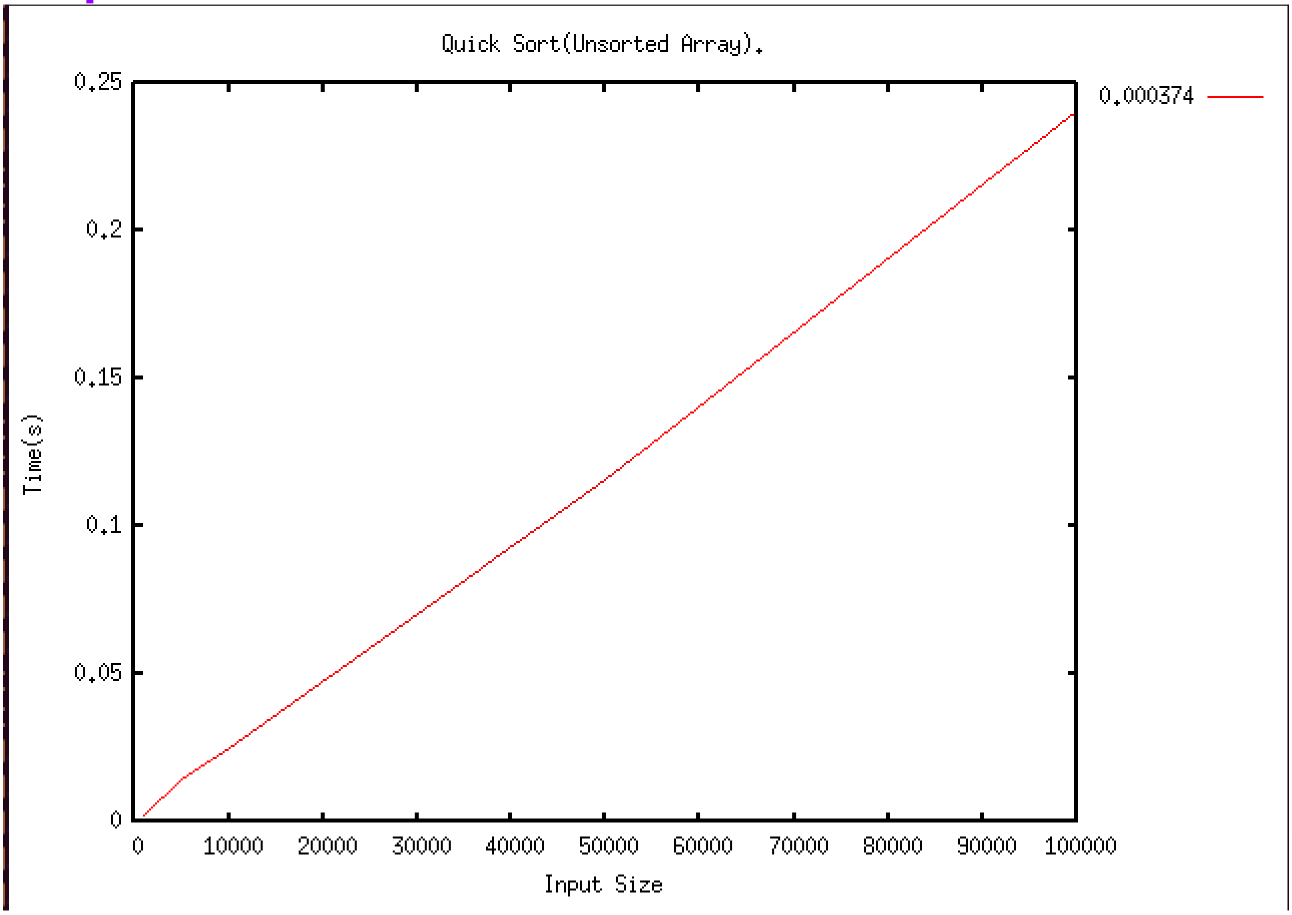


3. Unsorted

Size	Time
500	0.000374
1000	0.002296
5000	0.014319

10000	0.025348
50000	0.11585
100000	0.240888

Graph :

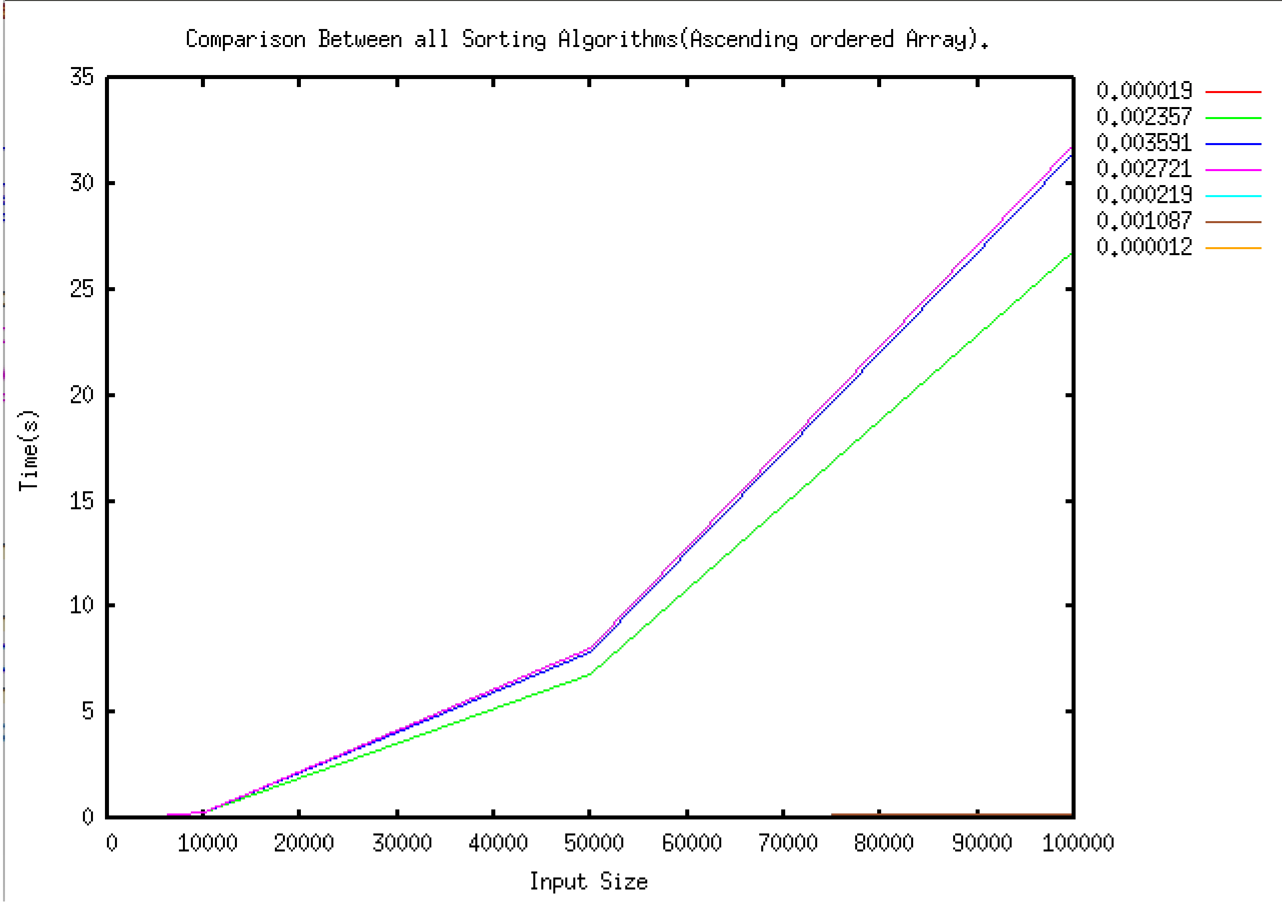


GRAPHICAL COMPARISON BETWEEN DIFFERENT SORTING ALGORITHMS

1).For Ascending ordered Array

Times starting with; represents

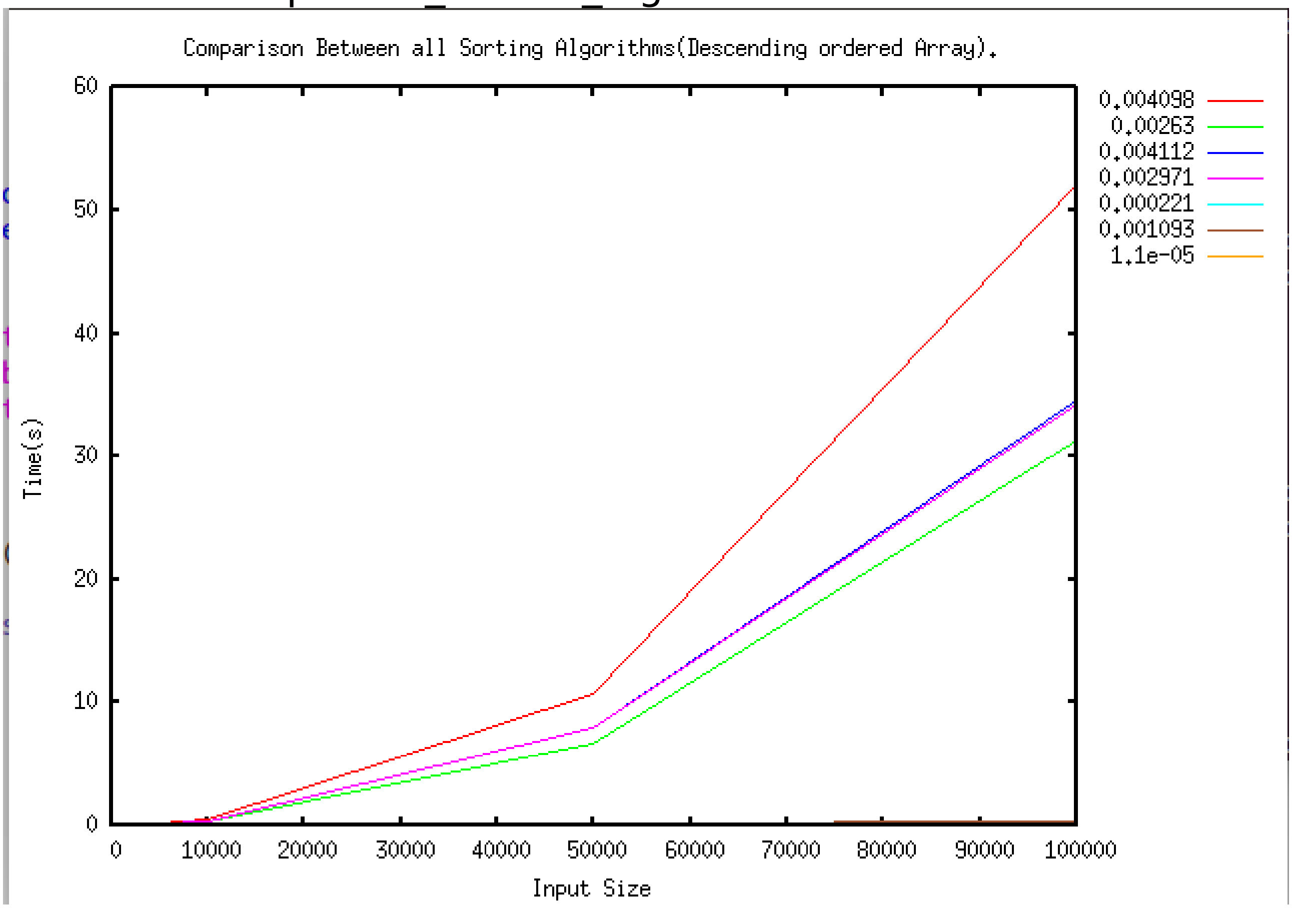
0.000019 - Insertion_algo
0.002357 - selection_algo
0.003591 - Rank_algo
0.002721 - Bubble_algo
0.000219 - Merge_algo
0.001087 - Quick_algo
0.000012 - Improved_bubble_algo



2).For Descending Ordered Array

Times starting with; represents

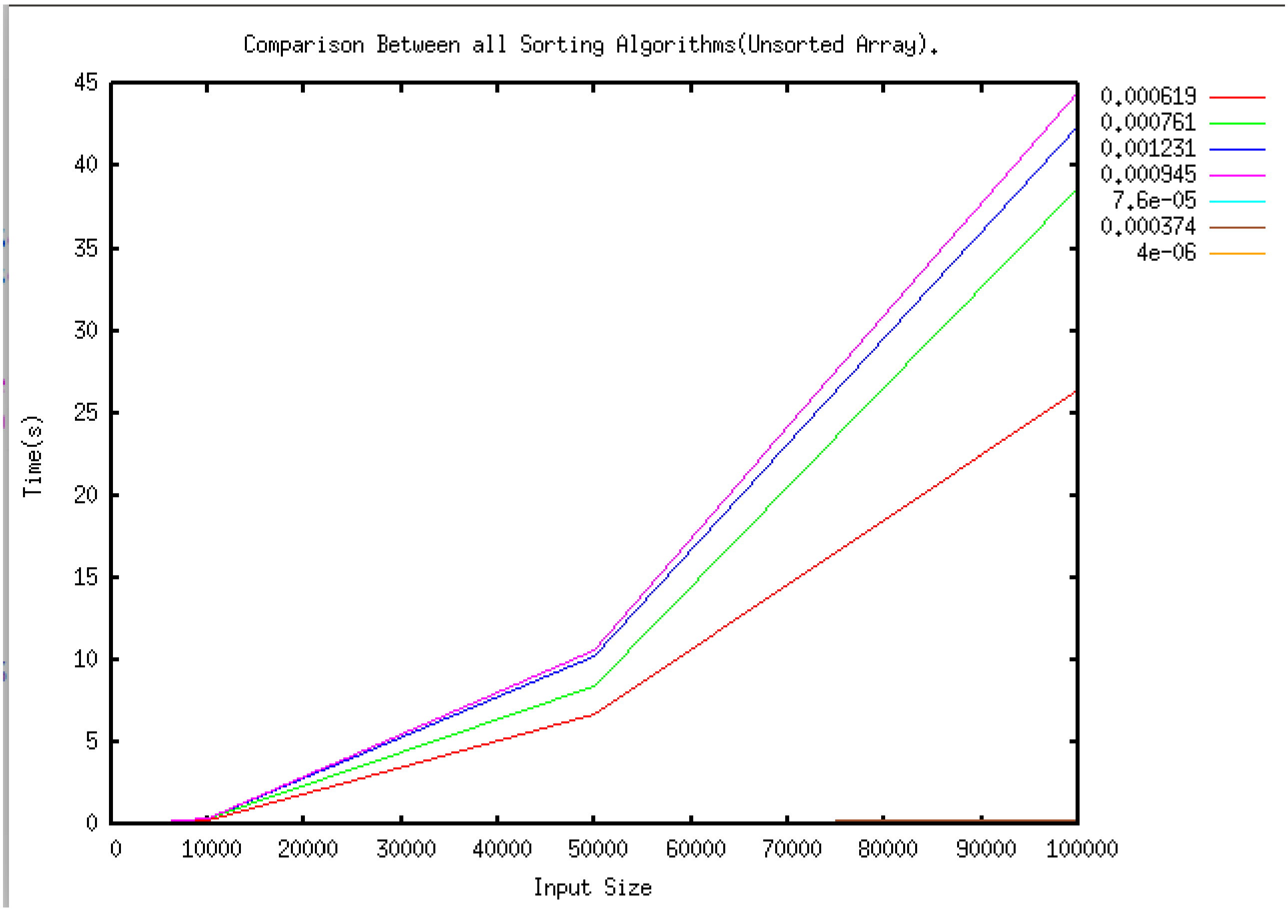
0.004098 - Insertion_algo
0.00263 - selection_algo
0.004112 - Rank_algo
0.002971 - Bubble_algo
0.000221 - Merge_algo
0.001093 - Quick_algo
1.1e-05 - Improved_bubble_algo



3).For Unsorted Array

Times starting with; represents

0.000619 - Insertion_algo
0.000761 - selection_algo
0.001231 - Rank_algo
0.000945 - Bubble_algo
7.6e-05 - Merge_algo
0.000374 - Quick_algo
4e-06 - Improved_Bubble_algo



TIME AND SPACE COMPLEXITY

Algorithm	Average	Best	worst	space
1.Bubble sort	O(n)	$O(n^2)$	$O(n^2)$	Constant
2.Insertion sort	$O(n^2)$	O(n)	$O(n^2)$	Constant

3.Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant
4.Rank sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	depends
5.Merge sort	$O(n*log(n))$	$O(n*log(n))$	$O(n*log(n))$	depends
6.Quick sort	$O(n*log(n))$	$O(n*log(n))$	$O(n^2)$	constant
7.Improved Bubble sort Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	Constant

THANKS