Advanced Data Structures and Algorithms

Assignment1

Running Time Analysis of Different Sorting Algorithms

By:

Hitesh Tarani

B13139

Bubble Sort

Pseudo Code

INPUT : A[1..n], array of integers

OUTPUT: Rearrangement of A such that A[1]≤A[2] ≤… ≤A[n]

BUBBLE-SORT (A)

// Sort by bubbling the smallest element to current position.

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

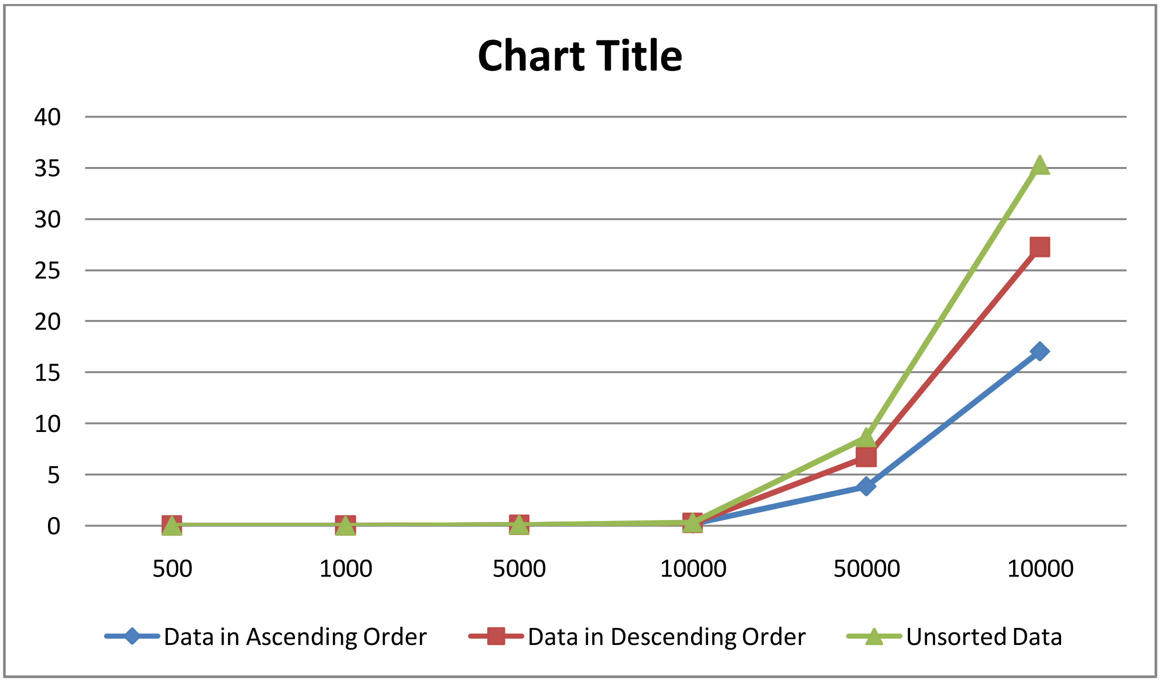
Execution time of Bubble Sort

Bubble Sort	Data in Ascending Order	Data in Descending Order	Unsorted Data
500	0.000468	0.000666	0.000801
1000	0.001551	0.003013	0.002443
5000	0.038537	0.067912	0.084762
10000	0.182936	0.266823	0.311252
50000	3.82375	6.67585	8.66513
10000	17.0451	27.2479	35.35
500000	-	-	-
1000000	-	-	-
5000000	-	-	-

Analysis of Bubble-sort Algorithm

1. It is clear from the data that bubble sort takes time proportional to n^2, for higher values of n in the average case. Hence time complexity of bubble sort is

$$f(n)=O(n^2)$$



2. **When data is already sorted, i.e. in ascending order, it takes minimum time as there is no need for swapping the elements The time complexity in best case is also $O(n^2)$.**
3. **When data is unsorted or is in descending order, the time complexity is $O(n^2)$, since if condition is executed mostly(unsorted) or every time(descending).**

Rank Sort

Pseudo Code

INPUT : A[1..n], array of integers

OUTPUT: Rearrangement of A such that A[1]≤A[2] ≤… ≤A[n]

```
RANK-SORT (A)
for j = 1 to n
    R[j] = 1
// Rank the n elements in A into R
for j = 2 to n
    for i = 1 to j - 1
        if A[i] <= A[j]
            R[j] = R[j] + 1
        else
            R[i] = R[i] + 1
// Move to correct place in U[1 . . n]
for j = 1 to n
    U[R[j]] = A[j]
// Move the sorted entries into A
for j = 1 to n
    A[j] = U[j]
```

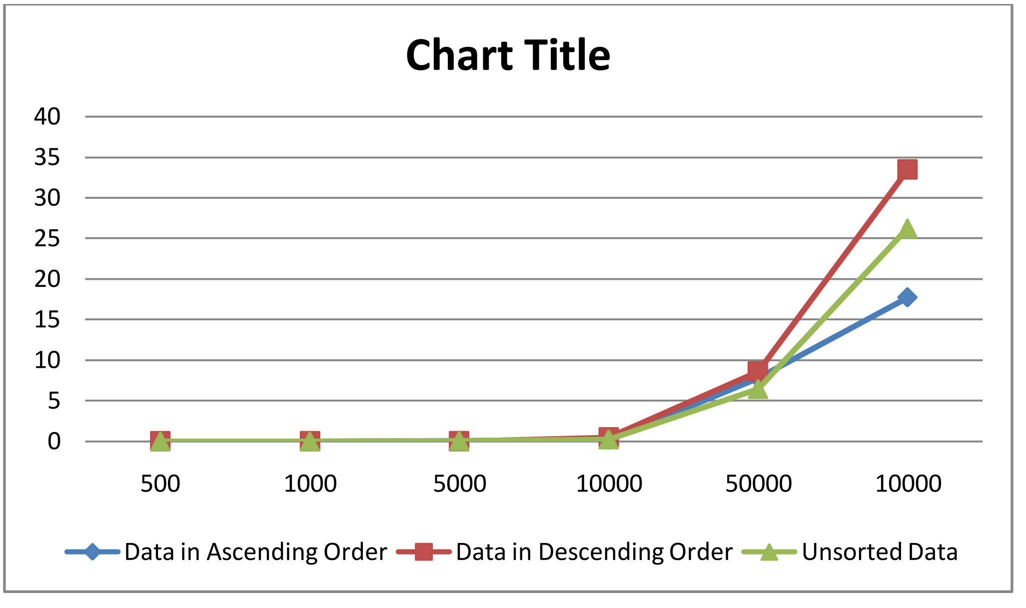
Execution time of Rank Sort

Rank Sort	Data in Ascending Order	Data in Descending Order	Unsorted Data
500	0.000444	0.00829	0.000645
1000	0.001701	0.003319	0.003616
5000	0.044327	0.02605	0.063798
10000	0.170303	0.471555	0.247407
50000	4.58656	8.5844	6.45994
10000	17.7091	33.4549	26.2432
500000			
1000000			
5000000			

Analysis of Rank-sort Algorithm

1. **It is clear from the data that rank sort takes time proportional to n^2, for higher values of n in the average case.**
   Hence overall time complexity of rank sort is

$$f(n)=O(n^2)$$



2. When data is already sorted, i.e. in ascending order, it takes minimum time. The time complexity in best case is $O(n^2)$.
3. The time taken is maximum in the worst case, when it is reversely sorted. The time complexity in worst case is also $O(n^2)$.
4. The average case takes time around between ascending and descending ordered data.

Insertion Sort

Pseudo Code

INPUT : A[1..n], array of integers

OUTPUT: Rearrangement of A such that A[1]≤A[2] ≤... ≤A[n]

INSERTION-SORT (A)

```
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
```

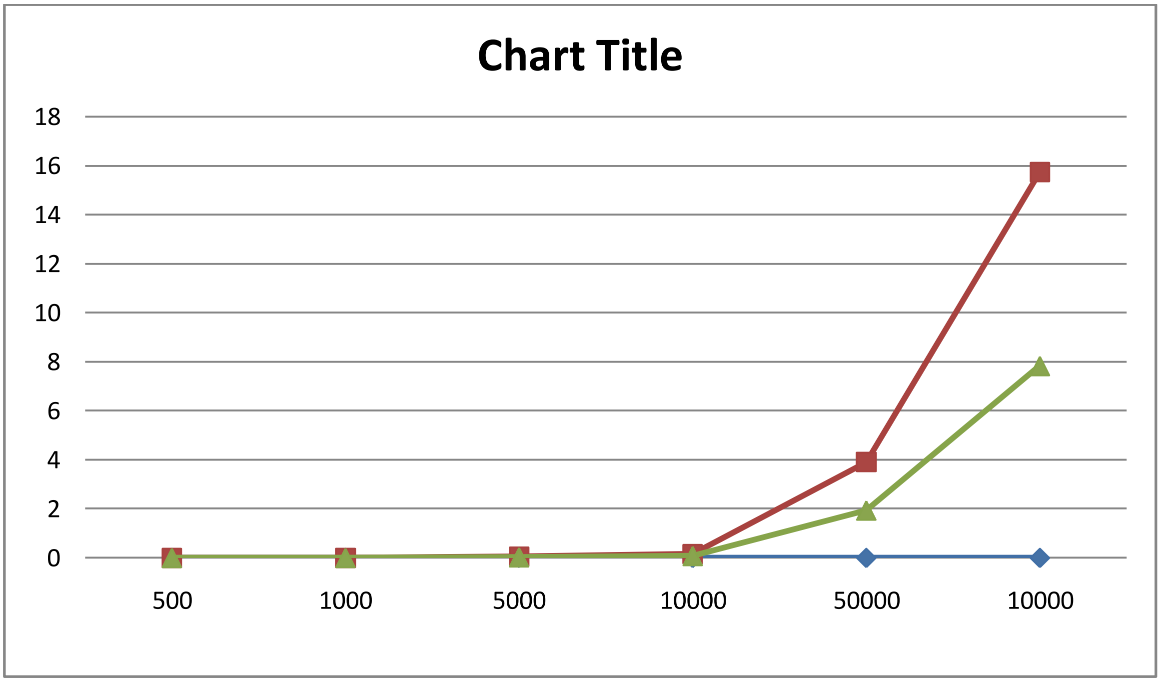
Execution time of Insertion Sort

Insertion Sort	Data in Ascending Order	Data in Descending Order	Unsorted Data
500	0.000003	0.000399	0.000215
1000	0.000005	0.001575	0.000798
5000	0.000022	0.05257	0.026755
10000	0.000044	0.156774	0.0796
50000	0.000318	3.90099	1.93606
10000	0.000574	15.7235	7.83003
500000	0.001163		201.379
1000000	0.004093		
5000000	0.20417		

Analysis of Insertion sort Algorithm

1. **It is clear from the data that insertion sort takes time proportional to n^2, for higher values of n in the average case. Hence time complexity of insertion sort is**

$$f(n)=O(n^2)$$



2. **When data is already sorted, i.e. in ascending order, it takes minimum time since it needs to compare just one element. Thus, the time complexity of this best case is nearly O(n) (can be observed from data as wall).**
3. **When data is unsorted, the time complexity is $O(n^2)$, as it has to compare many preceding elements.**
4. **When data is reversely sorted, time complexity is also $O(n^2)$, as it has to compare every preceding element.**

Selection Sort

Pseudo code

INPUT : A[1..n], array of integers

OUTPUT: Rearrangement of A such that A[1]≤A[2] ≤… ≤A[n]

```
SELECTION-SORT (A)
//Determine the largest element and move it to A[n], next largest element into A[n-1] and so on.
sorted = false
j = n
while j > 1 and sorted = false
        pos = 1
        sorted = true
        // Find the position of the largest element
        for i = 2 to 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
```

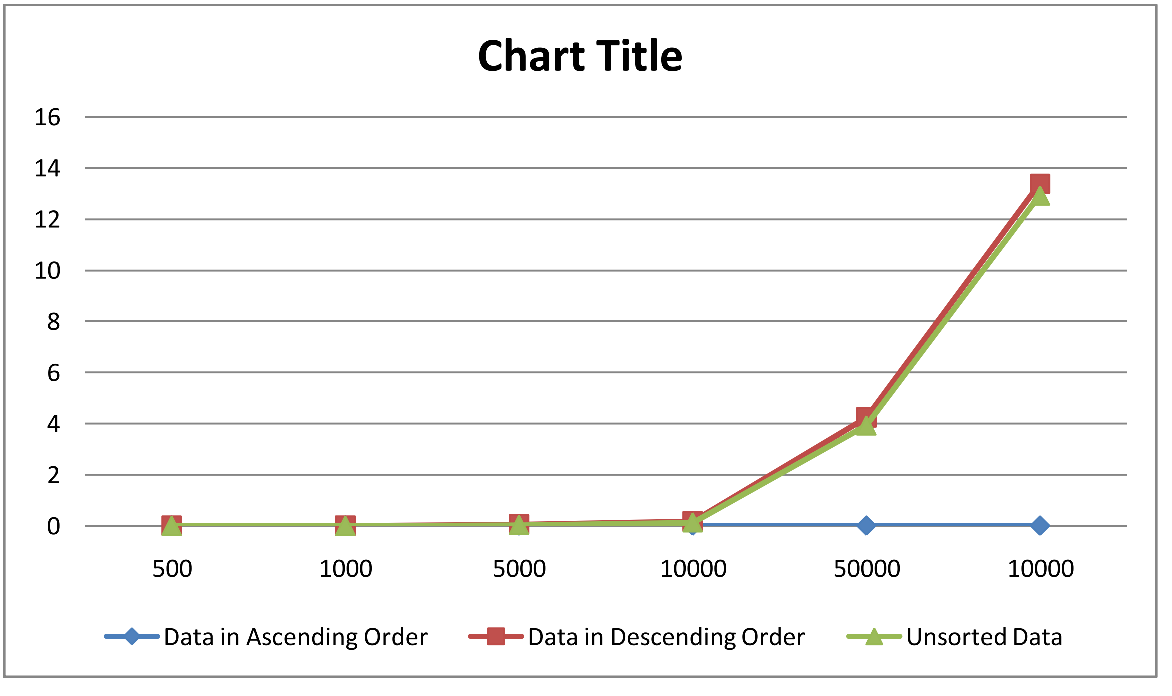
Execution time of Selection Sort

Selection Sort	Data in Ascending Order	Data in Descending Order	Unsorted Data
500	0.000003	0.000433	0.000413
1000	0.000004	0.001745	0.001305
5000	0.000017	0.046272	0.032312
10000	0.000028	0.16546	0.131588
50000	0.000138	4.23811	3.92678
10000	0.000275	13.38	13.9334
500000	0.00163		
1000000	0.002769		
5000000	0.013994		

Analysis of Selection sort Algorithm

1. It is clear from the data that selection sort takes time proportional to n^2, for higher values of n in the average case.
   Hence time complexity of selection sort is

$$f(n)=O(n^2)$$



2. When data is already sorted, i.e. in ascending order, it takes minimum time since it needs to run loop only once. The value of sorted will be true and hence loop will run only once. Thus, similar to insertion sort, the time complexity of this best case is nearly O(n) (can be observed from data as wall).
3. When data is unsorted, the average time complexity is $O(n^2)$, as it has to run loop almost all times, since the data is fully sorted only after running the loop every time.
4. When data is reversely sorted, time complexity is also $O(n^2)$, as it has to run loop every time.

Merge Sort

Pseudo Code

MERGE-SORT (A, *p*, *r*)

```
if  p < r  then
q = (p + r)/2
        MERGE-SORT (A, p, q)
        MERGE-SORT (A, q+1, r)
        MERGE (A, p, q, r)
```

Initial call: **MERGE-SORT (A, 1, *n*)**

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
```

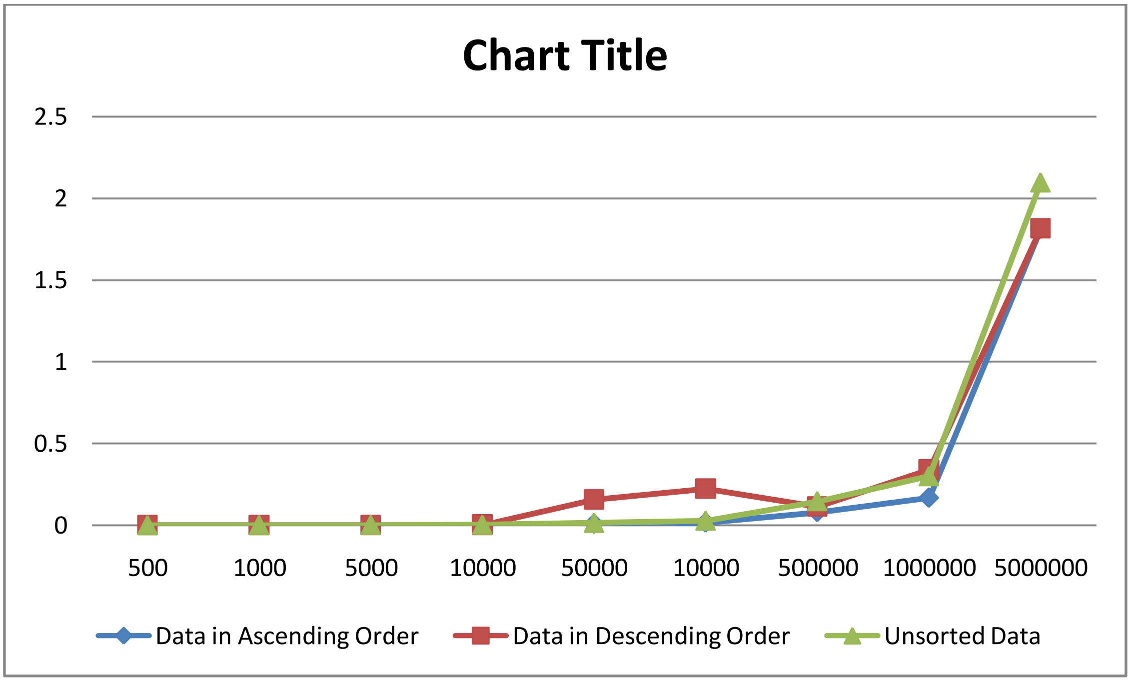
Execution time of Merge Sort

Merge Sort	Data in Ascending Order	Data in Descending Order	Unsorted Data
500	0.00005	0.00008	0.000137
1000	0.000105	0.000168	0.000292
5000	0.000752	0.001343	0.001599
10000	0.001252	0.001916	0.00249
50000	0.007678	0.15482	0.013902
10000	0.015163	0.22233	0.027717
500000	0.07965	0.112503	0.146751
1000000	0.166565	0.338429	0.299474
5000000	1.81068	1.81472	2.09675

Analysis of Merge sort Algorithm

1. It can be observed that Merge sort takes time proportional to n*log n, for higher values of n in the average case.
   Hence time complexity of merge sort is

$$f(n)=O(n*\log n)$$



2. **Here, it can be seen that the time taken by program for ascending, descending or unsorted (random) case is around the same.**
3. **Hence, it is not so good for already sorted, but very good for rest of the cases.**

Quick Sort

Pseudo Code

QUICK-SORT (A, *p*, *r*)

1. **if** ***p*** **<** ***r*** **then**
2. ***q*** **= PARTITION (A, *p*, *r*)**
3. **QUICK-SORT (A, *p*, *q*)**
4. **QUICK-SORT (A, *q*+1, *r*)**

Initial call: QUICK-SORT (A, 1, *n*)

PARTITION (A, *p*, *r*)

```
pivot = A[r]     // Pivot
i = p - 1
j = r + 1
while TRUE
          do
                    j = j - 1
     while A[j] > pivot
          do
                    i = i + 1
     while A[i] < pivot
          if j > i
                    exchange A[i] with A[j]
          else if j = i
                              return j - 1
                    else
                              return j
```

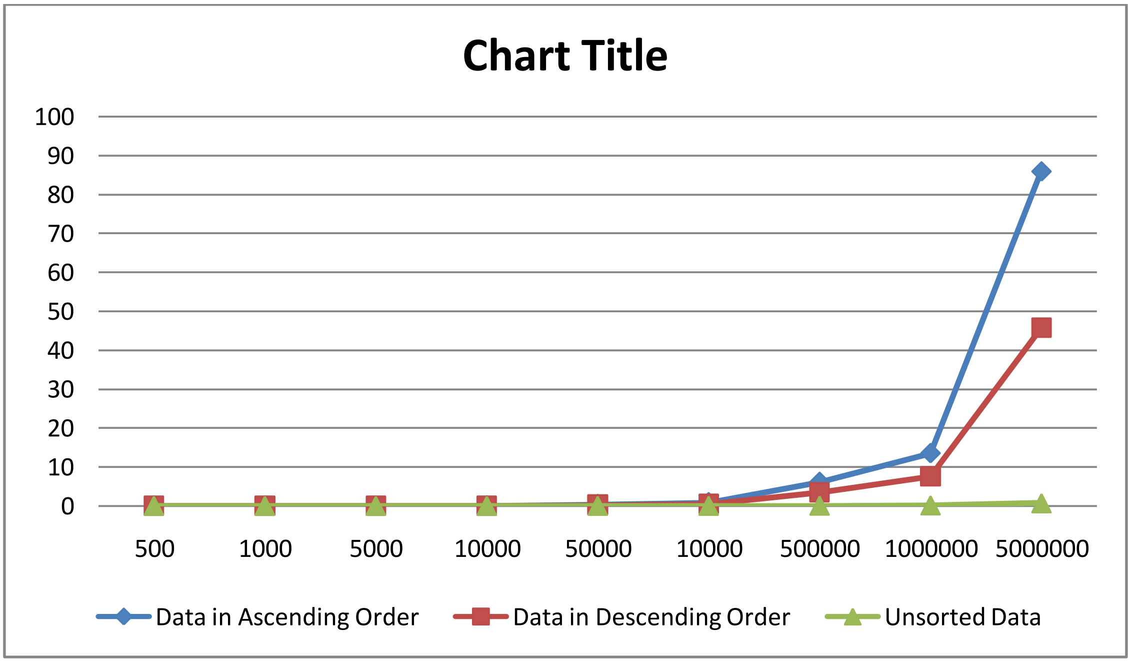
Execution time of Quick Sort

Quick Sort	Data in Ascending Order	Data in Descending Order	Unsorted Data
500	0.000332	0.000296	0.000061
1000	0.001219	0.001128	0.000115
5000	0.018721	0.012034	0.00061
10000	0.050695	0.031781	0.001222
50000	0.398149	0.232668	0.006252
10000	0.877284	0.535567	0.014383
500000	6.11559	3.50565	0.063415
1000000	13.4972	7.51897	0.128449
5000000	85.923	45.7129	0.675445

Analysis of Quick sort Algorithm

1. **It can be observed that Quick sort takes time proportional to n*log n, for higher values of n in the average case. Hence time complexity of quick sort is**

$$f(n)=O(n*\log n)$$



2. **In the case of unsorted case, this program is very efficient.**
3. **But in case of more sorted or reverse sorted case, it's efficiency (time taken) is very high, compared to the average case.**

Modified Bubble-Sort

Pseudo Code

INPUT : A[1..n], array of integers

OUTPUT: Rearrangement of A such that A[1]≤A[2] ≤… ≤A[n]

BUBBLE-SORT (A)

```
// Sort by bubbling the smallest element to current position.
j = n and sorted =false
while  j ≥ 2 and sorted is false
        Sorted=true
// Bubble up the smallest element to its correct position
        for  i = 1 to j - 1
            if  A[i] > A[i + 1]
                sorted=false
                temp = A[i]
                A[i] = A[i + 1]
                A[i + 1] = temp
        j = j - 1
```

Analysis of Modified Bubble sort Algorithm

1. This Modified Bubble Sort algorithm greatly decreases the time taken for already sorted data. Now it's time complexity is of O(n).
2. But marginally increases the time for both average and unsorted data.

Modified Rank Sort

Pseudo Code

INPUT : A[1..n], array of integers

OUTPUT: Rearrangement of A such that A[1]≤A[2] ≤… ≤A[n]

RANK-SORT (A)

```
for i < n
        R[i] = 0
for i<n
        j=i-1
         if arr[j] < = arr[i] and R[j]==j
                R[i]=i
        else
        while j>=0
                if arr[j] <= arr[i]
                        R[i]++
                else
                        rank[j]++
        j--
```

```
// Move to correct place in U[1 . . n]
14.   for  j = 1 to n
15.              U[R[j]] = A[j]
// Move the sorted entries into A
16.   for  j = 1 to n
17.              A[j] = U[j]
```

Analysis of Modified Rank-sort Algorithm

1. This Modified Rank-Sort algorithm greatly decreases the time taken for already sorted data.
2. But it also marginally increases the time for both average and unsorted data.

Comparison of different Algorithms

For Ascending data

1. **In case of already sorted data, selection sort, insertion sort, modified bubble-sort and modified rank-sort are very good. Their time complexity is of O(n).**
2. **The time complexity of merge sort is of O(n*log n) and so its time taken is a little more.**
3. **For others, the time complexity is of O(n^2) and the time taken is large.**

For Descending data

4. **In case of reversely sorted data, merge sort is very good. Its time complexity is of O(n*log n).**
5. **The time complexity of quick sort is of O(n^2), but as it has low constant values in its time complexity, its time taken is more.**
6. **For others, the time complexity is of O(n^2) and the time taken is very large.**

For Unsorted data

1. **In case of unsorted data, quick sort is very good as the coefficients for polynomial of time taken by algorithm are very small. Its time complexity is of O(n*log n).**
2. **In case of unsorted data, merge sort is also good as its time complexity is of O(n*log n).**
3. **For others, the time complexity is of O(n^2) and the time taken is very large.**