I. INTRODUCTION
Sorting is one of the most important operations in computer programming. Various sorting algorithms are available today, but not a single algorithm can be stated best for multiple problems. Different type of problem has to be dealt with different algorithm which is optimal for it. A sorting algorithm may not have minimum time complexity and minimum space complexity for a problem, so we have the concept of time-space tradeoff. Time-space tradeoff implies that we can minimize the time needed for the execution of an algorithm at the cost of memory space required for it.

A. Calculating the Complexity of an Algorithm: We need to find out the complexity of the algorithm to see how much efficient it is. We have defined the complexities of each of the sorting algorithm using Big Oh notation because it gives the longest running time for any input of size \( n \) and an upper bound on the running time for any input. Thus, it guarantees that the algorithm will never take longer than this.

B. Optimum Sorting Algorithm: From among the different sorting algorithms, some are best for one type of problems and others are best for other type of problems. It is difficult to say which sorting algorithm is optimal. However, we have compared the different algorithms to describe the type of problem they are best for. This will help in choosing the optimal sorting algorithm according to user requirements.

C. Comparison of different sorting algorithms: Different sorting algorithms are compared by analyzing the time complexity and space complexity using \( n \) input data. For better understanding and easy comparison we have used the same example for sorting, using different algorithms. Here we discuss the advanced sorting techniques in detail with algorithms and examples.

II. FUNDAMENTAL SORTING ALGORITHMS
A. Selection Sort: - Selection sort has the basic principle according to which it finds out the smallest element in an array with each pass and places it in its proper place until the array being sorted completely [6].

1. Algorithm: Selection sort
Description: - A is the array of \( n \) element to be sorted. Min denote the smallest element. LOC is a location variable
STEP: 1 Iterate through Array of \( n \) element.
   Repeat step 2 to 5 When \( K=1,2,3…………………N-1 \)
STEP: 2 Set \( Min=A[K], LOC=K \)
STEP: 3 Repeat for \( T=K+1, K+1…………N \)
STEP: 4 IF \( Min>A[T] \) Update Min and LOC
STEP: 5 \( Min=A[LOC], LOC=T \)
STEP: 7 EXIT.

2. Example:-
\[
A = [25, 20, 10, 15, 2, 0, 3, 7]
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( K=1, LOC ) ( C=6 )</td>
<td>25</td>
<td>20</td>
<td>10</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>( K=2, LOC ) ( C=5 )</td>
<td>0</td>
<td>20</td>
<td>10</td>
<td>15</td>
<td>2</td>
<td>25</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>( K=3, LOC ) ( C=7 )</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>( K=4, LOC ) ( C=8 )</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>( K=5, LOC ) ( C=7 )</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>20</td>
<td>25</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>( K=6, LOC ) ( C=8 )</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>25</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>Sorted</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>

So we see here that the selection sort has the basic principle according to which it finds out the smallest element in an array with each pass and places it in its proper place until the array being sorted completely. It is very slow, and it has its time complexity \( O(n^2) \) for...
both worst-case and best-case due to large number of comparison it performed. In some cases it performs better than bubble sort as fewer swaps are required.

B. Insertion Sort: Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and it perform sorting by taking element from list one by one and insert them in their proper position into new ordered list.

1. Algorithm: Insertion sort
Description: - A is the Array of n element to be sorted. Temp is the variable that keeps the record of element. PTR is the Pointer.

STEP: 1 Initialize the element
Set A [0] = - ∞

STEP: 2 Repeat Step 3 to 5 when k=2, 3 …N

STEP: 3 Set Temp= A[k] and PTR=K-1.

STEP: 4 Repeat While Temp < A [PTR].
   b. Set PTR=PTR-1


STEP: 6 EXIT.

2. Example:-
A: - 25 10 15 2 0 3 7

<table>
<thead>
<tr>
<th>Data</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
<th>Sorted list</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
<td>20</td>
<td>15</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>1st pass</td>
<td>20</td>
<td>25</td>
<td>10</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2nd pass</td>
<td>10</td>
<td>20</td>
<td>25</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>3rd pass</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4th pass</td>
<td>2</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>0</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5th pass</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>6th pass</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>7</td>
</tr>
<tr>
<td>7th pass</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>Sorted list</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>2</td>
</tr>
</tbody>
</table>

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists. It has time complexity O(n^2) like bubble sort and selection sort but I will perform sorting faster than Bubble sort and Selection sort due to fewer comparison it performed[1].

C. Bubble Sort: - Bubble work by comparing each item with next item to it and swap them if required.it is oldest and simplest sorting algorithm in use.

1. Algorithm:-Bubble sort
Description: - A is the Array of N element.PTR denotes the pointer.

STEP: 1 Repeat step 2and 3 for K=1 to N-1

STEP: 2 Set PTR=1 // initializing PTR

STEP: 3 Repeat When PTR<=N-K.
   b. SET PTR=PTR+1

STEP: 4 EXIT.

2. Example:-
A: - 25 10 15 2 0 3 7


```
20 10 15 2 0 3 7
```

2. Now 25 is fixed to proper place from here we need make 7-1=6 comparison for 2nd pass taking 20 as PTR then we get following result by comparing 20 with each element.

```
10 15 2 0 3 7 20
```

3. From here 25 and 20 are fixed now we need to make 6-1=5 comparison to for 3rd pass. As 10<15 No swapping so we take 15 as PTR initially and after 3rd pass we got result as follow.

```
10 2 0 3 7 15 20 25
```

4. In the 4th pass 15, 20, 25 are positioned so we make 5-1=4 comparison and we get following result.

```
20 0 3 7 10 15 25
```

5. In the 5th pass we need make 4-1= 3 comparison, As 10, 15, 20, 25 are positioned, After this pass we get following result.

```
0 2 3 7 10 15 20 25
```

Here we get sorted list. Bubble sort is approaches to the problem by making multiple passes, through the algorithm.it performed sorting by comparing two consecutive values and swap them if they are out of order.

Bubble sort has time complexity O(n^2) like selection and insertion sort but it is much slower than selection sort and insertion sort due to large number of comparison it performed [2].

D. Quick sort: - Quick sort is a divide and conquer algorithm with two phases, a partition phase and a sort phase.

1. Algorithm: Quick Sort
Description: - A is the Array of N element. P, R is the variable used for comparison

QUICK_SORT (A, P, R)

STEP: 1 P>R THEN

STEP: 2 Q= PARTITION (A, P, R)

STEP: 3 QUICKSORT (A, P, Q-1)

STEP: 4 QUICKSORT (A, Q+1, R)

STEP: 5 EXIT.

Here is the sub-algorithm for partition

PARTITION (A, P, R)

STEP: 1 X=A [R]

STEP: 2 I=P-1

STEP: 3 FOR J=P TO R-1

STEP: 4 DO IF A [I] <=X

STEP: 5 THEN I=I+1

STEP: 6 SWAP (A [I], A [J])

STEP: 7 SWAP (A [I+1], A [R])

STEP: 8 RETURN (I+1).

2. Example:-
A: - 25 20 10 15 2 0 3 7 Set pivot =7 partition the list into two
sub-list by placing all element which are greater than 7 on right list , and smaller element in left side list.

| 2 | 0 | 3 | 7 | 25 | 20 | 10 | 15 |

Now set pivot=2 for left list and pivot=15 for right list

| 0 | 2 | 3 | 7 | 10 | 15 | 20 | 25 |

Here we get sorted array.

| 0 | 2 | 3 | 7 | 10 | 15 | 20 | 25 |

Quick sort has complexity of O(n log n) [11]. Quick sort is faster than common sorting algorithm, it is better than heap sort, merge sort etc. Quick sort could be a best choice for sorting theoretically but practically it is little too complex to implement on large data set. Although it is fast and efficient but if the list is already sorted it gives horrible result [5].

E. Merge Sort: Merge Sort is a Divide and Conquer Algorithm with two phase, Partition phase and Merge phase. Merge sort splits the list to be sorted into two halves, and sort them recursively.

1. Algorithm: Merge sort

Description: A is the Array of N element. N is the Number of element.

STEP: 1 IF N=1, THEN RETURN
STEP: 2 Divide Array A [1…………N] into two halves
1. A[1…………N/2]
2. A[((N/2)+1)………N]
STEP: 3 Sort two arrays A [1…………N/2] and A[((N/2)+1)………N] recursively.
STEP: 4 Merge sorted arrays into single one
STEP: 5 EXIT.

2. Example:
A:- 25 20 10 15 2 0 3 7
STEP: 1 Divide array into two halves.

| 25 | 20 | 10 | 15 | 2 | 0 | 3 | 7 |

Merge sort is slightly faster than heap sort for larger data set, but it require twice the memory of heap sort because of second array it used. Due to its recursive nature it does not suited for application run on machine with limited memory. It has time complexity of O(n log n) [4].

III. ADVANCED SORTING ALGORITHMS

A. Library sort: Library sort, also called the gapped insertion sort is same as insertion sort but with gaps in array to accelerate the subsequent insertions. Library Sort is a combination of bucket sort, insertion sort and counting sort. First of all, a bucket sort is done by using some hash function and then a count is maintained for each bucket which counts the number of elements in the bucket. And finally the values are inserted using insertion sort.

1. Algorithm: library sort

Description: A is the array of n elements to be sorted, SIZE is the number of elements in array i.e. n, B[i] represents bucket formed using a hash function, C[B[i]] gives the count of elements in bucket B[i], POS is the variable to find position in new Array B where element is to be inserted.

STEP: 1 Bucket sort the elements of array A using a hash function such that each elements gets some value B[i]
STEP: 2 Maintain a count C[B[i]] of elements in each bucket B[i] for i =1,2,3,4…..
STEP: 3 Initialize i=0 and take a new array B [SIZE]
STEP: 4 WHILE i<=SIZE-1
THEN calculate POS for element A[I] from bucket B[i] as:
STEP: 5 IF POS is EMPTY
THEN insert A[I] at B[POS]
ELSE POS=POS+1 and GOTO
STEP: 6 INSERTION SORT (B)
STEP: 7 END.

2. Flowchart: library sort

3. Example:
A: 25 20 10 15 2 0 3 7

Comparison Based Identification Of Sorting Algorithm For A Problem

69
Comparison Based Identification Of Sorting Algorithm For A Problem

B1: 0 2 3 C[B1] = 3
B2: 7
C[B2] = 1


1. I=0, A[0]=25, 1<=7 (True)
   a. 25->B1
   d. I=0+1 = 1

2. I=1, A[1]=20, 1<=7 (True)
   a. 20->B1
   = 6
   d. I=1+1 = 2 and so on till we get the following

3. I=8, I<=7 (False)

B2: 10 15 20

I=0,
A[0]=25,
I<=7 (True)
I=0+1=1

I=1,
A[1]=20,
I<=7 (True)
I=1+1=2

I=2,
A[2]=10,
I<=7 (True)
B[7]=10
I=2+1=3

I=3,
A[3]=15,
I<=7 (True)
B[8]=15
I=3+1=4

I=4,
A[4]=20,
I<=7 (True)
B[9]=20
I=4+1=5

I=5,
A[5]=25,
I<=7 (True)
B[10]=25
I=5+1=6

I=6,
A[6]=10,
I<=7 (True)
I=6+1=7

I=7,
A[7]=3,
I<=7 (True)
B[12]=3
I=7+1=8

I=8,
A[8]=7,
I<=7 (True)
I=8+1=9

I=9,
A[9]=0,
I<=7 (True)
B[14]=0
I=9+1=10

I=10,
A[10]=2,
I<=7 (True)
B[15]=2
I=10+1=11

I=11,
I<=7 (True)
B[16]=15
I=11+1=12

I=12,
A[12]=20,
I<=7 (True)
B[17]=20
I=12+1=13

I=13,
A[13]=25,
I<=7 (True)
B[14]=25
I=13+1=14

So, here we can see that library sort reduces the time to sort using insertion sort by providing an almost sorted list. The gapped insertion sort or the library sort accelerates the subsequent insertions in the array. Thus, it is much faster than the basic sorting techniques. Library sort is much efficient for completely unordered list. And also for larger input set library sort is better than other basic sorting methods.

B. Gnome Sort: Gnome Sort is similar to insertion sort except that moving an element to its proper place is accomplished by a series of swaps as in bubble sort. This is similar to the way we sort flower pots placed in a line. Starting from the second we compare each pot with its previous pot, if it is larger than step forward to next pot otherwise swap the pots and move one step backward.

1. Algorithm: gnome sort

Description: A is the array of n elements to be sorted, SIZE is the number of elements in array i.e. n, and TEMP is a temporary variable used for swapping.

STEP: 1 Initialize I=1 and J=2
STEP: 2 WHILE (I!= SIZE-1)
   IF (A [I-1] <= A [I])
      THEN I=J and J++;
      [increment I and J]
      IF (I=0)
          THEN I=1
          [incrementing J]
STEP: 3 END.

2. Flowchart: gnome sort

3. Example:

   A: 25 20 10 15 2 0 3 7
   A [0, 1, . . . . .7 ]
   SIZE=8

   1. I=1 and J=2, I<=SIZE-1 (I<=7, True)
      b. A: 20 25 10 15 2 0 3 7
      c. I=1+1=2 and so on till we get the following

   2. I=1 and J=2, I<=SIZE-1 (I<=7, True)
      b. I=J=2
      J=J+1=2+1=3
      c. A: 20 25 10 15
      d. I=I-1=1
      e. IF(I=0)(True)
      THEN I=1
      2. I=1 and J=2, I<=SIZE-1 (I<=7, True)
      b. I=J=2
      J=J+1=2+1=3
      c. A: 20 25 10 15
      d. I=I-1=1
      e. IF(I=0)(False)
      And so on till we get the following

   3. I=2 and J=3, I<=SIZE-1 (I<=7, True)
      b. I=J=2
      J=J+1=2+1=3
      c. A: 20 25 10 15
      d. I=I-1=1
      e. IF(I=0)(False)
      And so on till we get the following

   4. I=3 and J=8, I<=SIZE-1 (I<=7, True)
      b. I=J=8
      J=J+1=8+1=9
      c. A: 0 2 3 7 10 15 20
      d. I=I-1=7
      e. IF(I=0)(False)
      And so on till we get the following

   5. I=8 and J=8, I<=SIZE-1 (I<=7, False).
Here, we see that gnome sort is also like insertion sort except that moving an element to its proper place is accomplished by a series of swaps as in bubble sort. The gnome algorithm is not much efficient but quite simple sort. Gnome algorithm is in no way better than insertion sort and hence is hardly preferred. However, this algorithm is used for small data input to avoid nested loops.

C. Stack Sort: In Stack Sort two stacks are used, one of which arranges the elements in ascending order and the second stack arranges the elements in descending order. At the end both the stacks are combined to form the sorted list [3].

1. Algorithm: stack sort
Description: A is the array of n elements to be sorted, SIZE is the number of elements in array i.e. n, S1 and S2 are two stacks., MIN is some integer less than any element of array A, MAX is some integer greater than any element of array A, PUSH is insertion operation into TOP of stack, POP is deletion operation from TOP of stack

STEP: 1 Initialize S1.TOP=MIN and
      S2.TOP=MAX

STEP: 2 WHILE I<= SIZE-1
      IF (A[I]> S1.TOP)
          MOVE elements from S2 to S1 until A[I]< S2.TOP
          INSERT A[I] at S2.TOP+1 and
          I++
      ELSE
          MOVE elements from S1 to S2 until A[I]> S1.TOP
          And PUSH A[I] at S1.TOP+1 and I++

STEP: 3 MOVE all elements from S2 to S1.

STEP: 4 END.

2. Flowchart: stack sort

3. Example:
   A: 25 20 10 15 2 0 3 7
   S1.TOP= -100  S2.TOP= 100  SIZE=8
   I=0, I<SIZE-1(0<7) True

   a. IF(A[0]> S1.TOP) i.e. IF(25> -10) True
   b. IF (A[0]> S2.TOP) i.e. IF(25> 100) False
   c. PUSH A[0] i.e. 25 at S2.TOP+1
   d. Therefore, S1: -100
      S2: 100 25
   e. I=I++ = 0+1 = 1

   a. IF(A[1]> S1.TOP) i.e. IF(20> -10) True
   b. IF (A[1]> S2.TOP) i.e. IF(20>25) False
   c. PUSH A[1] i.e. 20 at S2.TOP+1
   d. Therefore, S1: -100
      S2: 100 25 20
   e. I=I++ = 1+1 = 2
And so on till we get the following

2. I=1, I<SIZE-1(1<7) True
   a. IF(A[1]> S1.TOP) i.e. IF(20> -10) True
   b. IF (A[1]> S2.TOP) i.e. IF(20>25) False
   c. PUSH A[1] i.e. 20 at S2.TOP+1
   d. Therefore, S1: -100
      S2: 100 25 20
   e. I=I++ = 1+1 = 2

   a. IF(A[7]> S1.TOP) i.e. IF(7> 2) True
   b. IF(A[7]> S2.TOP) i.e. IF(7> 3) True
   c. MOVE 3 from S2 to S1
      i. S1: -100
      ii. S2: 100 25 20
   d. Therefore, S1: -100
      S2: 100 25 20
   e. I=I++ = 7+1 = 8

4. I=8, I<SIZE-1(8<7) False
   a. MOVE all elements from S2 to S1
      i. S1: -100
      ii. S2: 100

Stack Sort technique requires the use of two stacks, thus its memory requirement is more. Stack Sort consumes too much time for reverse ordered input list.

D. Deq Sort: Deq stands for double ended queues. In this, the element is first inserted at the tail of the deq. For the second element, two comparisons are made. First comparison is with the tail element, if the new element is greater than the tail element than the new element is appended to the queue, otherwise a second comparison is made with the head element of the queue and if found smaller new element is prepended
Comparison Based Identification Of Sorting Algorithm For A Problem

to the deq. And the process repeats. Whenever the element is found to be greater than the head element a new sublist is created and comparisons are made on the new list.

1. Algorithm: deq_sort
   Description: A is the array of n elements to be sorted, SIZE is the number of elements in array i.e. n, B is the sorted array
   STEP: 1 Initialize I=1, J=1 and S.TAIL=A[0]
   STEP: 2 WHILE I < SIZE
       IF (A[I]>S[J].TAIL)
       THEN S[J].TAIL=A[I], I++
       ELSE
           IF (S[J].HEAD=EMPTY)
           THEN S[J].HEAD=A[I], I++
           ELSE
               IF (A[I] < S[J].HEAD)
               THEN PREPEND A[I] to S[J].HEAD, I++
               ELSE
                   J++ and Create a new Sub list S[J] and S[J].TAIL=A[I], I++
   STEP: 3 IF I > SIZE
       THEN K= 0
       WHILE (K < SIZE)
       THEN Select the minimum among the HEAD of all S[J] ADD it to B[K]
   STEP: 4 END.

2. Flowchart: deq_sort

3. Example:

   A: 25 20 10 15 2 0 3 7
   SIZE=8

   1. I=1, J=1, S[TAIL]=25
   a. IF I < SIZE (1<8) True
   c. IF S[J].HEAD = EMPTY False
   d. S[J].HEA D=20, I++ (i.e. I=2)
   e. Therefore now S[J]= 20 25

   2. I=2, J=1, S[TAIL]=25
   a. IF I < SIZE (2<8) True
   c. IF S[J].HEAD = EMPTY False
   f. Therefore now S[J]=10 20 25
   And so on till we get the following

   3. I=7, J=3, S[TAIL]=3
   a. IF I < SIZE (7<8) (True)
   b. IF A[I] > S[J].TAIL (7>3) (True)
   c. Append A[I] i.e. 7 to S[J], I++(I=8)
   d. Therefore now S[J]=10 20 25 7

   4. I=8, J=3, S[TAIL]=7
   a. IF I < SIZE (8<8) (False)
   b. S[J]= 7 15 20
   c. S[J]= 7 2 15
   d. B[0]=0
   e. S[J]= 7 2 15
   f. S[J]= 7 2
   g. B[1]=2
   h. S[J]= 7 2
   And so on till we get the following

   5. I=9, J=3, S[TAIL]=7
   a. IF I < SIZE (9<8) (False)
   b. S[J]= 7 2 15
   c. S[J]= 7 2 15
   d. B[0]=0
   e. S[J]= 7 2 15
   f. S[J]= 7 2
   g. S[J]= 7 2
   h. B[1]=2
   i. S[J]= 7 2
   Therefore
   B = 0 2 3 7 10 15 20 25

DEQ or the double ended queue sorting technique is very efficient technique. Ordered insertion is easier than stack sort. There is no need to pop the elements instead a new sub list can be created. DEQ Sort is better for reverse ordered input list.

E. Heap Sort: - Heap sort is a improved version of straight selection sort .it work by building the heap out of data items and then removing the largest node from heap and placed it at the end of Array then again it reconstruct the heap, remove the largest node, placed it at end of Array, the process is continued until we get sorted list [9],[10].
Comparison Based Identification Of Sorting Algorithm For A Problem

1. Algorithm: Heap Sort

Description: - A is the Array of n element to be sorted. Size is the number of element in array. L is used for Left element in heap. R is used for right element.

STEP: 1 Build Heap (A)
STEP: 2 for I= length (A) down to 2
STEP: 3 exchange A [1] and A[i]
STEP: 4 heapsize=heapsize-1
STEP: 5 Heapify(A,1)

Build heap.

STEP: 1 heapsize=length (A)
STEP: 2 for I=floor (length/2) down to 1
STEP: 3 Largest =Left, Else
STEP: 4 Largest=R
STEP: 5 If(R<=heapsize) and A[R]>A[Largest]
STEP: 6 Largest=R
STEP: 7 If (Largest !=I)
STEP: 9 Heapify(A, Largest)

2. Example:-
A: - 25 20 10 15 2 0 3 7
1. Building the heap from given array

25
20
15
10

2. Delete root and replace with last element.

7 20 10 15 2 0 3 25

3. Build heap

4. Delete top element again

3 15 10 7 2 0 20 25

5. Build Heap Again

Sorted array:

15 7 10 3 2 0 20 25

Hence we left with this sorted array.

Heap sort is the slowest of the O(n log n) algorithms but it does not require recursion or multiple array to work, which make it perfect choice for sorting of extremely large data set over merge sort and quick sort algorithms. But heap sort is slower than merge sort and quick sort [11],[8].
F. Shell Sort: Shell sort is named after its inventor D. L. shell. It performed sorting by comparing element which are at certain distance and insert them at their proper place using insertion sort. The value of distance is half the number of input and is halved after each pass [7].

1. Algorithm: Shell Sort
   Description: - A is the Array of N element. Temp is the variable to store the value temporarily. Size Denote the size of Array. D denote gap between to element.
   STEP: 1 D=Size
   STEP: 2 while (D>1)
   STEP: 3 D = (D+1)/2
   STEP: 4 for (i = 0;i<Size-D; i++)
   STEP: 5 Temp=A [i+D]
   STEP: 7 A [i]=Temp
   STEP: 8 EXIT.

2. Example:--
   A:- 25 20 10 15 2 0 3 7

Now we will sort this array using shell sort

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D=3</td>
<td>25</td>
<td>20</td>
<td>10</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>D=2</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>D=1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>sorted</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
</tr>
</tbody>
</table>

Shell sort is the most efficient of the O(n^2)Class of sorting algorithm. It is suitable for sorting list having less than 5,000 items. Although it is the efficient algorithm but it is relatively slower than merge sort, heap sort, quick sort. It is very complex relative other algorithm of O(n^2) class.

G. Proxmap Sort:- Proxmap is a sorting algorithm that work by partitioning the array of data items into sub-array.

1. Algorithm: Proxmap Sort
   Description: - A is the Array of N element.
   Step: 1 Define mapping, map a key to sub array of destination array A2
   Step: 2 Bucket sorts all elements which have same key.
   Step: 3 then, interchange the key
   Step: 4 fine Grain sorting will be Done

2. Example:--
   A:- 2.5 2.1 1.1 1.5 0.2 0.3 7.1

   Key 0 1 2 3 4 5 6

   Array: 2.5 2.1 1.1 1.5 0.2 0.3 6.1

Now mapping will be done as shown in table.

<table>
<thead>
<tr>
<th>S.No</th>
<th>Sort Name</th>
<th>Average Case Complexity</th>
<th>Worst Case Complexity</th>
<th>Advantage(s)</th>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Proxmap</td>
<td>O(n)</td>
<td>O(n^2)</td>
<td>Faster than comparison based algorithm</td>
<td>Use extra memory for storing key</td>
</tr>
<tr>
<td>2.</td>
<td>Merge</td>
<td>O(nlogn)</td>
<td>O(nlogn)</td>
<td>Well suited for large data set</td>
<td>At least twice the memory requirement than other sorts.</td>
</tr>
<tr>
<td>4.</td>
<td>Library</td>
<td>O(nlogn)</td>
<td>O(n^2)</td>
<td>Stable comparison and can be run as online algorithm.</td>
<td>Requires extra space for the gaps.</td>
</tr>
<tr>
<td>5.</td>
<td>Heap</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>Well suited for large data set and does not use massive recursion.</td>
<td>Less efficient.</td>
</tr>
<tr>
<td>6.</td>
<td>Gnome</td>
<td>O(n')</td>
<td>O(n')</td>
<td>No nested loops, tiny code size and stable.</td>
<td>Not much efficient.</td>
</tr>
<tr>
<td>7.</td>
<td>Stack</td>
<td>O(n log n)</td>
<td>O(n')</td>
<td>Best performance among the sorting by insertion algorithms in all three cases.</td>
<td>Requirement of two stacks hence more memory.</td>
</tr>
<tr>
<td>8.</td>
<td>Deqsort</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>Better for reverse ordered lists.</td>
<td>Managing multiple deqs is not easy.</td>
</tr>
</tbody>
</table>

This is the sorted array.
Proxmap uses technique similar to hashing to assign an item to its correctly sorted positioned in a container. It sort with help of an address computation, A key is shifted in the first pass to the proximity of the final destination. It has worst case complexity O(n^2) and best –case complexity O(n). It is faster than the comparison based algorithms. We can avoid worst-case by using good map key.

III. COMPARATIVE STUDY

Advanced sorting algorithms:

Comparison Based Identification Of Sorting Algorithm For A Problem
CONCLUSION

In this paper, different advanced sorting algorithms have been studied. Each algorithm has been first represented in the form of pseudo code which describes the step by step execution and then it is also graphically represented through flowchart and explained with the help of examples.

Further, the advantages and disadvantages of sorting algorithms are discussed and time complexity of the different algorithms are compared and tried to find out which sorting algorithm is good in which situation, since never a single algorithm can be stated best for every kind of problem statement. The best sorting algorithm is the one which suits the user problem statement and provides the best solution with minimum time and minimum space complexity (memory can be traded off against time).

REFERENCES


***