Book

A Simplified Approach to Data Structures

Prof.(Dr.)Vishal Goyal, Professor, Punjabi University Patiala

Dr. Lalit Goyal, Associate Professor, DAV College, Jalandhar

Mr. Pawan Kumar, Assistant Professor, DAV College, Bhatinda

Shroff Publications and Distributors

Edition 2014

Sorting Techniques

Introduction

Sorting algorithms arrange items in a set according to a predefined ordering criterion. The sorting may be ascending or descending

Example:--

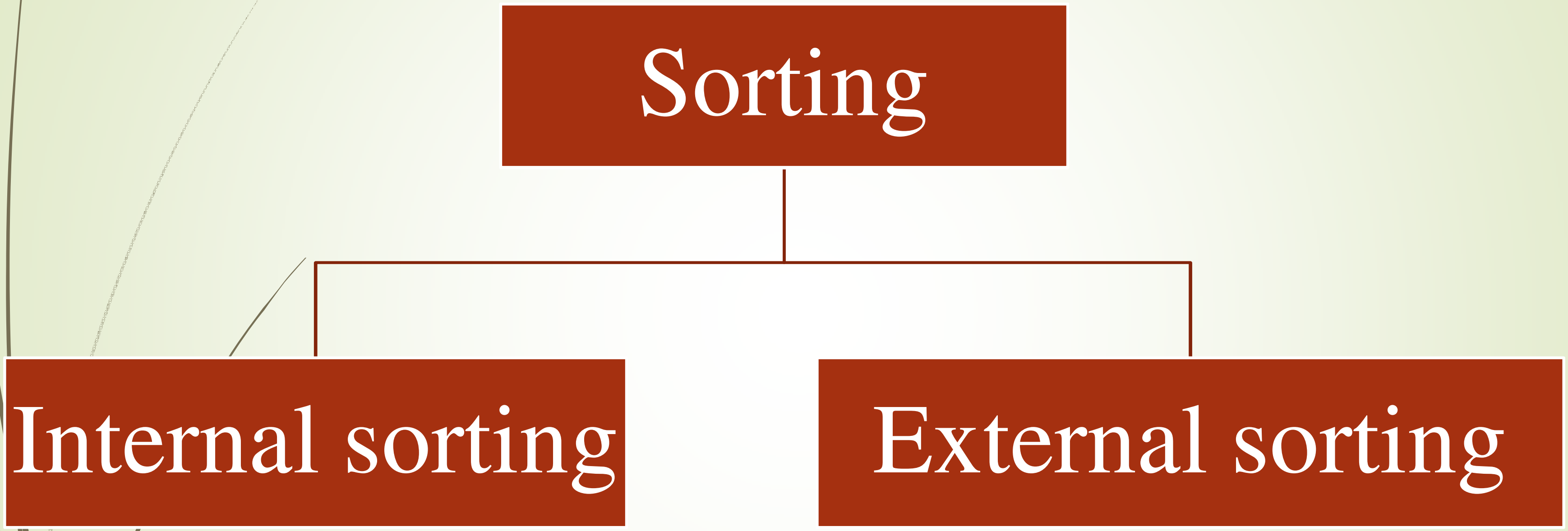
Unsorted list {20,30,10,50,40}

Ascending order {10,20,30,40,50}

Descending order {50,40,30,20,10}

Characteristics of algorithms :

- The type and amount of data to be stored.
- Amount of memory space required by sorting algorithm.
- Whether data is in random, partially, or in reverse order.
- Complexity of the sorting algorithm and the ease with which it can be implemented.


Sorting
Internal sorting
External sorting

Selection Sort

- Selection sort is the sorting algorithm which is in-place comparison sort.
- This algorithm does not use any extra space for sorting the elements of the array.
- The idea behind the selection sort is to find the smallest element in the array and place it with the element at the first position in the array.
- Then, find the next smallest element of the array by starting the search from the 2nd position to the last position of the array and replace it with the element at the 2nd position in the array.

Example

Consider the unsorted array A of size 8 shown below

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
22	35	17	8	13	44	5	28

<u>1st Pass</u>: The smallest element is 5 from A[1] to A[8], which is at 7th position. Exchange the values at A[1] and A[7], which results in the array as :

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
22	35	17	8	13	44	**5**	28

2nd Pass: The smallest element is 8 from A[2] to A[8], which is at 4th position. Exchange the values at A[2] and A[4], which results in the array as :

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
5	35	17	8	13	44	22	28

3rd Pass: The smallest element is 13 from A[3] to A[8], which is at 5th position. Exchange the values at A[3] and A[5], which results in the array as :

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
5	8	17	35	13	44	22	28

4th Pass: The smallest element is 17 from A[4] to A[8], which is at 5th position. Exchange the values at A[4] and A[5], which results in the array as :

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
5	8	13	**35**	**17**	44	22	28

5th Pass: The smallest element is 22 from A[5] to A[8], which is at 7th position. Exchange the values at A[5] and A[7], which results in the array as :

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
5	8	13	17	**35**	44	**22**	28

6th Pass: The smallest element is 28 from A[6] to A[8], which is at 8th position. Exchange the values at A[6] and A[8], which results in the array as :

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
5	8	13	17	22	**44**	35	**28**

7th Pass: The smallest element is 28 from A[7] to A[8], which is at 7th position. No exchange will take place as the element is already at its proper position.

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
5	8	13	17	22	28	35	44

no exchange due to **28 <35 and 28 <44**

Selection Sort Algorithm

Algorithm to sort an array ‘A’ with ‘n’ elements :

```
Step 1:Repeat Steps 2 to 5 For i=1 To n-1
Step 2:        Set Min = A[ i ] And Flag = False
Step 3:        Repeat Step4 For j = i+1 To n
Step 4:                  If A[ j ] < Min Then
                              Set Min = A[ j ]
                              Set Pos = j And Flag = True
                         [End If]
               [End Loop2]
```

```
Step 5:    If Flag = True Then
              Set Temp = A[ i ]
              Set A[ i ] = A[Pos]
              Set A[Pos] = Temp
           [End If]
        [End Loop1]
Step 6: Exit
```

Analysis of Selection Sort

Comparison to place the smallest element at first position in first pass is n-1. Comparison to place the second smallest element at second position in second pass is n-2. Comparison to place the third smallest element at third position in third pass is n-3 and so on. In the last pass there will be only one comparison for last two elements. Total comparisons will be,

$$f(n) = (n-1) + (n-2) + (n-3) + \ldots\ldots + 3 + 2 + 1$$

$$= n \times (n-1)/2$$

$$\approx O(n^2)$$

Complexity = $O(n^2)$

Analysis of Selection Sort

- Selection sort has same complexity in best case, average case, as well as in worst case.
- This is because the number of comparisons will remain same (n * (n-1)/2) to find the smallest element in unsorted part of the array.
- But, in case of best case, the time consumption in exchanging the element of array will be saved, as no exchange will be required, because the array is already sorted in best case scenario.

Insertion sort

- Insertion is also an in-place comparison sorting technique.
- This algorithm is also known as online sorting method because it sorts elements as it receives the elements.
- Insertion sort is stable as it does not change the relative position of the elements with equal value.
- The idea behind this sorting method is to sort all elements by inserting one element each time from the unsorted part of the list into the sorted part of the list.

Example :--

Consider an unsorted array A of size 8 shown below

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
22	35	17	8	13	44	5	28

Initially, we consider that the whole list of elements is divided into two parts i.e. sorted part consisting of the first element of array and the unsorted part consisting of n-1 elements.

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
22	35	17	8	13	44	5	28

Here, the first element A[2](=35) of the unsorted part is compared with the element A[1](=22) in the sorted part of the list.

First pass:

d

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
22	35	17	8	13	44	5	28

Now, comparing element A[3](=17) the first element of the unsorted part of the list, with the elements of the sorted part of the list i.e. A[2](=35) and A[1](=22), the element A[3] will be inserted at the first position.

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
22	17	35	8	13	44	5	28

Second pass:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
17	22	35	8	13	44	5	28

Now, it's the turn for the element A[4](=8),which will be compared with the elements in the sorted part of the list until an element smaller than A[4] is found.

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
17	22	8	35	13	44	5	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
17	8	22	35	13	44	5	28

As the elements A[3],A[2] and A[1] are greater than A[4](=8), so, these, elements will be shifted one position towards right and the element A[4] will be inserted at position A[1].

Third pass

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	17	12	35	13	44	5	28

Now, the sorted part of the list has four sorted elements and remaining four elements are present in the unsorted part. In the next step , the elements A[5](=13) will be inserted at the position A[2] after shifting elements A[2],A[3] and A[4]one position towards right.

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	17	22	13	35	44	5	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	17	13	22	35	44	5	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	13	17	22	35	44	5	28

In next step, element A[6] will remain at the same Position.

Fourth Pass:

Next element A[7] will be inserted at the position A[1]

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	13	17	22	35	44	5	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	13	17	22	35	5	44	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	13	17	22	5	35	44	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	13	17	5	22	35	44	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	13	5	17	22	35	44	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	5	13	17	22	35	44	28

Fifth Pass:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
5	8	13	17	22	35	44	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
5	8	13	17	22	35	28	44

In final step, the A[8](=28) will be inserted at position A[6] and the resulting sorted list will be:

After all the steps we got final sorted list as given below:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
5	8	13	17	22	28	35	44

Insertion Sort Algorithm

Sorts an array ‘A’ with ‘n’ elements using the sort technique.

```
Step1: Repeat steps 2 to 4 for i=2 to n
Step2:      Set Temp=A[i] and k=i-1
Step3:      Repeat While Temp<A[k] AND k>0
                Set A[k+1]=A[k]
                Set k=k-1
            [End loop]
Step 4:     Set A[k+1]=Temp
        [End loop]
Step 5:Exit
```

Analysis of Insertion Sort

The worst case for this sorting technique is, when elements of the list are in the reverse order of the desired one. In the worst case, each element of the unsorted part will be compared with all the elements of the sorted part of the list. Thus, in this case,

To position the 2nd element of the list at proper position, No of comparisons =1

To position the 3rd element of the list at proper position, No of comparisons =2

To position the 4th element of the list at proper position, No of comparisons =3

:

:

To position the n^th element of the list at proper position, No of comparisons =n-1

So, the total no. of comparisons in the worst case.

$f(n)=1+2+3+4+5+6+\ldots..+(n-2)+(n-1)$

$=n*(n-1)/2$

$=O(n^2)$

Thus, for the worst case the complexity of the insertion sort is O (n^2). The best case for this algorithm is when elements are already sorted. Then the complexity will be O(n). however, for the average case complexity will be O(n^2).

Merge sort

- Merge sort is one of the best sorting techniques which is based on the **divide and conquer** strategy.
- The Merge Sort reiterates on the list of elements by dividing it into smaller portions and sorting each sub-portion.
- The Merge Sort divides partitions in half by using the formula:

  Mid=(Lower+Upper)/2

  Here, Lower is the lower index and Upper is the upper index of array.

- Merge Sort reiterates on both the newly formed partitions by dividing them in half and continue the process.
- This process of dividing stop when the partition size reaches to one item.
- At this point it has created many one-item lists. Any one-item list is naturally in sorted order.
- The next step is to merge these one-item lists together for creating the sorted list.
- To combine two sorted lists, the merge sort compare successive pairs of elements, one from each list.

- If the list **A** has any element **<** than all the elements of list **B**, then it will be chosen to be append to the aggregated list or vice versa.
- And when all the elements of one list are added to the aggregated list then all the remaining elements of other list will be append directly to the aggregated list.
- This merging process takes at least **(n/2)** comparisons but not more than **(n-1)** comparisons.
- Merge Sort repeats the process of combining sorted sub-lists into larger sorted sub-lists until all have been successfully integrated into a single sorted list.

Example

To understand the concept and working of merge sort algorithm, lets us consider an unsorted array **A** of size 8 : **41,35,17,8,13,44,5,28**

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
41	35	17	8	13	44	5	28

Algorithm start splitting array **A** into sub-arrays A[1:4] and A[5:8] of size four each as shown below:

A[1:4] **A[5:8]**

A[1]	A[2]	A[3]	A[4]
41	35	17	8

A[5]	A[6]	A[7]	A[8]
13	44	5	28

Now, the sub-arrays A[1:4] and A[5:8] are split into arrays (A[1:2] & A[3:4]) and (A[5:6] & A[7:8]) respectively of size two each as shown below:

A[1:2] **A[3:4]** **A[5:6]** **A[7:8]**

A[1:2]

A[1]	A[2]
41	35

A[3:4]

A[3]	A[4]
17	8

A[5:6]

A[5]	A[6]
13	44

A[7:8]

A[7]	A[8]
5	28

Now, at this point, arrays A[1:2] , A[3:4] , A[5:6] , A[7:8] are split into final one element sub –arrays (A[1:1] & A[2:2]),(A[3:3] & A[4:4]) , (A[5:5 & A[6:6]) and (A[7:7] & A[8:8]) respectively as shown

A[1:1] A[2:2] A[3:3] A[4:4] A[5:5] A[6:6] A[7:7] A[8:8]

Now the sub-arrays A[3:3] and A[4:4] are merged to produce sorted array A[3:4] of size two. At this stage array can be pictorially shown as:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
35	41	17	8	13	44	5	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
35	41	8	17	13	44	5	28

Next, the sub-array a[1:2] and A[3:4] are merged to produce a sorted sub array A[1:4] of size 4 which is shown below:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	17	35	41	13	44	5	28

Now, the pending task of sorting the unsorted sub-array A[5:8] will be assumed as task of diving and sorting the sub-array A[1:4] is over.

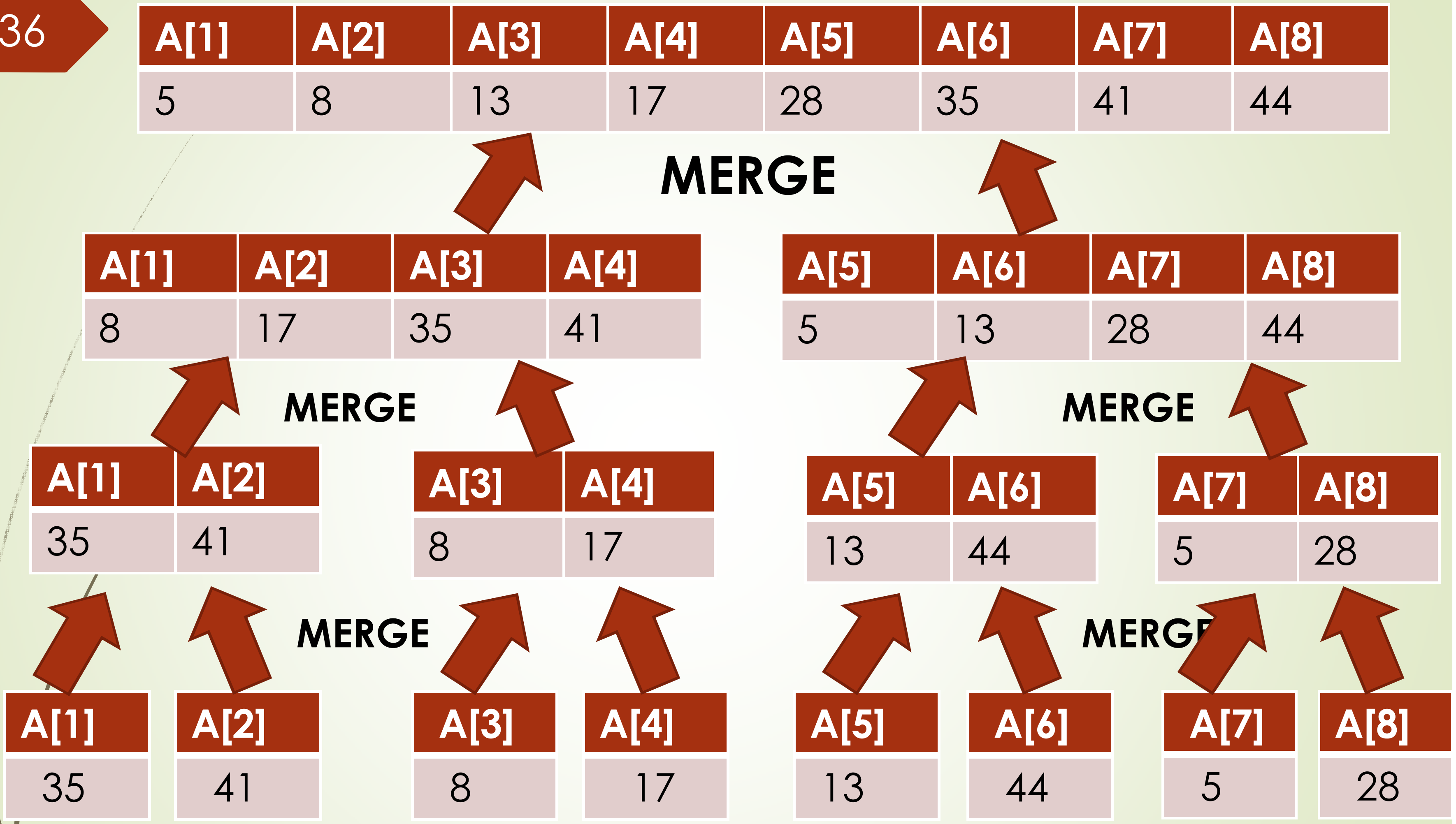
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	17	35	41	13	44	5	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	17	35	41	13	44	5	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	17	35	41	13	44	5	28

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	17	35	41	5	13	28	44

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
5	8	13	17	28	35	41	44


A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8]
5 8 13 17 28 35 41 44
MERGE
A[1] A[2] A[3] A[4]
8 17 35 41
A[5] A[6] A[7] A[8]
5 13 28 44
MERGE
MERGE
A[1] A[2]
35 41
A[3] A[4]
8 17
A[5] A[6]
13 44
A[7] A[8]
5 28
MERGE
MERGE
A[1] 35
A[2] 41
A[3] 8
A[4] 17
A[5] 13
A[6] 44
A[7] 5
A[8] 28

Algorithm of Merge Sort

Following Merge Sort algorithm sorts an unsorted list of elements using recursion and function ***MergeLists().***

Algorithm: Sorts an array A[L:U] using an auxiliary array 'B'

```
MergeSorting(Lower , Upper)
Step1:If Upper>Lower Then
Step2:      Set Mid = (Lower + Upper)/2
Step3:      Call MergeSorting(Lower , Mid)
Step4:      Call MergeSorting(Mid+1 , Upper)
Step5:      Call MergeLists(Lower , Mid , Upper)
        [End If]
Step6: Return
```

Algorithm: Merges two sorted sub-lists into a single sorted list.

```
MergeLists(Lower, Mid, Upper)
Step1:  Set Lb1 = Lower , Lb2 = Mid+1
        Set Ub1 = Mid , Ub2 = Upper
        Set Ub1 = Mid , Ub2 = Upper
        Set k = 1
Step2:Repeat Step3 While Lb1<Ub1 AND Lb2<Ub2
 Step3:   If A[Lb1]<A[Lb2] Then
                Set B[k] = A[Lb1]
                Set Lb1 = Lb1+1
                Set k = k+1
          Else
                Set B[k] = A[Lb2]
                Set Lb2 = Lb2+1
                Set k = k+1
            [End If]
        [End Loop]
Step4:   If Lb1>Ub1 Then
            While Lb2<=Ub2
                Set B[k] = A[Lb2]
                Set A[Lb2] = A[Lb2]+1
                Set k = k+1
            [End Loop]
Else
            While Lb1<=Ub1
                Set B[k] = A[Lb2]
                Set A[Lb2] = A[Lb2]+1
                Set  k = k+1
            [End Loop]
        [End If]
```

```
Else
            While **Lb1<=Ub1**
                Set **B[k] = A[Lb2]**
                Set **A[Lb2] = A[Lb2]+1**
                Set  **k = k+1**
            [End Loop]
        [End If]

Step5:   Repeat For **k = Lower to Upper**
            Set **A[k] = B[k]**
         [End Loop]
Step6:   Return
```

Analysis of Merge Sort

- In the first pass, sub-lists of size one are merged.
- In the second pass, the size of sub-lists being merged is 2.
- In the **k**th pass sub-lists being merged will be of size $\mathbf{2^{(k-1)}}$**.**
- A total of ***logn*** passes will be performed over the data.
- Since, two files can be merged in linear time, each pass of merge sort takes ***O(n)*** time.
- As there are ***logn*** passes, the total time complexity is ***O(n log n).***
- Merge sort is not ***in-place*** algorithm as it requires extra space to store the sorted elements.
- Merge sort algorithm is ***stable*** as the replicated elements do not exchange their positions.

Shell sort

- Shell sort is a generalization of insertion sort.
- Shell sort is named after its inventor Donald Shell who developed the technique in 1959.
- Shell sort is in-place comparison sort.
- Shell sort is also known as diminishing increment sort.
- This is because, the gap between the elements to be compared decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared with gap equal to one.

Basic Idea

- Set a gap **h** that is smaller than the number of elements in the list and larger than 1.
- Group all the elements of the list in **h** groups by putting elements that are **h** positions away from each other into the same group.
- Sort each group by exchanging locations of elements in the same group. Each group will have sorted elements and such a group is called **h-sorted**.
- Following the idea with decreasing value of **h**(value of **h** may be taken any but smaller than previous **h**) ending in 1, will produce the list of sorted elements.

Example

Consider a list of 12 elements :

1	2	3	4	5	6	7	8	9	10	11	12
64	85	20	55	09	17	97	88	49	71	27	30

considering the gap **h** as 5,3 and 1 respectively,

Pass 1:The gap is h = 5 and group is calling 5-sorting.

5-sorting performs the insertion sort on the five sub arrays $A_1(a_1,a_6,a_{11})$, $A_2(a_2,a_7,a_{12})$, $A_3(a_3,a_8)$, $A_4(a_4,a_9)$, $A_5(a_5,a_{10})$

1	2	3	4	5	6	7	8	9	10	11	12
64	85	20	55	90	17	97	88	49	71	27	30

5-sorting performs the insertion sort on the five sub arrays

$A_1(a_1,a_6,a_{11})$, $A_2(a_2,a_7,a_{12})$, $A_3(a_3,a_8)$, $A_4(a_4,a_9)$, $A_5(a_5,a_{10})$

- Rewriting the elements of five sub-arrays (A_1,A_2,A_3,A_4,A_5) column wise, we get five sub-lists as,

A_1	A_2	A_3	A_4	A_5
64	85	20	55	09
17	97	88	49	71
27	30			

- Now, apply the insertion sort on these five lists separately, we get

A_1	A_2	A_3	A_4	A_5
17	30	20	49	09
27	85	88	55	71
64	97			

The list of elements after 1st pass becomes,

1	2	3	4	5	6	7	8	9	10	11	12
17	30	20	49	09	27	85	88	55	71	64	97

Pass 2:The gap is h = 3 and group is calling 3-sorting.

3-sorting performs the insertion sort on the five sub arrays $A_1(a_1,a_4,a_7,a_{10})$, $A_2(a_2,a_5,a_8,a_{11})$, $A_3(a_3,a_6,a_9,a_{12})$

- Rewriting the elements of three sub-arrays (A_1,A_2,A_3) column wise, we get three sub-lists as,

A_1	A_2	A_3
17	30	20
49	09	27
85	88	55
71	64	97

- Now, apply the insertion sort on these three lists separately, we get

A_1	A_2	A_3
17	09	20
49	30	27
71	64	55
85	88	97

The list of elements after 2nd pass become,

1	2	3	4	5	6	7	8	9	10	11	12
17	09	20	49	30	27	71	64	55	85	88	97

Pass 3:The gap is h = 1 and group is calling 1-sorting.

3-sorting performs the insertion sort on the five sub arrays $A_1(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12})$ as shown below:

1	2	3	4	5	6	7	8	9	10	11	12
17	09	20	49	30	27	71	64	55	85	88	97

Now, apply the insertion sort on the single list, we get,

1	2	3	4	5	6	7	8	9	10	11	12
09	17	20	27	30	49	55	64	71	85	88	97

- The choice of gap size sequence h is not fixed and can be chosen arbitrarily. We have considered the gap size sequence as 5,3 and 1. Various researchers have suggested different gap size sequences.
- **Donald Shell** (the inventor of shell sort) proposed the gap size sequence as **FLOOR** ($n/2^k$), for k being the integer starting with 2. The gap sequence comes out to be n/2, n/4, n/8,……1.
- **Knuth** proposed the gap size sequence as $(3^k – 1)/2$, for k being the integer. The gap sequence comes out to be 1, 4,14, 40,121.
- **Pratt** proposed the gap size sequence as successive number of the form(2^p3^q). The gap sequence comes out to be 1, 2, 3, 4, 6, 8, 9, 12…

Algorithm of shell sort

Algorithm: Sort an array 'A' of size 'n' using Shell sort. The gap size is taken as original shell sequence i.e. n/2, n/4, n/8,…1

```
Step1 :  gap = n/2
            // gaps being used are of shell sequence
Step 2:  Repeat Steps 3 to 10 while gap>= 1
Step 3:      i = gap + 1
Step 4:      Repeat Steps 5 to 9 while i<n
Step 5:          Temp = A[i]
Step 6:          j = i
Step 7:          Repeat while j>= gap AND
                  A[j-gap] > Temp
                  A[j] = A[j-gap]
                  j =  j-gap
```

```
                    [End Loop]
Step 8:             **A[j] = Temp**
Step 9:             **i = i+1**
               [End Loop]
Step10:        **gap = gap/2**
          [End Loop]
Step11:  Exit
```

Analysis of shell sort

- Complexity of the shell sort is dependent upon the gap size sequence chosen.
- It has been empirically found that the shell sort performs better than insertion sort, bubble sort, and selection sort.
- The worst case complexity of the shell sort lies between **$O(n\log_2 n)$** and **$O(n^2)$** and on the average it is approximately **$O(n^{1.25})$**.

Radix Sort

- Radix Sort is a Invention of **Herman Hollerith** in year 1887 while working on tabulating machines.
- The radix sort is **non-comparison integer sort** method in which the comparison of elements to be sorted is not performed irrespective to other methods.
- Different from other methods like bubble sort , insertion sort, selection sort in which elements are compared with each other to perform sorting . But in radix sort structure of element is taken care rather than comparing the elements ,the structure is taken care.

- Individual digits of each element are taken care.
- Individual digits share the same significant position and value.
- Radix sort is known to be integer sort method.
- Even then this is not limited to sorting integers only.
- Can be applied to floating point , alpha-numeric elements or any other type.

Types of Radix Sort

MSD Radix Sort

The sorting starts from most significant digit(from left most digit to right most digit).MSD radix sort can be applied to sorting of string.

LSD Radix Sort

The sorting from least significant digit(from right most digit to left most digit). LSD radix sort is applied to numeric number sorting.

Example

To understand radix sort , consider the following sets of numbers

Example 1 :

005,017,008,785,890,889,431,,052,443,099,692

The given elements are numbers LSD radix sort will be applied to sort elements in ascending order.

Three passes require to sorting elements because elements given to be three digit numbers.

First pass

- The digit at 1’s place is taken care.
- Elements are written according to sorted order of this least significant digit.

Second pass

- The digit at 10’s place is taken care.
- Elements are written according to sorted order of this least 10’s place digit.

Third pass

- The digit at 100’s place is taken care.
- Elements are written according to sorted order of this least 100’s place digit.

Example

0	0	5
0	1	7
0	0	8
7	8	5
8	9	0
8	8	9
4	3	1
0	5	2
4	4	3
0	9	9
6	9	2

Pass =1

8	9	0
4	3	1
0	5	2
6	9	2
4	4	3
0	0	5
7	8	5
0	1	7
0	0	8
0	8	9
0	9	9

Pass=2

0	0	5
0	0	8
0	1	7
4	3	1
4	4	3
0	5	2
7	8	5
8	8	9
8	9	0
6	9	2
0	9	9

Pass=3

0	0	5
0	0	8
0	1	7
0	5	2
0	9	9
4	3	1
4	4	3
6	9	2
7	8	5
8	8	9
8	9	0

Algorithm of Radix sort

Algorithm: Sorts the elements using Radix sort.

// b is the maximum number of digits in each element

```
Step 1:     Repeat for **k=1 to b.**

Step 2:         Sort the elements in a stable way
                looking  at **kth** digit only.

            [End Loop]

Step 3: Exit
```

Analysis of Radix sort

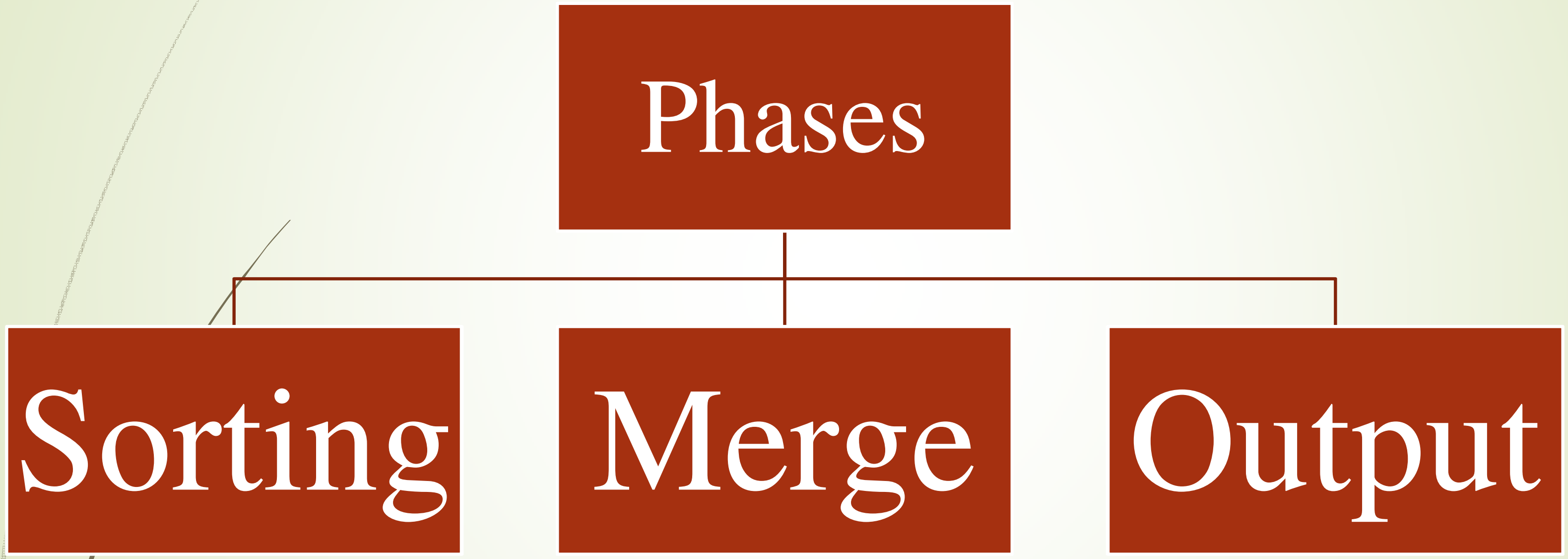
- The complexity of radix sort can be calculated depending upon the number of passes.
- The number of passes in the radix sort is as many as the number of digits in the elements.
- In each pass **n** elements are sorted looking at **k**$^{\mathbf{th}}$ digit only.
- Complexity of radix sort is **O(b n)** .Here n is Number of elements & **b** is maximum number of digit in each element.

- We can compare the complexity with **$O(n\log_2 n)$** and cannot guarantee that **O(b n)** is less than

  $O(n\log_2 n)$.
- This is because if we have **n-1** elements with a few digits but one element have more than **n** digits, than the values of **b** will becomes more than **n** and complexity will become more than **$O(n^2)$.** So, if the number of digits is less in the elements then the radix sort performs well .

External Sorting

- **External Sorting** is the sorting technique that can handle **huge** amount of data .
- Data is as large that cannot be accommodated in Main memory. Instead data must reside in a slower external Memory.
- The main concern with external sorting to minimize disk access.
- Usually, the external sorting is referred as **file sorting.**


Phases
Sorting
Merge
Output

Sorting Phase : chunks of data small enough that can fit .
In main memory and read ,sorted using an internal sorting technique.

Merge Phase : the sorted sub files are combined into large file.

Output Phase :the merged files are written out to the
external storage disk.

Example

Take example of sorting 12000 records which cannot be Accommodated in main memory as a whole simultaneously. To sort these records, external sorting technique is used.

Solution :

- First of all , these all records are divided into chunks Say chunks of 2000 records(which can be accommodated into the main memory) and apply the internal sorting method on all these 6 chunks .
- We get the 6 sorted sub files (each sub file contains 2000 sorted records) which are now sorted individually.

- Let these files are F1 having records 1 to 2000, F2 having records 2001 to 4000,F3 having records 4001 to 6000 and so on.
- Merge all these 6 sorted sub files starts. We use **2-way** merge in which, we use a method of merging **2 input files** at once.

Merge Pass 1

First of all files **F1** and **F2** are merged to get a file say **F7** which contains sorted record from 1 to 4000 and this file is written back to external storage media.

In second step , the files **F3** and **F4** are merged to get a file **F8** which sorted records from 4001 to 8000 and this
File is written back to external storage media.

In third step, the files **F5** and **F6** are merged to get file **F9** which contains the sorted record from 8001 to 12000 and this file written back to external storage media . Now, the problem is reduced to merge the three files **F7**, **F8**
And **F9** .

Merge Pass 2:

First of all files **F7** and **F8** are merged to get a file say **F10** which contains sorted records from 1 to 8000 and this file is written back to external storage media .
As only one file **F9** remaining, so in this pass , we leave this file as it is. And now the problem is reduced to merge the two files **F10** and **F9**.

Merge Pass 3:

Here , the files **F10** containing records 1 to 8000 and file **F9** containing records 8001 to 12000 are merged to create a single file which contains all the records 1 to 12000 which are now and this file is written back to external storage media.