UNIT - III

SEARCHING

- Searching means to find whether a particular value is present in an array or not.
- If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.
- However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
- Searching techniques are
    1. **Linear search,**
    2. **Binary search** and
    3. **Fibonacci Search**

LINEAR SEARCH:

- Linear search is a technique which traverses the array sequentially to locate given item or search element i.e., Key element.
- In Linear search, we access each element of an array one by one sequentially and see whether it is desired element or not. We traverse the entire list and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.
- A search is successful then it will return the location of desired element
- If A search will unsuccessful if all the elements are accessed and desired element not found.
- Linear search is mostly used to search an unordered list in which the items are not sorted.

Linear search is implemented using following steps...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the first element in the list.

Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function

Step 4 - If both are not matched, then compare search element with the next element in the list.

Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.

Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Example:

Consider the following list of elements and the element to be searched...

Step 1:

search element (12) is compared with first element (65)

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

Both are not matching. So move to next element

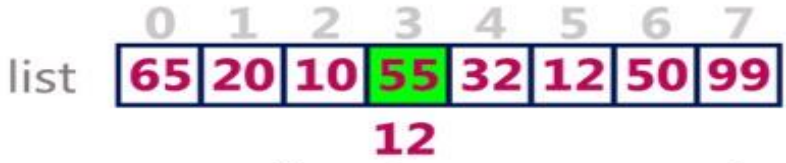
Step 3:

search element (12) is compared with next element (10)

Both are not matching. So move to next element

Step 4:

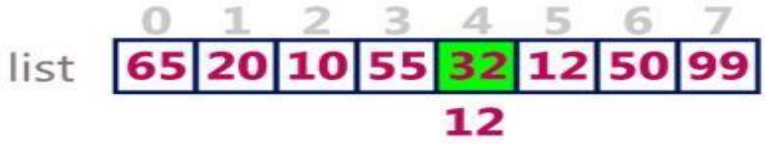
search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)



Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

BINARY SEARCH:

- Binary search is the search technique which works efficiently on the **sorted lists**. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.
- Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Basic Idea:

We are given a ***sorted list*** of elements and a key. We start the search process by comparing the keywith middle element of the list. Here, one of the following 3 cases may occur:

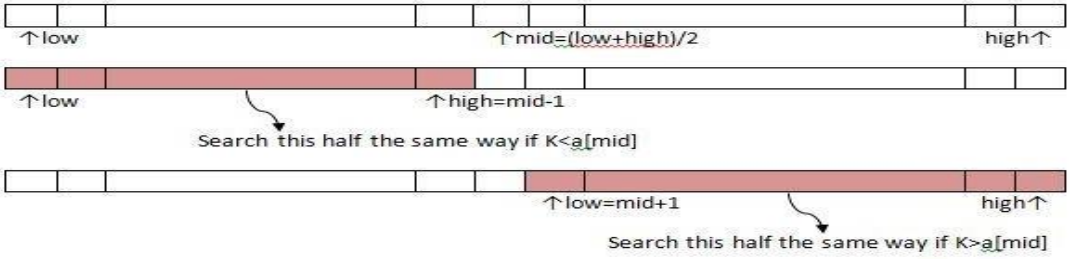
Case 1: key is matched with middle element. Search is successful; we stop the search by returning the position of key

Case 2: key is less than middle element. In this case, we restrict our binary search (we apply recursively) only to the first part of the list.

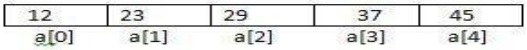
Case 3: key is greater than middle element. In this case, we restrict our binary search only to the second part of the list.

In either of the above cases (if key is not matched with middle element of the list i.e. case-2 or case-3), we restrict our search only to half of the list, thereby reducing list size by half at each step.

If the key is not available in the given list (unsuccessful search case), then because of applying the above process recursively the list will be divided into single element. Even this element will also be not matched with key, thereby, we can say that the search is unsuccessful.



Example: Consider an array a



and K=37,

Array	Variables	Comparison
Pass 1: 12 23 29 37 45 0 1 2 3 4 ↑ ↑ ↑ low mid high	low=0 high =4 mid =(low+high)/2 =(0+4)/2=2	 29=37 (False)
Pass 2: Since 37>29 consider right half of the array. 12 23 29 37 45 0 1 2 3 4 low↑ ↑high mid↑	low =mid+1=2+1=3 high =4 mid =(3+4)/2≈3	 37=37(Key element found)

Algorithm:

Step 1 - Read the search element from the user.

Step 2 - Find the middle element in the sorted list.

Step 3 - Compare the search element with the middle element in the sorted list.

Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.

Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.

Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.

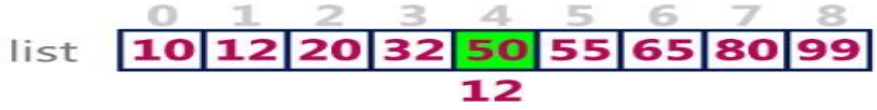
Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Example:

search element 12

Step 1:

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

Example 2:

search element 80

Step 1:

search element (80) is compared with middle element (50)

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

FIBONACCI SEARCH:

- **Fibonacci search** is an efficient search algorithm based on **divide and conquer** principle that can find an element in the given **sorted array** with the help of Fibonacci series in **O(log N)** time complexity. This is based on Fibonacci series which is an infinite sequence of numbers denoting a pattern which is captured by the following equation:

  F(n)=n **if n<=1**

  F(n)=F(n-1)+F(n-2) **if n>1**

  - where F(i) is the ith number of the Fibonacci series where F(0) and F(1) are defined as 0 and 1 respectively.
- The first few Fibonacci numbers are: **0,1,1,2,3,5,8,13....**

  F(0) = 0, F(1) = 1

  F(2) = F(1) + F(0)= 1 + 0 = 1F(3) = F(2) +

  F(1) = 1 + 1 = 2

  F(4) = F(3) + F(2) = 1 + 2 = 3 and so continues the series
- Other searches like binary search also work for the similar principle on splitting the search space to a smaller space but what makes Fibonacci search different is that it divides the array in **unequal parts** and operations involved in this search are **addition and subtraction** these arithmetic operations takes place simple and hence **reducing the work load of the computing machine**.

Algorithm:

- Let the length of given array be **n [0. n-1]** and the element to be searched be **x**.
- Then we use the following steps to find the element with minimum steps:

1. Find the **smallest Fibonacci number greater than or equal to n**. Let this number be **f(M)**

Let the two Fibonacci numbers preceding it be **f(M-1)** and **f(M-2)**.

F(M) = F(Size of array)

F(M-1) = F(M) - 1

F(M-2) = F(M-1) -1

i (index) = min (offset + F(M-2) , n-1) //Offset = -1

2. While the array has elements to be checked:

-> Compare x with the last element of the range covered by f(M-2)

-> If **x** matches, return index value

-> Else if **x is less** than the element, move the three Fibonacci variables two Fibonacci down, Indicating removal of approximately two-third of the unsearched array from rear end. Not Reset offset to index

-> Else x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Indicating removal of approximately one-third of the unsearched array from front end.

3. Since there might be a single element remaining for comparison, check if F(M-1) is '1'. If Yes, compare x with that remaining element. If match, return index value.

Example: The Elements in array & Search key is

Search_Key 85

elements	10	22	35	40	45	50	80	82	85	90	95
Index	0	1	2	3	4	5	6	7	8	9	10

Initially the Fibonacci series is …

0	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**21**	**34**
1	2	3	4	5	6	7	8	9	10
					F(m-2)	F(m-1)	F(m)		

To calculate index position **i = min(offset + F(m-2), n-1),** Initially offset value is -1.

F(m)	**F(m-1)**	**F(m-2)**	**Offset**	**i(index)**	**a[i]**	**Consequence**
13	8	5	-1	(-1+5,10) = 4	45	1 steps down, Reset offset
8	5	3	4	(4+3, 10)=7	82	1 steps down, Reset offset
5	3	2	7	(7+2, 10) =9	90	2 steps down
2	1	1	7	(7+1, 10) = 8	85	Return i

Finally our desired element is **found at the location of 8.**

SORTING:

Definition: Sorting is a technique to rearrange the list of records(elements) either in ascending or descending order, Sorting is performed according to some key value of each record.

The sorting can be divided into two categories. These are:

- Internal Sorting
- External Sorting

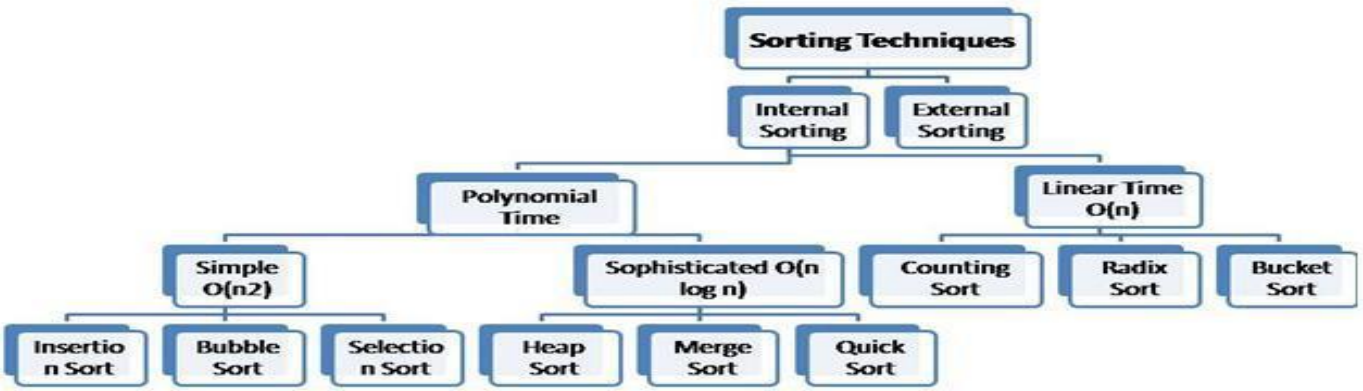


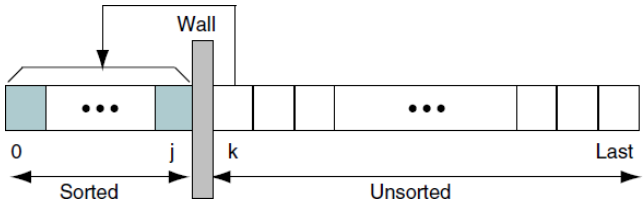
Fig . Classification Of Sorting Techniques

- **Internal Sorting:** When all the data that is to be sorted can be accommodated at a time in the main memory (Usually RAM). Internal sortings has five different classifications: bubble, insertion, shell, selection, merging, heap, quick, Counting, Radix / Bucket sort
- **External Sorting:** When all the data that is to be sorted can't be accommodated in the memory (Usually RAM) at the same time and some have to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc.

  Ex: Natural, Balanced, and Polyphase

INSERTION SORT:

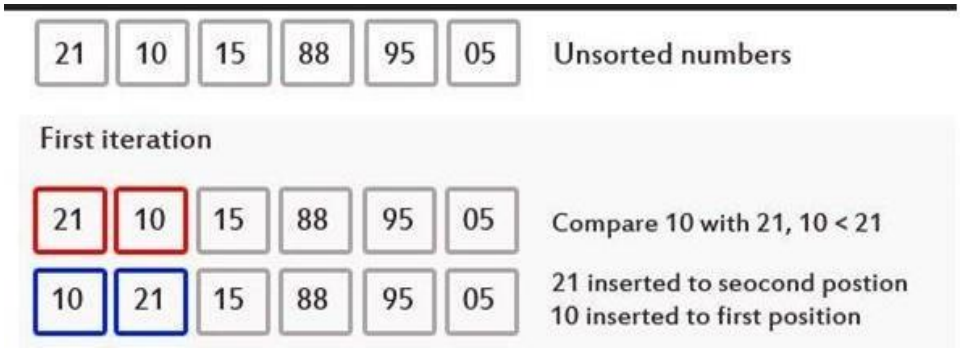
- In Insertion sort the list can be divided into two parts, one is sorted list and other is unsorted list. In each pass the first element of unsorted list is transfers to sorted list by inserting it in appropriate position or proper place.
- The similarity can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.



Following are the steps involved in insertion sort:

1. We start by taking the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards
2. We compare the key element with the element(s) before it, in this case, element at index 0:
   - If the key element is less than the first element, we insert the key element before the first element.
   - If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the proper position.
4. And we go on repeating this, until the array is sorted.

Example:



Second iteration

10	21	15	88	95	05	Compare 15 with 21, 15 < 21
10	?	21	88	95	05	21 inserted to third postion
10	?	21	88	95	05	Compare 15 with 10, 15 > 10
10	15	21	88	95	05	15 got inserted to second position 10 inserted to first position

Third iteration

10	15	21	88	95	05	Compare 88 with 21, 88 > 21
10	15	21	88	95	05	Compare 88 with 15, 88 > 15
10	15	21	88	95	05	Compare 88 with 10, 88 > 10

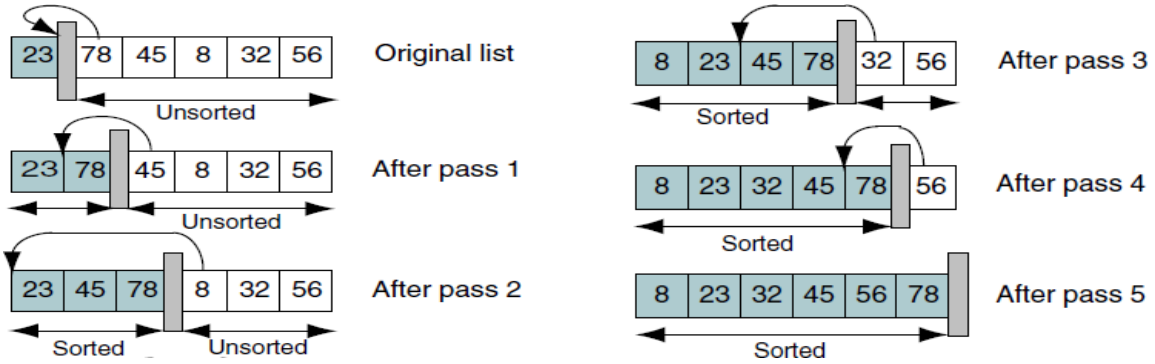
Fourth iteration

10	15	21	88	95	05	Compare 95 with 88 , 95 > 88
10	15	21	88	95	05	Compare 95 with 21, 95 > 21
10	15	21	88	95	05	Compare 95 with 15, 95 > 15
10	15	21	88	95	05	Compare 95 with 10, 95 > 10

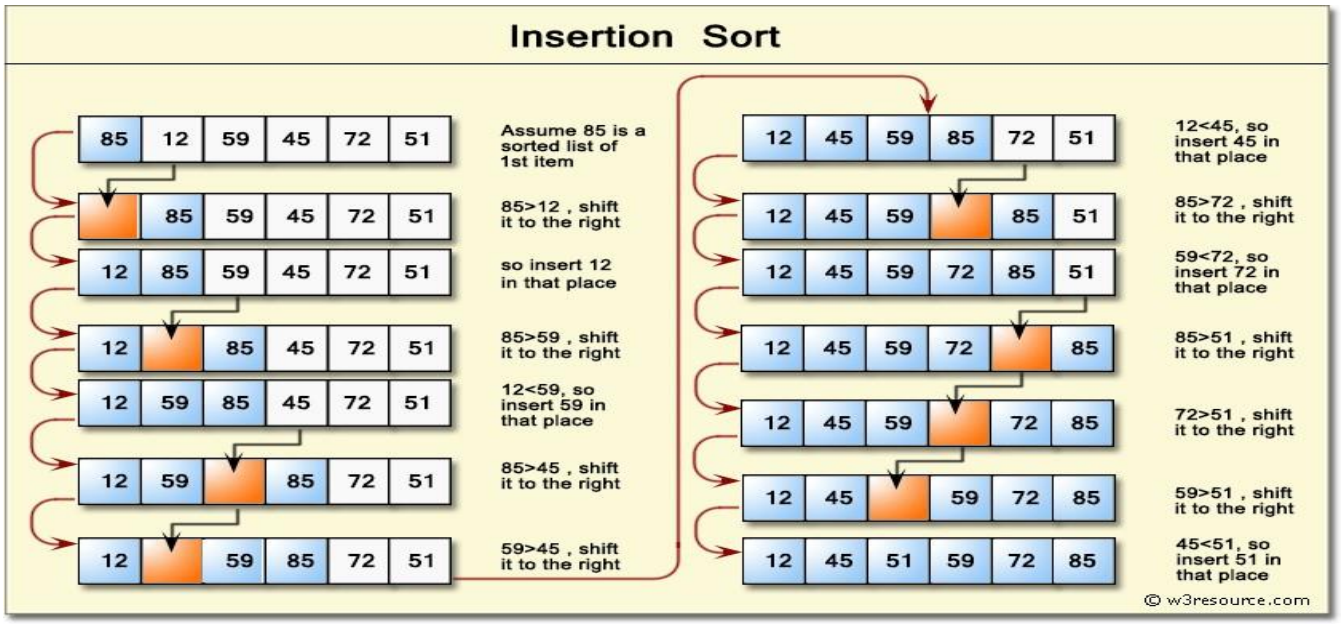
Fifth iteration

10	15	21	88	95	05	Compare 05 with 95, 05 < 95
10	15	21	88	?	95	95 inserted to last position
10	15	21	?	88	95	Compare 05 with 88, 05 < 88 88 inserted to next position
10	15	?	21	88	95	Compare 05 with 21, 05 < 21 21 inserted to next position
10	?	15	21	88	95	Compare 05 with 15, 05 < 15 15 inserted to next position
05	10	15	21	88	95	Compare 05 with 10, 05 < 10 10 inserted to next position and 05 inserted to first position

Example 1:



Example 2:



Analysis (or) Time complexity of Insertion sort:

Best case: If the list is already sorted order, then it is the best case for insertion sort. In this case the inner loop is not executed at all. Therefore, the complexity of insertion sort in the best case in O(n).

Worst case: If the list is sorted in reverse order (in descending order), then it is the worst case for insertion sort. In this case, the inner loop is executed completely i.e,

In pass-1, the number of iterations of inner loop = 1

In pass-2, the number of iterations of inner loop = 2

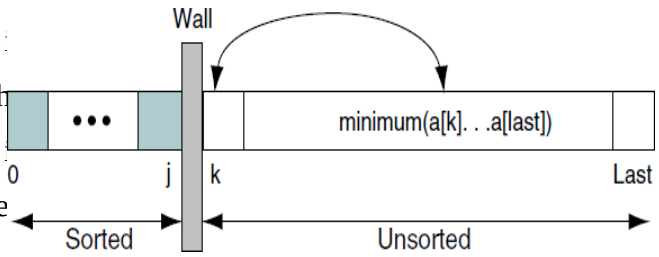
In pass-3, the number of iterations of inner loop = 3

In pass (n-1), the number of iterations of inner loop = (n-1).

Therefore, total number of comparisons = 1+2+3+….+(n-1) = O(n2).

SELECTION SORT

- Given a list of data to be sorted, we simply select the smallest item and place it in a sorted list. These steps are then repeated until we have sorted all of the data.
- In first step, the smallest element is search in the list, once the smallest element is found, it is exchanged with the element in the first position.
- Now the list is divided into two parts. One sorted list other is unsorted list. Find out th smallest element in the unsorted list and it exchange with the starting position of unsorte list, after that it will add in to sorted list.



- This process is repeated until all the elements are sorted.

Ex: asked to sort a list on paper.

Algorithm: **SELECTION SORT(ARR, N)**

Step 1: Repeat Steps 2 and 3 for K = 1 to N-1

Step 2: CALL SMALLEST(ARR, K, N, Loc)

Step 3: SWAP A[K] with ARR[Loc]

Step 4: EXIT

Algorithm for finding minimum element in the list.

SMALLEST (ARR, K, N, Loc)
Step 1: [INITIALIZE] SET Min = ARR[K]

Step 2: [INITIALIZE] SET Loc = K

Step 3: Repeat for J = K+1 to N

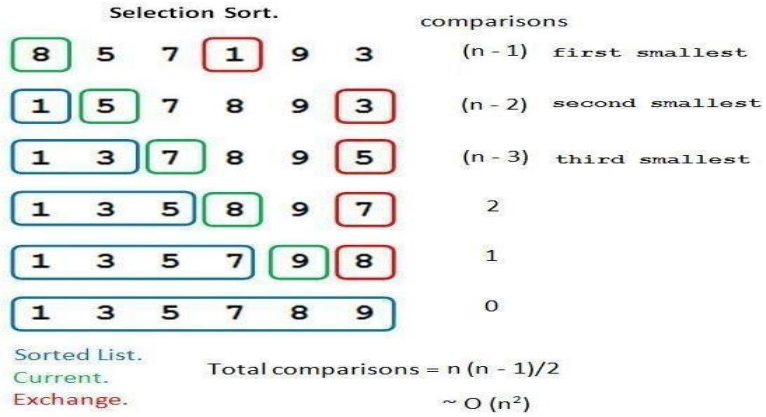
IF Min > ARR[J]

SET Min = ARR[J]
SET Loc = J

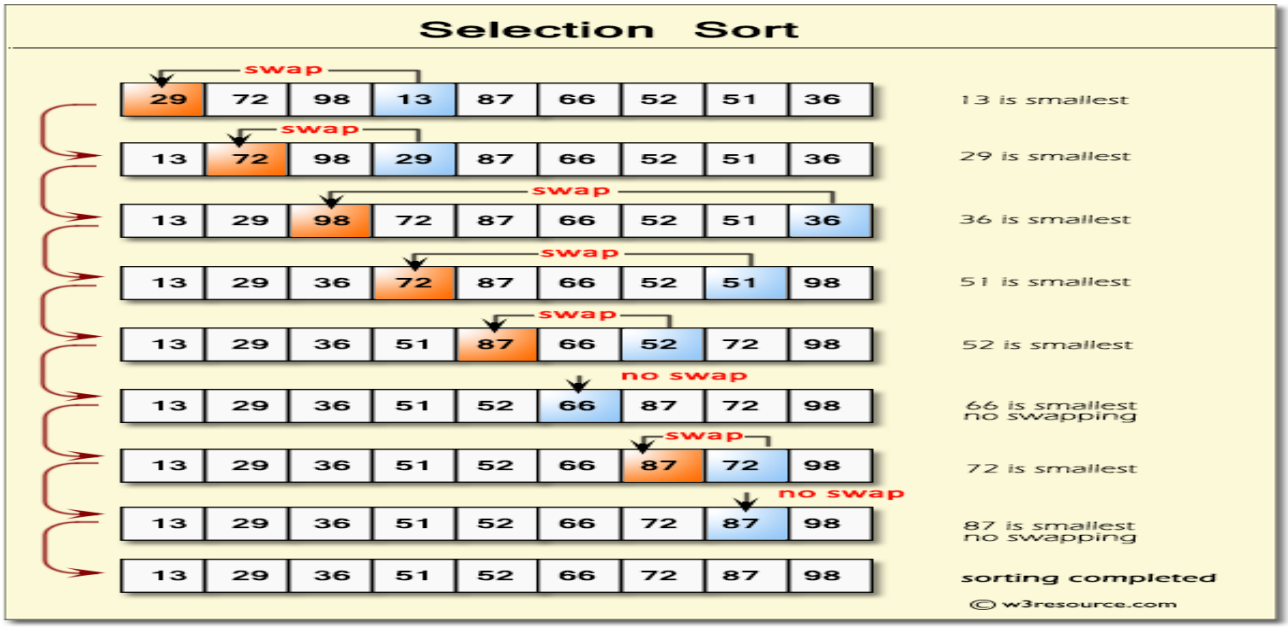
[END OF IF]

[END OF LOOP]

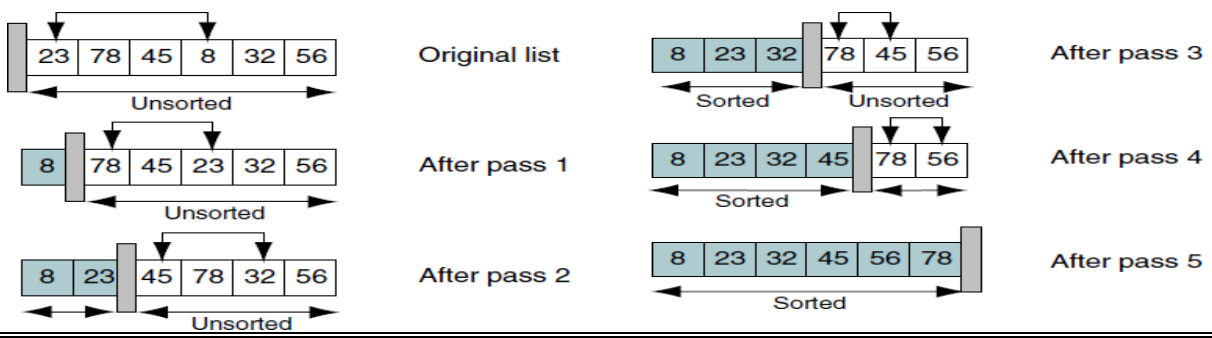
Step 4: RETURN Loc



Example 1:



Example 2: Consider the elements 23,78,45,88,32,56



Time Complexity:

Number of elements in an array is 'N' Number of

passes required to sort is 'N-1'

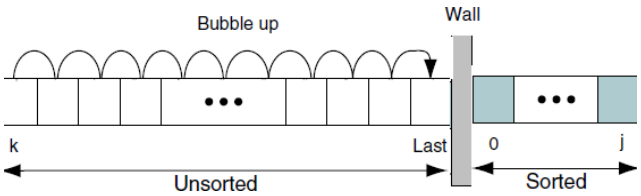
Number of comparisons in each pass is 1st pass N-1, 2nd Pass N-2 …

Time required for complete sorting is: $T(n) <= (N-1)*(N-1) T(n) <= (N-1)^2$

Finally, The time complexity is $O(n^2)$.

BUBLLE SORT

- Bubble Sort is also called as Exchange Sort
- In Bubble Sort, each element is compared with its adjacent element
  a) If he first element is larger than the second element then the position of the elements are interchanged.
  b) Otherwise, the position of the elements is not changed.
  c) The same procedure is repeated until no more elements are left for comparison.
- After the 1st pass the largest element is placed at (N-1)th location. Given a list of *n* elements, the bubble sort requires up to *n* – 1 passes to sort the data.



Algorithm:

BUBBLE SORT(ARR, N)

Step 1: Read the array elements

Step 2: i:=0;

Step 3: Repeat step 4 and step 5 until i<n

Step 4: j:=0;

Step 5: Repeat step 6 until j<(n-1)-i

Step 6: if A[j] > A[j+1]

Swap(A[j],A[j+1])

End if

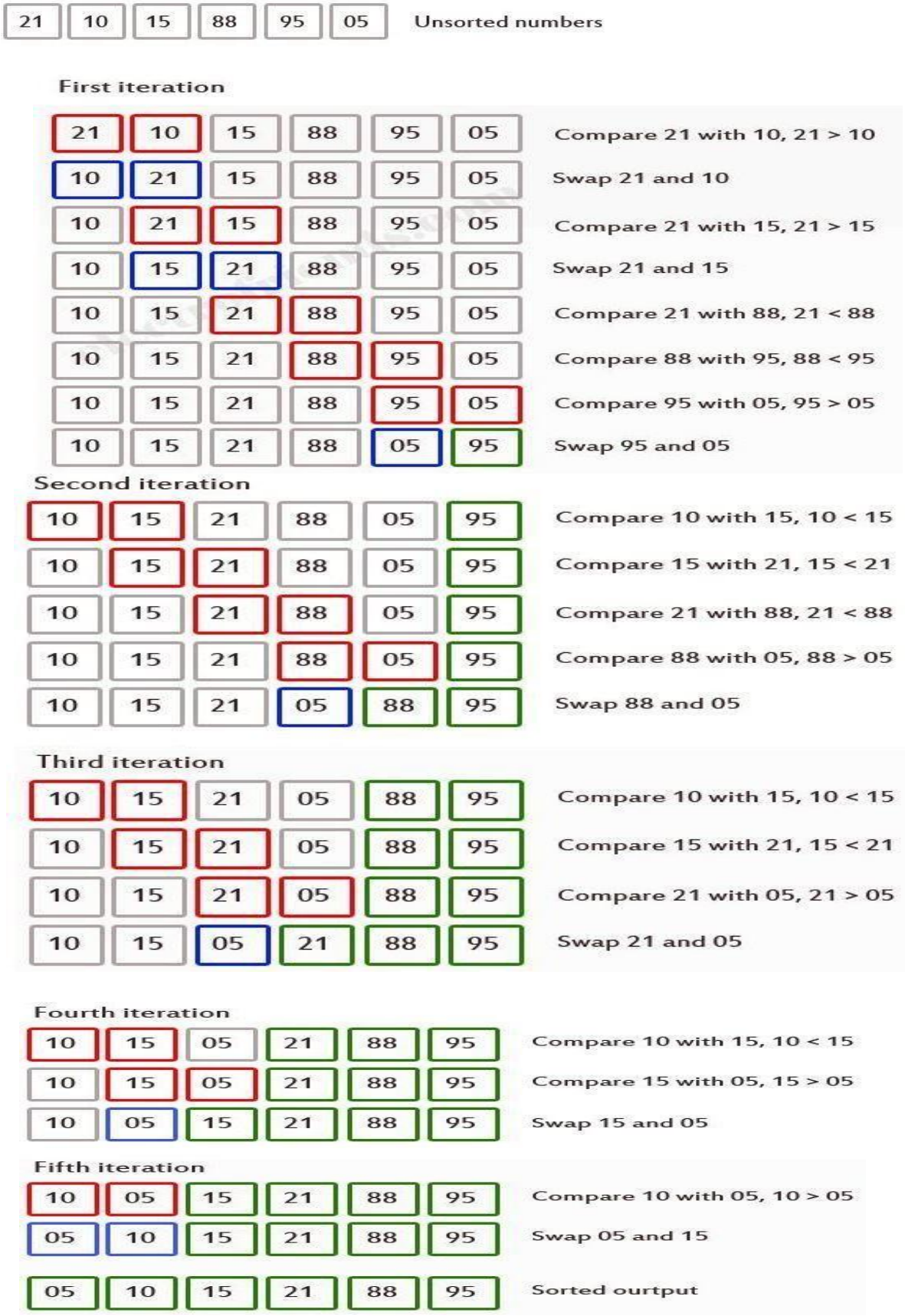
End loop 5

End loop 3

Step 7: EXIT

Example 1:

Example 2:



Time Complexity:

Number of elements in an array is 'N' Number of passes required to sort is 'N-1'

Number of comparisons in each pass is 1st pass N-1, 2nd Pass N-2 …

Time required for complete sorting is: $T(n) <= (N-1)*(N-1) T(n) <= (N-1)^2$

Finally, The time complexity is $O(n^2)$.

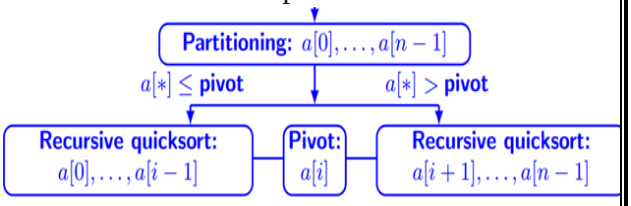
QUICK SORT:

- Quick sort follows **Divide and Conquer** algorithm. It is dividing array in to smaller parts based on partitioning and performing the sort operations on those divided smaller parts. Hence, it works well for large datasets.

So, here are the steps **how Quick sort** works in simple words.

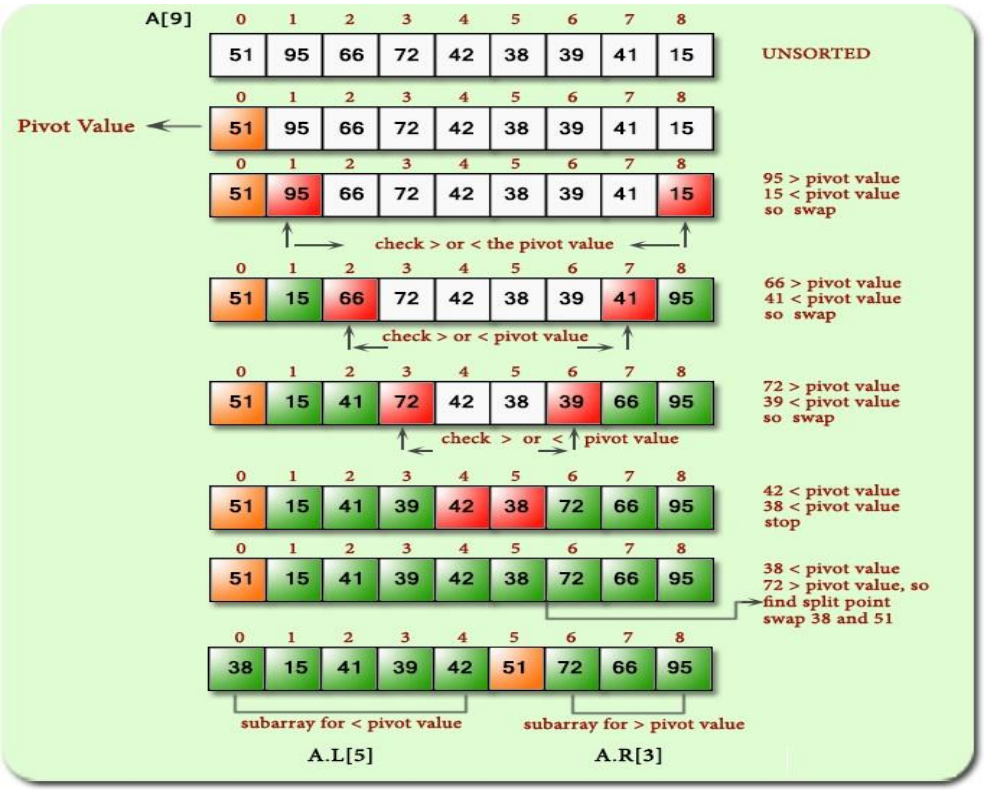
1. First select an element which is to be called as **pivot** element.
2. Next, compare all array elements with the selected pivot element and arrange them in such a waythat, an element less than the pivot element are to its left and greater than pivot is to it's right.
3. Finally, perform the same operations on left and right side elements to the pivot element.

How does Quick Sort Partitioning Work



1. First find the **"pivot"** element in the array.
2. Start the left pointer at first element of the array.
3. Start the right pointer at last element of the array.
4. Compare the element pointing with left pointer and if it is less than the pivot element, then move the left pointer to the right (add 1 to the left index). Continue this until left side element is greater than or equal to the pivot element.
5. Compare the element pointing with right pointer and if it is greater than the pivot element, then move the right pointer to the left (subtract 1 to the right index). Continue this until right side element is less than or equal to the pivot element.
6. Check if left pointer is less than or equal to right pointer, then swap the elements in locations of these pointers.
7. Check if index of left pointer is greater than the index of the right pointer, then swap pivot element with right pointer.

Example:



Algorithm:

```
quickSort(array, lb, ub)
{
   if(lb< ub)
   {
      pivotIndex = partition(arr, lb, ub);
      quickSort(arr, lb, pIndex - 1);
       quickSort(arr, pivotIndex+1, ub);
   }
}
```

RADIX SORT:

- Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort.
- Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.
- During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the n^{th} pass, where n is the length of the name with maximum number of letters.
- When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant (LSD) to the most significant (MSD) digit. While sorting the numbers, we have **ten** buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the **length** of the number having maximum number of digits.

Example 1: Sort the given numbers using radix sort.345, 654, 924, 123, 567, 472, 555, 808, 911

- In the first pass, the numbers are sorted according to the digit at ones place.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

- After this pass, the numbers are collected bucket by bucket. In the second pass, the numbers are sorted according to the digit at the tens place.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

- In the third pass, the numbers are sorted according to the digit at the hundreds place.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

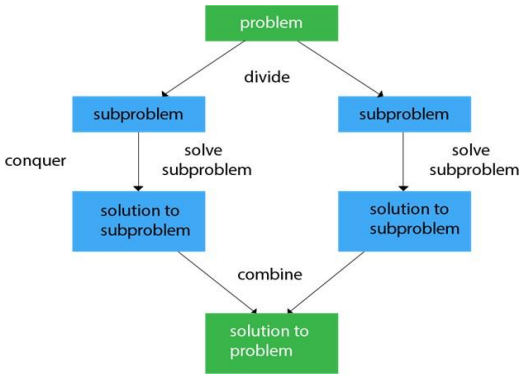
- The numbers are collected bucket by bucket. After the third pass, the list can be given as final sorted list. 123, 345, 472, 555, 567, 654, 808, 911, 924.

Algorithm:

1. Let **A** be a linear array of **n** elements **A[1], A[2], A[3]...A[n]**. Digit is the total number of digit in the largest element in array **A**.
2. Input n number of elements in an array A.
3. Find the total number of digits in the largest element in the array.
4. Initialize i=1 and repeat the steps 4 and 5 until(i<=Digit).
5. Initialize the bucket j=0 and repeat the steps 5until (j<n).
6. Compare the i^{th} position of each element of the array with bucket number and place it in thecorresponding bucket.
7. Read the elements (S) of the bucket from 0^{th} bucket to 9^{th} bucket and from the first position to the higherone to generate new array A.
8. Display the sorted array A.
9. Exit

Divide and Conquer:

- Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.



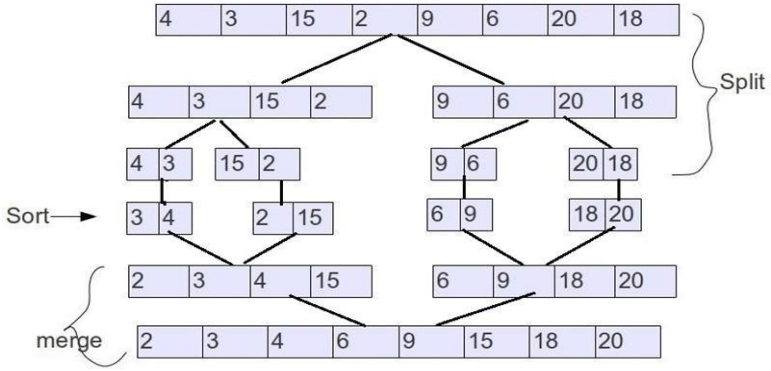
- Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of sub-problems.
2. **Conquer:** Solve every sub-problem individually, recursively.
3. **Combine:** Put together the solutions of the sub-problems to get the solution to the whole problem.

MERGE SORT

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.



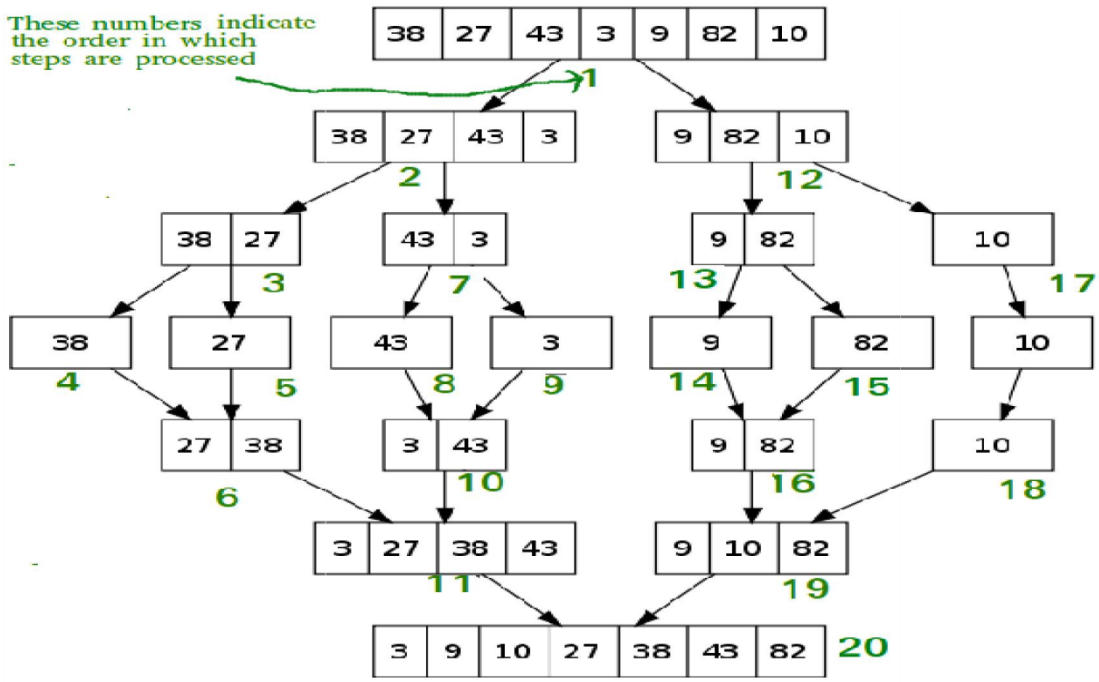
The Merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub arrays into one.

MergeSort(arr[], l, r)

If l<r

1. Find the middle point to divide the array into two halves:
   middle m = (l+r)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



MergeSort Algoritm:

```
MergeSort(A, lb, ub )
{
   If  lb<ub
         {
            mid = floor(lb+ub)/2;
            mergeSort(A, lb, mid);
            mergeSort(A, mid+1, ub);
            merge(A, lb, ub , mid);
         }
}
```

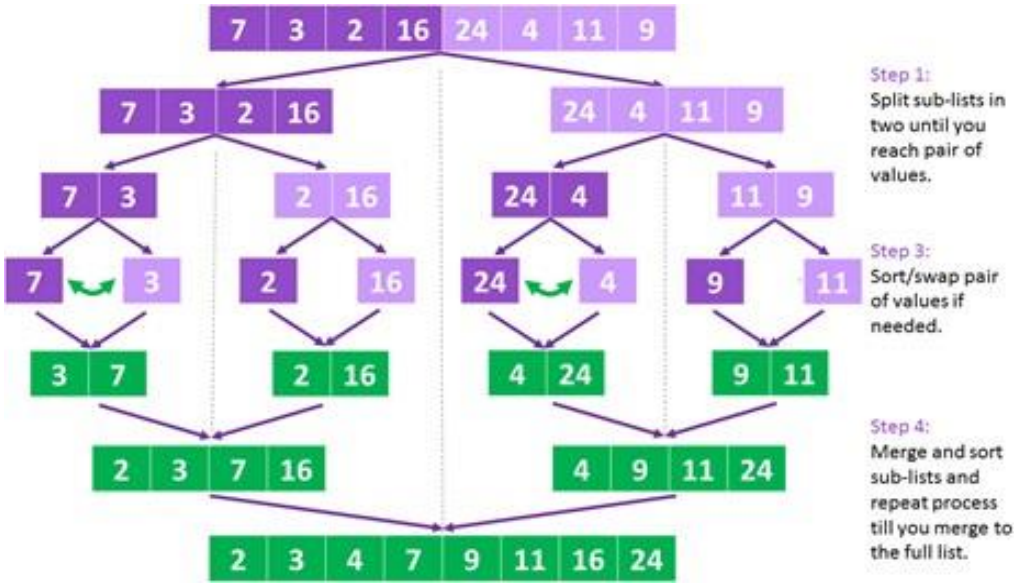
Implementation Recursive Merge Sort:

- The merge sort starts at the Top and proceeds downwards, "split the array into two, make a recursive call, and merge the results.", until one gets to the bottom of the array-tree.

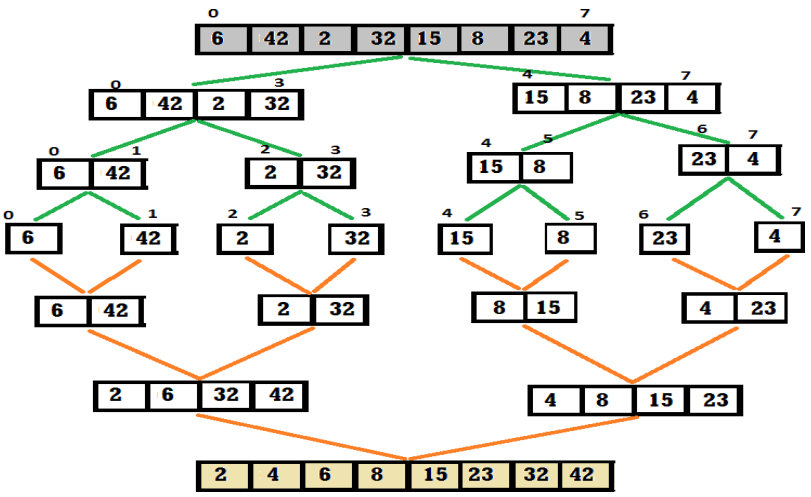
Example: Let us consider an example to understand the approach better.

1. Divide the unsorted list into n sub-lists based on mid value, each array consisting 1 element
2. Repeatedly merge sub-lists to produce newly sorted sub-lists until there is only 1 sub-list remaining. This will be the sorted list

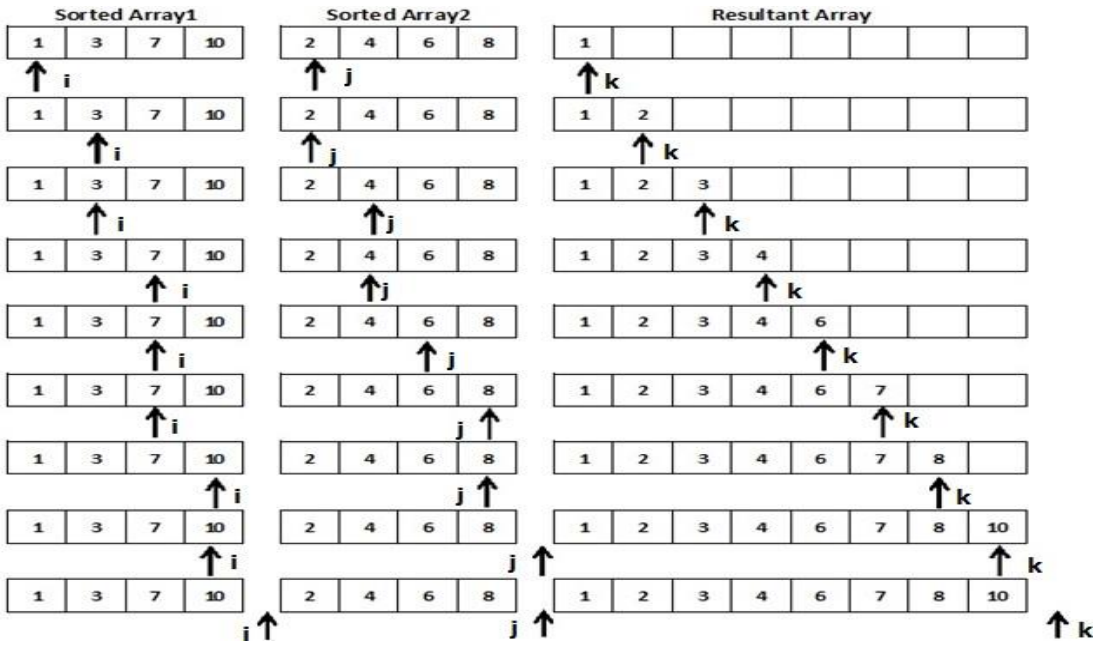
Recursive Mere Sort Example:



Example 2:



Two- Way Merge Sort:



Merge Algorithm:

Step 1: set i,j,k=0

Step 2: if A[i]<B[j] then

copy A[i] to C[k] and increment i and k

else

copy B[j] to C[k] and increment j and k

Step 3: copy remaining elements of either A or B into Array C.

BASIS FOR COMPARISON	QUICK SORT	MERGE SORT
The partition of elements in the array	The splitting of a array of elements is inany ratio, not necessarily divided into half.	The splitting of a array of elements is inany ratio, not necessarily divided into half.
Worst case complexity	O(n2)	O(nlogn)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sortingalgorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
Additional storage space requirement	Less(In-place)	More(not In-place)
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Locality of reference	good	poor

Shell Sort:

Shell Sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shell Sort is to allow exchange of far items. Inshell Sort, we make the array h-sorted for a large value of h. We keep reducing thevalue of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element is sorted.

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm.This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval.

This interval is calculated based on Knuth's formula as −

Knuth's Formula

h = h * 3 + 1

Where −h is interval with initial value 1

This algorithm is quite efficient for medium-sized data sets as its average and worst- case complexity of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items. And the worst case space complexity is O(n).

Algorithm: Following is the algorithm for shell sort.

Step 1 − Initialize the value of *h*

Step 2 − Divide the list into smaller sub-list of equal interval *h*

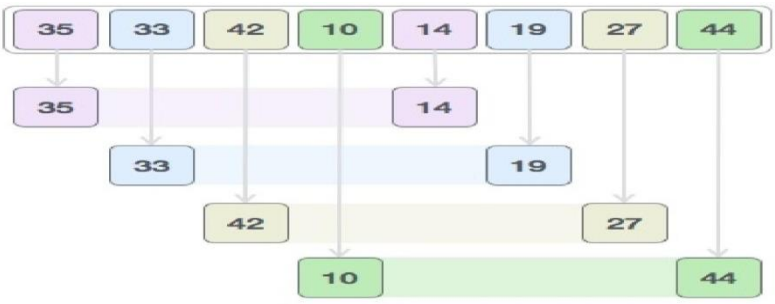
Step 3 − Sort these sub-lists using insertion sort

Step 4 − Repeat until complete list is sorted.

How Shell Sort Works?

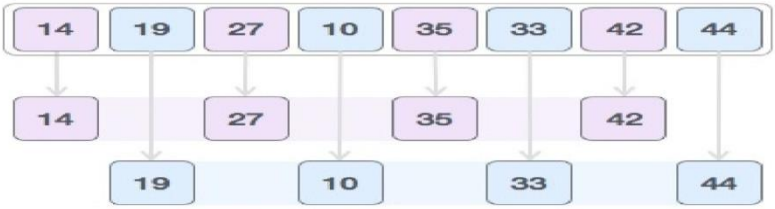
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions.

Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}

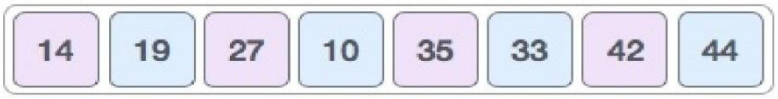


We compare values in each sub-list and swap them (if necessary) in the original array.After this step, the new array should look like this –

Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19,10, 33, 44}



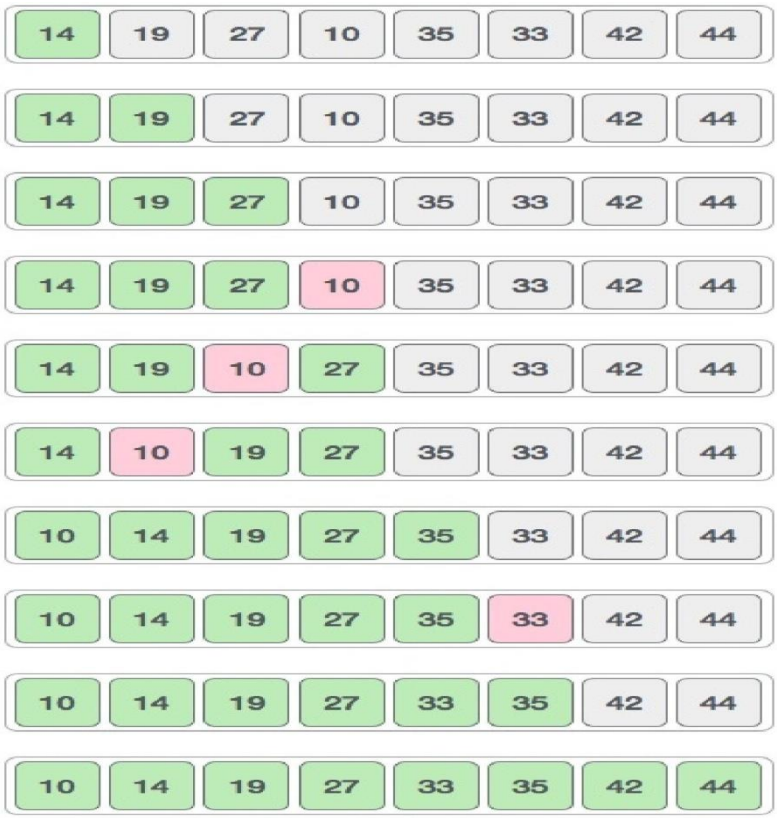
We compare and swap the values, if required, in the original array. After this step, thearray should look like this –



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertionsort to sort the array.

Following is the step-by-step depiction –

We see that it required only four swaps to sort the rest of the array.



Heap Sort:

Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.

A min heap is an almost complete binary tree such that the value of each node is lessthan or equal to those in its children.

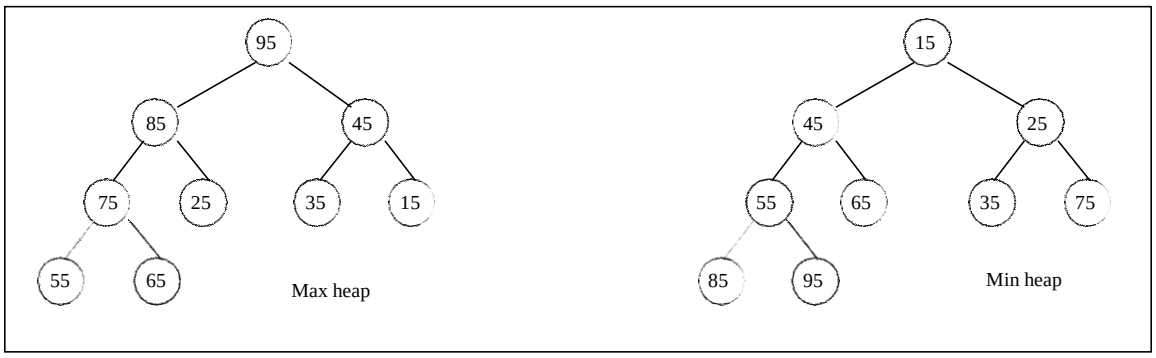


Figure 2.1 shows the maximum and minimum heap tree.

Representation of Heap Tree:

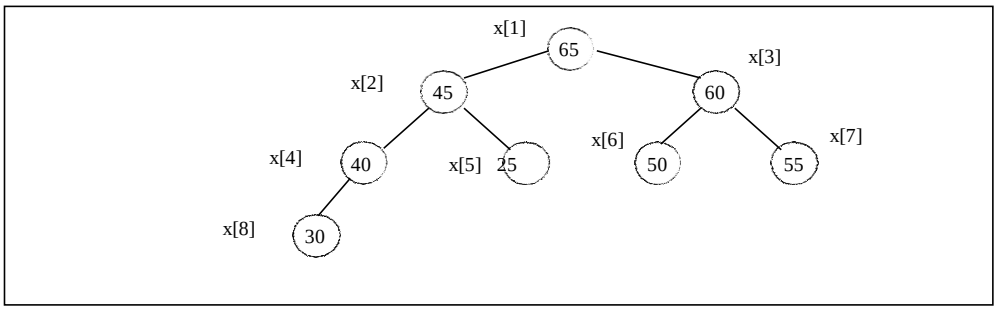
Since heap is a complete binary tree, a heap tree can be efficiently represented usingone dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location *i* can be found in location 2**i*.
- The right child of an element stored at location *i* can be found in location 2*i + 1.
- The parent of an element stored at location *i* can be found at location floor (*i*/2).

For example let us consider the following elements arranged in the form of array asfollows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30

The elements of the array can be thought of as lying in a tree structure. A heap treerepresented using a single array looks as follows:



Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we haveto adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted nodeto the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchanges as well as further comparisons is no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place.

Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

Max_heap_insert (a, n)

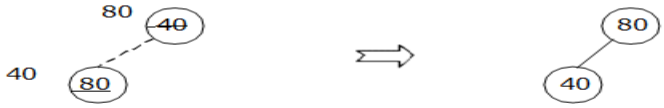
```
{
        //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]integer i, n;
        i = n;
        item = a[n]  ;
        while ( (i > 1) and (a[  i/2  ] < item ) do
        {
                a[i] = a[ i/2   ] ;// move the parent downi = i/2  ;
        }
        a[i] = item ;
        return true ;
}
```

Example:

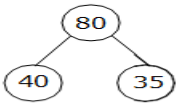
Form a heap by using the above algorithm for the given data 40, 80, 35, 90, 45, 50, 70.

1. Insert 40:

2. Insert 80:

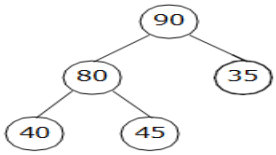


3. Insert 35:



4. Insert 90:

5. Insert 45:



6. Insert 50:

7. Insert 70:

Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap thetree as stated below:
  - Let newly modified root node be the current node. Compare itsvalue with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node
  - Make X as the current node.
  - Continue re-heap, if the current node is not an empty node.

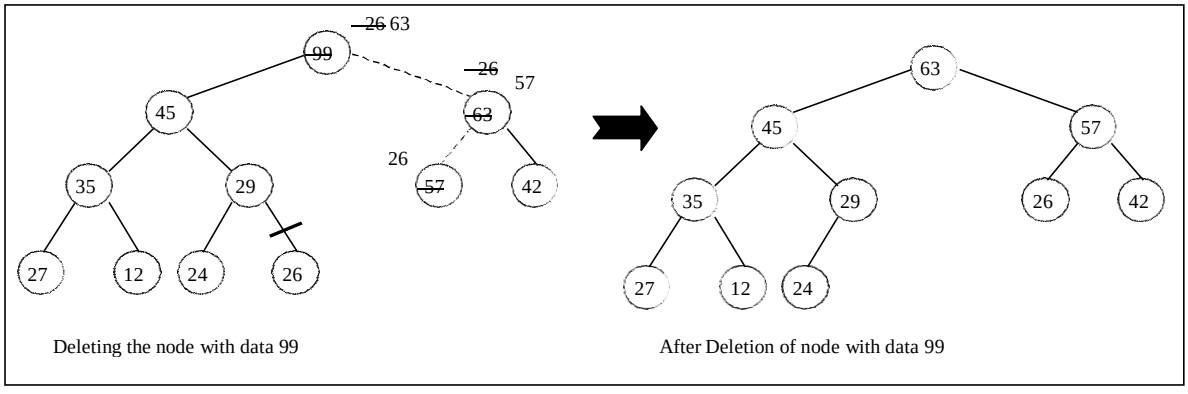
The algorithm for the above is as follows:

delmax (a, n, x)

```
// delete the maximum from the heap a[n] and store it in x
{
        if (n = 0) then
        {
                write ("heap is empty");
                return false;
        }
            X= a[1];
           a[1] = a[n];
           adjust(a, 1, n-1);
           return true;
} adjust (a, i, n)
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to form asingle
heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j <- j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j))
                 then break; // a position for item is found
                else
                 a[ ⌊ j / 2 ⌋ ] = a[j] // move the larger child up a level
                j = 2 * j;
        }
        a [ ⌊ j / 2 ⌋ ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now, 26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appear as the leave node, hence re-heap is completed.

This is shown in figure 2.3.



Deleting the node with data 99 After Deletion of node with data 99

Comparing different Sorting Algorithms:

Shell Sort:

The shell sort is by far the fastest of the class of sorting algorithms. It is more than 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, itsclosest competitor.

Heap Sort:

It is the slowest of the sorting algorithms but unlike merge and quick sort it does notrequire massive recursion or multiple arrays to work

Merge Sort:

The merge sort is slightly faster than the heap sort for larger sets, but it requires twice the memory of the heap sort because of the second array.

Quick Sort:

The quick sort is an in-place, divide-and-conquer, massively recursive sot. It can be said as the faster version of the merge sort. The efficiency of the algorithm is majorlyimpacted by which element is chosen as the pivot point. The worst-case efficiency of the quick sort is when the list is sorted and left most elements are chosen as the pivot.
As long as the pivot point is chosen randomly, the quick sort has an algorithmic complexity of O(nlogn)

Time Complexities All the Searching & Sorting Techniques:

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)