Unit – V

Syllabus:

- Searching - Linear search, Binary search, Fibonacci search-Hashing
- Sorting- Definition-Insertion sort, Selection sort, Exchange (Bubble sort, Quick sort), merging (Merge sort)-Iterative and recursive merge sort, Shell sort- Radix sort-Heap sort.

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 processgives the location of that value in the array.
- However, if the value is not present in the array, the searching process displays an appropriatemessage and in this case searching is said to be unsuccessful.
- Searching techniques are ***linear search, binary search*** and ***Fibonacci Search***

LINEAR SEARCH:

- Linear search is a technique which traverses the array sequentially to locate given item or search 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.

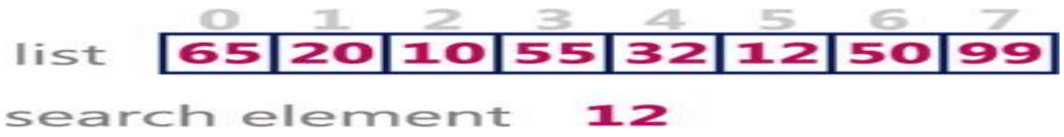
Step 2 - Compare the search element with the first element in the list.

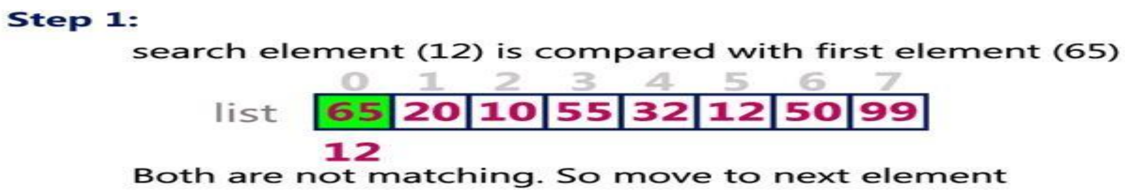
Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function**Step 4 -** If both are not matched, then compare search element with the next element in the list. **Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.

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

Example:

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





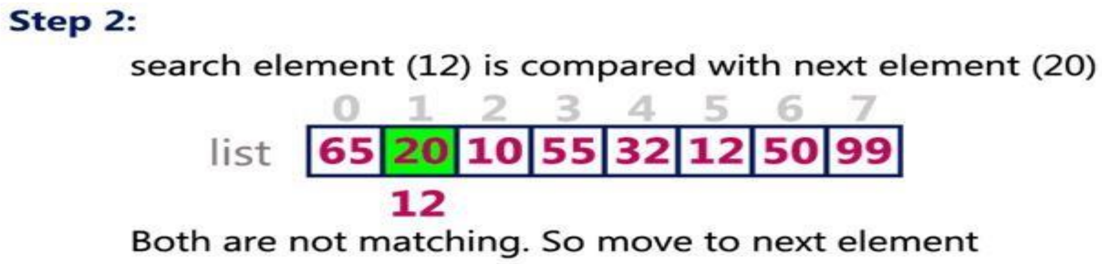
Step 1:

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

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element



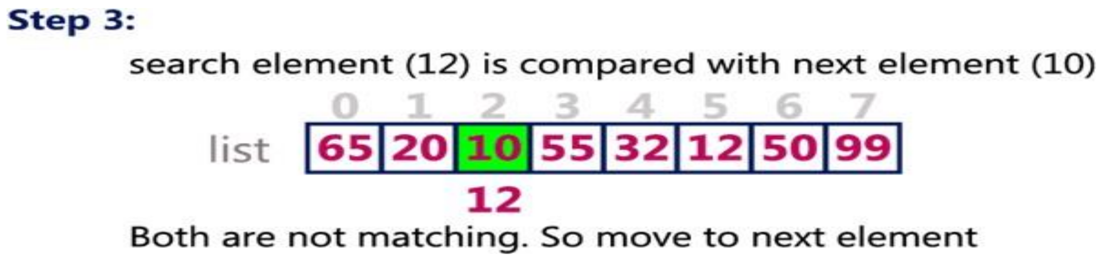
Step 2:

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

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element



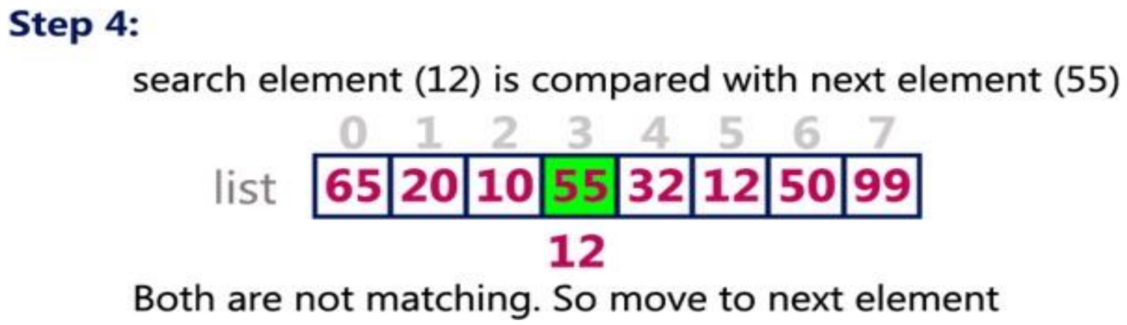
Step 3:

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

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element



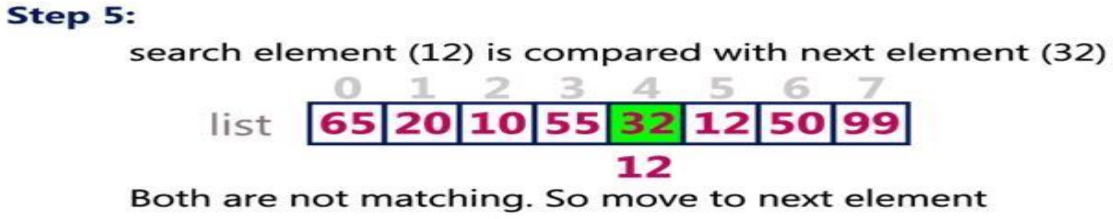
Step 4:

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

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element



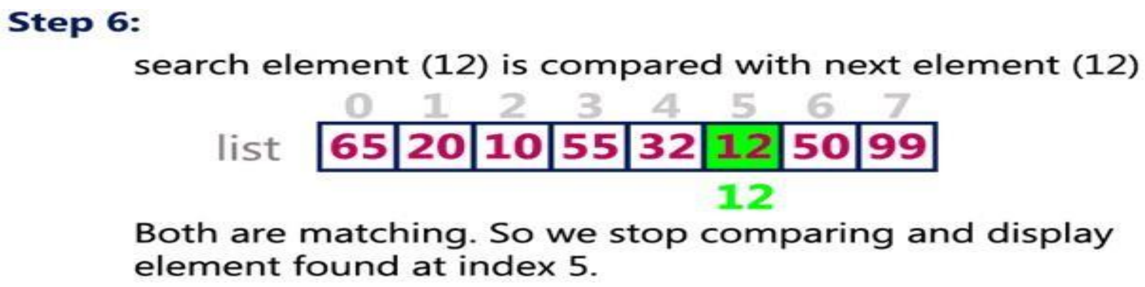
Step 5:

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

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

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 orderto search an element into some list by using binary search technique, we must ensure that the listis 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.

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.

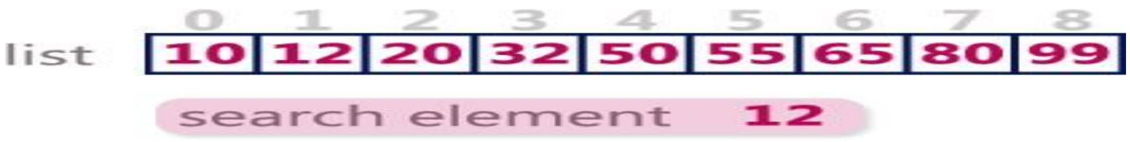
Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the leftsub list of the middle element.

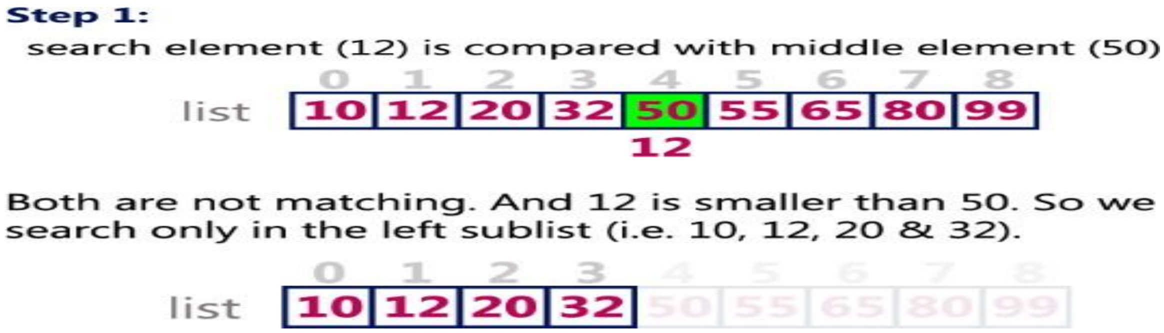
Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the rightsub list of the middle element.

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:

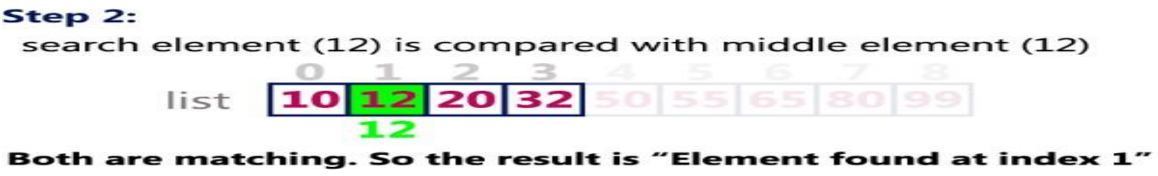




Step 1:

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

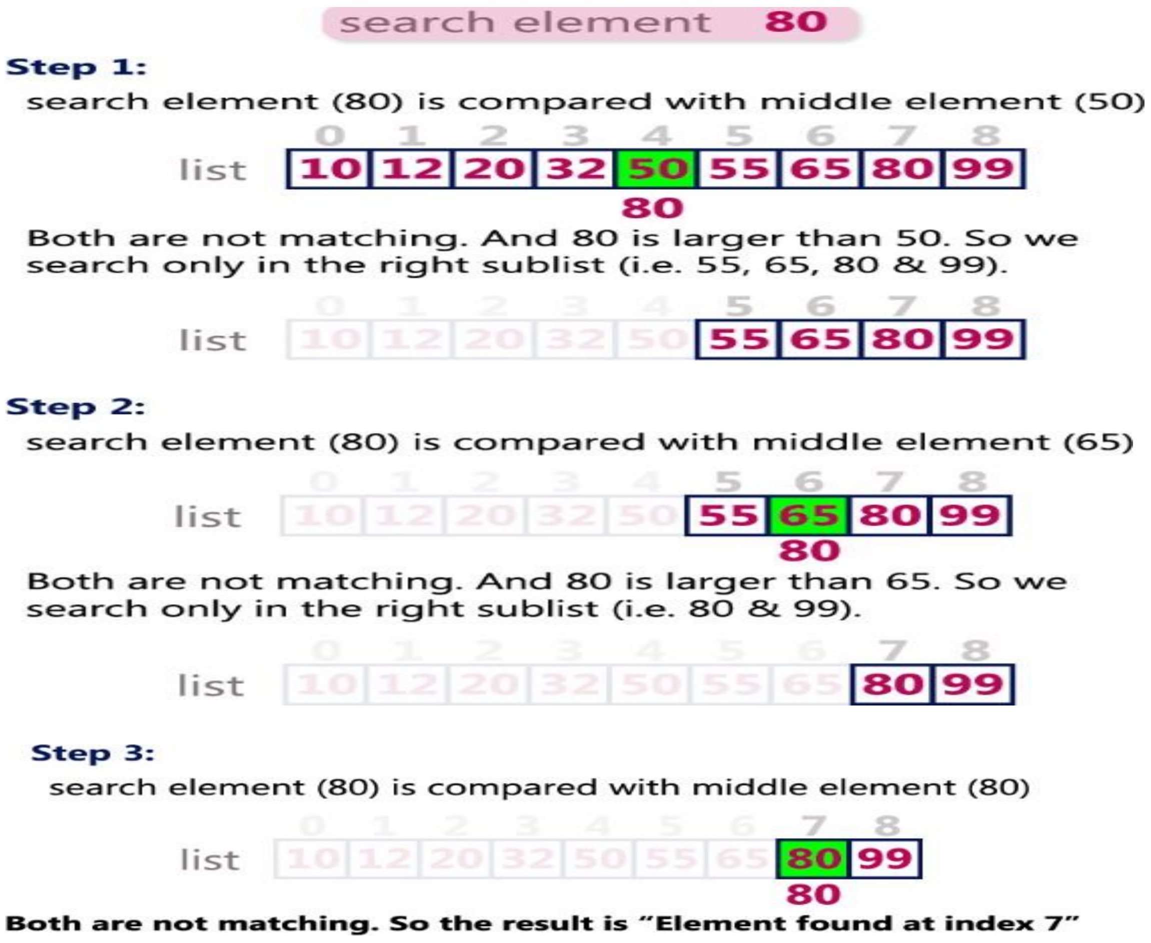
Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).



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:



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"

<u>FIBONACCI SEARCH:</u>

- **Fibonacci search** is an efficient search algorithm based on **divide and conquer** principle that canfind 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 denotinga 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 i^{th} 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 = 1

F(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 fromfront 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.**

Hashing:

Hashing in the **data structure** is a technique of mapping a large chunk of data into small tables using a hashing function. It is also known as the message digests function. It is a technique that uniquely identifies a specific item from a collection of similar items.

It uses hash tables to store the data in an array format. Each value in the array has assigned a unique index number. Hash tables use a technique to generate these unique index numbers for each value stored in an array format. This technique is called the hash technique.

You only need to find the index of the desired item, rather than finding the data. With indexing, you can quickly scan the entire list and retrieve the item you wish. Indexing also helps in inserting operations when you need to insert data at a specific location. No matter how big or small the table is, you can update and retrieve data within seconds.

Hashing in a data structure is a two-step process.

1. The hash function converts the item into a small integer or hash value. This integer is used as an index to store the original data.
2. It stores the data in a hash table. You can use a hash key to locate data quickly.

Hash Function

The hash function in a data structure maps arbitrary size of data to fixed-sized data. It returns the following values: a small integer value (also known as hash value), hash codes, and hash sums.

hash = hashfunc(key)

index = hash % array_size

The function must satisfy the following requirements:

- A good hash function is easy to compute.
- A good hash function never gets stuck in clustering and distributes keys evenly across the hash table.
- A good hash function avoids collision when two elements or items get assigned to the same hash value.

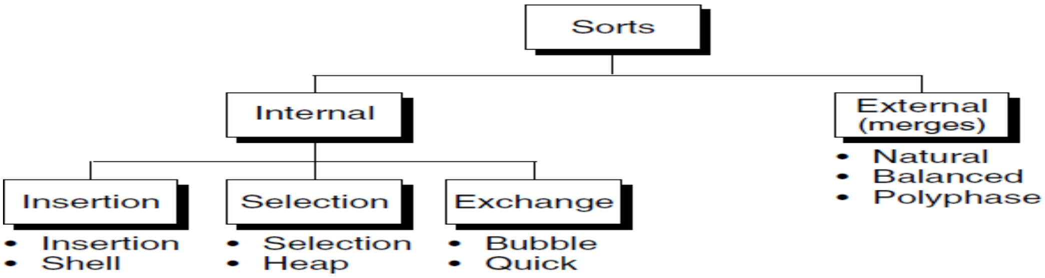
SORTINGS:

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

Categories of Sorting:

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

- Internal Sorting
- External Sorting

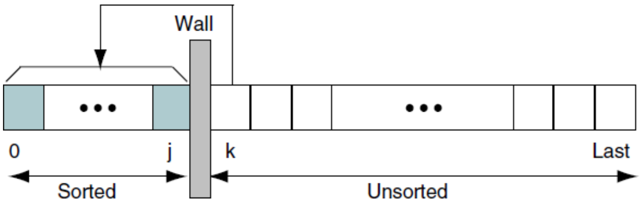


- **Internal Sorting:** When all the data that is to be sorted can be accommodated at a time in the main memory (Usually RAM). Internal sorting has five different classifications: insertion, selection, exchanging, merging, and distribution sort
- **External Sorting:** When all the data that is to be 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:

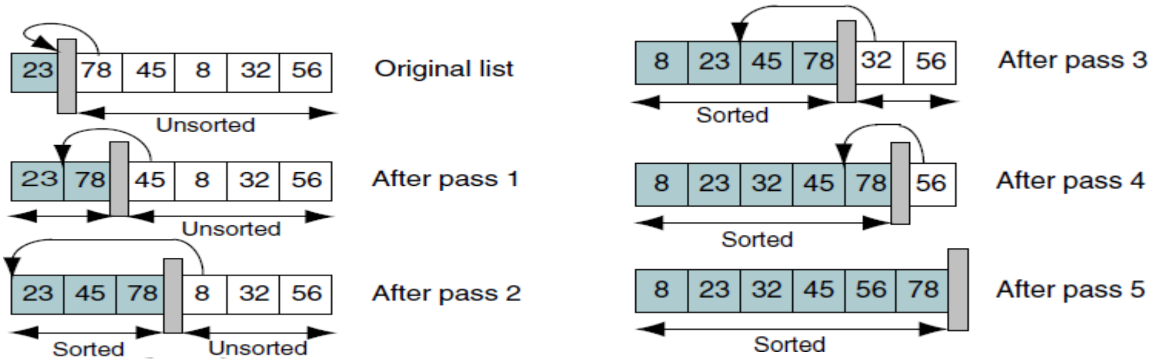
- In Insertion sort the list can be divided into two parts, one is sorted list and other is unsorted list. In each pass the first element of unsorted list is transfers to sorted list by inserting it in appropriateposition or proper place.
- The similarity can be understood from the style we arrange a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



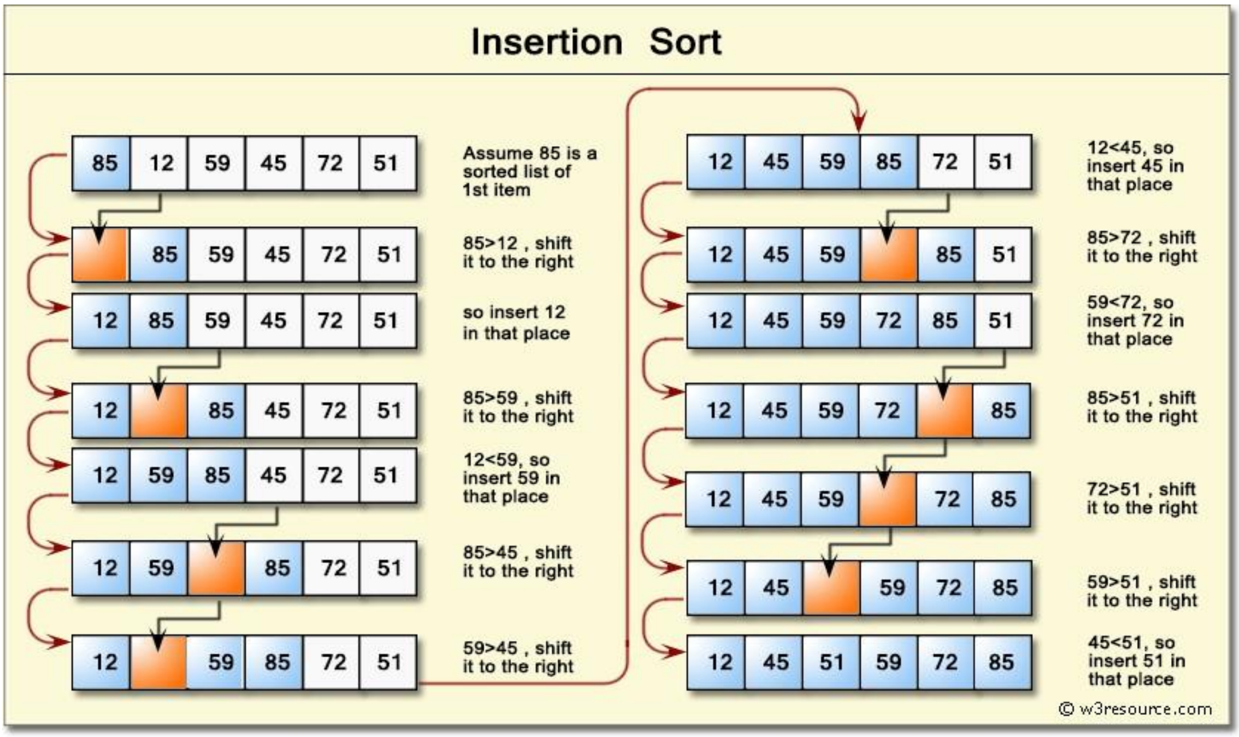
Following are the steps involved in insertion sort:

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 firstelement.
   - 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 1:

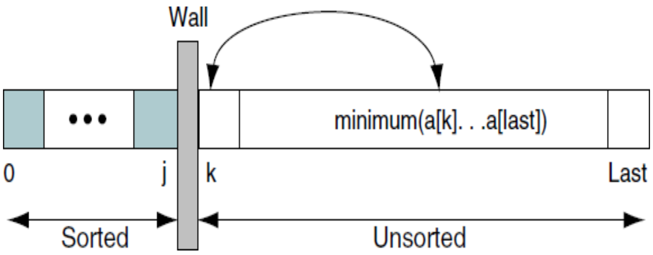


Example 2:


Insertion Sort
85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place
© w3resource.com

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 is sorted list other is unsorted list. Find out the smallest element in the unsorted list and it is exchange with the starting position of unsorted list, after that it will added in to sorted list.



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

asked to sort a list on paper.

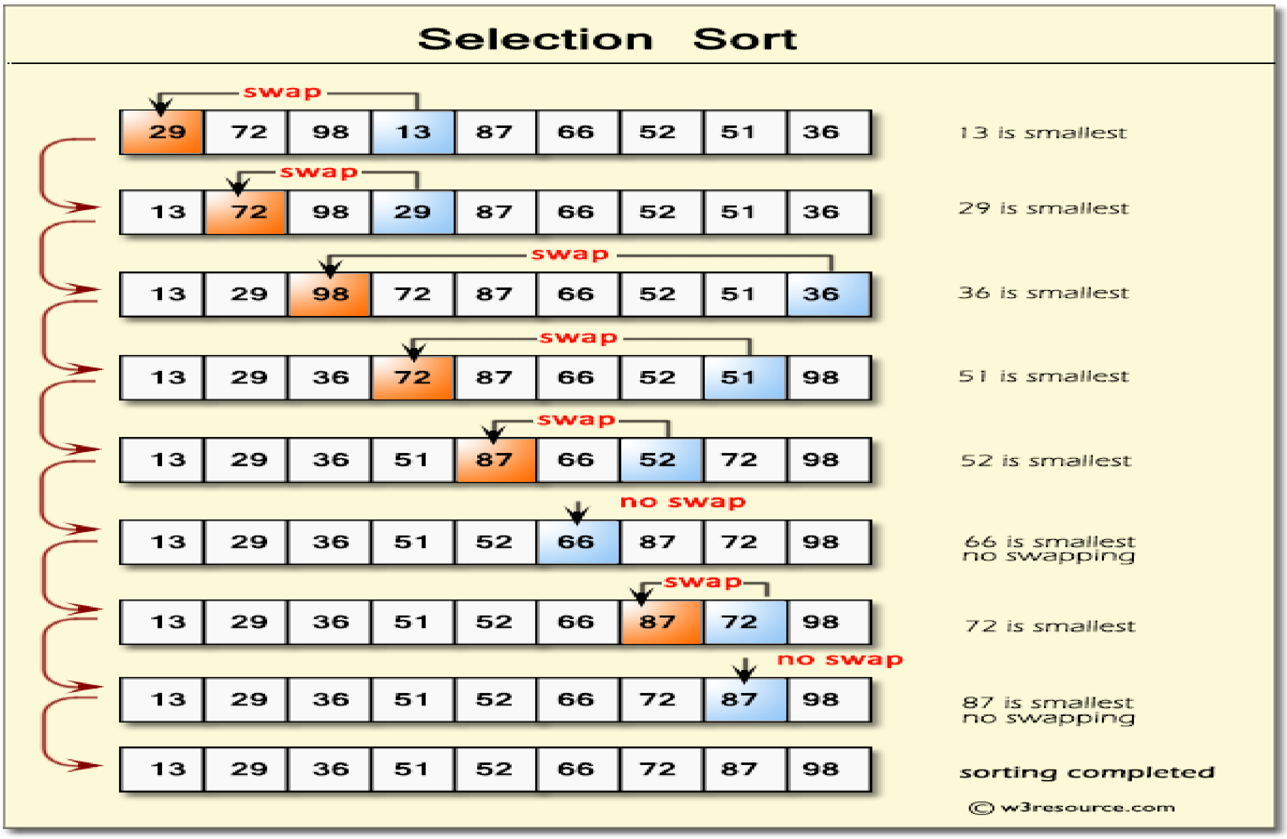
Algorithm:

```
SELECTION SORT (ARR, N)
Step 1: Repeat Steps 2 and 3 for K = 1 to N-1Step
2: CALL SMALLEST (ARR, K, N, Loc)
Step 3: SWAP A[K] with ARR[Loc]
Step 4: EXIT
```

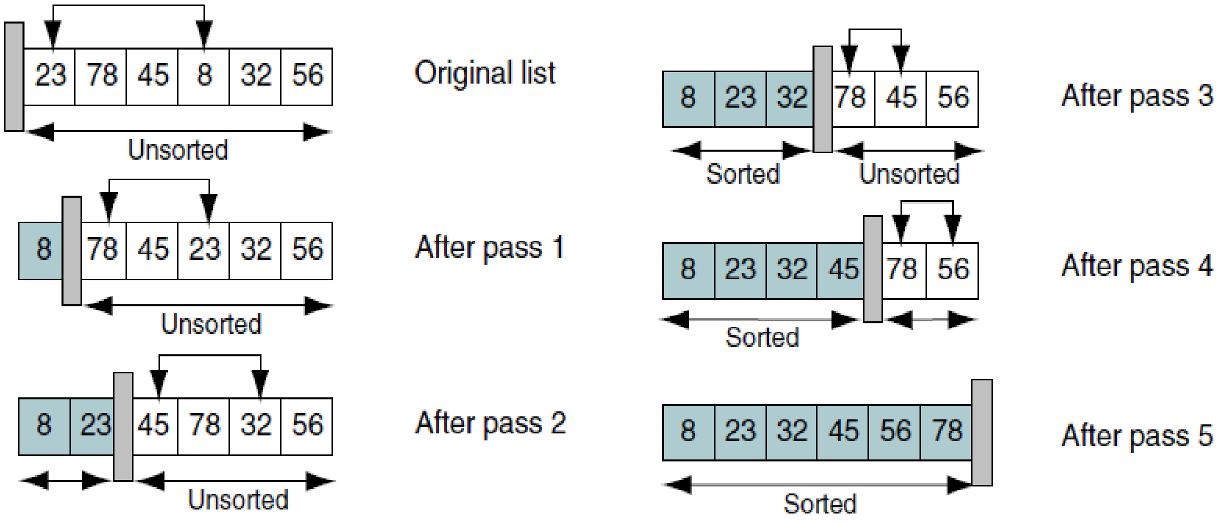
Algorithm for finding minimum element in the list.

```
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:

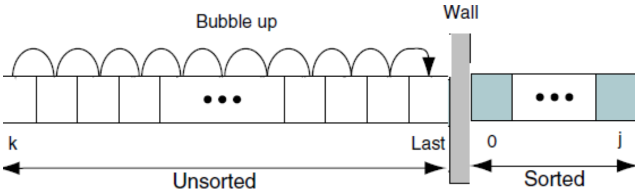
T(n) <= (N-1)*(N-1)

T(n) <= $(N-1)^2$

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

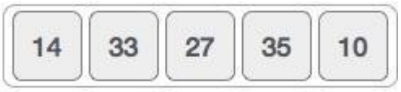
BUBBLE SORT:

- 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 are 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 tosort the data.



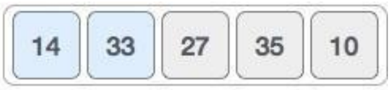
Example 1:

- We take an unsorted array for our example.



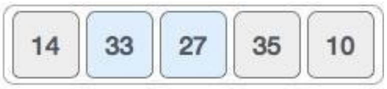
14 33 27 35 10

- Bubble sort starts with very first two elements, comparing them to check which one is greater.



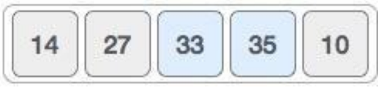
14 33 27 35 10

- In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33with 27. We find that 27 is smaller than 33 and these two values must be swapped.



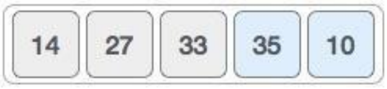
14 33 27 35 10

- Next we compare 33 and 35. We find that both are in already sorted positions.



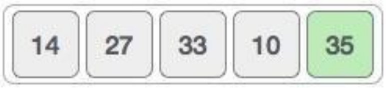
14 27 33 35 10

- Then we move to the next two values, 35 and 10. We know then that 10 is smaller 35.



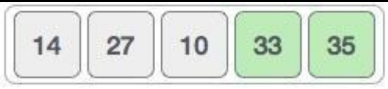
14 27 33 35 10

- We swap these values. We find that we have reached the end of the array. After one iteration, thearray should look like this −

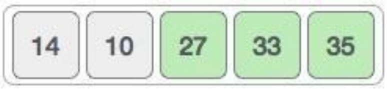


14 27 33 10 35

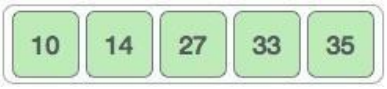
- To be defined, we are now showing how an array should look like after each iteration. After thesecond iteration, it should look like this



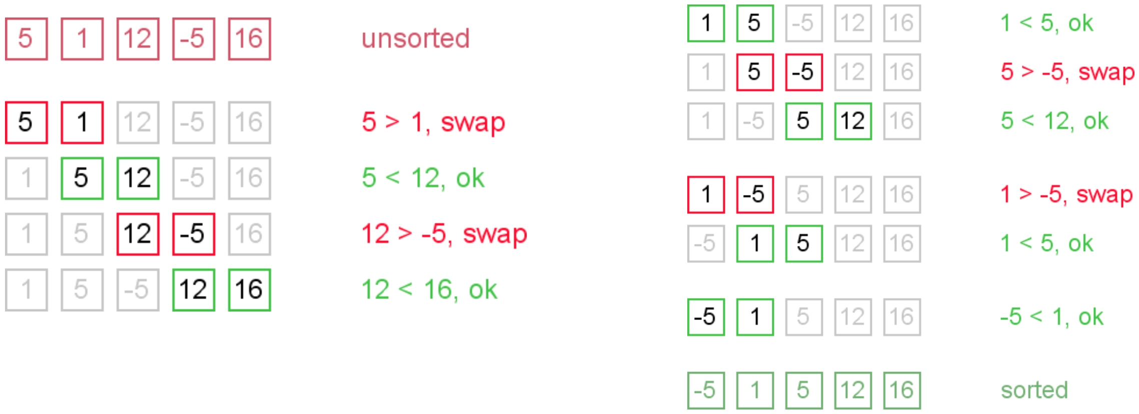
□ Notice that after each iteration, at least one value moves at the end.



□ And when there's no swap required, bubble sorts learns that an array is completely sorted.



Example 2:



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
```

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:

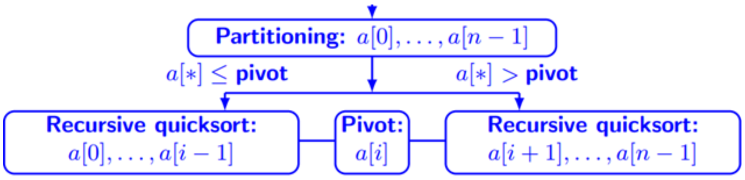
$T(n) <= (N-1)*(N-1)$

$T(n) <= (N-1)^2$

Finally, The time complexityis

$O(n^2)$.

QUICK SORT:


Partitioning: $a[0], \dots, a[n-1]$
$a[*] \leq$ pivot
$a[*] >$ pivot
Recursive quicksort: $a[0], \dots, a[i-1]$
Pivot: $a[i]$
Recursive quicksort: $a[i+1], \dots, a[n-1]$

- 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.

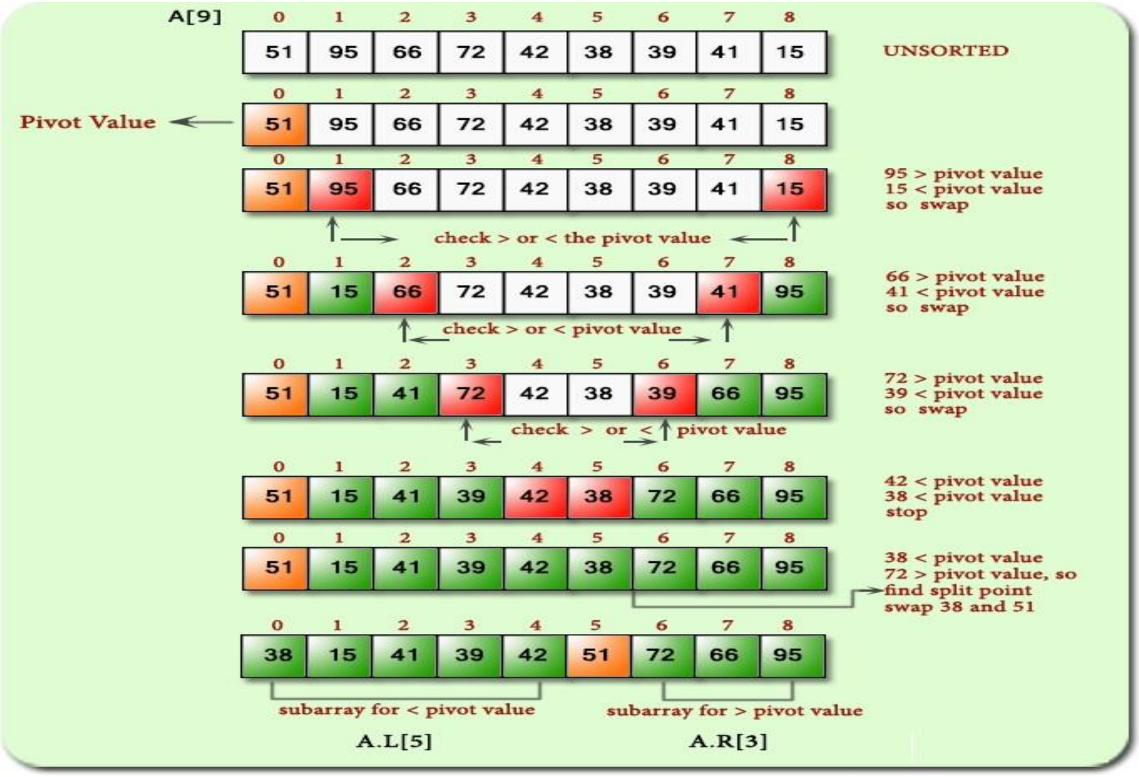
How does Quick Sort Partitioning Work

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 movethe left pointer to the right (add 1 to the left index). Continue this until left side element is greaterthan 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 elementwith right pointer.

Algorithm:

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

Example:



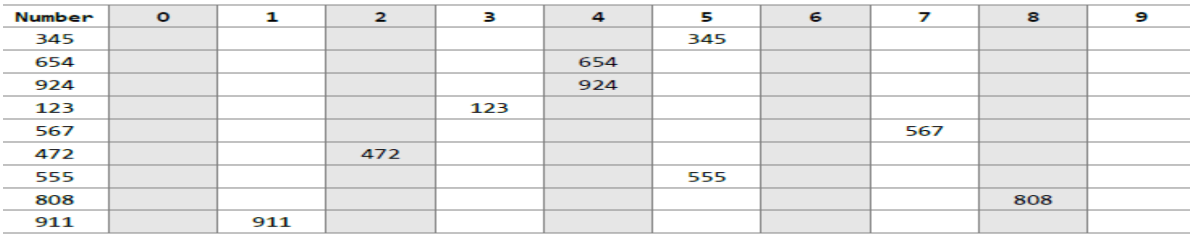
RADIX SORT:

- 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 areused 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 nth pass, where n is thelength 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 numberof passes will depend on the **length** of the number having maximum number of digits.

Example 1: Sort the numbers given below using radix sort.345,

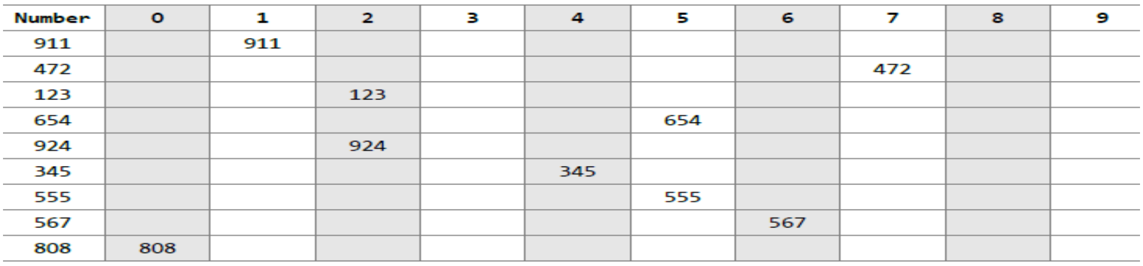
654, 924, 123, 567, 472, 555, 808, 911

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



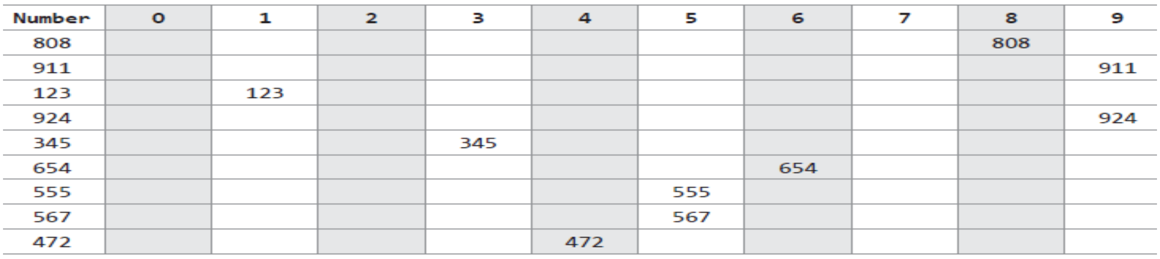
Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

- 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 finalsorted list. 123, 345, 472, 555, 567, 654, 808, 911, 924.

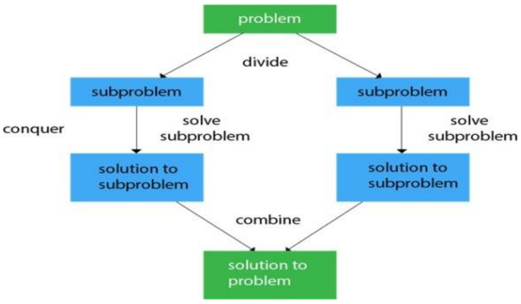
Algorithm:

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 the corresponding bucket.
7. Read the elements (S) of the bucket from 0th bucket to 9th bucket and from the first position to the higher one to generate new array A.
8. Display the sorted array A.
9. Exit.

Divide and Conquer:

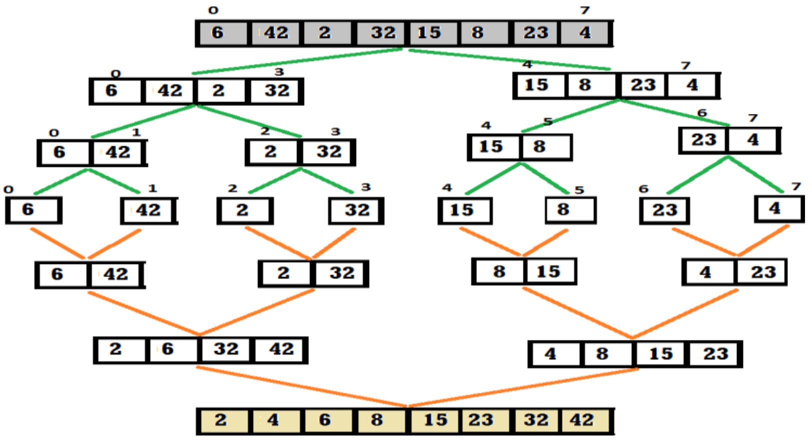
- Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solvingthe problem 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-problemsto 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 sub lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list.

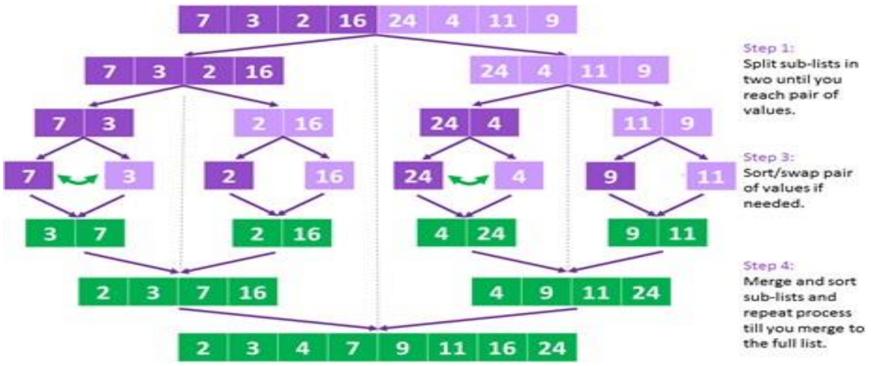
Implementation Recursive Merge Sort:

- 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:



Step 1:
Split sub-lists in two until you reach pair of values.

Step 3:
Sort/swap pair of values if needed.

Step 4:
Merge and sort sub-lists and repeat process till you merge to the full list.

Example 2:
MergeSort Algoritm:

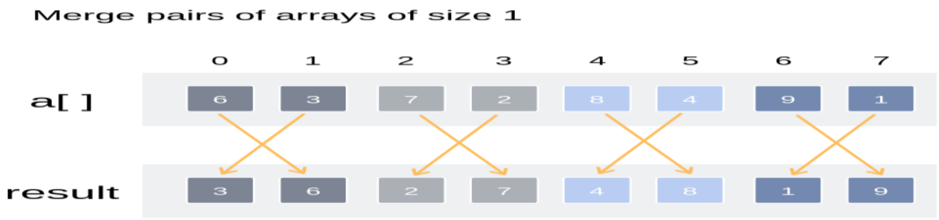
```
MergeSort(A, lb, ub )
{
    If  lb<ub
        {
            mid = floor(lb+ub)/2;
            mergeSort(A, lb, mid)
            mergeSort(A, mid+1, ub)
            merge(A, lb, ub , mid)
        }
}
```

Iterative Merge Sort:

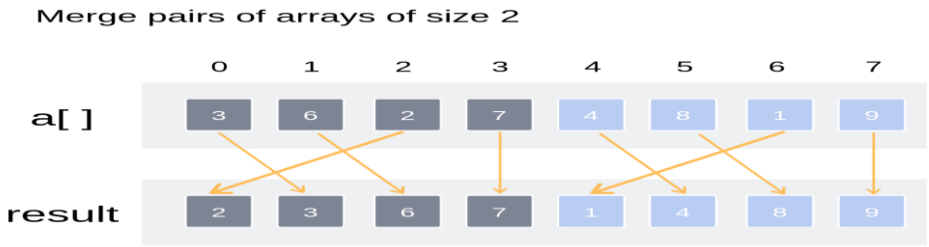
The Bottom-Up merge sort approach uses iterative methodology. It starts with the "single-element" array, and combines two adjacent elements and also sorting the two at the same time. The combined-sorted arrays are again combined and sorted with each other until one single unit of sorted array is achieved.

Example: Let us understand the concept with the following example.

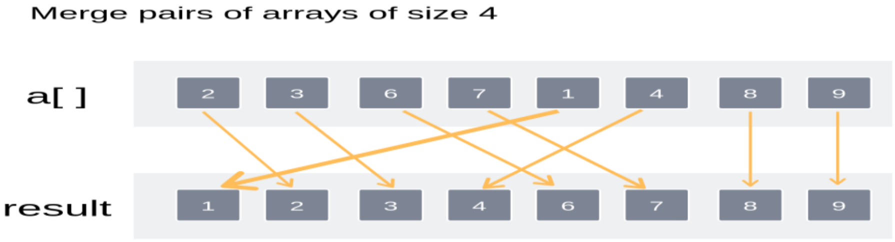
Iteration 1:



Iteration 2:

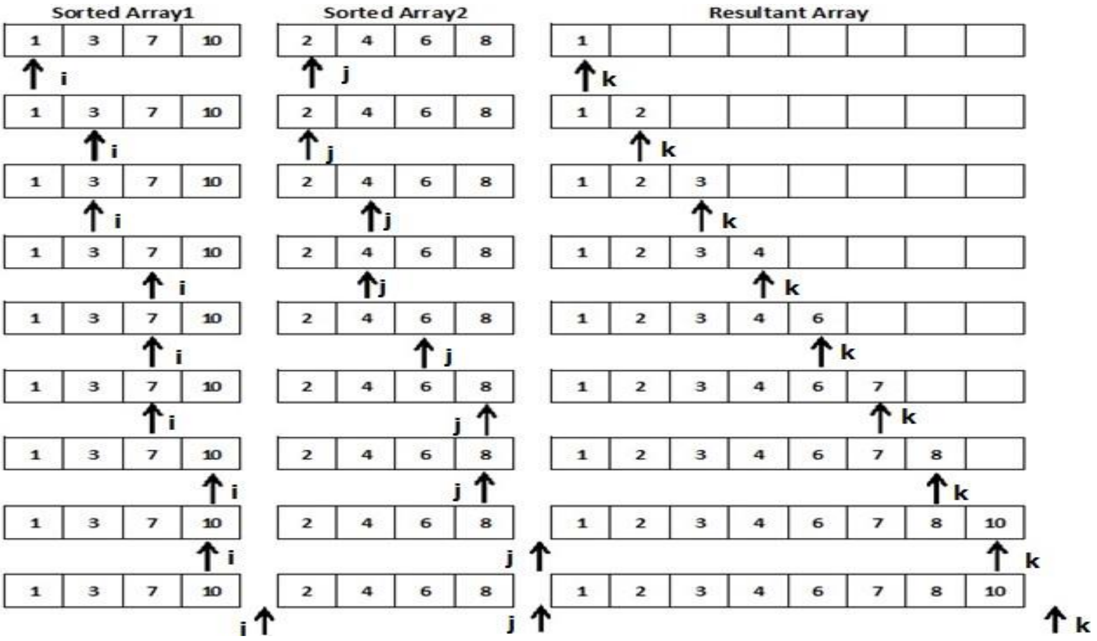


Iteration 3:



Thus the entire array has been sorted and merged.

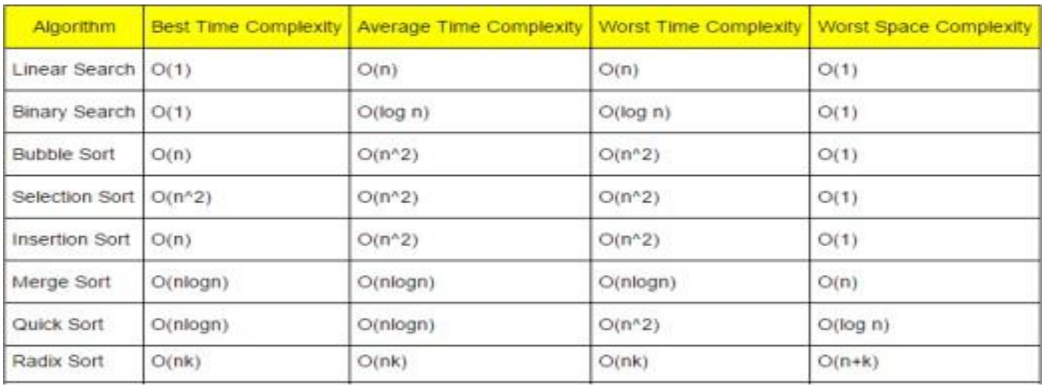
Two- Way Merge Sort:



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.
```

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)

SHELL SORT:

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

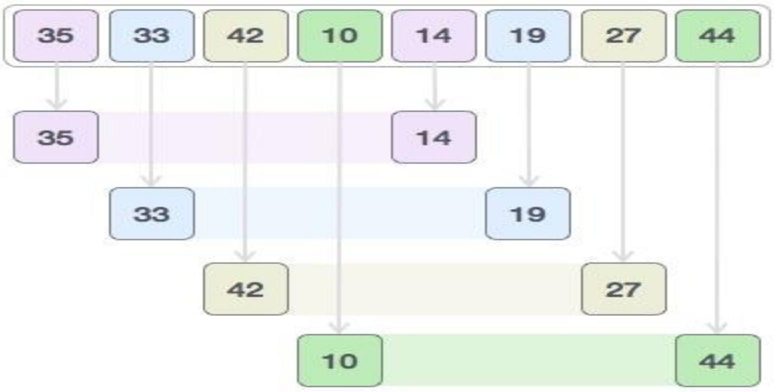
h = h * 3 + 1

Where −

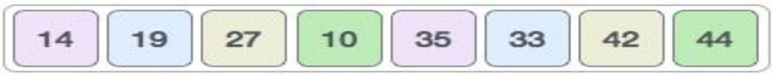
h is interval with initial value 1

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).

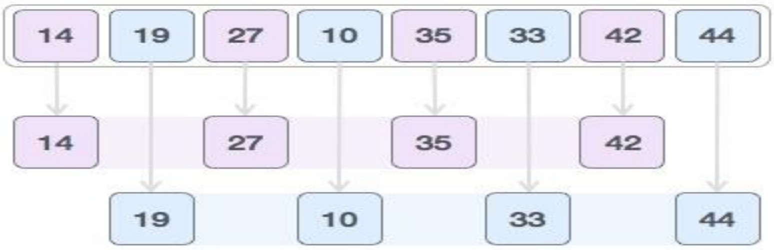
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



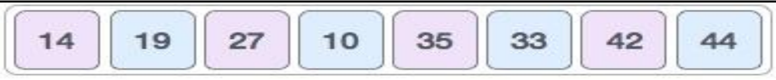
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −



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



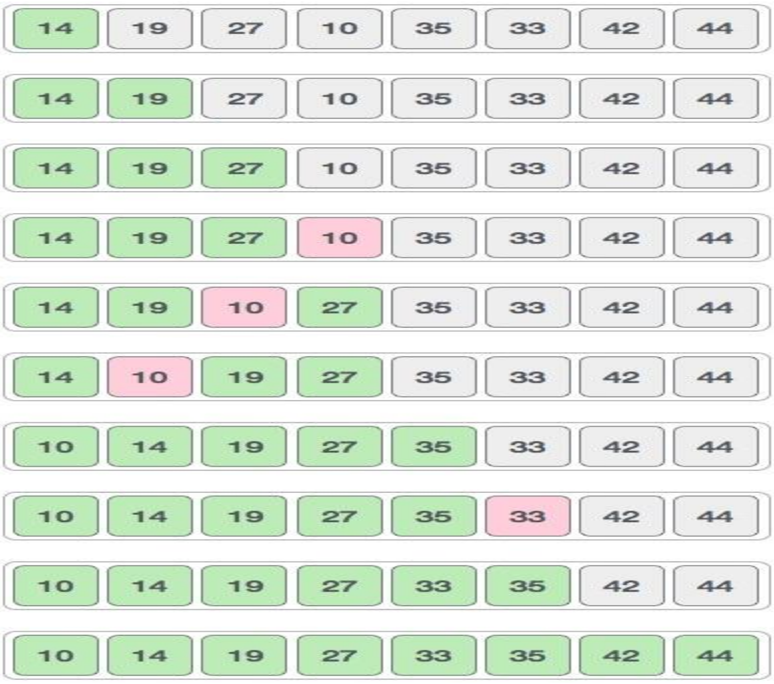
We compare and swap the values, if required, in the original array. After this step, the array should look like this −



14	19	27	10	35	33	42	44

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

Following is the step-by-step depiction −



14	19	27	10	35	33	42	44
14	19	27	10	35	33	42	44
14	19	27	10	35	33	42	44
14	19	27	10	35	33	42	44
14	19	10	27	35	33	42	44
14	10	19	27	35	33	42	44
10	14	19	27	35	33	42	44
10	14	19	27	35	33	42	44
10	14	19	27	33	35	42	44
10	14	19	27	33	35	42	44

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

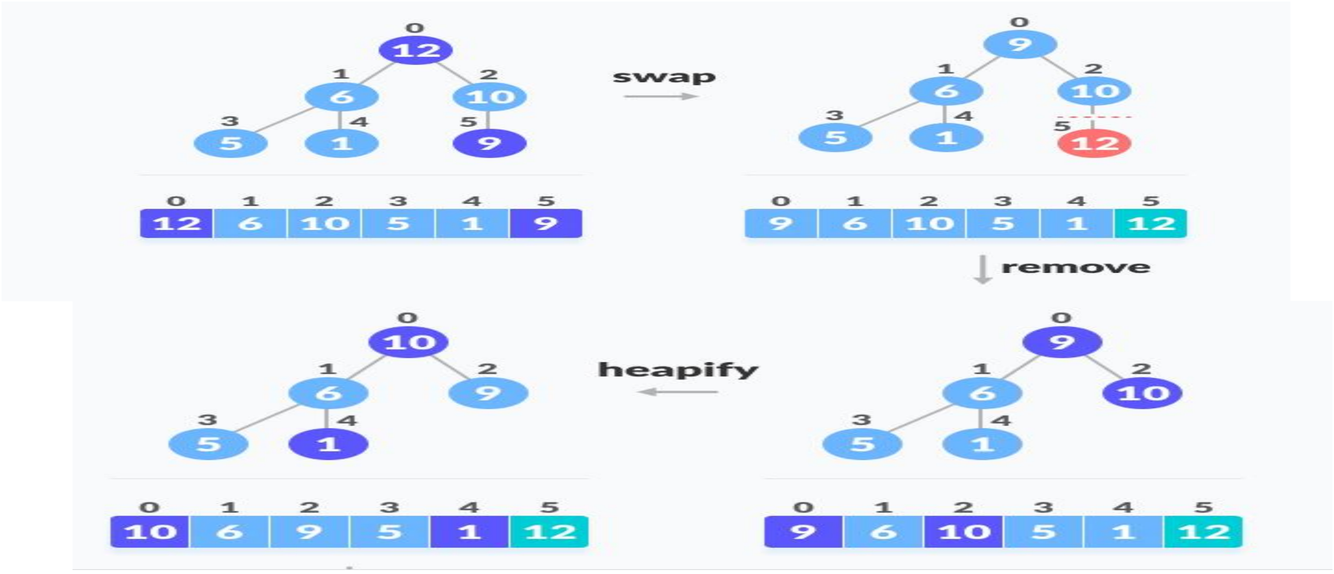
Algorithm

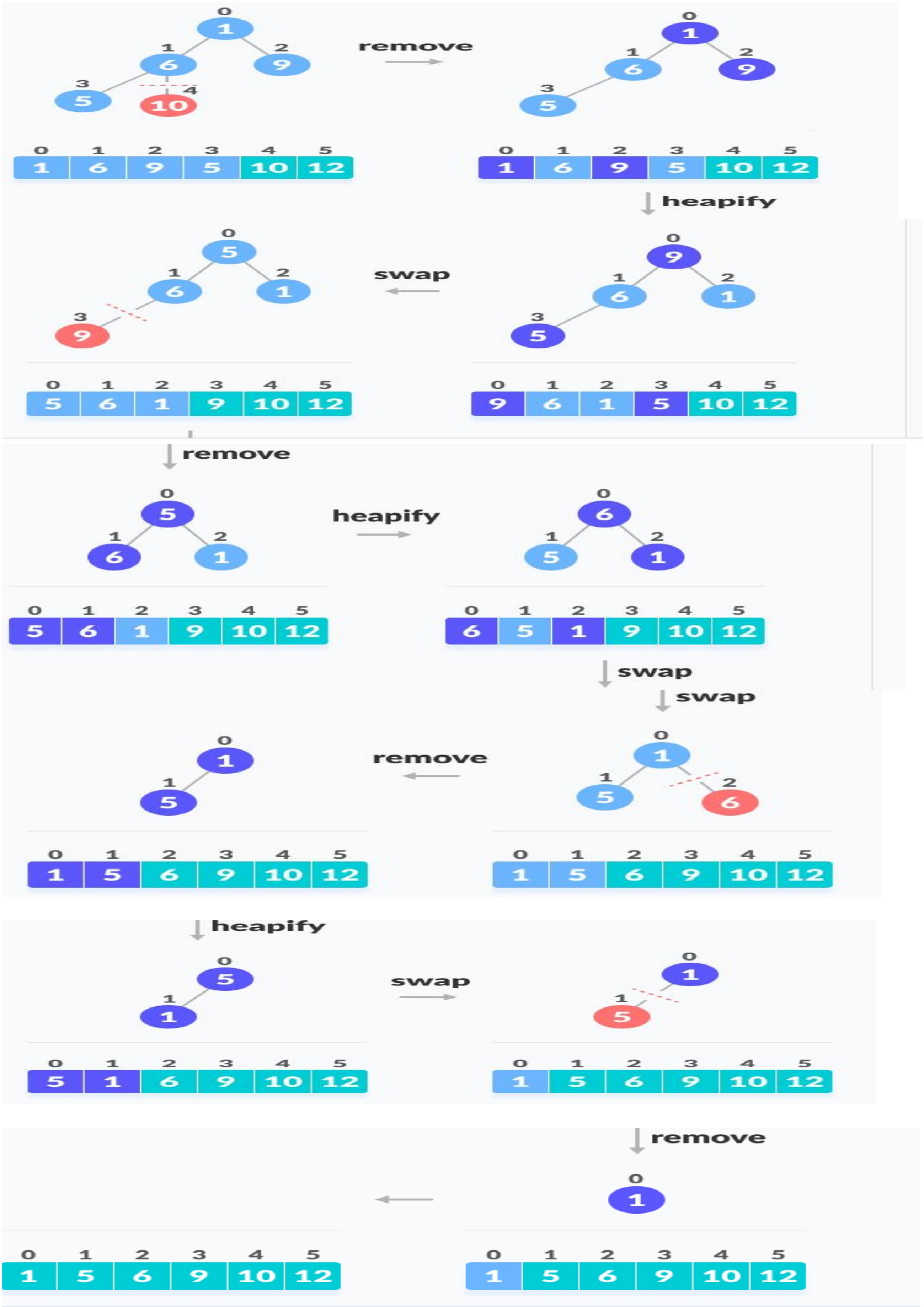
Following is the algorithm for shell sort.

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 3 − Repeat until complete list is sorted

HEAP SORT:

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.





Swap, Remove, and Heapify

Time Complexity

Best	O(nlog n)

Worst	O(nlog n)
Average	O(nlog n)
Space Complexity	O(1)